BASIC 2 Plus

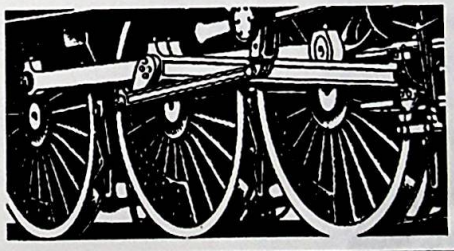
USER GUIDE &
QUICK REFERENCE

LOCOMOTIVE
SOFTWARE

BASIC 2 Plus

Locomotive Software's BASIC
for the Amstrad PC1512/1640 and other PCs
running Digital Research's GEM Environment Manager

User Guide and Quick Reference



LOCOMOTIVE SOFTWARE

Whilst every effort has been made to verify that this software works as described, it is not possible to test any program of this complexity under all possible conditions or circumstances. Therefore BASIC 2 Plus is supplied 'as is' without warranty of any kind either express or implied.

The particulars given in this manual are given by Locomotive Software in good faith. However, BASIC 2 Plus is subject to continuous development and improvement, and it is acknowledged that there may be errors or omissions in this manual. If you experience any difficulty with the use of this product, please write to:

Locomotive Software Ltd
Allen Court
Dorking
Surrey RH4 1YL

Please enclose a disc containing any relevant programs. We will return it with our reply.

Locomotive Software reserves the right to revise this manual without prior warning.

Written by Martin S Taylor
Published by Locomotive Software Ltd
Printed by Ashford Colour Press, Gosport, Hants

First Published 1989
ISBN 1 85195 018 4

Preface

This User Guide and Quick Reference is one of two instruction manuals supplied with BASIC 2 Plus. The other is the Language Reference Manual.

The User Guide section of this manual explains how to use the BASIC 2 Plus program. It describes how to enter programs into your computer, how to save them, and so on. There is also a complete description of BASIC 2 Plus's powerful mouse-driven editor, and a large section on the debugging tools available.

The User Guide does not give a full description of how the commands work. For that, you must refer to the Language Reference Manual, the second book supplied with BASIC 2 Plus. This descibes the language in great detail. However, the User Guide does give a 'Quick Reference' summary of the syntax of all the commands available in BASIC 2 Plus, each with brief explanation of what they do.

If you have programmed in BASIC 2, the predecessor of BASIC 2 Plus, you will notice some new facilities. Such structures as the **SELECT CASE** statement and the block **IF** statement make structured programming much easier. Routines are now implemented, and it is now possible for your program to be split into different program units – a main program and several modules. Read Chapters 5, 6, and 7 in theLanguage Reference Manual for details.

When programming, you will see that you now have a powerful mouse-driven 'cut-and-paste' editor to work with. There are lots of facilities for debugging your programs, too. But perhaps the first thing you'll notice, especially if you are running very large programs, is the speed of BASIC 2 Plus. One of the standard test programs runs over *six* times faster in BASIC 2 Plus than in BASIC 2.

IMPORTANT: *Before running any **BASIC 2** program under BASIC 2 Plus, you are advisedfirst to scan through the program in the Edit window to see if any of the variables are being shown in capital letter. If any are, then you will need to change the name of this variable because this name is now used as a BASIC 2 Keyword.*

You are also advised to use BASIC 2 Plus's Check-out option to 'Pre-scan' your program before running it (as described in the section on 'Programming' in Chapter 1 of the User Guide).

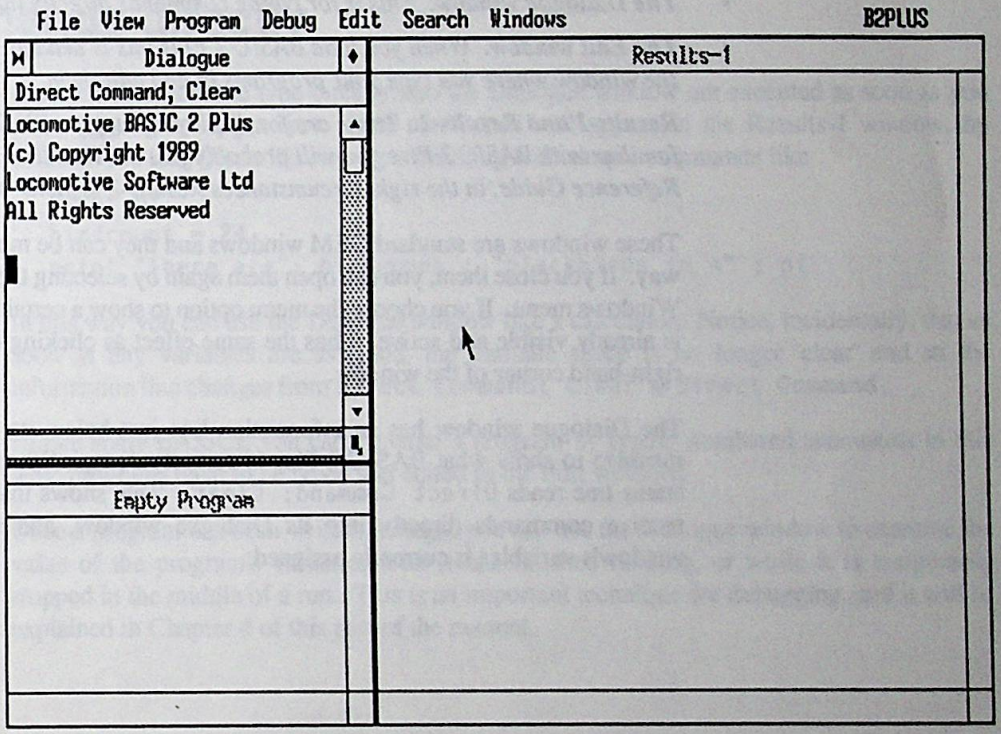
Contents

8909

BASIC's Basics

BASIC 2 Plus works with GEM, the window manager on your computer. This guide assumes that you have GEM installed on your computer and that you have some experience of using it; specifically, it assumes that you know how to select, scroll, resize, and close windows, how to pull down menus and select options from them, and that you understand how to select files from a disc using the Item Selector Dialog. If you aren't familiar with these tasks, read about GEM in the manual that came with your computer.

Assuming, then, that you are reasonably familiar with GEM and the features it offers, make a back-up copy of your BASIC 2 Plus disc, load GEM, and then double click on the file B2PLUS.APP to load BASIC. You should see the BASIC 2 Plus screen; it looks like this:



If you see BASIC 2 Plus start to open but you are almost immediately returned to the GEM desktop, you probably haven't got enough memory available. Try removing fonts from your system, or freeing memory in other ways. (You can discover how much free memory there is available by typing `PRINT FRE` in the Dialogue window.)

There are various things to note about the screen. As with most GEM applications, there is a menu bar at the top of the screen providing assorted commands which are always available for you to use.

Notice the windows on the screen. Generally when BASIC 2 Plus starts up, there are four of these, though one of them (Results-2) is covered up by the Edit window and the Dialogue window. (If you are short of memory, Results-2 may not be displayed at all. This isn't a serious problem, though it may restrict you a little if you need to make full use of BASIC 2 Plus's output facilities.)

The four windows are:

- ***The Dialogue window.*** *This is for typing commands directly into BASIC 2 Plus.*
- ***The Edit window.*** *When you load BASIC 2 Plus this is headed* `Empty Program`*. This is the window where you type your programs in and edit them.*
- ***Results-1 and Results-2.*** *These are for displaying output. While you are becoming familiar with BASIC 2 Plus you will probably just use Results-1, though as explained in the Reference Guide, in the right circumstances Results-2 is faster and holds more information.*

These windows are standard GEM windows and they can be moved and resized in the usual way. If you close them, you can open them again by selecting the appropriate option from the Windows menu. If you choose the menu option to show a certain window when that window is already visible and active, it has the same effect as clicking on the 'grow box' in the top right-hand corner of the window.

The Dialogue window has an information line just below its name bar. This displays a message to show what BASIC 2 Plus is curently doing. When you load the program, the status line reads `Direct Command; Clear`. This shows that BASIC 2 Plus is ready to receive commands directly into its Dialogue window, and that none of the Dialogue window's variables is currently assigned.

FRE

Use FRE is a function which returns the amount of unused memory.

Syntax FRE

CLEAR

Use CLEAR discards all variables.

Syntax CLEAR

Note CLEAR may only be in Direct Mode.

Typing commands directly

Commands which you type directly into the Dialogue window are executed as soon as you press ↵. Any output from these commands will appear in the Results-1 window, by default. Try this, if you like, by experimenting with simple commands like

```
print pi
birdcount = 24
print "There are"; birdcount ; "blackbirds in a" ; pi
```

In this way you can use the Dialogue window like a calculator. Notice, incidentally, that as soon as any variables are assigned, the variable space is no longer 'clear' and so the information line changes from `Direct Command; Clear` to `Direct Command.`

Unlike some BASICs, you cannot write a program by typing numbered statements in this way: programs have to be typed and edited in the Edit window.

Once a program has been written, though, you can use the Dialogue window to examine the value of the program's variables after it has finished running, or while it is temporarily stopped in the middle of a run. This is an important technique for debugging, and it will be explained in Chapter 4 of this part of the manual.

Programming

To write a simple program, click on the Edit window (the one with `Empty Program` in the title bar) to make it active, and then enter the text of your code. If you make mistakes, you can erase them with the ⌫ key or you can use BASIC 2 Plus's sophisticated built-in editor described in Chapter 2.

Notice how each line appears in the Edit window exactly as you type it; upper and lower case letters appear as they are input from the keyboard. But as soon as you press ↵ (or move the cursor to a new line by any other means) BASIC 2 Plus's keywords are transformed into capital letters, while your invented names and identifiers are transformed into lower case. Notice also how carriage return is displayed as ▼.

When your program is ready, select `Run` from the Program menu. The first thing BASIC 2 Plus does is pre-scan the program, checking for syntax and other simple errors. This means that elementary errors in your code will be discovered very quickly, even if they appear in a rarely executed part of the program.

If there are any pre-scan errors, a message will appear in the Dialogue window, and in the Edit window the cursor will move to the first offending line. You can then correct the error, and select `Run` again.

When you have removed all the pre-scan errors, selecting `Run` will start to run the program. This time the program will stop at run-time errors which the pre-scan could not discover:

RUN

Use `RUN` starts executing the current program.

Syntax `RUN`

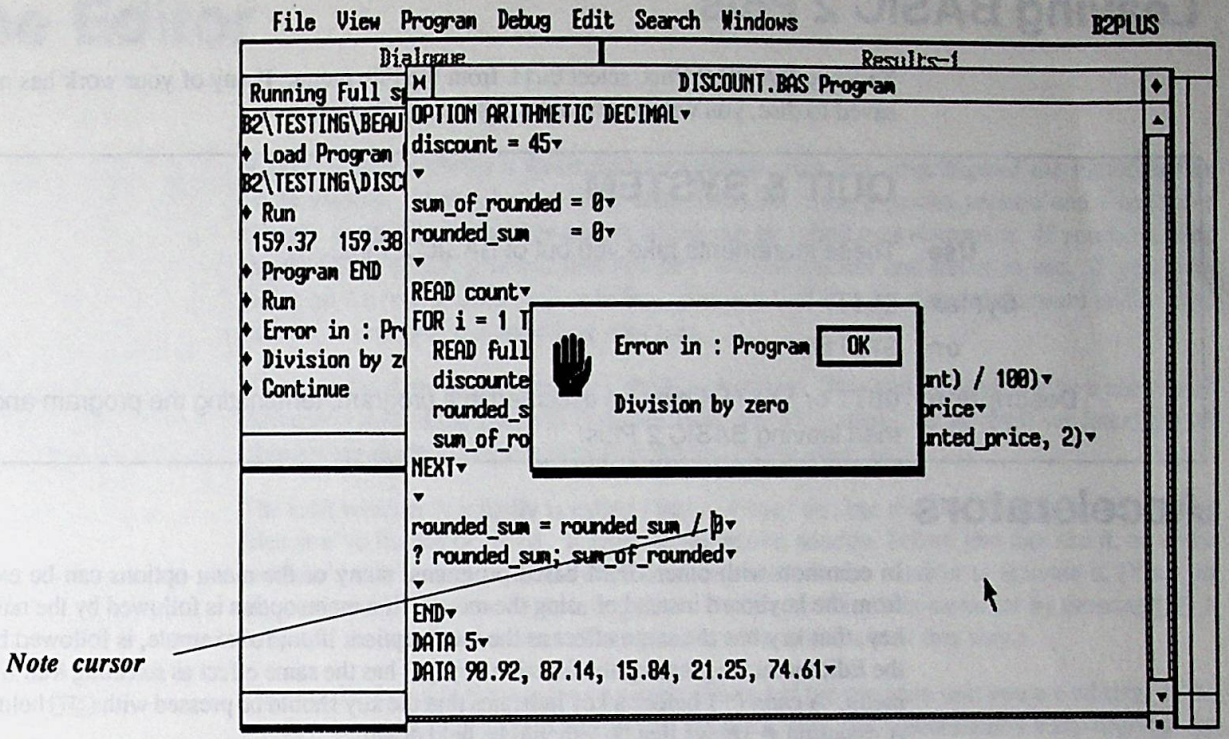
Description `RUN` may be executed in a program, terminating the program and starting it all over again.

EDIT

Use `EDIT` takes you into the edit window.

Syntax `EDIT`

Note `EDIT` may only be in Direct Mode.



errors such as division by zero, or references to a non-existent file, for example. When a run-time error is discovered, the program halts and an Alert is displayed with a brief indication of what the problem is. As with syntax errors, the cursor will position itself near the offending statement in the Edit window. For example, see the picture on the following page.

When a program stops because of a run-time error, you can be sure that execution has at least started, and so in tracking down the bug it may be helpful if to inspect the program's variables. (You can do this by entering `PRINT` *variable-name* into the Dialogue window.) Changing the program will mean losing the contents of these variables, and you may not wish to do this; so before you can edit your program after a run-time bug, a dialogue appears warning you that you are about to destroy the value held in your program's variables.

If you need to pre-scan a program without running it, select the option `Check out` on the Program menu.

If the program runs successfully, it will finish by displaying an Alert with the words `Program END`. Clear the dialogue by clicking on `OK`.

Leaving BASIC 2 Plus

To leave BASIC 2 Plus, select `Quit` from the File menu. If any of your work has not been saved to disc, you will be given the opportunity to save it.

QUIT & SYSTEM

Use These statements take you out of BASIC 2 Plus.

Syntax `QUIT`

or `SYSTEM`

Description `QUIT` or `SYSTEM` may be executed in a program, terminating the program and then leaving BASIC 2 Plus.

Accelerators

In common with other GEM-based programs, many of the menu options can be executed from the keyboard instead of using the mouse. If a menu option is followed by the name of a key, that key has the same effect as the menu option. `Run`, for example, is followed by f9 in the Edit menu; so pressing the function key [F9] has the same effect as selecting `Run` from the menu. A caret (^) before a key indicates that the key should be pressed with [Ctrl] held down. A diamond ◆ shows that [Alt] should be held down.

Accelerators for commands in the Windows menu are not available when a program is running, and for this reason they are not displayed in the menu. They are:

[F1]	Show Results-1
[F2]	Show Results-2
[F3]	Show Edit or Results-3
[F4]	Show Dialogue or Results-4
[Ctrl] + [F1]	Hide Results-1
[Ctrl] + [F2]	Hide Results-2
[Ctrl] + [F3]	Hide Edit or Results-3
[Ctrl] + [F4]	Hide Dialogue or Results-4
[Alt] + [F3]	Show Edit
[Alt] + [F4]	Show Dialogue

`Show Edit` also displays the Dialogue window.

`Show Results-3` and `Show Results-4` are not available unless these windows have been opened on a stream. See Chapter 9 of the Reference Manual to see how to do this.

The Editor

BASIC 2 Plus includes a powerful editor which takes full advantage of the facilities that GEM affords. Many changes have been made since the previous version and these have brought it into line with other editors which run in a similar environment. If you have used the BASIC 2 editor, you will find this new version quicker and easier to use. If you have never used a 'cut & paste' editor before, you will find it to be a straightforward and natural method of entering and amending your text.

All editing is done in one window – the Edit window. This can be used to give a number of different views of the program (selected from the View menu, as we shall see later) but all changes are made in the text view that BASIC 2 Plus starts in.

The Edit window is initially is called `Empty Program`, but this title changes to reflect any files you've loaded or saved. It must be the active window before you can use it, so if the window's title is dimmed, click the mouse anywhere in the window to activate it. (You can also activate the window by selecting Show Edit in the Windows menu (or by pressing F3); in fact, if the window has been closed, you have to activate it in this way.)

The text displayed in the window is simply the text of the program unit you are editing. If the window is not wide enough to hold any of the lines, they will automatically wrap round to the next line of the window and a small mark will appear at the beginning and the end of the broken line. This does not imply a new line of the program; program lines are always separated with typed carriage return characters, which are displayed on the screen as ▼.

Inserting text is quite straightforward. Click at the point in the program where you wish to insert the extra text; the cursor will move to that place. Now simply type the additional text, and it will be added at the position marked by the cursor.

The Carriage return symbols behave quite logically when you insert text. If you want to append more characters to the end of a line, you must take care to put the insertion point before the carriage return symbol, or the characters will be inserted after the carriage return, that is, at the start of the next line.

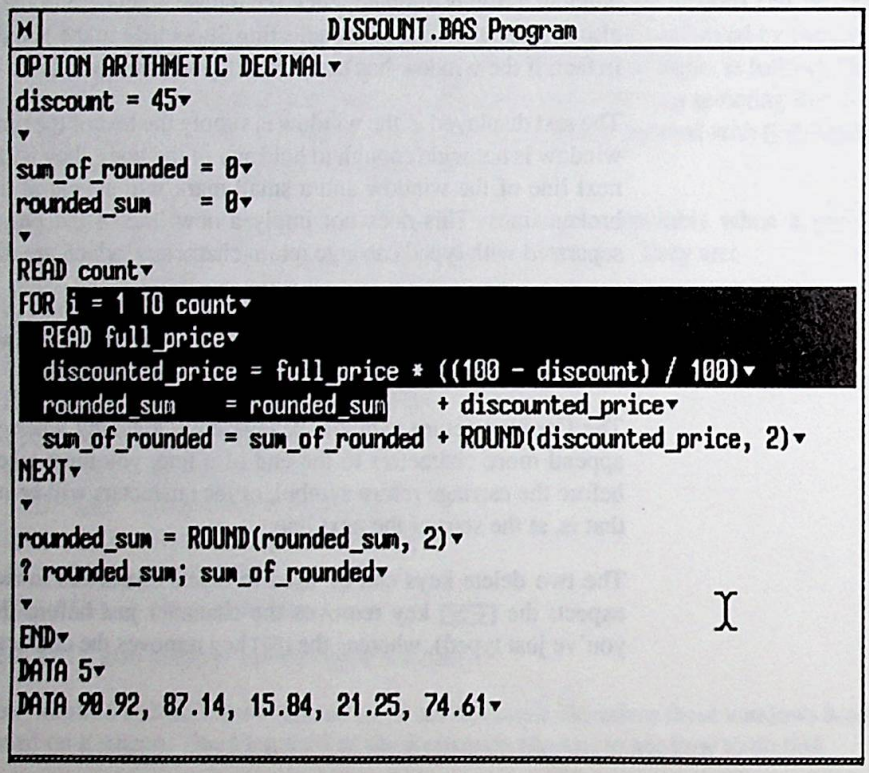
The two delete keys can be used to delete characters in two different directions as you'd expect: the ←Del key removes the character just before the cursor (that is, the character you've just typed), whereas the Del key removes the character just after it.

Selecting text

One important facility which the editor provides is a means to handle whole blocks of text quickly and conveniently. This is one area where the editor has been changed from BASIC 2. Instead of selecting the beginning and end of the block separately, they are selected by 'dragging' the pointer across text to select a continuous block, as described in the next few paragraphs.

Selecting a block

Begin by positioning the pointer at one end of the block you wish to select. Click the left-hand mouse button, but do not release it. Now 'drag' the mouse across the text (without releasing the button) and as you do so you will see that the block of text between the pointer and the place you originally clicked will become highlighted. (The exact colour change will depend on the system you are using.) Move the pointer to the other end of the block of text, and release the button. The whole block will stay highlighted, and the block is now selected and ready for you to operate on.



If you position the pointer inaccurately and find that the selected block is not what you want, clear the selection by clicking and releasing the mouse button anywhere on the window. The highlighting will clear as you do this. (On colour monitors there may be some flickering of colours as you do this.)

Selecting a Large Block

If you need to select more text than will fit into the window, select the start of the block as described above. Then, with the mouse button still held down, move the pointer beyond the top or bottom edge of the window. The text will scroll, highlighting as it goes. When enough text is in view, release the mouse button at the other end of your selected block. (When the text is scrolling, it will move rather faster if the cursor is well outside the window. If you want it to scroll slowly only move the cursor slightly beyond the edge.)

If the block you want to select is very large, there is an alternative technique. Using the method already described, select a small piece of text at one end of the block. Then move to the other end using the scroll bars, and press the ⇧ key as you click the mouse button in the appropriate place. All the text from the small block to the Shift-click will be selected, and will be highlighted on the screen.

Deleting a Block

To delete a block of text, select it and then press either of the delete keys. When a block of text is selected, ←Del and Del both have the same effect. Or, to replace the block with different text, select the block and then type. Typing when there is a block selected replaces the block with whatever is typed.

Cut and Paste

Instead of deleting a block of text, you may want to transfer it to another part of the program. The way to do this is by 'cutting and pasting' using two of the options available on the Edit menu.

Begin by selecting the block of text, as explained above. Do not delete it, but select `Cut` from the Edit menu. You will see the text disappear from your program. It has not been annihilated, though; it has been moved to a special area called the Clipboard. (Later, we shall see how to inspect the contents of the Clipboard and edit them directly.)

Position the cursor at the point where you want the text moved to. Select `Paste` (again from the Edit menu) and the contents of the clipboard will be inserted at the point selected.

Pasting text does not delete it from the clipboard, so you can paste it again in many different places if you like.

If you want to copy a block of text without deleting it from its original position, select it in the usual way and then choose `Copy` from the Edit menu. This replaces the Clipboard's contents with a copy of the selected text. The new contents of the Clipboard can be pasted wherever you like in the program.

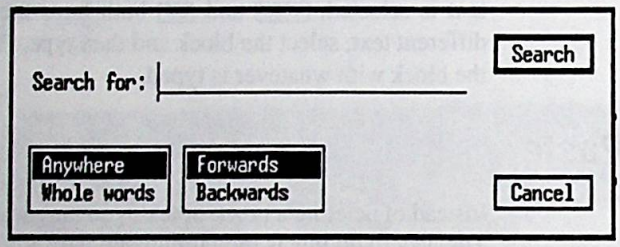
In fact, the Clipboard is maintained when one file is saved and another loaded, so you can use this technique to transfer lines of code from one program to another.

If you explicitly want to empty the Clipboard of text, the Edit menu's `Clear` option does this. This is different from the way `Clear` works on some other 'Cut and Paste' systems.

Search and Replace

In common with many editors, there is a facility for finding a text string and, if necessary, substituting another. All the options in connection with this are available through the Search menu.

To find a specified string, choose `Search` from the menu. BASIC 2 Plus displays a dialogue box prompting you to insert the string to search for:



Type the search string into the box, select the direction you wish to search in, and click on the Search box. BASIC 2 Plus will search from the current position of the cursor until it finds the string.

Pressing the ↵ key has the same effect as clicking on the SEARCH button. Because of this there is no way to search for a string containing a carriage return.

If you choose the `Whole Words` option instead of `Everywhere` in the dialogue box, only whole words will be found. This means that the specified string will only be found if it appears in the program delimited by characters other than letters (including Greek and accented letters), numbers, and the underline character. Note that this does not restrict the range of characters which may be included in a search string; it simply defines the characters which delimit a word.

For example, if a line of the program read

```
LET frederick$.details.title$ = "The Great"
```

If the `Whole Words` option was selected, `Search` would not find this line if the search string was `fred`, but it would find it if the search string was `frederick` or `frederick$`. This is because in the first case `frederick` is delimited by a space character before and a `$` behind, and in the second case `frederick$` is delimited by a space in front and a full stop behind. Since the space character, the dollar, and the full stop are all characters which can be used to delimit whole words, both `frederick` and `frederick$` can be found by `Search` with the `Whole Words` option.

If the editor finds an occurrence of the string but not the specific occurrence you need, you can repeat the search (starting from the end of the string found by the first search) by selecting `Search again` from the Edit menu.

Note that, since the repeated search starts from the end of the string found by the first search, the two strings cannot overlap. For example, searching for `issi` will stop at the word `Mississippi`. If you then choose `Search again`, the editor will not find the second occurrence of `issi` in `Mississippi`, because the search starts after the second letter `i` in the word.

Replacing

Replacing is similar to search except that when it finds the string the editor will substitute another. When you choose `Replace...`, you will be prompted for the two strings.

`Replace...` includes the option of `Manual` or `Automatic` replacement. `Automatic` replaces all the occurrences of the first string with the second without further ado. `Manual` stops at each occurrence and asks you whether you wish to replace or not.

`Replace...`, like `Search again`, continues its search from the end of the previous string.

Merge

The Merge option on the Edit menu is used to insert text from one file into another. When you select Merge, a dialogue box will prompt you for a file to insert. This must be a text file; it can have been prepared using the BASIC 2 Plus editor itself or another text editor.

Outlining

Large programs in BASIC 2 Plus may well include a number of *routines* – subprograms and functions. (How these fit into the structure of a program is explained in Chapter 6 of the Language Reference) When there are a great many routines in a program, it can be difficult to keep track of them all, so to help with this, the editor provides the Outline view of a program or module.

If, when you are editing a program or module, you select the `Outline` option in the View menu, the first line of every routine in that program or module is displayed in the Edit window. The order these lines are displayed in corresponds to the order in which the routines are declared in the file.

Outline view provides a convenient way of skipping around a file when editing it; if you need to change a line in one of the routines, there is no need to hunt up and down the file, trying to find where the routine is declared. Simply select the Outline view and double-click on the routine you wish to edit. You will be returned to editing mode, with the routine of your choice displayed in the window.

You cannot edit directly with the Outline view selected, but you can copy text to the Clipboard. This is particularly useful when routines are exported to and from modules; to make sure the routine's name is declared correctly in the IMPORT block, select the Outline view and Copy the name to the clipboard. From there, it can be pasted directly to the importing module, and you can be sure of not introducing typing errors.

When the Outline view is displayed, the `List...` option in the File menu changes to `List Outline...`. Selecting it will print a copy the Outline view of the program.

There is an accelerator for selecting the Outline view; instead of selecting Outline in the View menu, press F7 to achieve the same effect. F6 returns you to normal editing.

Other Features

Options

Selecting `Options` on the Edit menu gives you access to three facilities offered by the editor: the means to turn off Insert mode, and the Auto-indent facility; it also allows you to set the tab interval. These facilites are described in the next three sections.

Insert Mode

Normally, when you type text into the `Edit` window, the characters you type are inserted at the cursor, and the rest of the line shuffles along to make room for the insertion. This is generally what you will want. Sometimes, though, it is more convenient for the characters typed in to overwrite the characters already on the screen. BASIC 2 Plus allows you to do this by turning off Insert mode. This is done by choosing `Options` from the Edit menu, and then selecting `Insert Off`.

Or, to save using the menus, you can toggle Insert mode (that is, turn it on if it's off, and off if it's on) by pressing the Ins key.

Auto-Indent

Some of the new block structures in BASIC 2 Plus (compound `IF` statements, `FOR` loops, and so on) are much easier to read if the body of the loop is indented by a couple of spaces. To make it more convenient to type leading spaces at the start of every line, BASIC 2 Plus provides the Auto-indent option.

When Auto-Indent is turned on, every time a carriage return is inserted into the program it is automatically followed by enough blank spaces to ensure that the first character typed on the new line is directly under the first character of the previous line. If the previous line had no leading spaces, no spaces will be generated at the start of the new line. Any or all of the spaces can be deleted if they are not required.

Tab Interval

When you press the ⇥ key, the editor inserts enough spaces to take the cursor to the next tab stop. By default, tab stops are every two character widths across the screen, but you can adjust this to whatever value you want by changing the value via `Options` on the Edit menu.

The editor always inserts spaces, not tab characters; this means that you can remove excess space a character at a time if you wish.

Renumber

`Renumber...` is a facility available under the Edit menu. It is not normally needed with BASIC 2 Plus, since BASIC 2 Plus uses line numbers very little. However, should you need (or prefer) to write a program with line numbers, `Renumber...` provides the easiest way to do this.

When you select `Renumber...`, a dialogue box appears with three options: `Remove`, `As Is`, and `Add`. You can specify the line number of the first numbered line and the step size, that is, the difference between each line number and the next.

Choosing `Remove`, `As Is`, or `Add` will rationalise the line numbers in your program `Add` will number all the lines in the program; `As Is` will number only those lines which already have numbers, and `Remove` will number only those lines which are referenced by `GOTO`, `GOSUB` or other BASIC 2 Plus commands. In all three cases the numbering will start at the number specified by `First line`, and will continue in steps specified by `Line No Step`.

Rationalising line numbers means more than just renumbering the lines; it also means that references to lines throughout the program are kept consistent. For example, if the program contains the line

```
GOTO 255
```

it will be changed to make sure control still passes to the correct place. If line 255 has been renumbered to 120, then the above line will be altered to

```
GOTO 120
```

File Management

This chapter describes the way BASIC 2 Plus manages files – how to load and save files to disc, and how to list them to the printer. This is done with commands from the File menu. The chapter also explains the other commands on that menu, such as `Load and Run...` and `All new`.

There is then a description of the 'Workspace view' of your program, an overall view of the structure which gives access to the main program and its modules. It is the means of changing the clipboard directly, and it allows you to edit the *Debug command screen*, an essential tool for debugging whose use will be explained in the next chapter.

The Current Workspace Component

Before an explanation of the File menu can be given, though, it is necessary to give a loose description of an important concept: the Current Workspace Component. (The precise description is given later in this chapter.)

There are twelve Workspace Components in all – 'slots' in your computer's memory into which a block of text can be loaded. At any time, one of these Workspace Components is the Current Workspace Component, and it is this that many of the File menu's options work on.

If you are new to BASIC 2 Plus, you can think of the Current Workspace Component as an area of your computer's memory where your program is stored. Later, when you have more experience and need to write programs with modules and use the Debug command screen, you should read the last section of this chapter to get an accurate grasp of the full picture.

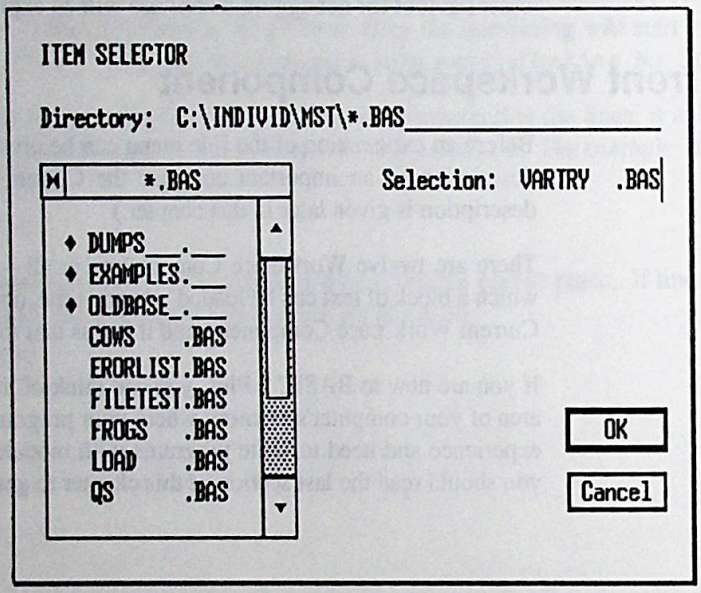
The File menu

The File menu appears as shown in the margin.

Some of the options may be dimmed, of course, to indicate that they are not available. `Save...` for instance, is not available until there is something in memory to save.

Save...

`Save...` is used to save the content of the Current Workspace Component to a disc file. If it has not been saved previously and was not loaded from a disc file, you will be prompted for the name of the file to save it in with this dialogue:

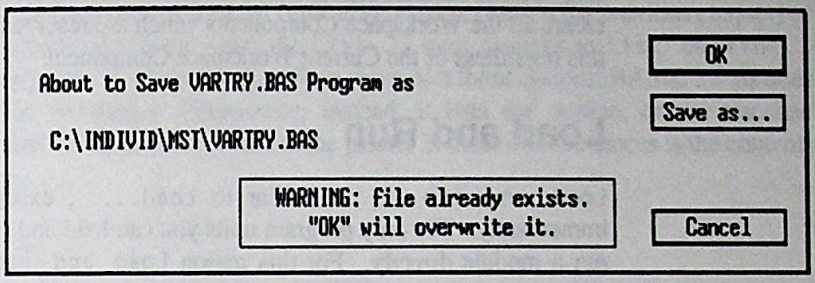


If you have saved previously, or if you did load the Current Workspace Component from a file, the edited version will be saved under the same name. In either case you are warned that BASIC 2 Plus is about to save the file. This gives you the chance to cancel the operation at the last minute, or to save under a different name.

Save as...

To save the content of the Current Workspace Component under a new name, select `Save as...`. This will always prompt you for the name of the file to save in.

If the file name you choose already exists, BASIC 2 Plus displays a warning:



You are then given the opportunity to save under a different name, or abort the save altogether.

Save all

Whereas `Save...` saves the contents of the Current Workspace Component, `Save all` saves the contents of all the Workspace Components which are reserved for programs or modules. It does this regardless of the Current Workspace Component. (It does not save the Debug command screen or the Clipboard.)

Load Program...

The option `Load Program...` changes, depending on the Current Workspace Component. If the Current Workspace Component is a module, for instance, this option will change to `Load Module...`. Or it can appear as `Load Debug...`, or `Load Clipboard...`.

However the option appears, when you select it you will be presented with a dialogue from which you should choose a file. This file will then be loaded into the Current Workspace Component. If loading would overwrite unsaved text, you are warned and given the choice of saving, of discarding, or of cancelling the whole operation.

New

New clears the Current Workspace Component so that it is available for you to type in fresh text. As with Load Program... you are warned if New is about to annihilate anything which has not been saved to disc.

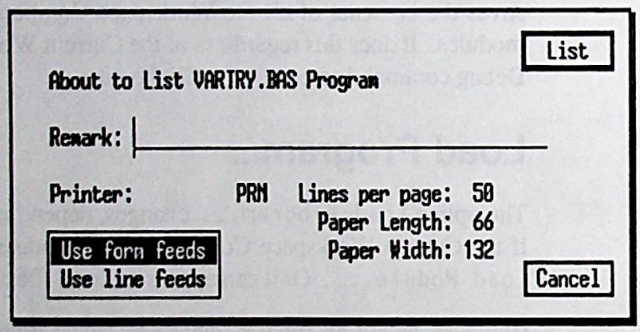
All new is similar, except that instead of clearing the Current Workspace Component, it clears all the Workspace Components which are reserved for programs or modules. It does this regardless of the Current Workspace Component.

Load and Run

Load and run... is similar to Load... , except that the file you load is run immediately. The only program units you can load and run are main programs: you cannot run a module directly. For this reason Load and run... always loads into the main program, irrespective of the Current Workspace Component.

List...

List... normally sends the contents of the Current Workspace Component to the printer. When you select it, the following dialogue is displayed:



You can adjust the number of lines per page, the paper length, and the paper width from this dialogue. You can also specify a remark to be printed at the head of the listing.

The options Use Form Feeds and Use Line Feeds control what BASIC 2 Plus sends to the printer when a new page is needed. If the former option is selected, a form feed character is sent; if the latter option is selected, BASIC 2 Plus sends enough line feed characters to take the printer to the top of the next page.

However, `List...` is slightly different from other options in the File menu; it changes according to the *view* you have selected.

Different views are selected from the View menu. If you select the 'standard' view (the second one on the View menu) or the Workspace view, `List...` works exactly as described above.

But if you have either selected Outline view (described in the previous chapter) or Traceback view (described in the next) the `List...` item changes to `List Outline...` or `List Traceback...` respectively. When you select these options, BASIC 2 Plus does not list the Current Workspace Component; instead, it lists the outline, or the traceback, of your program. In fact, what is listed to the printer in these circumstances is the contents of the Edit screen.

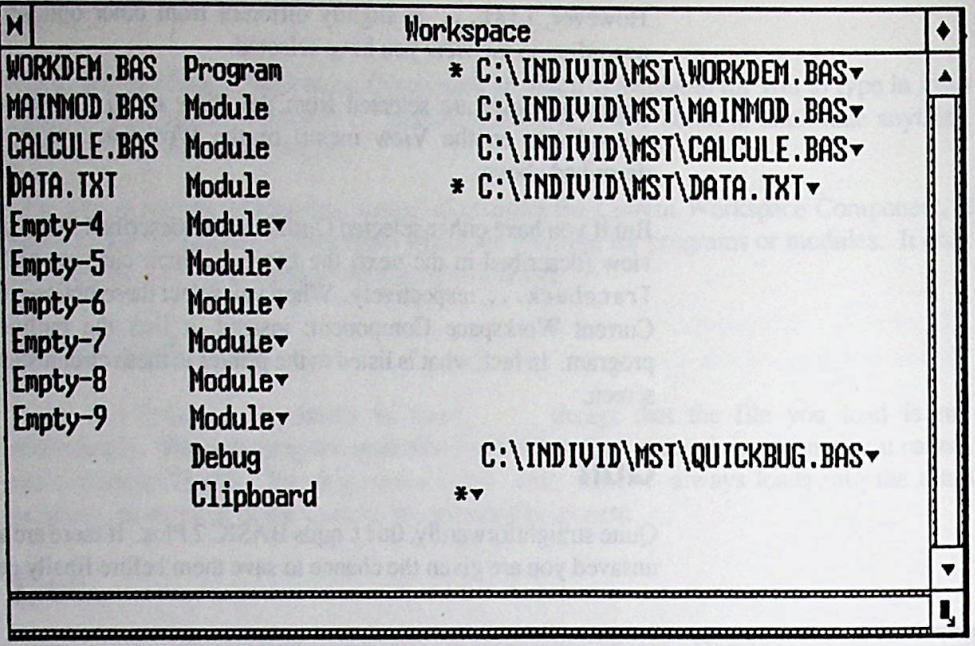
Quit

Quite straightforwardly, `Quit` quits BASIC 2 Plus. If there are any programs or modules still unsaved you are given the chance to save them before finally quitting.

Workspace view

As was briefly mentioned above, BASIC 2 Plus programs may comprise several 'program units': one main program and a number of modules (up to nine of which can be active at once). Each program unit is stored in a separate file. The workspace view of your program shows the names of all the program units together with the files they are stored in. It also gives access to the 'Debug command screen' and direct access to the Clipboard. There are thus twelve possible 'Workspace Components' which hold the main program, up to nine modules, the Debug command screen and the Clipboard. (The Debug command screen contains a list of extra commands which you can have executed at any point in your program. Its use will be explained in the next chapter.)

Selecting `Workspace` from the View menu will display the Workspace view of your program in the `Edit` window. (If the `Edit` window is closed you won't be able to see this view without selecting `Show Edit` from the Windows menu.)



The Workspace view consists of twelve lines, one for each possible workspace component, and each line has three parts.

The first is the name of the program or module currently occupying the component (in the case of the Debug command screen and the Clipboard this first part of the line is left blank).

The second part of the line shows what kind of component it is. It can be any of:

- ***Program***
- ***Module*** ***Binary Module***
- ***Debug***
- ***Clipboard***

(Binary modules are modules written not in BASIC 2 Plus, but directly in machine code. Their use is explained in the Language Reference.)

The third part of the line gives the name of the file that the component's contents were last saved to or loaded from. If any changes have been made (ie. the version held in the computer's memory is different from the version held on disc) the filename is preceded with an asterisk.

Of the twelve possible workspace components, the Current Workspace Component is the one marked by the cursor. Changing the Current Workspace Component is simply a matter of clicking the mouse to move the cursor to a new line. It is irrelevant where on the line the cursor appears.

Double-clicking on a line of the workspace view will make that component the current component, and immediately display its contents on the Edit screen. Or you can always display the contents of the Current Workspace Component by selecting the second option on the View menu. This option changes to reflect what the Current Workspace Component is; as you move the cursor in the workspace view this option will change to Program, Module, Debug, or Clipboard as appropriate.

You cannot edit a component directly in workspace view, though you can use `Copy` to transfer information to the Clipboard. And for obvious reasons, you cannot use `Cut`, `Copy`, or `Paste` when you are editing the Clipboard directly.

Debugging

This chapter introduces BASIC 2 Plus's debugging tools, facilities which enable you to test your programs thoroughly and efficiently, as well as to track down and correct the bugs.

Starting and Stopping the Program

The most necessary of all the debugging tools supplied with BASIC 2 Plus is the means to stop a program at any time. There are a number of ways to do this, depending on exactly what you want.

The simplest way to halt a progam temporarily is just to move the pointer into the menu bar. This is a feature of GEM rather than BASIC 2 Plus, and it applies to any program which runs under GEM. When the program is to continue, just move the pointer back into the main part of the screen.

Since this is a feature of GEM, you can't issue any BASIC 2 Plus commands when the program is stopped in this way. Instead, BASIC 2 Plus allows a User Break. When the program is running, you can either select `Stop` from the Program menu, or press Ctrl-C. The program will stop running and an Alert will appear on the screen with the word `Break.` Click on `OK` to clear this. The cursor in the Edit window will be positioned at the command which was being executed at the time of the break, so that, for example, you can see which particular part of the program was taking so long.

When you want to continue, you can select either `Continue` from the Program menu, (which will let the program carry on from the point where it was interrupted) or `Re-run`, which will re-run the program from the beginning.

CONT

Use `CONT` resumes program execution (if possible).

Syntax `CONT`

Note `CONT` may only be run in Direct Mode.

Direct Command Debugging

One of the most important features of the User Break is that you can issue commands while the program is temporarily halted. Provided you haven't altered the code at all, you can type commands into the Dialogue window and they will be executed immediately. This is enormously useful for finding out the values of variables, checking on the amount of free memory, etc. The only restriction is that you cannot execute statements which transfer control: `GOTO`, for example, or `SUB` and `FUNCTION` calls.

While it is quite permissible to use the `PRINT` command to interrogate variables, `PRINT` will send its output to the default stream. (This is the Results-1 window unless you change it.) If an essential part of your program concerns the layout of output, it may be inconvenient to perform your debug testing with `PRINT` statements.

Instead, there is the ? command. This is an easy way of finding out the value of a variable without disturbing the Results windows. (It's also quicker to type than `PRINT`.) The syntax is:

? *variable-name*

and this will display the value of the variable in the Dialogue window. You can also use the ? command in your program, if you wish; it is one way of putting trace statements into your code. However, there is a much better way which uses the Debug command screen, as we shall see in a few sections.

A very useful feature of BASIC 2 Plus is that variables retain their value even after the program has finished running. You can interrogate them with the ? command until you either amend the program, or execute one of the commands `LOAD`, `NEW`, or `CLEAR`. `LOAD` and `NEW` will be explained in the next chapter. `CLEAR` is available specifically to clear the values of all variables. When you execute it you will see the Information line change back to `Direct Command; Clear`.

Debug Points

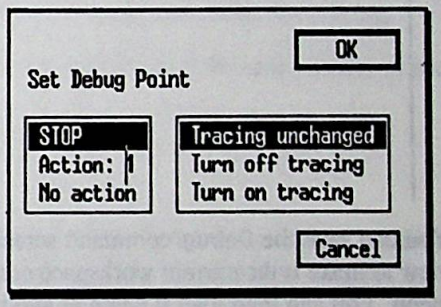
Debug points are special markers that you can put into your program to make BASIC 2 Plus carry out certain actions. They perform the job which in other BASICs is done by trace statements – special statements put into the program which are only used to track down errors. Debug points are better than trace statements for four reasons:

- ***Debug statements can be inserted and removed without restarting a program.***
- ***All a program's debug points can be activated, deactivated, or removed with one command.***
- ***The commands executed by debug statements do not need to be changed individually.***
- ***Debug statements can turn tracing on and off. (The section on Tracing explains this fully.)***

There are two stages to using debug points; defining and inserting them, and running the program with Debug enabled.

Inserting and deleting Debug Points

To insert a debug point, put the cursor where you want to insert it, and select `Set Debug Point...` from the Debug menu. A dialogue will appear giving you two choices of three options each.



One choice concerns tracing; it will be explained in the next section.

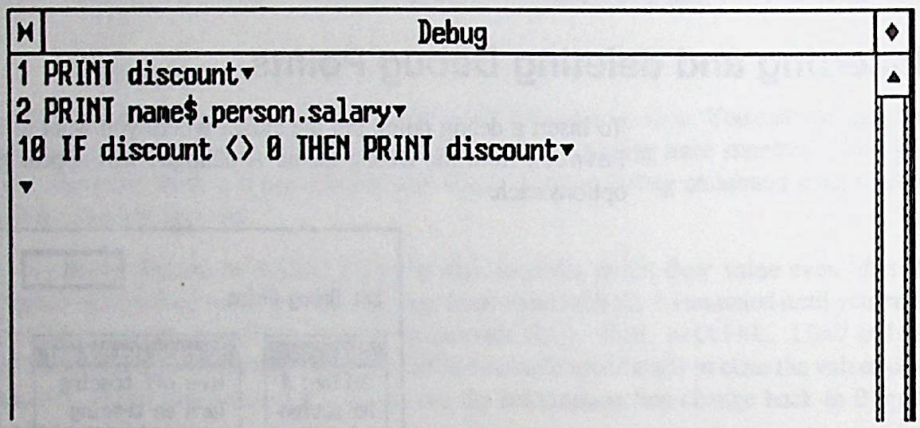
The other choice concerns what you want to happen when the program reaches the debug point. You can choose between:

- ***STOP***
- ***Action: 1***
- ***No action***

The first and last are self-explanatory. The second, if selected, means that at the debug point the program will carry out Debug Action number 1. There is a cursor immediately after the figure 1 enabling you to change it to another number. You can specify any Action number from 1 to 62.

The action itself is specified in the Debug command screen, which is accessed via the Workspace view of the program as follows:

Select `Workspace` from the View menu, and double click on the Debug line. The Edit window will change to show a window (initially blank) headed with the title `Debug`. Into this screen you can type BASIC 2 Plus commands, each beginning with a line number from 1 to 62. You cannot use commands which change the flow of control of the program – `GOTO`s or routine calls, for example –but you can have any other command which will fit onto one line: `IF` statements and even `SELECT` statements are permitted provided you do not exceed the limit of 255 characters.



You can save the `Debug` command screen to a file if you wish. Select it in the Workspace view to make it the current workspace component and then choose `Save...` from the File menu. You can then load it again at another time by selecting the `Debug` command screen, and choosing `Load Debug...` from the File menu.

To delete a debug point, move the cursor to it and select `Clear Debug` Point from the Debug menu. `Clear all Debug`, also on the Debug menu, removes all the debug points from the current workspace component.

Note that you can only have one debug point on each statement of the program, and you cannot have debug points at all on blank lines.

Using Debug Points

In the Program menu there are seven options which control the debugging of the program:

- ***Full speed (or Debug disabled)***
- ***Debug enabled***
- ***Show Trace – TROFF***
- ***Show Trace – TRON***
- ***Single Step***
- ***‹‹TROFF›› enabled***
- ***‹‹TROFF›› disabled***

The first option is `Full speed` only if there are no debug points in the current workspace component; it changes to `Debug disabled` as soon as you add any. If either of these options is selected, all debug points are totally ignored. (Of course, if `Full speed` is selected, there are no debug points to ignore.) The difference between `Full speed` and `Debug disabled` is that when there are debug points in the program, BASIC 2 Plus has to invest a small amount of time looking for them in order to ignore them. `Debug disabled` thus runs slightly slower than `Full speed`.

Debug enabled means that the program obeys `STOP` and `Action` commands in debug points, but ignores any commands concerning the tracing.

The remaining options are described in the next section, where tracing is explained fully.

Tracing

Often when debugging it is useful to be able to follow the flow of control around a program, watching to see which branch of an `IF` statement is taken, for example, or following a chain of subprograms or modules.

Program tracing provides the way to do this. When tracing is switched on, the program runs very slowly so that the statement being processed can be displayed in the Edit window. The cursor is always shown on this line, so by following the position of the cursor you can watch the flow of control directly. If something happens which you want to investigate, you can quickly move the cursor to the menu bar, select `Stop`, and then enter some direct debug commands.

Turn trace on and off

There are two ways to turn tracing on and off: directly, from the Debug menu, or by means of Debug points in the program.

The menu options which control tracing are `Show Trace - TROFF` and `Show Trace - TRON`. `Show Trace - TRON` turns tracing on, and `Show Trace - TROFF` turns it off again. The difference between `Show Trace - TROFF` and `Debug enabled` is that the former allows the tracing to be changed using debug points, whereas the latter always ignores all tracing commands.

The tracing options in debug points should now be self-explanatory. `Turn on tracing` turns tracing on (provided that one of the `Show Trace` options is selected in the Program menu) and `Turn off tracing` turns it off again. If the tracing is changed by a debug point inside a subprogram or module, it reverts to its original value on exit.

However, debug points which turn tracing off can be overridden. The last two items on the Program menu are `<<TROFF>> enabled` and `<<TROFF>> disabled`. When the second of these is selected, all `<<TROFF>>` debug points are ignored.

Single Step

Single step mode enables you to execute your program one statement at a time.

If you start a program by selecting the second item on the Program menu – `Start Step` – the option `Single Step` will automatically be selected in the lower part of that menu before the pre-scan of the program. If the pre-scan finds no errors in the program, `Start Step` will change to `Step`, and you will be able to 'step' through the program, one statement at a time, by repeatedly selecting `Step` (or more likely using its accelerator key F10).

In fact, you can select this option at any time the program is stopped. Hit F10, and BASIC 2 Plus will go into single step mode. `Single Step` will be selected in the lower part of the program menu, and one statement will be executed. When you no longer wish to single step, select one of the other options (`Full Speed`, `Debug Disabled`, `Debug Enabled`, `Show Trace - TROFF` or `Show Trace TRON`) and then select `Continue`, and the program will continue in your chosen mode.

If you select `Continue` without choosing one of the above options, `Debug Enabled` will automatically be selected for you.

Debug points still operate when the program is in single step mode. Specifically, any `<<TROFF>>` and `<<TRON>>` points will take the program out of single step mode and back in again. This means that, for example, you can put a `<<TROFF>>` point at the beginning of each of your routines, so that when single stepping through a program, the trace will skip through the routines at full speed, returning to single step again when control returns to the main program.

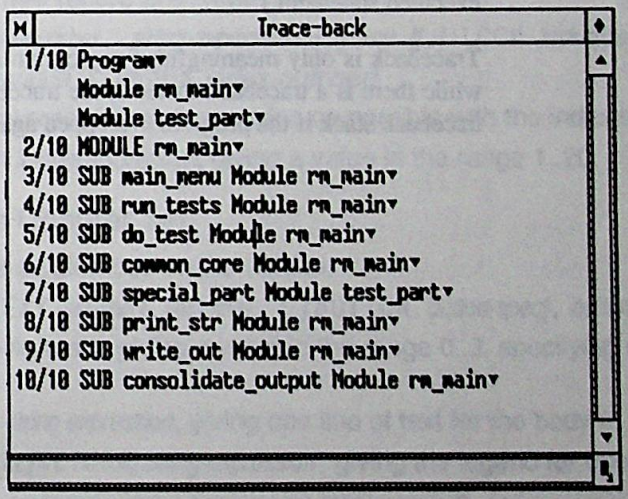
If you wish to disable this feature, (so that the program continues to single step even if it hits a `<<TROFF>>`) select `<<TROFF>> disabled` at the foot of the Program menu.

Traceback

Traceback (not to be confused with trace) is another way of viewing the structure of a program. However, it is different from the Workspace view or the Outline view in that it is only meaningful while the program is temporarily stopped.

When you stop a program and select `Traceback` from the View menu, BASIC 2 Plus displays the names of all the active routines, modules, and subroutines (including calls to error handling routines and DEF functions) in the order of their calling. Routines which are active on more than one level (because of recursion) are displayed an appropriate number of times.

The names are preceded by two numbers separated by a slash. The second of these is the number of levels currently active; the first indicates the position of each routine in the hierarchy.



In the example on the previous page, for instance, the main program has loaded two modules, `rm_main` and `test_part`. Of these, `rm_main` has been called from the main program, and is still active. We can see this, since `MODULE rm_main` is marked as the second active call out of ten.

This module has called `main_menu`, a subprogram which is to be found in `MODULE rm_main`. `rm_main` has called `run_tests`, and so the traceback continues, listing all ten active calls.

Variables in context

When you're viewing a traceback, you can determine the context in which any ? commands are executed. To do this, put the cursor onto the name of the routine or module you want to investigate. This makes any ? commands yield the value of variables in the context of that routine or module – that is, the value they hold in that routine or module. This context is always displayed in the information line, just below the title bar of the Dialogue window.

So by using traceback you can examine the value of a variable in an active routine even if it has the same name as a variable in a nested routine. Furthermore, in a recursive routine you can examine the value of a variable at each level of the recursion process.

If you select a routine in this way, and then switch from the traceback view to the standard editing view, the edit window will show the 'traceback context', that is, it will show the selected routine with the cursor positioned immediately after the statement which called the next routine in the chain. (Obviously this does not apply to the last routine in the chain, since it doesn't contain an active call. Instead, the cursor is positioned immediately after the last executed statement.)

Traceback is only meaningful if a program is temporarily halted. If you resume execution while there is a traceback in view, the traceback window will clear; it will display the new traceback stack if the program is stopped again before it terminates.

Syntax Summary

ABS (see Chapter 3)

Use ABS is a function which returns the absolute value of its argument.

Syntax ABS(*argument*)

where *argument* is a *numeric-expression*.

ACOS (see Chapter 3)

Use ACOS is a function which returns the angle whose cosine takes the given value.

Syntax ACOS(*argument*)

where *argument* is a *numeric-expression*, giving a value in the range –1..+1.

ADDKEY (see Chapter 13)

Use ADDKEY adds a new key for the current record.

Syntax ADDKEY *[#]stream-number[,] new-key [[,]* LOCK *lock-type]*

where *new-key* is: KEY *key-value [[,]* INDEX *index-number]*

where *key-value* is an *expression*, giving a value compatible with the index's key type

index-number is an *integer-expression*, giving a value in the range 1..20.

ADDREC (see Chapter 13)

Use ADDREC adds a new record to a Keyed File.

Syntax ADDREC *[#]stream-number, string-expression [,] key [[,]* LOCK *lock-type]*

where *key* is: **KEY** *key-value [[,]* **INDEX** *index-number]*

where *key-value* is an *expression*, giving a value compatible with the index's key type

index-number is an *integer-expression*, giving a value in the range 1..20.

ALERT (see Chapter 14)

Use ALERT produces an alert box on the screen.

Syntax ALERT *icon [,]* TEXT *text-line [, text-line]... [,]* BUTTON *button-spec[, button-spec]...*

where: *icon* is an *integer-expression* giving a value in the range 0..3, specifying which icon is to be used.

Each*text-line* is a *string-expression*, giving one line of text for the body of the alert box.

Each *button-spec* is *[*RETURN*]string-expression* , giving the legend for one button.

Note: *References given to the appropriate chapter of the Language Reference Guide.*

ASC (see Chapter 4)

Use ASC is a function which converts characters to their numeric character code representation.

Syntax ASC(*argument*)

ASIN (see Chapter 3)

Use ASIN is a function which returns the angle whose sine takes the given value.

Syntax ASIN(*argument*)

where *argument* is a *numeric-expression*, giving a value in the range –1..+1.

ATAN & ATN (see Chapter 3)

Use These functions return the angle whose tangent takes the given value.

Syntax ATAN(*argument*)
ATN(*argument*)

where *argument* is a *numeric-expression.*

ATAN2 (see Chapter 3)

Use ATAN2 is a function which returns the angle between the x axis and the line from the origin to the given point.

Syntax ATAN2(*x* , *y*)

where *x* and *y* are *numeric-expressions.*

BIN$ (see Chapter 4)

Use BIN$ is a function which converts an integer value to a string of binary digits.

Syntax BIN$(*argument* [, *min-digits*])

where *argument* and *min-digits* are *integer-expressions*; *min-digits* must be in the range 0..32.

BOX (see Chapter 12)

Use BOX draws a box on a graphics device.

Syntax BOX *[#stream-number,] point, width, height [[,] attribute]...*

BUTTON (see Chapter 14)

Use BUTTON is a function which returns the state of the mouse button.

Syntax BUTTON *[(button-number)]*

where *button-number* is an *integer-expression*, giving a value in the range 1..15.

CALL (see Chapter 6)

Use CALL is used to invoke a subprogram.

Syntax *[*CALL*] sub-identifier [(] actual-parameter [, actual-parameter]... [)]*

where *sub-identifier* is a *numeric-identifier* corresponding to the *sub-identifier* of a SUB.

actual-parameter is *expression*
or *general-variable*
or *actual-array*

actual-array is *array-name* [()],
or *string-general-variable . record-name . vector-field-name* [()]

CALL MODULE (see Chapter 7)

Use CALL MODULE invokes the body of a module.

Syntax CALL MODULE *module-identifier* [(*actual-parameter* [, *actual-parameter*]...)]

where *actual-list* is *actual-parameter* [, *actual-parameter*]...

actual-parameter is *expression*
or *general-variable*
or *actual-array*

actual-array is *array-name* [()],
or *string-general-variable . record-name . vector-field-name* [()]

CEILING & CEIL (see Chapter 3)

Use These functions return the value of their argument rounded to an integer. Rounding is upwards, that is, towards plus infinity.

Syntax CEILING(*argument*)
CEIL(*argument*)

where *argument* is a *numeric-expression*.

CHDIR & CD (see Chapter 15)

Use These statements change the current directory.

Syntax CHDIR *file-name*
CD *rest-of-line*

where *rest-of-line* is everything up to the end of the line

CHDIR$ (see Chapter 15)

Use CHDIR$ is a function which returns the current directory.

Syntax CHDIR$*[(drive)]*

where *drive* is a *string-expression*, giving a single character string.

CHR$ (see Chapter 4)

Use CHR$ is a function which converts a character code to a single character string.

Syntax CHR$(*argument*)

where *argument* is an *integer-expression* in the range 0..255.

CINT (see Chapter 3)

Use CINT is a function which returns the value of its argument rounded to an integer. Rounding is towards the nearest integer, with halves being rounded away from zero. The result must be in the range -2147483648..+2147483647.

Syntax CINT(*argument*)

where *argument* is a *numeric-expression*.

CIRCLE (see Chapter 12)

Use CIRCLE draws part or all of a circle on a graphics device.

Syntax CIRCLE *[#]stream-number,] point, radius [[,] attribute]...*

where *attribute* is: PART *start-angle, end-angle*
or: FILL *[ONLY] [WITH fill-style]*
or: WIDTH *line-width*
or: STYLE *line-style*
or: START *line-start-style*
or: END *line-end-style*
or: COLOUR *colour-number*
or: COLOR *colour-number*
or: MODE *write-mode*

radius is an *integer-expression*, giving a positive, non-zero value.

start-angle and *end-angle* are *numeric-expressions*, giving positive, or zero, values.

CLOSE (see Chapter 8)

Use CLOSE closes streams.

Syntax CLOSE

or CLOSE *[#]stream-number [, [#]stream-number]...*

CLOSE WINDOW (see Chapter 14)

Use CLOSE WINDOW closes the specified window.

Syntax CLOSE WINDOW *window-number*

where *window-number* is an *integer-expression*, giving a value in the range 1..4.

CLEAR (see User Guide Chapter 1)

Use CLEAR discards all variables.

Syntax CLEAR

CLEAR RESET (see Chapter 9)

Use CLEAR RESET sets all windows to their initial state, and, in Direct Mode, discards all variables.

Syntax CLEAR RESET

CLS (see Chapters 11 & 12)

Use CLS clears a virtual screen.

Syntax CLS *[RESET]*

or CLS *[#]stream-number [[,] RESET]*

CONSOLIDATE (see Chapter 13)

Use As soon as a keyed file is changed it is marked "inconsistent". CONSOLIDATE causes all outstanding information to be written to the file, and clears the "inconsistent marker.

Syntax CONSOLIDATE *[#]stream-number*

CONST (see Chapter 2)

Use CONST declares and sets named constants.

Syntax CONST *simple-variable = expression [, simple-variable = expression]...*

CONT (see User Guide Chapter 4)

Use CONT resumes program execution (if possible).

Syntax CONT

COS (see Chapter 3)

Use COS is a function which returns the cosine of its argument.

Syntax COS(*argument*)

where *argument* is a *numeric-expression*.

CURRENCY$ (see Chapter 11)

Use CURRENCY$ is a function which returns the current "currency string".

Syntax CURRENCY$

DATA (see Chapter 2)

Use DATA is used to include numeric or string data in the code of the program itself, rather than inputting it at run time.

Syntax DATA *[data-literal] [, [data-literal]] ...*

where *data-literal* is *[sign] numeric-literal*
or *string-literal*
or *first-char [[character]... last-char]*

sign is + or -

first-char is any printable character excluding space, comma & double quotes

character is any printable character excluding comma

last-char is any printable character excluding space & comma

DATE (see Chapter 3)

Use DATE is a function which converts a date in the form of a string to a date as a number of days since 31st December 1899.

Syntax DATE*[(argument)]*

where *argument* is a *string-expression.*

DATE$ (see Chapter 4)

Use DATE$ is a function which converts a date as a number of days since 31st December 1899 to a date in the form of a string

Syntax DATE$*[(argument)]*

where *argument* is an *integer-expression*, giving a positive, non-zero, value.

DECLARE (see Chapter 2)

Use DECLARE declares simple variables and arrays.

Syntax DECLARE *declared-object [, declared–object]...*

where *declared-object* is *simple-variable*
or *formal-array*

DECLARE CONST (see Chapter 7)

Use DECLARE CONST declares named constants.

Syntax DECLARE CONST *simple-variable [, simple-variable]...*

DEC$ (see Chapter 4)

Use DEC$ is a function which converts a numeric value to a formatted decimal string.

Syntax DEC$(*argument* , *template*)

where *argument* is a *numeric-expression.* and *template* is a *string-expression.*

DEF (see Chapter 6)

Use DEF is used to define an expression function.

Syntax DEF *function-identifier* [(*parameter* [, *parameter*]...)] = *result-expression*

where *function-identifier* is an *identifier*, and *result-expression* is an *expression* of the same type. Each *parameter* must be a *simple-variable.*

DEG (see Chapter 3)

Use DEG is a function to converts angles in radians to degrees.

Syntax DEG(*argument*)

where *argument* is a *numeric-expression.*

DEL (see Chapter 15)

Use DEL deletes files. (See also ERASE and KILL)

Syntax DEL *rest-of-line*

where *file-filter* is a *string-expression*

rest-of-line is everything up to the end of the line

DELKEY (see Chapter 13)

Use DELKEY deletes a key for the current record.

Syntax DELKEY *[#]stream-number[,] key [[,]* LOCK *lock-type]*

where *key* is: KEY *key-value [[,]* INDEX *index-number]*

or: AT *position-string*

where *key-value* is an *expression*, giving a value compatible with the index's key type

index-number is an *integer-expression*, giving a value in the range 1..20.

position-string is a *string-expression*, giving a value once returned by POSITION$.

DIM (see Chapter 2)

Use DIM is used to specify the size and number of elements in one or more arrays.

Syntax DIM *array-declaration [, array-declaration]...*

where *array-declaration* is *array-name (dimension-bounds [, dimension-bounds]...) [storage-class]*

array-name is an *identifier* giving the name, and type, of the array

dimension-bounds is *[lower-bound* TO *]upper-bound*

where *lower-bound* and *upper-bound* are both *integer-expressions* giving values in the range -32768..32767 and *lower-bound* ≤ *upper-bound*.

storage-class is *integer-class [*KEY*]*
or IEEE4 orIEEE8
or FIXED *length*

where *integer-class* is one of BYTE, UBYTE, WORD, UWORD, INTEGER

length is an *integer-expression* giving a value in the range 1..4096.

DIMENSIONS (see Chapter 2)

Use DIMENSIONS is a function used to determine the number of dimensions of ar. array.

Syntax DIMENSIONS(*actual-array*)

DIR (see Chapter 15)

Use DIR produces directory listings. (See also FILES.)

Syntax DIR *rest-of-line*

where *rest-of-line* is everything up to the end of the line

DISPLAY (see Chapter 15)

Use DISPLAY lists the contents of a file. (See also TYPE.)

Syntax DISPLAY *[#stream-number,] file-name*

DISTANCE (see Chapter 12)

Use DISTANCE is a function which returns the distance from the cursor to a given point, in user coordinates.

Syntax DISTANCE(*[#stream-number,] point*)

DO, LOOP (see Chapter 5)

Use The DO statement is used for repeating a sequence of statements when the repetition is governed by some condition which may be tested before or after the execution of the loop.

Syntax DO *[terminating-condition]* : *loopbody* : LOOP *[terminating-condition]*

where *terminating-conditions* are either

WHILE *truth-value*

or UNTIL *truth-value*

loopbody is a sequence of BASIC 2 Plus statements, separated in the usual way with colons or new lines.

DRIVE (see Chapter 15)

Use DRIVE sets the current drive.

Syntax DRIVE *drive*

where *drive* is a *string-expression*, giving a single character string.

EDIT (see User Guide Chapter 1)

Use EDIT takes you into the edit window.

Syntax EDIT

ELLIPSE (see Chapter 12)

Use ELLIPSE draws part or all of a circle on a graphics device.

Syntax ELLIPSE *[[#stream-number,] point, x-radius, aspect [[,] attribute]...*

where *attribute* is: PART *start-angle, end-angle*

or: FILL *[ONLY] [WITH fill-style]*

or: WIDTH *line-width*

or: STYLE *line-style*

or: START *line-start-style*

or: END *line-end-style*

or: COLOUR *colour-number*

or: COLOR *colour-number*

or: MODE *write-mode*

x-radius is an *integer-expression*, giving a positive, non-zero value.

aspect is a *numeric-expression*, giving a positive, non-zero value.

start-angle and *end-angle* are *numeric-expressions*, giving positive, or zero, values.

ELLIPTICAL PIE (see Chapter 12)

Use ELLIPTICAL PIE draws a elliptical "pie" section on a graphics device.

Syntax ELLIPTICAL PIE *[#]stream-number,] point, x-radius, start-angle, end-angle, aspect [[,] attribute]...*

where *attribute* is: FILL *[ONLY] [WITH fill-style]*
or: WIDTH *line-width*
or: STYLE *line-style*
or: COLOUR *colour-number*
or: COLOR *colour-number*
or: MODE *write-mode*

x-radius is an *integer-expression*, giving a positive, non-zero value.

aspect is a *numeric-expression*, giving a positive, non-zero value.

start-angle and *end-angle* are *numeric-expressions*, giving positive, or zero, values.

END (see Chapter 5)

Use END is used to terminate execution of the program when it has completed its allotted tasks, as opposed to STOP, which is used in debugging.

Syntax END

ENVIRON$ (see Chapter 15)

Use ENVIRON$ is a function which information from the "environment" in the form of a string

Syntax ENVIRON$(*argument*)

where *argument* is either an *integer-expression*, giving a positive, non-zero, value.
or a *string-expression.*

EOF (see Chapter 10)

Use EOF is a function, which returns TRUE if there is nothing more to be input from the given stream.

Syntax EOF(*[#]stream-number*)

EPS (see Chapter 3)

Use EPS is used to give a measure of the significance of a number.

Syntax EPS(*argument*)

where *argument* is a *numeric-expression.*

ERASE (see Chapter 15)

Use ERASE deletes files. (See also KILL and DEL.)

Syntax ERASE *rest-of-line*

where *rest-of-line* is everything up to the end of the line

ERR (see Chapter 16)

Use ERR is a function which returns the number of the latest error.

Syntax ERR

ERROR (see Chapter 16)

Use ERROR raises an error.

Syntax ERROR *integer-expression*

ERROR$ (see Chapter 16)

Use ERROR$ is a function which returns the message associated with an error number.

Syntax ERROR$(*integer-expression*)

EXIT DO, EXIT FOR (see Chapter 5)

Use EXIT DO and EXIT FOR are used to get out of a loop from the middle without waiting for the loop to complete. EXIT DO and EXIT FOR are used to leave DO and FOR loops respectively

Syntax EXIT DO

or EXIT FOR

EXIT MODULE (see Chapter 7)

Use EXIT MODULE is used to leave a module before the final END MODULE statement.

Syntax EXIT MODULE

EXIT SUB, EXIT FUNCTION (see Chapter 6)

Use EXIT SUB and EXIT FUNCTION are used to terminate a subprogram or routine function before the terminating END SUB or END FUNCTION is reached.

Syntax EXIT SUB

or EXIT FUNCTION

EXP (see Chapter 3)

Use EXP is the exponentiation function, the inverse of the natural logarithm.

Syntax EXP(*argument*)

where *argument* is a *numeric-expression*.

EXPORT (see Chapter 7)

Use EXPORT makes a program's global variables available to any modules it calls.

Syntax EXPORT *exported-object* [, *exported-object*] ...

where *exported-object* is *simple-variable* or *formal-array*

formal-array is *array-identifier* ([,] ...)

EXTENT (see Chapter 12)

Use The EXTENT function returns the length a string will be when printed.

Syntax EXTENT({#}*stream-number*,] *string-expression*)

or EXTENT({#}*stream-number*[,]] *print-function* [[,] *print-function*]... [,] *string-expression*)

FALSE (see Chapter 3)

Use FALSE is a function which always returns 0, the value which BASIC 2 Plus deems to represent the Boolean value 'false'. (See also OFF.)

Syntax FALSE

FD (see Chapter 12)

Use FD moves the cursor forward a given distance. (See also FORWARD.)

Syntax [MOVE] FD {#}*stream-number*,] *distance* [[,] *attribute*]...

where *attribute* is: WIDTH *line-width*

or: STYLE *line-style*

or: START *line-start-style*

or: END *line-end-style*

or: COLOUR *colour-number*

or: COLOR *colour-number*

or: MODE *write-mode*

distance is an *integer-expression*, giving a positive, non-zero value.

FILES (see Chapter 15)

Use FILES produces directory listings. (See also DIR.)

Syntax FILES *[#stream-number,] [file-filter]*

where *file-filter* is a *string-expression*

FIND$ & FINDDIR$ (see Chapter 15)

Use These functions are provided for finding files and directories.

Syntax FIND$(*file-filter [, ordinal]*)

FINDDIR$(*file-filter [, ordinal]*)

where *file-filter* is a *string-expression*

ordinal is an *integer-expression*, giving a value in the range 1..32767

FIX (see Chapter 3)

Use FIX is a function which returns the value ofits*argument* rounded to an integer. Rounding is towards zero. (See also TRUNC.)

Syntax FIX(*argument*)

where *argument* is a *numeric-expression*.

FLOOD (see Chapter 12)

Use FLOOD fills an area bounded by a given colour.

Syntax FLOOD *[#stream-number,] point, [, boundary-colour] [[,] attribute]...*

where *attribute* is: FILL WITH *fill-style*

or: COLOUR *colour-number*

or: COLOR *colour-number*

or: MODE *write-mode*

boundary-colour is a *colour-number*.

NB: FLOOD may have no effect, depending on the device driver.

FLOOR (see Chapter 3)

Use The FLOOR function returns the value of itsr argument rounded to an integer. Rounding is downwards, that is, towards minus infinity. (See also INT.)

Syntax FLOOR(*argument*)

where *argument* is a *numeric-expression*.

FONT$ (see Chapter 12)

Use FONT$ is a function which returns the name of a font.

Syntax FONT$(*[#stream-number,] font-ordinal*)

where *font-ordinal* is an *integer-expression*, giving a positive, non-zero value.

FOR (see Chapter 5)

Use The FOR statement is used to repeat a group of statements where the number of repetitions is known or can be calculated at the time the loop is entered.

Syntax FOR *loopcounter* = *start* TO *end* [STEP *stepsize*] : *loopbody* :NEXT...

where *loopcounter* is a *simple-variable*, not a VAR or ref formal parameter.

start, *end*, and *stepsize* are numeric expressions

loopbody is a sequence of BASIC 2 Plus statements, separated in the usual way with colons or new lines.

FORWARD (see Chapter 12)

Use FORWARD moves the cursor forward a given distance. (See also FD.)

Syntax [MOVE] FORWARD *[#stream-number,] distance [[,] attribute]...*
[MOVE] FD *[#stream-number,] distance [[,] attribute]...*

where *attribute* is: WIDTH *line-width*
or: STYLE *line-style*
or: START *line-start-style*
or: END *line-end-style*
or: COLOUR *colour-number*
or: COLOR *colour-number*
or: MODE *write-mode*

distance is an *integer-expression*, giving a positive, non-zero value.

FRAC (see Chapter 3)

Use FRAC is a function which returns the fraction part of the value of its argument.

Syntax FRAC(*argument*)

where *argument* is a *numeric-expression.*

FRE (see User Guide Chapter 1)

Use FRE is a function which returns the amount of unused memory.

Syntax FRE

FREEFILE (see Chapter 8)

Use FREEFILE is a function which returns on unused stream number.

Syntax FREEFILE

FUNCTION (see Chapter 6)

Use FUNCTION is used to define a routine function.

Syntax FUNCTION *function-identifier* [(*formal-list*)] [EXPORT] :
function-body :
END FUNCTION

where *function-identifier* is *identifier*,
formal-list is *formal-functionlist* [*separator formal-functionlist*] ...
formal-functionlist is VAL *simple-var-list*
or CONST *simple-var-list*
or [VAR] *formal-var-list*
simple-var-list is *identifier* [, *identifier*] ...
formal-var-list is *formal-var* [, *formal-var*] ...
formal-var is *identifier* or *formal-array*
formal-array is *array-identifier* ([,] ...)
separator is ; or ,
function-body is a sequence of BASIC 2 Plus statements.

GET (see Chapter 13)

Use GET gets data from Random or Keyed Files.

Syntax GET [#]*stream-number*, *string-general-variable* [[,] *position*] [[,] LOCK *lock-type*]

where *position* is as described in POSITION.

GOSUB (see Chapter 6)

Use GOSUB is used to call a subroutine elsewhere in the program.

Syntax GOSUB *location*

GOSUB can be spelt as two words thus: GO SUB.

GOTO (see Chapter 5)

Use GOTO is used to jump unconditionally to another part of the program.

Syntax GOTO *location*

GOTO may be spelt as two words thus: GO TO

GRAPHICS (see Chapter 12)

Use GRAPHICS sets graphics output attributes.

Syntax GRAPHICS *[#stream-number] [[,] attribute]...*

where *attribute* is: CURSOR *cursor-type*
or: FILL *[STYLE] [WITH] fill-style*
or: *[LINE]* WIDTH *line-width*
or: *[LINE]* STYLE *line-style*
or: *[LINE]* START *line-start-style*
or: *[LINE]* END *line-end-style*
or: MARKER *marker-number*
or: MARKER SIZE *marker-size*
or: COLOUR *colour-number*
or: COLOR *colour-number*
or: MODE *write-mode*

where *cursor-type* is an *integer-expression*, giving a value in the range 1..3.

GRAPHICS...RESTORE (see Chapter 12)

Use GRAPHICS...RESTORE turns on or off window contents restore.

Syntax GRAPHICS *[#stream-number[,]]* RESTORE *truth-value*

GRAPHICS...UPDATE (see Chapter 12)

Use GRAPHICS...UPDATE forces graphics output to be acted on.

Syntax GRAPHICS *[#stream-number[,]]* UPDATE *[NEW]*

HEADING (see Chapter 12)

Use HEADING is a function which returns the current heading.

Syntax HEADING*[(][#]stream-number)]*

HEX$ (see Chapter 4)

Use HEX$ is a function which converts an integer value to a string of hexadecimal digits.

Syntax HEX$(*argument [, min-digits]*)

where *argument* and *min-digits* are *integer-expressions*; *min-digits* must give a value in the range 0..32.

IF (single line version) (see Chapter 5)

Use This version of the IF statement is used to choose between two courses of action. It is retained only for compatibility with earlier versions of BASIC.

Syntax
```
IF testvalue [ colons ] THEN [ instructions ] [ [ colons ] ELSE elseinstructions ]
```
or
```
IF testvalue [ colons ] GOTO location [ [ colons ] ELSE elseinstructions ]
```

where *testvalue* is a *truth-value*

instructions and *elseinstructions* are either (a) a line number, or (b) a sequence of BASIC statements, each separated by *colons*

colons is a sequence of one or more colons, but not new lines.

IF...END IF (see Chapter 5)

Use IF...END IF is used to choose between various courses of action depending on the values of various expressions.

Syntax
```
IF testvalue1 [ : ] THEN [ : ] statements1 [ : ]
[ ELSEIF testvalue2 [ : ] THEN [ : ] statements2 [ : ] ]
[ ELSEIF testvalue3 [ : ] THEN [ : ] statements3 [ : ] ]
.....
[ ELSE [ : ] elsestatements [ : ] ]
END IF
```

where the *testvalues* are *truth-values*, and *statements1*, *statements2*,... and *elsestatements* are sequences of BASIC 2 Plus statements, separated with colons or new lines in the usual way.

IMPORT (see Chapter 7)

Use IMPORT is used to specify the routines, constants and variables which are imported into a program unit from a module.

Syntax
```
IMPORT [ MODULE module-identifier [ ( formal-list ) ] ] :
   [ import-declaration : ] ...
END IMPORT
```

where *module-identifier* is a *numeric-identifier*,

import-declaration is SUB *sub-identifier* [(*formal-list*)]
or FUNCTION *function-identifier* [(*formal-list*)]
or DECLARE CONST *simple-var-list*
or DECLARE *formal-var-list*

sub-identifier is *numeric-identifier*

function-identifier is *identifier*

formal-list	is	*formal-sublist [separator formal-sublist] ...*
formal-sublist	is	VAL *simple-var-list*
	or	CONST *simple-var-list*
	or	*[* VAR *] formal-var-list*
simple-var-list	is	*identifier [, identifier] ...*
formal-var-list	is	*ormal-var [, formal-var] ...*
formal-var	is	*identifier* or *formal-array*
formal-array	is	*array-identifier* (*[,] ...*)
separator	is	; or ,

INKEY (see Chapter 10)

Use INKEY is a function which returns a numeric value corresponding to a key press.

Syntax INKEY

INKEY$ (see Chapter 10)

Use INKEY$ is a function which returns a string value corresponding to a key press.

Syntax INKEY$

INPUT (see Chapter 10)

Use INPUT inputs numeric or string values from sequential input devices.

Syntax INPUT *[#stream-number[,]]* [AT(*column*; *line*) *] [;] [prompt] general–variable [, general–variable]... [;]*

INPUT$ (see Chapter 10)

Use INPUT$ is a function which returns a string containing a given number of characters read from a given stream.

Syntax INPUT$(*#stream-number, count*)

or INPUT$(*count, [#]stream-number*)

where *count* is an *integer-expression*, giving a value in the range 1..4096.

INSTR (see Chapter 4)

Use INSTR is a function which is used to find out if one string is a substring of another.

Syntax INSTR(*[start ,] search-string , target-string*)

where *start* is an *integer-expression* in the range 1..4096, and *search-string* and *target-string* are *string-expressions.*

INT (see Chapter 3)

Use The INT function returns the value of itsr argument rounded to an integer. Rounding is downwards, that is, towards minus infinity. (See also FLOOR.)

Syntax INT(*argument*)

where *argument* is a *numeric-expression.*

KEY (see Chapter 13)

Use KEY is a function which returns the current key value for a Keyed File.

Syntax KEY(*[#]stream-number*)

KEY$ (see Chapter 13)

Use KEY$ is a function which returns the current key value for a Keyed File.

Syntax KEY$(*[#]stream-number*)

KEYSPEC (see Chapter 13)

Use KEYSPEC creates a new index in a Keyed File, and declares the type of key for the index.

Syntax KEYSPEC *[#]stream-number [,]* INDEX *index-number [key-spec] [unique]*

where *index-number* is an *integer-expression*, giving a value in the range 1..20.

key-spec is: BYTE, UBYTE, WORD, UWORD or INTEGER
or: FIXED *string-length*

where *string-length* is an *integer-expression*, giving a value in the range 1..30.

unique is: *[,]* UNIQUE *truth-value*

KILL (see Chapter 15)

Use KILL deletes files. (See also DEL and ERASE.)

Syntax KILL *file-filter*

where *file-filter* is a *string-expression*

LABEL (see Chapter 1)

Use LABEL is used to define a named loaction.

Syntax LABEL *label*

or *label*:

LBOUND (see Chapter 2)

Use The LBOUND function may be used to determine the lower bound of one of the dimensions of an array. (See also LOWER.)

Syntax LOWER(*actual-array [, index-number]*)

where *index-number* must be an *integer-expression*.in the range 1 to 7.

LCASE$ (see Chapter 4)

Use The LCASE$ function converst characters to lower case. (See also LOWER$.)

Syntax LCASE$(*argument*)

where *argument* is a *string-expression*

LEFT (see Chapter 12)

Use LEFT changes the current heading. (See also LT, RIGHT and RT.)

Syntax LEFT *[#stream-number ,] angle-change*

where *angle-change* is a *numeric-expression.*

LEFT$ (see Chapter 4)

Use LEFT$ is a function to truncate a string by selecting only its leftmost characters.

Syntax LEFT$(*argument , length*)

where *argument* is a *string-expression* and *length* is an *integer-expression* returning a value in the range 0..4096.

LEN (see Chapter 4)

Use LEN is a function to find the length of a string. (How long is a piece of string?)

Syntax LEN(*argument*)

where *argument* is a *string-expression*

LET (see Chapter 2)

Use LET assigns the value of an expression to a general–variable.

Syntax *[* LET *] general–variable = expression*

LINE (see Chapter 12)

Use LINE draws a line between two points, possibly via a number of intermediate points.

Syntax LINE *[#stream-number,] point, [point,]... point [[,] attribute]...*

where *attribute* is: WIDTH *line-width*
or: STYLE *line-style*
or: START *line-start-style*
or: END *line-end-style*
or: COLOUR *colour-number*
or: COLOR *colour-number*
or: MODE *write-mode*

LINE INPUT (see Chapter 10)

Use LINE INPUT inputs a complete line from sequential input devices.

Syntax LINE INPUT *[#stream-number[,]] [*AT(*column; line*)*] [;] [prompt] string-general–variable [;]*

where the *[,]* following the *stream-number* is required if none of AT, PROMPT or the first *[;]* are present.

column and *line* are *integer-expressions* giving the screen position at which to start the prompt etc.

prompt is *prompt-string prompt-separator,*

where *prompt-string* is PROMPT *string-expression* or *string-literal*

and *prompt-separator* is , or ;.

LOAD MODULE (see Chapter 7)

Use LOAD MODULE is used to load a module into memory.

Syntax LOAD MODULE *module-identifier , filename*

where *module-identifier* is *numeric-identifier*

filename is *string-expression*

LOC (see Chapter 13)

Use LOC is a function which returns the current record number.

Syntax LOC(*[#]stream-number*)

LOCATE (see Chapters 11 & 14)

Use LOCATE moves to a given character and line position.

Syntax LOCATE *[#stream-number,] column; line*

LOCK (see Chapter 13)

Use LOCK sets a new record lock.

Syntax LOCK *[#]stream-number, lock-type [[,] position]*
LOCK *[#]stream-number [[,] position] [,]* LOCK *lock-type*

where *position* is as described in POSITION.

LOF (see Chapter 13)

Use LOF returns the length of a file.

Syntax LOF(*[#]stream-number*)

LOG (see Chapter 3)

Use LOG is a function which returns the logarithm to base e of its argument – the natural logarithm.

Syntax LOG(*argument*)

where *argument* is a *numeric-expression*, giving a positive, non-zero, value.

LOG2 (see Chapter 3)

Use LOG2 is a function which returns the logarithm to base 2 of its argument.

Syntax LOG2(*argument*)

where *argument* is a *numeric-expression*, giving a positive, non-zero, value.

LOG10 (see Chapter 3)

Use LOG10 is a function which returns the logarithm to base 10 of its argument.

Syntax LOG10(*argument*)

where *argument* is a *numeric-expression*, giving a positive, non-zero, value.

LOWER (see Chapter 2)

Use The LOWER function may be used to determine the lower bound of one of the dimensions of an array. (See also LBOUND.)

Syntax LBOUND(*actual-array [, index-number]*)

where *index-number* must be an *integer-expression.*in the range 1 to 7.

LOWER$ (see Chapter 4)

Use The LOWER$ functionconverts characters to lower case. (See also LCASE$.)

Syntax LOWER$(*argument*)

where *argument* is a *string-expression*

LPRINT (see Chapter 11)

Use LPRINT is just like print, except that the output is always to stream 0.

Syntax LPRINT *[print-item]...*

LSET (see Chapter 4)

Use LSET is used to assign to a *string-general-variable* without changing the length of the string it holds.

Syntax LSET *string-general-variable* = *string-expression*

LT (see Chapter 12)

Use LT changes the current heading. (See also LEFT, RIGHT and RT.)

Syntax LT *[#stream-number,] angle-change*

where *angle-change* is a *numeric-expression.*

LTRIM$ (see Chapter 4)

Use LTRIM$ is a function to remove leading spaces from strings.

Syntax LTRIM$(*argument*)

where *argument* is a *string-expression*

MAX (see Chapter 3)

Use MAX is a function which returns the value of its largest argument.

Syntax MAX(*argument, argument [, argument]* ...)

where *argument* is *numeric-expression.*

MAXNUM (see Chapter 3)

Use MAXNUM is a function which returns the largest representable number.

Syntax MAXNUM

MD (see Chapter 15)

Use MD makes a new directory. (See also MKDIR.)

Syntax MD *rest-of-line*

where *rest-of-line* is everything up to the end of the line

MID$ – Function (see Chapter 4)

Use MID$ is a function to select a substring from a given string.

Syntax MID$(*argument*, *start* [, *length*])

where *argument* is a *string-expression*

start and *length* are *integer-expressions*. *start* must give a value greater than zero; *length* must give a value greater than or equal to zero. Neither may be greater than 4096.

MID$ – Statement (see Chapter 4)

Use MID$ assigns a string to a substring.

Syntax MID$(*string-general-variable*, *start* [, *length*]) = *string-expression*

where *start* and *length* are *integer-expressions*. *start* must give a value greater than zero; *length* must give a value greater than or equal to zero. Neither may be greater than 4096.

MIN (see Chapter 3)

Use MIN is a function which returns the value of its smallest argument.

Syntax MIN(*argument*, *argument* [, *argument*] ...)

where *argument* is *numeric-expression*.

MKDIR (see Chapter 15)

Use These statements make a new directory. (See also MD.)

Syntax MD *rest-of-line*

where *rest-of-line* is everything up to the end of the line

MODULE (see Chapter 7)

Use MODULE is used to define a module – a program unit which may be accessed and executed by a BASIC 2 Plus program.

Syntax
```
MODULE module-identifier [ ( formal-list ) ]
    module-body
END MODULE
```

MOVE (see Chapter 11)

Use MOVE moves the graphics cursor.

Syntax MOVE [#*stream-number*,] *point*

NAME (see Chapter 15)

Use NAME changes the name of a file. (See also REN.)

Syntax NAME *old-file-name* AS *new-file-name*

where *old-file-name* and *new-file-name* are *file-name* s

NEXT (see Chapter 5)

Use Terminates the sequence of statements which make up the body of a FOR loop.

Syntax NEXT *[loopcounter1] [, [loopcounter2]] [, [loopcounter3]]* ...

OFF (see Chapter 3)

Use OFF is a function which always returns 0, the value which BASIC 2 Plus deems to represent the Boolean value 'false'. (See also FALSE.)

Syntax OFF

ON (see Chapter 3)

Use ON is a function which always returns –1. (See also TRUE.)

Syntax ON

ON ERROR (see Chapter 16)

Use ON ERROR specifies what action is to be taken in the event of an error.

Syntax ON ERROR STOP

or ON ERROR GOTO *location*

or ON ERROR EXIT *place*

where *place* is MODULE, SUB or FUNCTION, as appropriate.

ON...GOSUB (see Chapter 6)

Use Like ON...GOTO, this statement is a relic from older BASICs, included to make BASIC 2 plus compatible.

Syntax ON *condition* GOSUB *[location1] [, [location2]] [, [location3]]* ...

where *condition* is an *integer-expression*

location1, *location2*, *location3*, ... etc are *locations*.

ON...GOTO (see Chapter 5)

Use ON...GOTO is a relic from older BASICs, included to make BASIC 2 plus compatible.

Syntax ON *condition* GOTO *[location1][, [location2]][, [location3]]* ...

where *condition* is an *integer-expression*

location1, *location2*, *location3*, ... etc are *locations*.

OPEN (see Chapter 8)

Use OPEN associates a given stream with a device or file.

Syntax OPEN *[#]stream-number [,]* WINDOW *window-number*

or OPEN *[#]stream-number [,]* INPUT *filename [[,]* LOCK *lock-type]*

or OPEN *[#]stream-number [,] [exist]* OUTPUT *filename [[,]* LOCK *lock-type]*

or OPEN *[#]stream-number [,] [exist]* APPEND *filename [[,]* LOCK *lock-type]*

or OPEN *[#]stream-number [,] [exist]* RANDOM *random-spec [[,]* LOCK *lock-type]*

or OPEN *[#]stream-number [,] [exist]* GRAPHICS *filename graphics-spec*

or OPEN *[#]stream-number [,]* PRINT *[printer-number]*

or OPEN *[#]stream-number [,]* DEVICE *device-number*

OPEN...APPEND (see Chapter 8)

Use OPEN file for output. (See also OPEN...OUTPUT.)

Syntax OPEN *[#]stream-number [,] [exist]* APPEND *filename [[,]* LOCK *lock-type]*

where *exist* is NEW or OLD

OPEN...DEVICE & OPEN...GRAPHICS (see Chapter 8)

Use These forms of OPEN associate a given stream with a GEM graphics device or a Metafile.

Syntax OPEN *[#]stream-number [,]* DEVICE *device-number*

OPEN *[#]stream-number [,] [exist]* GRAPHICS *filename graphics-spec*

where *exist* is NEW or OLD

device-number is an *integer-expression*, giving a value in the range 0..32767.

graphics-spec is *[[,]* SIZE *m-width*,*m-height] [[,]* SPACE *p-width*[,*p-height*]]

m-width and *m-height* are *numeric-expressions*, giving positive, non-zero, values.

p-width and *p-height* are *integer-expressions*, giving values in the range 1..32768.

OPEN...INPUT (see Chapter 8)

Use OPEN file for input.

Syntax OPEN *[#]stream-number [,] INPUT filename [[,] LOCK lock-type]*

OPEN...OUTPUT (see Chapter 8)

Use OPEN file for output. (See also OPEN...APPEND.)

Syntax OPEN *[#]stream-number [,] [exist]* OUTPUT *filename [[,] LOCK lock-type]*

where *exist* is NEW or OLD

OPEN...PRINT (see Chapter 8)

Use OPEN...PRINT associates a given stream with a simple printer device.

Syntax OPEN *[#]stream-number [,] PRINT [printer-number]*

where *print-number* is an *integer-expression*, giving a value in the range 0..5 :

0 = PRN
1..3 = LPT1 to LPT3
4..5 = COM1 to COM2

OPEN...RANDOM (see Chapter 13)

Use OPEN...RANDOM associates a given stream with a random or keyed file.

Syntax OPEN *[#]stream-number [,] [exist]* RANDOM *random-spec [[,] LOCK lock-type]*

where *random-spec* is *filename [[,] INDEX filename] [[,] LENGTH record-length]*

record-length is an *integer-expression*, giving a value in the range 1..4096.

OPEN...WINDOW (see Chapter 8)

Use OPEN...WINDOW associates a given stream with a given screen window.

Syntax OPEN *[#]stream-number [,] WINDOW window-number*

where *window-number* is an *integer-expression*, giving a value in the range 1..4.

OSERR (see Chapter 16)

Use OSERR is a function which gives more information about the latest "operating system dependent" error.

Syntax OSERR

PI (see Chapter 3)

Use PI is a function which returns the representable value closest to the mathematical constant π.

Syntax PI

PIE (see Chapter 12)

Use PIE draws a circular "pie" section on a graphics device.

Syntax PIE *[#stream-number,] point, radius, start-angle, end-angle [[,] attribute]...*

where *attribute* is: FILL *[ONLY] [WITH fill-style]*
or: WIDTH *line-width*
or: STYLE *line-style*
or: COLOUR *colour-number*
or: COLOR *colour-number*
or: MODE *write-mode*

radius is an *integer-expression*, giving a positive, non-zero value.

start-angle and *end-angle* are *numeric-expressions*, giving positive, or zero, values.

PLOT (see Chapter 12)

Use PLOT plots "markers" at the points given.

Syntax PLOT *[#stream-number,] point [, point] ... [[,] attribute]...*

where *attribute* is: MARKER *marker-number*
or: SIZE *marker-size*
or: COLOUR *colour-number*
or: COLOR *colour-number*
or: MODE *write-mode*

POINT (see Chapter 12)

Use POINT sets a new current heading.

Syntax POINT *[#stream-number,] angle*

where *angle* is a *numeric-expression.*

POINTSIZE (see Chapter 12)

Use POINTSIZE is a function which returns an available size of a font.

Syntax POINTSIZE(*[#stream-number,] font-ordinal, point-size*)

where *font-ordinal* and *point-size* are *integer-expressions*, giving positive, non-zero values.

POS (see Chapters 9 & 12)

Use These function return the current cursor position in characters and lines. (See also VPOS)

Syntax `POS[(`*[#]stream-number*`)]`

POSITION (see Chapter 13)

Use `POSITION` moves to a new position in a Random or Keyed file.

Syntax `POSITION` *[#]stream-number* `[,]` *position* `[[,] LOCK` *lock-type]*

where *position* is: `NEXT`
or: `AT` *record-number*
or: `AT` *position-string*
or: `KEY` *key-value* `[[,] INDEX` *index-number]*
or: `INDEX` *index-number*

where *key-value* is an *expression*, giving a value compatible with the index's key type.
index-number is an *integer-expression*, giving a value in the range 1..20.
position-string is a *string-expression*, giving a value once returned by `POSITION$`.

POSITION$ (see Chapter 13)

Use `POSITION$` is a function which returns a string which unambiguously specifies the current position.

Syntax `POSITION$(`*[#]stream-number*`)`

PRINT (see Chapter 11)

Use `PRINT` outputs numbers and strings.

Syntax `PRINT` *[#stream-number[,]] [print-item]...*

where the *[,]* following the *stream-number* is required if the first *print-item* is an *expression*.
each *print-item* may be: *expression*
or *separator*
or *print-command*– see Print Commands box
or *print-function* – see Print Functions box
or `USING` *string-expression* `;`

provided that no *expression* immediately follows another.
separator is `,` or `;`

PRINT COMMANDS (see Chapter 11)

Use These may be used in PRINT or LPRINT statements.

Syntax AT(*column*;*line*)
TAB(*integer-expression*)
ADJUST(*integer-expression*)

PRINT FUNCTIONS (see Chapter 11)

Use These may be used in PRINT or LPRINT statements, and in the EXTENT function.

Syntax ZONE(*integer-expression*)
MARGIN(*integer-expression*)
EFFECTS(*integer-expression*, *integer-expression*)
MODE(*write-mode*)
COLOUR(*colour-number*)
COLOR(*colour-number*)
FONT(*font-ordinal*)
POINTS(*point-size*)
ANGLE(*integer-expression*)

PROGPATH$ & PROGFILE$ (see Chapter 7)

Use These functions return the current file name of the main program.

Syntax PROGPATH$
PROGFILE$

PUT (see Chapter 13)

Use PUT puts data to Random or Keyed Files.

Syntax PUT *[#]stream-number*, *string-expression [[,] position] [[,]* LOCK *lock-type]*

where *position* is as described in POSITION.

QUIT (see User Guide Chapter 1)

Use SYSTEM takes you out of BASIC 2 Plus. (See also SYSTEM.)

Syntax SYSTEM

RAD (see Chapter 3)

Use RAD is a function to converts angles in degrees to radians.

Syntax RAD(*argument*)

where *argument* is a *numeric-expression.*

RANDOMIZE (see Chapter 3)

Use RANDOMIZE is used to set the seed for BASIC 2 Plus's pseudo-random number generator.

Syntax RANDOMIZE *[seed-value]*

where *seed-value* is a *numeric-expression.*

RD (see Chapter 15)

Use These statements delete (remove) a directory. (See also RMDIR.)

Syntax RD *rest-of-line*

where *rest-of-line* is everything up to the end of the line

READ (see Chapter 2)

Use READ is used to read data from a DATA statement into *general–variables.*

Syntax READ *general–variable [, general–variable]* ...

RECORD (see Chapter 2)

Use RECORD is used to specify a record structure which can be imposed on strings.

Syntax RECORD *record-name* ; *field-definition [, field-definition]* ...

where *field-definition* is: *numeric-field-name [vector-field] [storage-class]*
or: *string-field-name [vector-field]* FIXED *length*

numeric-field-name is a *numeric-identifier*, giving the name of a numeric field

string-field-name is a *string-identifier*, giving the name of string field

vector-field is (*dimension-bounds*)
or [*dimension-bounds*]

where *dimension-bounds* is as in DIM.

storage-class is *integer-class [*KEY*]*
or IEEE4 orIEEE8

where *integer-class* is one of BYTE, UBYTE, WORD, UWORD, INTEGER

length is an *integer-expression* giving a value in the range 1..4096.

REN (see Chapter 15)

Use REN changes the name of a file. (See also NAME.)

Syntax REN *rest-of-line*

where *rest-of-line* is everything up to the end of the line

REPEAT (see Chapter 5)

Use The REPEAT statement is used for repeating a sequence of statements when the repetition is governed by some condition which may be tested before or after the execution of the loop.

Syntax REPEAT : *loopbody* : UNTIL *condition*

where *condition* is a *truth-value*

loopbody is a sequence of BASIC 2 Plus statements, separated in the usual way with colons or new lines.

REPOSITION (see Chapter 13)

Use REPOSITION moves to a new position in a Keyed file, without changing record.

Syntax REPOSITION *[#]stream-number [,] key [[,] LOCK lock-type]*

where *key* is: KEY *key-value [[,] INDEX index-number]*

where *key-value* is an *expression*, giving a value compatible with the index's key type

index-number is an *integer-expression*, giving a value in the range 1..20.

RESTORE (see Chapter 2)

Use RESTORE is used to explicitly move the pointer which indicates the next DATA item to be read in.

Syntax RESTORE *[location]*

RESUME (see Chapter 16)

Use RESUME signals the end of "error state".

Syntax RESUME

or RESUME NEXT

or RESUME EXIT *place*

or RESUME *location*

or RESUME STOP

where *place* is MODULE, SUB or FUNCTION, as appropriate.

RETURN (see Chapter 6)

Use RETURN is used to terminate a subroutine.

Syntax RETURN

RIGHT (see Chapter 12)

Use These statements change the current heading. (See also RT, LEFT and LT.)

Syntax RIGHT *[#stream-number,] angle-change*

where *angle-change* is a *numeric-expression.*

RIGHT$ (see Chapter 4)

Use RIGHT$ is a function to truncate a string by selecting only its rightmost characters.

Syntax RIGHT$(*argument , length*)

where *argument* is a *string-expression* and *length* is an *integer-expression* returning a value in the range 0..4096.

RMDIR (see Chapter 15)

Use RMDIR deletes (removes) a directory. (See also RMDIR.)

Syntax RD *rest-of-line*

where *rest-of-line* is everything up to the end of the line

RND (see Chapter 3)

Use RND is a function which returns a pseudo-random number.

Syntax RND*[(argument)]*

where *argument* is an *integer-expression*, giving a value in the range 1..65535.

ROUND (see Chapter 3)

Use ROUND is function which returns the value of its argument rounded to a specific number of decimal places.

Syntax ROUND(*argument[, rounding]*)

where *argument* is a *numeric-expression* and *rounding* is an *integer-expression.*

RSET (see Chapter 4)

Use RSET is used to assign to a *string-general-variable* without changing the length of the string it holds.

Syntax RSET *string-general-variable* = *string-expression*

RT (see Chapter 12)

Use RT changes the current heading. (See also RIGHT, :LEFT and LT.)

Syntax RT *[#stream-number,] angle-change*

where *angle-change* is a *numeric-expression.*

RTRIM$ (see Chapter 4)

Use RTRIM$ is a function to remove trailing spaces from strings.

Syntax RTRIM$(*argument*)

where *argument* is a *string-expression*

RUN (see User Guide Chapter 1)

Use RUN starts executing the current program.

Syntax RUN

SCREEN (see Chapter 9)

Use SCREEN sets the type and size of a Virtual Screen

Syntax SCREEN *[#stream-number [,]]* GRAPHICS *[size] [[,] window-attribute]...*

SCREEN *[#stream-number [,]]* TEXT *[size] [[,] window-attribute]...*

SCREEN *[#stream-number [,]]* TEXT FLEXIBLE *[[,] window-attribute]...*

where *size* is *width* [FIXED], *height* [FIXED]

window-attribute is: MAXIMUM *width, height*

or: MINIMUM *width, height*

or: UNIT *width, height*

or: INFORMATION *truth-value*

width and *height* are *integer-expressions*, giving positive, non-zero values.

SELECT CASE (see Chapter 5)

Use SELECT CASE is used to choose one course of action from several, the choice being determined by the value of a given expression.

Syntax

```
SELECT CASE selector-expression [:]
   [CASE testvalues1 [ , testvalues1a ][ , testvalues1b ] ... : statements1 : ]
   [CASE testvalues2 [ , testvalues2a ][ , testvalues2b ] ... : statements2 : ]
   .....
   [CASE ELSE : elsestatements : ]
END SELECT
```

where *statements1*, *statements2*, *statements3*, ..., and *elsestatements* are sequences of BASIC statements separated by colons or new lines in the usual way

testvalues may be either of the form

expression [TO *end-expression*]

or IS *relation limit-expression*

where all the *expressions* must be of the same type as *selector-expression*.

SELECTOR (see Chapter 15)

Use SELECTOR runs an "item" selector dialogue.

Syntax

```
SELECTOR directory-string [, selection-string]
```

or

```
SELECTOR , selection-string
```

SELPATH$ & SELFILE$ & SELWILD$ (see Chapter 15)

Use These functions return parts of the values stored for the SELECTOR dialogue.

Syntax

```
SELPATH$
SELFILE$
SELWILD$
```

SET (see Chapter 12)

Use SET changes the text output attributes for a given stream.

Syntax SET *[#stream-number[,]] text-attribute [[,]text-attribute]...*

where each *text-attribute* may be one of :

ZONE *zone-size*
MARGIN *margin-position*
EFFECTS *effects-on, effects-off*
MODE *write-mode*
COLOUR *colour-number*
COLOR *colour-number*
FONT *font-ordinal*
POINTS *point-size*
ANGLE *text-angle*
WRAP *truth-value*

SGN (see Chapter 3)

Use SGN is a function which returns the sign of the value of its argument.

Syntax SGN(*argument*)

where *argument* is a *numeric-expression.*

SHAPE (see Chapter 12)

Use SHAPE draws a polygon shape.

Syntax SHAPE *[#stream-number,] point, point, point [, point]... [[,] attribute]...*

where *attribute* is: FILL *[ONLY] [WITH fill-style]*
or: WIDTH *line-width*
or: STYLE *line-style*
or: COLOUR *colour-number*
or: COLOR *colour-number*
or: MODE *write-mode*

SHARED (see Chapter 6)

Use SHARED makes global variables directly available to routines.

Syntax SHARED *shared-object [, shared-object] ...*

where *formal-var* is *simple-variable* or *array-identifier* (*[,] ...*)or *array-identifier* [*[,] ...*]

SIN (see Chapter 3)

Use SIN is a function which returns the sine of its argument.

Syntax SIN(*argument*)

where *argument* is a *numeric-expression.*

SQR (see Chapter 3)

Use SQR is a function which returns the (positive) square root of its argument.

Syntax SQR(*argument*)

where *argument* is a *numeric-expression.* returning a positive, or zero value.

STOP (see Chapter 5)

Use STOP is used to halt the execution of the program in a condition where the variables can be inspected or altered.

Syntax STOP

STR$ (see Chapter 3)

Use STR$ is a function to convert a numeric value to string form.

Syntax STR$(*argument*)

where *argument* is a *numeric-expression.*

STREAM (see Chapter 8)

Use The STREAM statement changes the default stream. The STREAM function returns the current stream.

Statement Syntax STREAM *[#]stream-number*

Function Syntax STREAM

STRING$ (see Chapter 4)

Use STRING$ is a function to generate a string of repeated characters.

Syntax STRING$(*length* , *argument*)

where *length* is an *integer-expression* in the range 0..4096 and *argument* is a non-null *string-expression* or an *integer-expression* giving a value in the range 0..255.

SUB (see Chapter 6)

Use SUB is used to define a subprogram.

Syntax SUB *sub-identifier* [(*formal-list*)] [EXPORT]
sub-body :
END SUB

where *sub-identifier* is a *name*

sub-body is a sequence of BASIC 2 Plus statements, separated with colons or new lines in the usual way.

SWAP (see Chapter 2)

Use SWAP exchanges the values of two general variables.

Syntax SWAP *general-variable* , *general-variable*

SYSTEM (see User Guide Chapter 1)

Use SYSTEM takes you out of BASIC 2 Plus. (See also QUIT.)

Syntax SYSTEM

TAN (see Chapter 3)

Use TAN is a function which returns the tangent of its argument.

Syntax TAN(*argument*)

where *argument* is a *numeric-expression.*

TEST (see Chapters 9 & 12)

Use TEST is a function which returns the colour of a point on a Graphics Screen.

Syntax TEST([#*stream-number*,] *point*)

NB: TEST may have no effect, depending on the device driver.

TEXT (see Chapter 11)

Use TEXT performs various operations available on text screens.

Syntax TEXT [#*stream-number*,] CLEAR *clear-area*

or TEXT [#*stream-number*,] DELETE [LINE]

or TEXT [#*stream-number*,] INSERT [LINE]

or TEXT [#*stream-number*,] FEED *count*

where *clear-area* is one of EOL, BOL, LINE, EOS, BOS, SCREEN

count is an *integer-expression.*

TIME (see Chapter 3)

Use TIME is a function which returns the number of hundredths of a second which have elapsed since midnight, to the nearest hundredth.

Syntax TIME

TIME$ (see Chapter 4)

Use TIME$ is a function which returns the number of hundredths of a second which have elapsed since midnight, to the nearest hundredth, in the form of a string.

Syntax TIME$

TIMER (see Chapter 3)

Use TIMER is a function which gives the number of seconds which have elapsed since midnight.

Syntax TIMER

TOWARD (see Chapter 12)

Use TOWARD is a function which returns the heading from the cursor to a given point.

Syntax TOWARD([#*stream-number*,] *point*)

TRUE (see Chapter 3)

Use TRUE is a functios which always returns –1. (See also ON.)

Syntax TRUE

TRUNC (see Chapter 3)

Use TRUNC is a function which returns the value of their *argument* rounded to an integer. Rounding is towards zero. (See also FIX.)

Syntax TRUNC(*argument*)

where *argument* is a *numeric-expression*.

TYPE (see Chapter 15)

Use TYPE lists the contents of a file. (See also DISPLAY.)

Syntax TYPE *rest-of-line*

where *rest-of-line* is everything up to the end of the line.

UBOUND (see Chapter 2)

Use UBOUND may used to determine the upper bound of one of the dimensions of an array. (See also UPPER.)

Syntax UBOUND(*actual-array [, index-number]*)

where *index-number* must be an *integer-expression*.in the range 1 to 7.

UCASE$ (see Chapter 4)

Use UCASE$ converts characters to upper case. (See also UPPER$.)

Syntax UCASE$(*argument*)

where *argument* is a *string-expression*

UNIQUE (see Chapter 13)

Use UNIQUE is a function which returns the next "unique" number for a Keyed File.

Syntax UNIQUE(*[#]stream-number*)

UNLOAD (see Chapter 7)

Use UNLOAD MODULE is used to unload a module.

Syntax UNLOAD MODULE *module-identifier*

UPPER (see Chapter 2)

Use UPPER may used to determine the upper bound of one of the dimensions of an array. (See also UBOUND.)

Syntax UPPER(*actual-array [, index-number]*)

UPPER$ (see Chapter 4)

Use UPPER$ converts characters to upper case. (See also UCASE$.)

Syntax UPPER$(*argument*)

where *argument* is a *string-expression*

USER...ORIGIN (see Chapter 12)

Use USER_ORIGIN changes where the origin of user coordinate space appears on the device.

Syntax USER *[#stream-number]* ORIGIN *point*

USER...SPACE (see Chapter 12)

Use USER_SPACE sets the extent of the user coordinate space.

Syntax USER *[#stream-number]* SPACE *width[, height]*

where *width* and *height* are *integer-expressions*, giving positive, non-zero values.

where *index-number* must be an *integer-expression*.in the range 1 to 7.

VAL (see Chapter 3)

Use VAL converts the string representation of a number to its numeric value.

Syntax VAL(*argument*)

where *argument* is a *string-expression*.

VERSION (see Chapter 15)

Use VERSION is a function which returns vesrion information.

Syntax VERSION(*query*)

where *query* is an *integer-expression*, giving a value in the range 0..4.

VPOS (see Chapters 9 & 12)

Use VPOS returns the current cursor position in characters and lines. (See also POS)

Syntax VPOS*[(][#]stream-number)]*

WHILE (see Chapter 5)

Use The WHILE statement is used for repeating a sequence of statements when the repetition is governed by some condition which may be tested before or after the execution of the loop.

Syntax WHILE *condition* : *loopbody* : WEND

where *condition* is a *truth-value*

loopbody is a sequence of BASIC 2 Plus statements, separated in the usual way with colons or new lines.

WHOLE$ (see Chapter 4)

Use WHOLE$ is a function which returns the whole of a fixed-length string.

Syntax WHOLE$(*argument*)

where *argument* is a *string-expression*

WINDOW (see Chapter 14)

Use WINDOW statements change the size, position etc. of Screen Windows.

Syntax WINDOW *[#stream-number]* CLOSE
or WINDOW *[#stream-number]* FULL *truth-value*
or WINDOW *[#stream-number]* OPEN
or WINDOW *[#stream-number]* SIZE *width, height*
or WINDOW *[#stream-number]* PLACE *point*
or WINDOW *[#stream-number]* TITLE *string-expression*
or WINDOW *[#stream-number]* INFORMATION *string-expression*
or WINDOW *[#stream-number]* MOUSE *mouse-form*
or WINDOW *[#stream-number]* CURSOR *truth-value*
or WINDOW *[#stream-number]* SCROLL *point*

where *mouse-form* is an *integer-expression*, giving a value in the range 0..7
width and *height* are *integer-expressions*, giving a value 1..5000

XACTUAL & YACTUAL (see Chapter 14)

Use XACTUAL and YACTUAL are functions which return the actual size of a window, in screen device pixels.

Syntax XACTUAL*[(][#]stream-number)]*
YACTUAL*[(][#]stream-number)]*

XBAR & YBAR (see Chapter 14)

Use XBAR and YBAR are functions which return the actual size of the scroll barsof a window, in screen device pixels.

Syntax XBAR*[(][#]stream-number)]*
YBAR*[(][#]stream-number)]*

XCELL & YCELL (see Chapter 12)

Use XCELL and YCELL are functions which return the size of a character cell, in user coordinates.

Syntax XCELL*[(][#]stream-number)]*
YCELL*[(][#]stream-number)]*

XDEVICE & YDEVICE (see Chapters 9 & 12)

Use XDEVICE and YDEVICE are functions which return the size of a graphics device, in device pixels.

Syntax XDEVICE*[(#]stream-number)]*

YDEVICE*[(#]stream-number)]*

XMETRES & YMETRES (see Chapters 9 & 12)

Use XMETRES and YMETRES are functions which return the size of a graphics device, in metres.

Syntax XMETRES*[(#]stream-number)]*

YMETRES*[(#]stream-number)]*

XMOUSE & YMOUSE (see Chapter 14)

Use These functions return the current position of the mouse pointer, in screen device pixels.

Syntax XMOUSE

YMOUSE

XPIXEL & YPIXEL (see Chapters 9 & 12)

Use XPIXEL and YPIXEL are functions which return the size of a device pixel, in user coordinates.

Syntax XPIXEL*[(#]stream-number)]*

YPIXEL*[(#]stream-number)]*

XPLACE & YPLACE (see Chapter 14)

Use XPLACE and YPLACE are functions which return the position of the bottom left hand corner of a window, in screen device pixels.

Syntax XPLACE*[(#]stream-number)]*

YPLACE*[(#]stream-number)]*

XPOS & YPOS (see Chapters 9 & 12)

Use XPOS and YPOS are functions which return the current cursor position, in user coordinates.

Syntax XPOS*[(#]stream-number)]*

YPOS*[(#]stream-number)]*

XSCROLL & YSCROLL (see Chapter 14)

Use XSCROLL and YSCROLL are functions which return the position of the window on the virtual screen, in user coordinates.

Syntax XSCROLL*[([#]stream-number)]*

YSCROLL*[(#]stream-number)]*

XUSABLE & YUSABLE (see Chapter 9)

Use XUSABLE and YUSABLE are functions which return the usable size of a graphics device, in device pixels.

Syntax XUSABLE*[(#]stream-number)]*

YUSABLE*[(#]stream-number)]*

XVIRTUAL & YVIRTUAL (see Chapter 9)

Use XVIRTUAL and YVIRTUAL are functions which return the size of a graphics virtual screen, or other graphics device, in user coordinates..

Syntax XVIRTUAL*[(#]stream-number)]*

YVIRTUAL*[(#]stream-number)]*

XWINDOW & YWINDOW (see Chapter 14)

Use XWINDOW and YWINDOW are functions which return the size of a window, in screen device pixels.

Syntax XWINDOW*[(#]stream-number)]*

YWINDOW*[(#]stream-number)]*

YASPECT (see Chapters 9 & 12)

Use YASPECT is a function which gives the aspect ratio of a user coordinate pixel.

Syntax YASPECT*[(#]stream-number)]*

ZONE (see Chapter 11)

Use ZONE sets the print zone size for the Dialogue Screen.

Syntax ZONE *zone-size*

Keywords

ABS
ACOS
ADDKEY
ADDREC
ADJUST
ALERT
AND
ANGLE
APPEND
ARC
AS
ASC
ASIN
AT
ATAN
ATAN2
ATN
BIN$
BOL
BOX
BUTTON
BYTE
CALL
CASE
CD
CEIL
CEILING
CHDIR
CHDIR$
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COLOR
COLOUR
CONSOLIDATE
CONT
COS
CURRENCY$
CURSOR
DATA
DATE
DATE$
DEC$
DECIMAL
DECLARE
DEF
DEG
DEGREES
DEL
DELETE
DELKEY
DEVICE
DIM
DIMENSIONS
DIR
DISPLAY
DISTANCE
DO
DRIVE
EDIT
EFFECTS
ELLIPSE
ELLIPTICAL
ELSE
ELSEIF
END
ENVIRON$
EOF
EOL
EOS
EPS
ERASE
ERR
ERROR
ERROR$
EXIT
EXP
EXPORT
EXTENT
FALSE
FD
FEED
FILES
FILL
FIND$
FINDDIR$
FIX
FIXED
FLEXIBLE
FLOOD
FLOOR
FONT
FONT$
FOR
FORWARD
FRAC
FRE
FREEFILE
FULL
FUNCTION
GET
GOSUB
GO
GOTO
GRAPHICS
HEADING
HEX$
IF
IMPORT
INDEX
INF
INFORMATION
INKEY
INKEY$
INPUT
INPUT$
INSERT
INSTR
INT
INTEGER
IS
KEY
KEYU$
KEYSPEC
KILL
LABEL
LBOUND
LCASE$
LEFT
LEFT$
LEN
LENGTH
LET
LINE
LOAD
LOC
LOCATE
LOCK
LOF
LOG
LOG10
LOOP
LOWER
LOWER$
LPRINT
LSET
LTRIM$

MARGIN
MARKER
MAX
MAXIMUM
MAXNUM
MIN
MINIMUM
MD
MIDS$
MKDIR
MOD
MODE
MODULE
MOUSE
MOVE
NAME
NEW
NEXT
NOT
OFF
OLD
ON
ONLY
OPEN
OPTION
OR
ORIGIN
OSERR
OUTPUT
PART
PI
PIE
PLACE
PLOT
POINT
POINTS
POINTSIZE
POS
POSITION
POSITION$
PRINT
PROMPT
PUT
QUIT
RAD
RADIAN
RANDOM
RANDOMIZE
RD
READ
RECORD
REM
REN
REPEAT
REPOSITION
RESET
RESTORE
RESUME
RETURN
RIGHT
RIGHT$
RMDIR
RND
ROUND
ROUNDED
RSET
RT
RTRIM$
RUN
SCREEN
SCROLL
SELECT
SELECTOR
SELFILE$
SELPATH$
SELWILD$
SET
SGN
SHAPE
SHARED
SIN
SIZE
SPACE
SQR
START
STEP
STOP
STR$
STREAM
STRING$
STYLE
SUB
SWAP
SYSTEM
TAB
TAN
TEST
TEXT
THEN
TIME
TIME$
TIMER
TITLE
TO
TOWARD
TRAP
TRUE
TRUNC
TYPE
UBOUND
UBYTE
UCASE$
UNIQUE
UNIT
UNLOAD
UNTIL
UPDATE
UPPER
UPPER$
USER
USING
UWORD
VAL
VAR
VERSION
VPOS
WEND
WHILE
WHOLE$
WIDTH
WINDOW
WITH
WORD
WRAP
XACTUAL
XBAR
XCELL
XDEVICE
XMETRES
XOR
XPIXEL
XPLACE
XPOS
XSCROLL
XUSABLE
XVIRTUAL
XWINDOW
YACTUAL
YBAR
YCELL
YDEVICE
YMETRES
YOR
YPIYEL
YPLACE
YPOS
YSCROLL
YUSABLE
YVIRTUAL
YWINDOW
ZONE

Important Notice

THE SOFTWARE CONTAINED IN THE DISKETTE PACKAGE IS SUPPLIED TO YOU ON THE TERMS AND CONDITIONS INDICATED BELOW. THE OPENING OF THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF SUCH TERMS AND CONDITIONS ARE NOT ACCEPTED BY YOU, YOU MUST RETURN THE UNOPENED PACKAGE TO THE PLACE OF PURCHASE AND YOUR MONEY WILL BE REFUNDED. NO REFUNDS WILL BE GIVEN WHERE THE PACKAGE HAS BEEN OPENED UNLESS THE PRODUCT IS FAULTY AND SUCH REFUND BECOMES PAYABLE UNDER CLAUSE 7 BELOW

In this notice, the terms:

'Locomotive' means Locomotive Software Limited

'The Program' means the programs known as B2PLUS.RSC and B2PLUS.APP on the diskette supplied in the diskette package.

1. Copyright

Material within The Program is copyright Locomotive.

Locomotive grants to the purchaser of this package a non-exclusive right to use The Program in accordance with these terms and conditions. Such Licence may be transferred only in accordance with Clause 3 below. Any other use or dealing not expressly authorised by these terms and conditions is strictly prohibited.

2. Use

The Program may only be used on a single machine or terminal at any one time but may be copied in support of that use. Any such copying is subject to there being no modification of The Program and in particular to the copyright notices of Locomotive being preserved in the copy. Save for copying as aforesaid, any other operations (including modification or translation from machine readable form) are expressly prohibited.

3. Transfer

The Program may be transferred to a third party provided the original and all copies are transferred or otherwise destroyed and provided further these terms and conditions are produced to that third party and prior to the transfer that party agrees and undertakes to observe and continue to observe the same. Without such transfer and undertaking any application of The Program or copies thereof by any other person will not be authorised by Locomotive and will be in breach of Locomotive's copyright and other proprietary rights.

4. Documentation

The documentation accompanying The Program is copyright Locomotive. However, no right to reproduce that documentation in part or in whole is granted by Locomotive. Should additional copies of the documentation be required for whatever reason, application must be made in writing to Locomotive which will be considered in its discretion.

5. Breach

If the user for the time being acts in breach of any of these terms and conditions it shall indemnify Locomotive against all loss suffered (including loss of profits) and the licence granted hereunder shall be deemed to be terminated forthwith. On termination the user shall deliver up to Locomotive all infringing and lawful copies of The Program.

6. Exclusions

Neither Locomotive nor any person authorised by it gives warranties or makes representations that The Program is error free or will meet functions required by the user. It shall be the responsibility of the user to satisfy itself that The Program meets the user's requirements. The Program is supplied on an 'as is' basis and save as expressly provided in these conditions all warranties of any nature (and whether express or implied) are excluded.

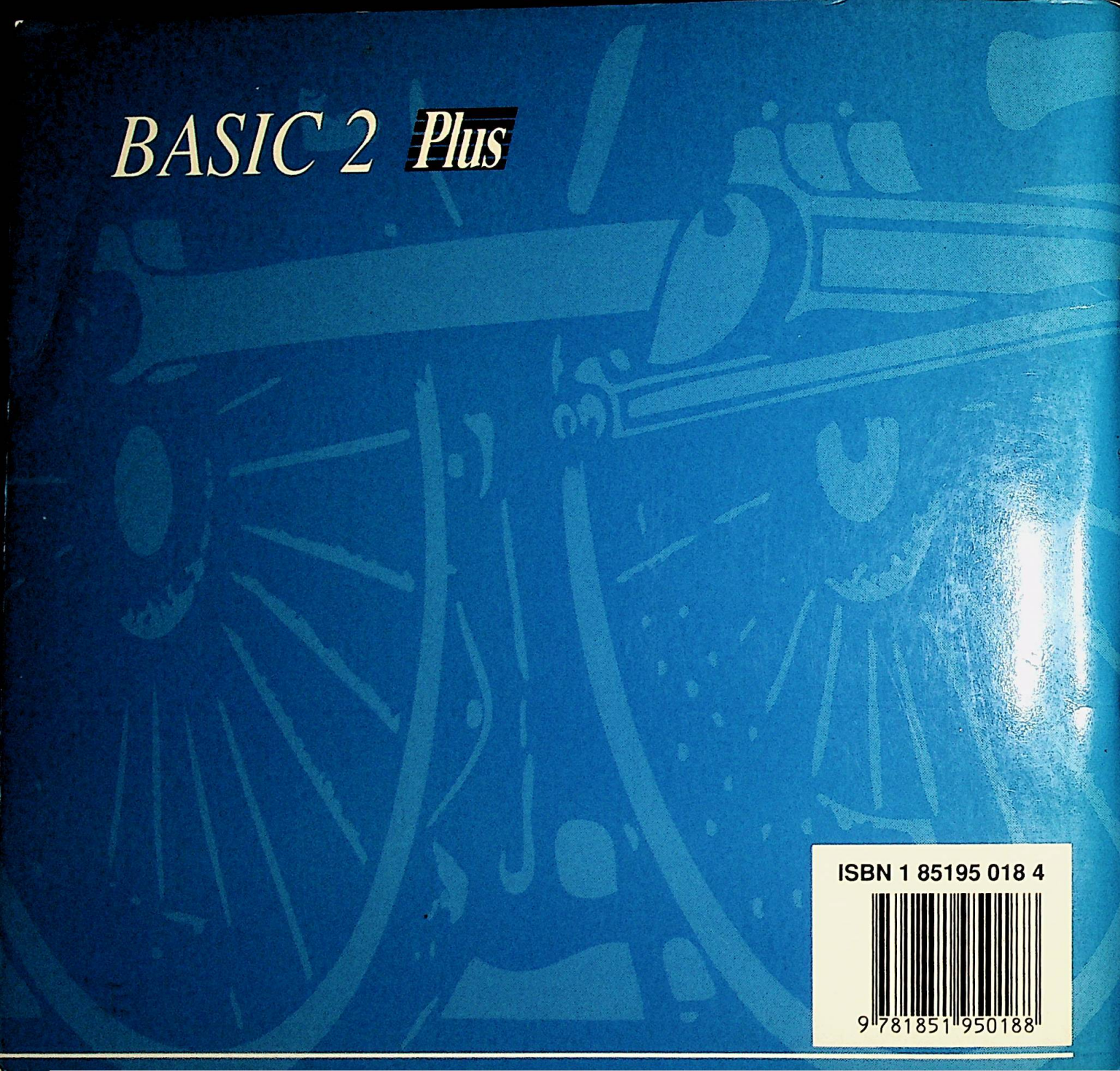
7. Liability

Locomotive warrants that the diskette on which the program is stored is free from material defect and through normal use will remain so for a period of 90 days after purchase. In the event of any breach of this warranty (or statutory warranty or conditions incapable of exclusion by these conditions) the responsibilities of Locomotive shall be limited to replacing the enclosed program or to returning the price paid for the same as they shall determine.

As the sole exception to the foregoing Locomotive will accept liability for death or personal injury resulting from its negligence. In no circumstance shall Locomotive be liable for any indirect or consequential costs damages or losses (including loss of business profits, operating time or otherwise) arising out of the use or inability to use the enclosed program and diskette whether or not the likelihood of damage was advised to Locomotive or its dealer.

This notice does not affect your statutory rights.

BASIC 2 Plus



ISBN 1 85195 018 4