National Physical Laboratory
Teddington
Middlesex TW11 0LW
Telephone 01 - 977 3222
Telex 262344

National Physical Laboratory

Microtext: an authoring system for computer dialogues

Microtext is a powerful but easy-to-use system which simplifies the production of a wide range of man–computer dialogues. Text, graphics and video can be used in one integrated presentation. Users with no computer experience can produce instructional material quickly and easily, while more experienced users can make use of many sophisticated facilities. Applications include interviewing systems, teaching packages, training courses, and interactive demonstrations and simulations.

Advantages of Microtext

The Microtext system comprises a simple but powerful author language combined with a wide range of author aids. Material to be presented can be quickly edited and reorganized, and its use can be controlled and monitored by the testing and tracing facilities provided.

The Microtext system isolates the content of the lesson material from the complexities of the computer, and thus considerably simplifies the author's task in producing lessons. The integral screen editor is used to prepare frames of text and graphics, and audio-visual presentation is controlled by simple commands. Each frame is terminated by simple branch instructions. The logic of the interaction is thus determined by a structure comparable to a numbered flow chart.

Applications

Microtext can be used for almost any interactive application which presents text, diagrams, or video to the user and prompts for a reply. It can be used for both obtaining information from the user in a 'form-filling' or interviewing situation, and giving information to the user in an instructional context. Commands can be incorporated to accept input from a light pen, touch screen or special keyboard, and to control peripherals such as clocks, timers, slide projectors, video-tape recorders and video-disc players.

Questionnaires: Microtext can be used to produce interactive questionnaires for use in interviewing and form-filling applications. The user can be asked multiple-choice questions, or prompted for answers which are either numeric or plain text. The results of the interview can be displayed, and summaries produced on a printer.

Education: Microtext enables teachers to produce their own computer-based lessons, and exchange material with other teachers who may be using different computers. Lessons can be written which assess the ability of the pupil and adapt the teaching level appropriately. It is particularly appropriate for remedial teaching since it can be used to hold the student's attention and thoroughly explore understanding of a topic.

Training: Microtext can significantly increase the productivity of a computer-based training department, by simplifying the process of producing instructional material, and where appropriate enabling the expert in a subject to author his own courseware. The training material may be purely descriptive, or if appropriate instruct the trainee in the use of external equipment. Reports can be generated to assist in the evaluation of both the trainee and the suitability of the training package.

Information: Microtext can be used as a videotex system for information retrieval. Applications can be menu-driven as in Viewdata, or can directly access information by identifying keywords in a plain language response. Microtext is also suitable for public displays. Information can be presented to the user in a dynamic fashion, and responses recorded for later analysis.

Expert advice: Microtext can guide a user through a complex external task, helping and prompting as required. This technique can be used both for training with simulated situations, and to provided detailed advice for genuine problems.

Technical specification

The *Microtext Language Interpreter* presents text and graphics on the screen, and accepts responses from the user. The screen can be divided into fixed and rolling parts, and responses can be prompted for at any position on the screen. Response matching includes single-character or keyword analysis, with optional numeric validation and range checks. Help menus can be supplied. Text is stored as a series of linked frames, and presentation and branching can be modified by the state of internal variables modelling the capabilities of the user. A concise textual summary of the interaction can be printed or archived as necessary. In Microtext-plus commands can be implemented to control peripherals such as slide projectors, video-disc players, light pens, touch screens, and special keyboards. The BBC implementation includes full colour and graphic support.

The *Microtext Editor* allows an author to create and modify Microtext modules. The full-screen editor incorporates character and line insert and delete, and frames can be created, copied, and printed, and complete modules can be loaded or saved to tape or disk. Modules can be run in test mode which includes status information and warnings of error conditions. Execution can be interrupted and variables or frames examined or modified before continuing.

The *Publishing System* enables applications using Microtext to be published in a secure form by producing a Delivery System which integrates the interpreter with an encoded application.

Implementations

The following suppliers are marketing versions of Microtext: Acornsoft (BBC model B tape and disc), Acorn Video Limited (Microtext-plus ROM for the BBC model B), and Transdata (CP/M and CP/M 86 machines). Versions for Commodore and Apple microcomputers are available, and licensees are being sought to market Microtext for these and other machines.

For further information on licensing arrangements contact Mr Robert Watson (extension 3980). General enquiries should be made to Mr David Barfoot (extension 3987), or Dr Nigel Bevan (extension 4011).

Microtext for the Commodore 64 & 128

64 Microtext is a new authoring system for interactive applications on the Commodore 64 and Commodore 128. Microtext is a frame-based authoring environment, developed at the National Physical Laboratory, and already available for the BBC Micro, Electron, Apple, IBM Micro, and CP/M systems. It is rapidly becoming accepted as a standard for Computer Based Training and related authoring on low-cost microcomputers, offering unique power and convenience to the non-computer-expert author.

64 Microtext is a new implementation of Microtext, by the original authors of the system on the BBC Micro. It offers significantly enhanced graphics facilities, including normal CBM graphics, BBC-compatible high resolution graphics, concurrent sprite movement and sound, and animation! Special facilities are as follows:

* Full Commodore graphics, using a very convenient full-screen editor, incorporating screen and border colour roll, and text colour override. Also supports multi-colour and extended background modes.

* BBC-compatible high-resolution mode, with $POINT, $LINE and $TRIANGLE. Can be used IN CONJUNCTION WITH normal Commodore graphics. 24K of user-space available in ALL graphics modes, including high resolution.

* Up to 512 user-defined characters, using a convenient $DEFCHAR command, allowing characters to be defined dynamically on a grid.

* Up to 8 sprites, also defined dynamically on a system-generated grid. The use of a concurrent scheduler allows sprites to be positioned anywhere on the screen, or SET IN MOTION without holding up normal Microtext execution.

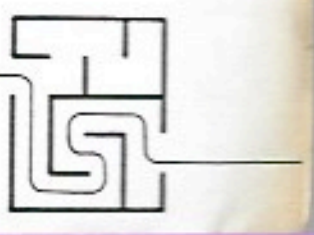
* $ANIMATE facility allowing up to 8 sprite definitions to be rotated while a sprite is stationary or moving, also concurrent with normal Microtext execution. Allows a wide range of animation effects to be created.

* Very powerful sound facilities, giving access to all SID-chip capabilities. Includes $VOICE, $FILTER, $SOUND and $NOTE for single note or chord formation. Sound is also handled concurrently, so that Microtext execution is not held up while sound is being output.

* Fully COMPUNET-compatible for remote learning applications. Microtext modules can be up/down-loaded to/from Compunet from anywhere in the country, usually at local phone rates.

Other facilities include user-port control and EXTENSION COMMAND facility, for control of special peripherals.

Microtext 64 is now available with full documentation for £45.00, from Ariadne Software Ltd, 273 Kensal Road, London W10. Tel (01) 960 0203.



CBM Microtext Demonstration

Sept '86, Ariadne Software Ltd.

Starting up

To get into Microtext, LOAD"MICROTEXT",8 (or LOAD"*",8) and RUN. This will cause the main system to be loaded, which takes about one minute.

Running demos

Once loading is complete, press any key to enter Command mode. To run demos, enter

```
RUN WEL.COM
```

which gives a menu of demo modules, as follows:

1. Escape from a hotel fire. Simple text-only module. Quite boring.

2-4. Microtext modes, Microtext commands, Basic concepts. Explanations of elementary Microtext, written in elementary Microtext. Useful when demonstrating editing.

5. Writing a program. Starts with a simple text-only Microtext question and answer, then adds bells and whistles. Text only, but quite pretty - illustrates more advanced Microtext coding.

6. Plug faultfinding. Illustrates use of CBM graphics (Microtext mode 0) as a (better) alternative to Teletext (mode 7) on BBC. Originally written on BBC using Teletext - transferred and auto-translated using a user-port link, then manually cleaned up.

7. Draw a bar chart. Illustates use of BBC-compatible high-res graphics (Microtext mode 3). Also transferred directly from the BBC Micro, with minor adjustment for different screen ratio.

8. User defined characters. Illustrates BBC-compatible user-defined character set, plus ability to define your own characters.

9. Reading a Micrometer. A Microtext "classic", also written in BBC high-res; transferred to CBM with only minor editing. Illustrates totally incomprehensible Microtext - original had to be crunched to minimum possible module-space to fit in the very limited memory available on the BBC!

0. 64 Microtext graphics. A guide to 64-specific graphics and sound. Illustrates graphics modes 0 to 3, sprite facilities, animation, concurrent sprite movement and sound generation. Rather pretty.

Breaking out

Demos 1 to 9 can be interrupted by entering a question-mark '?' when the program is paused for user-input - this will cause a return to the main demo menu. This is a special application of a more general "user help" facility.

Alternatively, demos can be halted when paused for user-input by pressing the STOP key. This gives an "interrupt menu" - pressing 's' for 'stop' will return to Command mode.

Any module can be "crashed out" of at any stage by holding down RUN-STOP and pressing RESTORE - this gives an immediate return to Command mode.

Editing

Having loaded a module - eg by LOAD COM.STA or by breaking out of a demo as above - it is possible to step forwards or backwards frame by frame using cursor-up and cursor-down; home goes immediately to start of module, and clear goes immediately to the end. When on any particular frame, it is possible to toggle between Command and Edit modes using RUN-STOP. Editing uses a full-screen editor, with special keys as follows:

cursor keys - move cursor (not into area reserved for frame number)
return - move cursor to next line
home - move cursor to top line
clear - disabled - use ERASE from Command mode
del - character delete
inst - character insert
colours - select cursor colour
rvs on/off - reverse field
shift-C= - flip character set

f3 - delete line upwards
f4 - insert line
f5 - auto-repeat on/off
f6 - cursor colour override on/off
f7 - screen colour roll
f8 - border colour roll

ctrl-R - insert carriage return (line break)
ctrl-P - display cursor coordinates.

Commands

For a summary of important commands, type HELP in Command mode. For a disk catalog, use CAT. For a CBM DOS command, use >.

Other queries

Microtext does NOT turbo-load, so it WILL run with an IEEE cartridge.

The Microtext disk is NOT copy protected. Special arrangements can be made for "multi-unit" deals for institutional users. A run-time only system is also available at a low rate - contact Ariadne.

Microtext supports "extension commands", allowing a set of extra commands to be merged with the system for control of peripherals like special keyboards, video players, etc. However, as far as we know, it is NOT possible to Genlock the 64 without spending a lot of money. Ariadne are available for assistance creating extension commands for special applications.

Microtext will run with a CBM modem attached - the system is highly Compunet-compatible, with a view to remote-learning. Modules can be up/down loaded to/from Compunet as s-files; utilities also exist to convert between Compunet multi-frame format and Microtext modules. An extension command is available allowing high-res graphics screens with sprite animation created using Microtext to be exported in a compressed format, in which they can be displayed as Compunet high-resolution "action-pages".

Microtext expects to talk to an MPS801 or compatible printer on the serial IEEE, or to a 4022 or compatible printer accessed via an IEEE cartridge. Other printers - eg on the user port - can be supported by writing suitable extension commands.

Availability

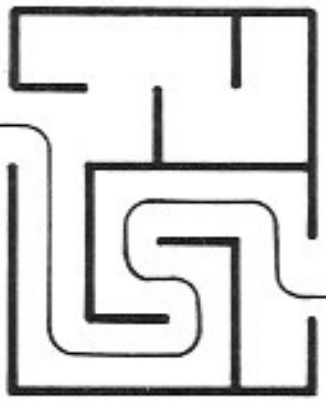
Packages consisting of the CBM Microtext authoring system, demonstration modules, and 120-page perfect-bound A4 manual are available (cheque with order) from Ariadne Software for #45.00 (incl VAT); these packages are available to dealers at #30.00 (incl VAT). Special deals for bulk orders, multi-user institutional orders, and run-time only systems are also available. Contact

Ariadne Software Ltd.
273 Kensal Road,
London W10 5DB.

Tel (01) 960 0203.

MICROTEXT

for the Commodore 64



MICROTEXT

for the Commodore 64

Original Microtext documentation edited by Nigel Bevan and Robert Watson, National Physical Laboratory.

64 version edited by David Parkinson, Ariadne Software Ltd.

Many people contributed to the original Microtext guide, including Philippa Bush, Steve Collins, Catherine Fear, Tony Mansfield, Dianne Murray, Len Rogers and Brian Walker.

Hanafi Houbart and Charles Langley contributed new material to the 64 version.

Contents

More advanced topics are marked with * and can be omitted on a first reading.

PART I - INTRODUCING MICROTEXT

1] What is Microtext?

Microtext is a powerful but simple to use system that enables interactive computer applications to be developed by people who need have no previous programming experience.

Microtext can be used to create almost any interactive application which presents text or diagrams to individuals or groups and prompts for a reply. It can be used both to obtain information from the user in a 'form-filling' or interviewing situation, and to give information to the user in an instructional context. The version for the Commodore 64 Microcomputer includes commands to support sophisticated sound and music as well as high-resolution graphics and Sprites.

Obtaining information

Microtext can be used for writing interactive questionnaires in interviewing and form-filling applications. The answers can take the form of multiple-choice responses, numeric data from a keypad, or plain text from the whole keyboard. The results of the interview can be shown in the form of a neat summary for each user.

Giving information

There are four main applications:

1 Education

Microtext enables teachers to write their own computer-based teaching material, and exchange lessons with other teachers who may be using different computers. Microtext lessons can assess the student's ability, and adapt the level of presentation to his or her requirements.

It is particularly appropriate for remedial teaching since it can be used to hold the student's attention and thoroughly explore his or her understanding of a topic. Microtext is also ideal for educational demonstrations for use in public displays. Information is presented to the user, and responses can be recorded for later analysis.

2 Training

Microtext allows the expert in a field to design and write his own training material. This may be purely descriptive, or it can contain simulations, or if appropriate instruct the pupil in the use of external equipment. The summaries and management facilities will assist in the evaluation of training packages, and material can easily be amended as

requirements change.

3 Information retrieval

Microtext incorporates the capabilities of a Viewdata information system, but can considerably reduce search times by branching on keywords rather than relying on numeric menus.

4 Expert advice

Microtext can guide a user through a complex external task, prompting for information when necessary, and giving expert advice. This technique can be used both for training with simulated problems, and as an expert assistant in real applications.

2] About this manual

This manual describes the implementation of Microtext on the Commodore 64 Microcomputer. It contains full details of the use of Microtext and assumes no previous experience of the system. It is, however, useful for you to have a basic familiarity with the Commodore 64 Microcomputer - for example switching on, loading disks or tapes, and the layout of the keyboard. You should, therefore spend some time with the system and the Commodore 64 Microcomputer User Guide if you have not used it before. The User Guide may also be useful if you are using any of the more specialised facilities of this implementation of Microtext, for example sound, or high resolution graphics.

To start with, work through chapters 3 and 4 of this manual, which take you step-by-step through loading procedures and the creation of a very simple Microtext program, and/or run the demonstration programs supplied with the system.

You should then read chapter 5, **Using Microtext**, which outlines the basic concepts of Microtext, and describes the structure of a Microtext program, and the facilities of the authoring system.

The subsequent chapters describe the detailed facilities of the system:

6 Text creation and editing describes how text is input to the system and subsequently edited. This section also discusses the the use of graphics and colour in standard low-resolution text mode (mode 0).

7 Response matching and branching gives full details of the powerful facilities of Microtext for testing user input and varying its responses accordingly.

8 Using variables describes the use of variables to store and manipulate text and numeric data.

9 Summary items covers the use of summaries, in which a report can be built up during a Microtext session recording, for example, the user's performance in tests.

10 Testing and running modules describes how programs are tested and run.

The remaining sections describe the special facilities built into CBM Microtext for graphics and sound:

11 Extended text modes explains how to make use of "multi-colour" and "extended background" text modes to produce a greater variety of displays.

12 User defined characters explains how to make your own graphics characters or complete character sets for special purposes like Greek characters or mathematical symbols.

13 High resolution graphics explains how to combine high resolution points, lines and triangles with the normal 64 display.

14 Animation and sprites covers the creation of large coloured graphic objects known as "sprites", and how to achieve movement and animation effects using them.

15 Sound explains how to use the 64s sound-synthesis chip to add a wide variety of sound effects to Microtext.

16 Miscellaneous commands covers how to control external devices using the User Port, and similar facilities.

Three appendices follow:

Appendix A Module Portability gives advice on the transfer of Microtext applications from one system to another.

Appendix B 64 Microtext command summary.

Appendix C Error messages.

Note that sections covering more advanced features are marked with '*' in the contents listing and in the text of the manual; these can be omitted on first reading.

3] Starting Microtext

3.1] Loading the Microtext system

First, to get the computer to accept Microtext commands it must understand the Microtext language. This is done by loading the Microtext System into the computer.

(If you make a typing error, you can correct it with the DELETE key. If you make a mistake and the computer won't respond, you can usually recover by pressing the STOP key, and then repeating the last command you gave.)

Switch on the computer and monitor (or TV receiver). The message:

```
**** COMMODORE 64 BASIC V2 ****

64K RAM SYSTEM 38911 BASIC BYTES FREE

READY.
```

or a similar message will appear on the screen, with a flashing reverse square (cursor) at the start of the next line.

The system will run from either tape or disk. Loading from tape or from the 1541 disk drive takes a while; however Microtext is fully IEEE-cartridge compatible, so users with such cartridges will have no difficulty using the faster drives available. The system has also been designed to operate with a Commodore modem attached to the machine, for convenient "remote learning".

To load Microtext off disk, put the disk in the drive, then type

```
LOAD "MICROTEXT",8
```

and press RETURN. The drives will turn for a few seconds, following which the machine will signal 'READY' - when this happens type

```
RUN
```

and press RETURN. The drives will turn again while the main system is loaded; eventually the Microtext title page will appear with the message PRESS ANY KEY. This will take you into "command mode", which is explained below.

To load from tape, type

```
LOAD "MICROTEXT"
```

and press RETURN. The system will prompt you to PRESS PLAY ON TAPE; the screen will then blank while Microtext is loaded. Eventually the screen will return with the 'READY' prompt displayed; type

RUN

and press RETURN to get the Microtext title page, then press any key to enter "command mode".

Command mode

Command Mode in Microtext is comparable to the 'READY' prompt in BASIC, and tells you the machine is ready to accept your command.

At the bottom of the screen is a coloured strip of two lines. The top of these is the 'status line' which tells you which mode you are in. Commands are typed on the bottom line.

If you are using cassette you will see that on the status line are the words:

Command Mode 0 TAPE

which confirm that the machine in use is a Cassette-based Commodore 64 Microcomputer. If you load something, the system will expect it to be loaded from cassette tape. Similarly, if it says:

Command Mode 0 DISK

it means that the system is expecting to load something from disk.

3.2] What next?

Once the initial display is on the screen, pressing any key will cause the system to enter Command Mode. Typing HELP lists the main commands available. The next chapter gives details of how to create your first program, or you could look at chapter 5 for an explanation of the facilities of the Microtext system.

Alternatively, you could run an introductory demonstration if you have not already done so; type

RUN WEL.COM

and follow the instructions given to you on the screen.

3.3] Stopping Microtext

When you have finished using Microtext, you may want to reset the machine and return to Commodore BASIC. You can do this by typing

EXIT

4] Creating your first program

This chapter gives step-by-step instructions on how to create a very simple Microtext program.

It introduces the basic concepts, and can be used by someone unfamiliar with computer programming.

4.1] What is a program?

A Microtext **program** is the complete package of Microtext commands and text that achieves the desired objective, i.e. a complete lesson for a student under instruction or a complete set of questions and replies constituting an interview.

A program is divided into one or more parts called **modules**. Each module consists of a number of **frames**.

A program could be likened to a book, a module to a chapter and a frame to an individual page.

To make a book comprising a number of chapters, the text has to be broken down into suitable pieces to fit the page. Similarly, each piece of program text is placed in a frame together with control information placed before and after it for the computer to interpret:

CONTROL INFORMATION

LINES OF TEXT

CONTROL INFORMATION

Frames are numbered in the range from 1 to 999 in ascending order. It is convenient to number in increments of five or ten to begin with - (e.g. 5 - 10 - 15 - 20 -) to allow room for insertions of frames later if needed, for instance to extend the original text.

4.2] Starting on paper - the flowchart

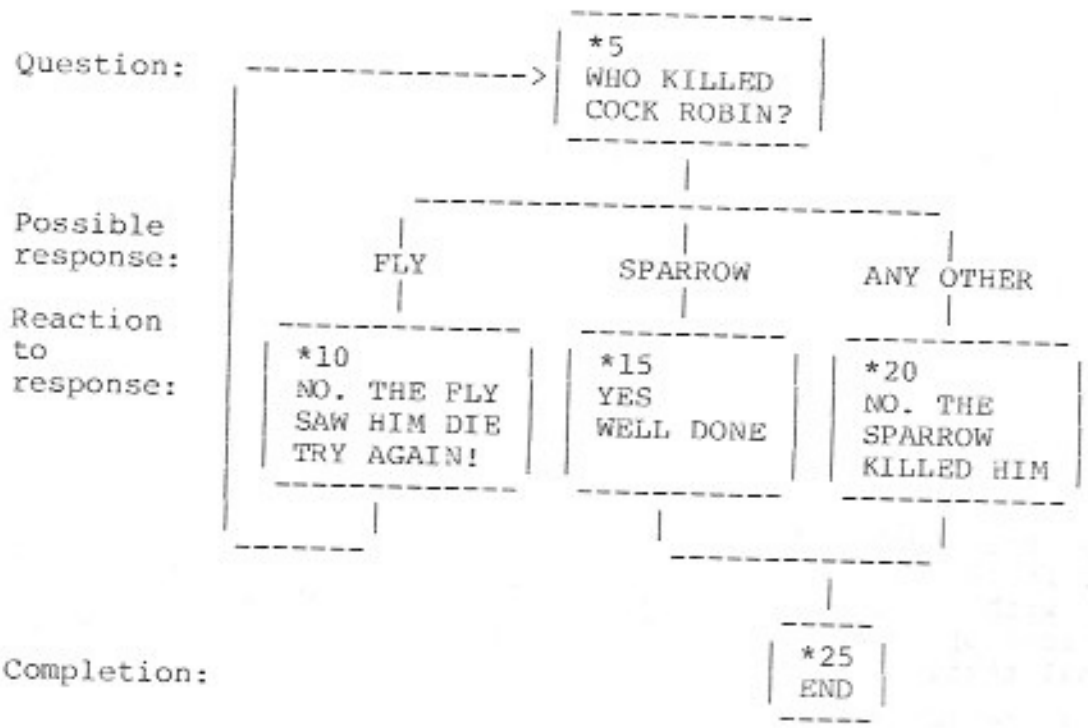
The initial design of a program should start on paper so that you can look at it and modify it to do exactly what is required.
This example will test knowledge of Cock Robin:

```
Who killed Cock Robin?
I, said the sparrow
With my bow and arrow
I killed Cock Robin.

Who saw him die?
I, said the fly
With my little eye
```

```
I saw him die.
..


All the birds of the air
Fell to sighing and sobbing
When they heard the bell toll
For poor Cock Robin.
```

The next few pages contain a straightforward example of a program based on the rhyme of Cock Robin. The program poses a question and expects more than one reply.

The easiest approach is to draw up a flowchart: first the question is asked, and then the replies determine what happens next:



This is a simple one-module program broken down into frames which are numbered in each box of the flowchart:

Frame 5 consists of the question and possible replies
Frame 10 shows the reaction to the reply FLY
Frame 15 shows the reaction to the reply SPARROW
Frame 20 shows the reaction to any other reply the user may give
Frame 25 shows that the program has come to an end.

The program can now be typed on the computer keyboard.

If you carry out the following sequence of operations step-by-step you will translate the flowchart into the program. It doesn't matter at this stage whether or not you understand all

the operations or the reasons for them: they will become clear once you have written the program one step at a time, and then examined what you have achieved.

4.3] Writing the program

1. Type FRAME 5 and then press RETURN (ie your command). Your frame number will now appear at the top of the screen as:

   *5

   This indicates that frame 5 has been created.

2. Press RUN/STOP and the bottom strip will show Edit Mode.

   The RUN/STOP key is used to switch between Command Mode and Edit Mode. In Edit Mode the machine is ready to accept text from you. The first line of the frame is reserved for the frame number and control information.

3. Type:

   WHO KILLED COCK ROBIN?

   (It doesn't matter whether upper or lower case is used.) The cursor is still on the second line.

4. Press RETURN and on the third line type:

   ?

   ? at the beginning of the line informs the computer that you expect a response from the user of more than one character. (If only one character is expected ! is used.)

5. Press RETURN and on the fourth line type:

   FLY=10

   This tells the computer that if the user replies FLY it should go to frame 10.

6. Press RETURN and type:

   SPARROW = 15

   (spaces before or after = are ignored).

7. Press RETURN and type:

   = 20

   This tells the computer that if the user replies anything other than FLY or SPARROW it should go to frame 20.

8. Press RUN/STOP. A row of dots will appear beneath your text t indicate the end of the first frame and the computer is now bac in Command Mode, ready for your next command. The screen shoul now have on it this display; which is your first frame.

DISPLAY ON SCREEN	WHAT IT MEANS
*5	- frame number
WHO KILLED COCK ROBIN?	- text
?	- pause for the user's response
FLY=10	- possible answers with
SPARROW=15	instructions to proceed to
=20	appropriate frame, ie
	branch addresses
.....	- delimiting dots

(If it doesn't, you have probably missed out one operation! To remove your first frame (but not the Microtext system) from the computer memory type: ERASE and press RETURN. You can then go back to step 1 and try again. Or you could try editing the frame - see Section 6.2.).

9. Type:

```
FRAME 10
```

and press RETURN followed by RUN/STOP to return to Edit Mode. The prompt cursor will again appear at the top of the screen under the *10.

10. Type:

```
NO. THE FLY SAW HIM DIE.
TRY AGAIN.
```

11. Press RETURN and type:

```
= 5
```

which tells the computer to return to frame 5 immediately after this frame.

12. Press RUN/STOP to end this frame with the delimiting dots and put the machine into Command Mode.

13. Type:

```
FRAME 15 <RETURN>
```

and press RUN/STOP to return to Edit Mode.

14. Type:

```
YES. WELL DONE!
```

```
PRESS SPACE BAR.
!
= 25
```

This will obtain a single character answer ('!' prompt) then branch to frame 25.

15. Press RUN/STOP to Command Mode and type:

```
FRAME 20 <RETURN>
```

16. Press RUN/STOP for Edit Mode then type:

```
NO. THE SPARROW KILLED HIM.
PRESS SPACE BAR.
!
= 25
```

17. Press RUN/STOP, and type:

```
FRAME 25 <RETURN>
```

The computer now has to be informed to end the program, so the END command is put in the final frame. Press RUN/STOP and type:

```
$END
```

18. Press RUN/STOP.

If you think you made any errors when you typed in the frames, you could look ahead to chapter 6 which explains how they can be edited.

4.4] Running the program

The program has now been written and entered in the computer's memory. If you have a printer, type:

```
PRINT 5-25
```

and then press RETURN; you should get the result shown below.

The program can now be put into action (or 'run') to see if it works! So, in Command Mode type:

```
RUN
```

and press RETURN. You can then try answering the Cock Robin question. At the end of the run press any key to go back to Command Mode.

The program can now be examined frame by frame by using the cursor up and down key (CRSR up-arrow and down-arrow). If you are satisfied, your first Microtext program has been created.

```
*5
WHO KILLED COCK ROBIN?
?
FLY   =10                          These are called BRANCH
SPARROW =15                        ADDRESSES - the frame to go
= 20                               to when a response is made.
.............................
*10
NO. THE FLY SAW HIM DIE
TRY AGAIN.
= 5
.............................
*15
YES. WELL DONE!
PRESS SPACE BAR.
!
= 25
.............................
*20
NO. THE SPARROW KILLED HIM.
PRESS SPACE BAR
!
= 25
.............................
```

This is the flowchart translated into Microtext.

To get a better understanding of each procedure, it is suggested that now you concoct your own flowchart with question and answers but within the structure of the Cock Robin example, and then proceed step-by-step with writing your own program, but using the same operating instructions.

Branch Addresses

The branch addresses above were typed on separate lines for clarity. They can alternatively be put on one line but each branch address must be followed by a comma to indicate to the computer the end of that particular address:

```
FLY =10, SPARROW =15, =20
```

The third branch address is called a DEFAULT BRANCH and responds to any answer other than FLY or SPARROW by going to frame 20. If the default branch were omitted from the program, the computer would give the message:

```
NOT UNDERSTOOD - TRY AGAIN
```

until either FLY or SPARROW was typed.

4.5] Saving the program

If your program is working properly it can be saved on tape or disk for replay as a memento of your first attempt.

1. Insert a blank tape, or a disk.

2. With the computer in Command Mode type

    SAVE ROB.STA

This saves the program with a name ROB.STA, as explained below.

If using **disk** the program will automatically be transferred.

If using **tape** the bottom of the screen will show the message

    PRESS RECORD & PLAY ON TAPE

As soon as you do this, the screen will blank while the computer saves the program onto the tape. Being a short program, it will not take long. When it has done this, the screen display will return, with the bottom line clear again. Press **Stop** on the cassette machine (This is to release the mechanism; the motor will have stopped of its own accord.)

3. To reload the program:

If using **disk**, in Command Mode type:

    LOAD ROB.STA

and then **press** RETURN. Your program will now load.

If using **tape**, **rewind** the tape, then in Command Mode type:

    LOAD ROB.STA

and **press** RETURN. The message 'PRESS PLAY ON TAPE' will appear. Press PLAY and wait; the screen will blank, then return when the computer has finished loading. Press Stop on the tape.

4.6] File names in Microtext

A 'filing system' - such as the 'DOS' used on Commodore disk drives - uses 'file names' to tell different programs apart. A useful Microtext convention is to split this file name into two parts separated by a full stop - a **PROGRAM-NAME** such as 'ROB' followed by a **MODULE-NAME** such as STA. On the Commodore 64 the whole file name (including the dot) can be up to 16 characters long; however, if you want to be compatible with other systems like the BBC Micro, both program and module names should be restricted to a maximum of three characters.

4.7] The HELP command

If you type HELP followed by RETURN you will see a list of the main commands available in Command Mode. You may find this useful to remind you of the facilities available.

PART II - A GUIDE TO MICROTEXT

5] Using Microtext

This chapter gives an overview of the basic concepts essential to an understanding of Microtext, and it is recommended that you read this chapter before going on to the rest of the manual or attempting a full implementation of a Microtext application. It would be useful if you had also read chapter 4, or worked through one of the demonstration programs to give yourself a basic familiarity with the system.

5.1] Organisation of material

Programs and modules

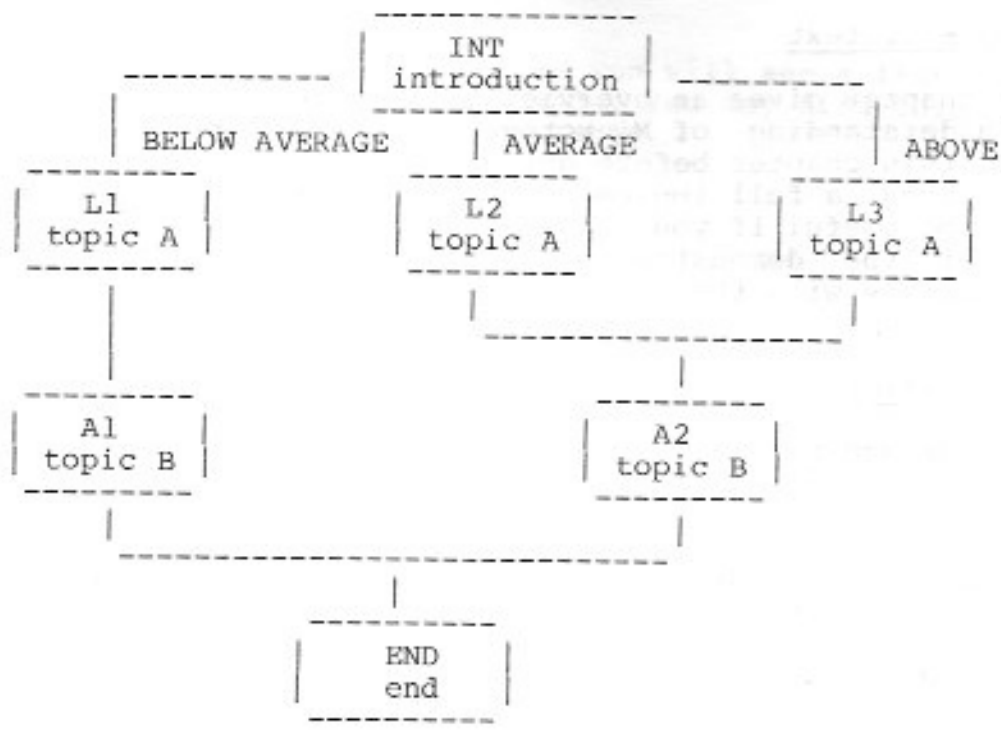
A program is a complete package of Microtext commands and text that achieves the desired objective, e.g. a complete lesson for a student under instruction, or a complete set of questions and replies constituting an interview.

A program is divided into one or more named parts called **modules**. A module is the unit in which text is stored on disk or tape and is the largest unit that is held in the computer's memory at any one time. Some guidance on the sizes of programs and modules is given in section 5.6.

This manual is written for the person producing the program, the **author**. The person for whom the program is written, whether student, interviewee, patient or whatever, is termed the **user**. The user may need some of the information in this manual. For example, they may need to load programs from disk or tape during a session. It is assumed that you, the author, will make available such information as the user needs, either via the screen or on paper. Some advice on this topic is given in section 10.7.

Let us now look at how these basic units of Microtext might be used in a potential application. The way in which the material is presented obviously varies a great deal depending on the topic, but we can look at a number of features likely to be common to any application involving the testing of the user's present state of knowledge and guiding him or her though the rest of the program efficiently on the basis of this test.

In the simple example outlined in Figure 5.1, we are providing tests and instruction for pupils on the subject of microcomputers. The responses made by the pupil will determine the order in which information is presented, and a report can be generated at the end. Two topics are covered: computer languages (A), and computer applications (B).



Firstly there is a common introductory module which provides t
user with guidance on how to proceed, asks for personal detai
and tests his or her knowledge of the subject. On the basis
the test, the user is classified as below average, average
above average in terms of knowledge of the subject. The auth
has then provided three treatments of computer languages, and t
on the topic of applications. Depending on the outcome of t
initial test, the user is presented with one of the thr
presentations on languages followed by an appropriate treatme
of computer applications. A final module, common to all user
could perform and print an analysis of each user's performance
the tests.

Module names

The modules are given names of up to sixteen characters.
distinction is made between upper and lower case. In the abo
example, the modules were:

INT Introduction and ability tests
L1 Language topic for below- average students
L2 Language topic for average students
L3 Language topic for above-average students
A1 Applications topic for below average students
A2 Applications topic for average and above-average studen
END Common end module

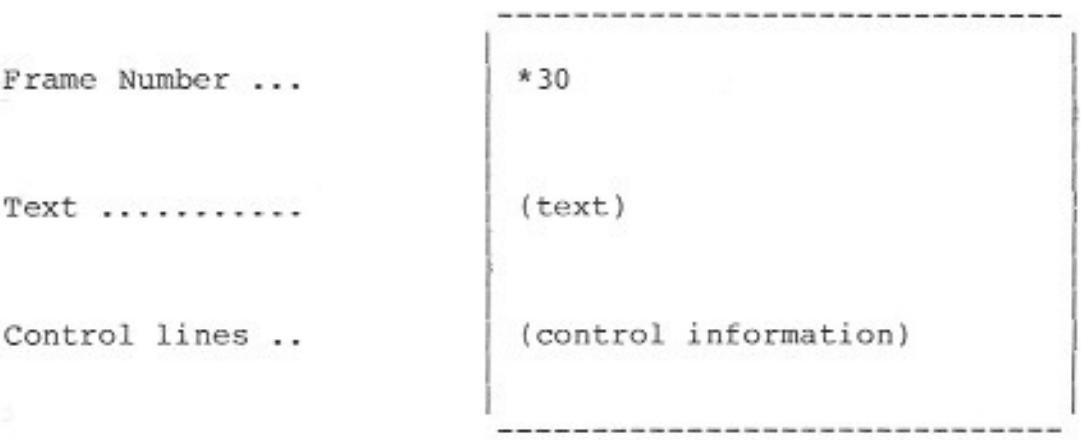
Since a single program may consist of a whole number of modul
like this, each stored seperately on disk or tape, it makes sen
to signify that they do belong to the same program in the nam

we give them to be stored under. A recommended Microtext convention is to use a common prefix followed by a dot, followed by the module name. Thus we could call the program on microcomputers MICRO, and the modules could be named MICRO.INT, MICRO.L1, MICRO.L2 and so on.

Frames

Each module in a program consists of a series of numbered frames. A frame is a unit of text that fits conveniently onto the computer screen. Take for instance the module MIC.A1 in our example. This would comprise a number of frames:

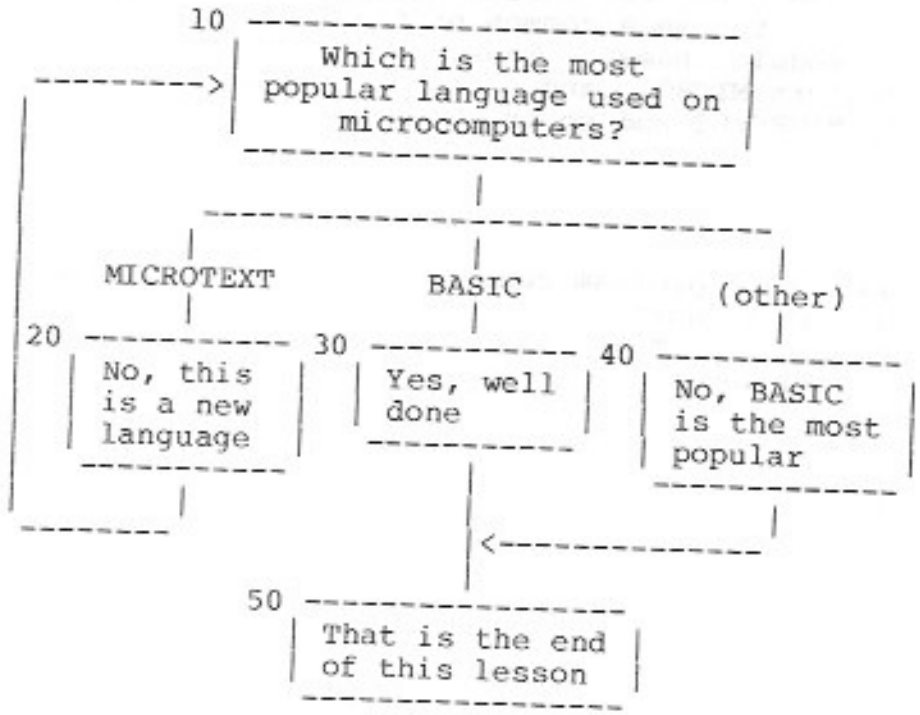
Each frame contains text and control information:



5.2] Building a module

Having looked at the overall structure of a Microtext program, let us now consider how a typical series of frames could be built up. In our example, part of the module on languages could test whether or not the user knows about BASIC.

In general, the program will provide both information and questions to the user. The author would normally structure the material into the form of a dialogue with the user on paper using a flowchart. The flowchart for the frames in this example is as follows:



The next step is to translate this into a Microtext program f the computer to interpret.

The example illustrated requires 5 frames. The first frame is question:

```
*10
Which is the most popular language
used on microcomputers?
```

The *10 indicates that this is frame number 10. Each box on t flowchart has a corresponding frame number. Other conventio that need to be known for this example are the use of t question mark, ?, for user input, and the equals sign, =, f 'goto'.

A question mark at the beginning of a line tells the computer t wait for a response from the user. The author should anticipat the answers the user might give, and list them with instructio as to where the computer should next go to in the program. Thi is done by adding an equals sign and the appropriate frame numbe to each listed response, eg MICROTEXT=20 means, if the answe 'MICROTEXT' is given, go to frame 20. More generally the equal sign is used whenever the computer must be told to move from on frame to another.

These rules are best illustrated by writing out the full sequenc of frames for this example. Each box on the flowchart is given number, and then the text in each box is written as a frame o Microtext:

```
*10                                    (frame number 1)
Which is the most popular
language used on micro-
computers?
?
MICROTEXT =20, BASIC =30               (response = frame)
= 40                                   (default branch)
.............................
*20
No, Microtext is a relatively new
language.

= 10                                   (go back to frame 1)
.............................
*30 (SCORE=5)                          (set the SCORE to 5)
Yes, well done.

= 50
.............................
*40 (SCORE=0)                          (set the SCORE to 0)
No, BASIC is the most popular.
It is not <ANS>.                       (ANS contains the user's)
                                       (last response)
= 50
 .............................
*50
Your score so far is <SCORE>           (contents of SCORE)
That is the end of the lesson.

= 100                                  frame)
**Score on languages
**is <SCORE>.                          (summary for report)
.............................
```

In this example, Microtext will display the text of the first frame, and then wait for the user to answer the question. If the answer is MICROTEXT it will branch to frame 2, if BASIC to frame 3 and set the score to 5, and if any other answer it branches by default to frame 4 and sets the score to 0. Frames 2, 3 and 4 all branch on to another frame without waiting for any further user input. The sequence eventually terminates with a branch to frame 10.

The example shows the use of a **variable** called SCORE which is used here to keep a numerical tally of the user's performance. After several dialogues of the kind illustrated the author could use this value to determine the user's level of knowledge and thus which series of modules he should follow through the remainder of the program.

The example also uses the **system variable** ANS in which Microtext automatically stores the last user response. The line starting with ** contains a **summary item** for inclusion in the final report.

Full details of all these and all the other Microtext facilitie are given in the following chapters.

5.3] Screen display

When Microtext is first loaded, the screen shows 23 blank lines followed by two lines at the bottom in "reverse field" containin status information. The top of the screen is used for text; th bottom two lines contain the **status line** and the **command line.**

The status line is removed when running a program, and in othe modes displays: the current system Mode (Command, Edit, or Test) the current CBM screen mode (initially 0); the current storag device (Tape, or Disk); the current program and module name and in certain circumstances, the current frame number.

The command line is used to type commands and display syste messages. When an error message is displayed, pressing any ke will cause the message to be cleared, and return to Command Mode. (The key pressed to clear the message will become the firs character of any subsequent command).

The top line of the screen is the header line, and includes th current frame number (which cannot be altered).

5.4] Microtext modes and system control

Microtext will, at any one time, be operating in one of four **Modes.** These are:

Command Mode
Edit Mode
Test Mode
Run Mode

Command Mode

Command Mode is used by the author to load and save modules, to step through the frames in a module, and to enter Edit Mode, Run Mode, or Test Mode. It is also possible to issue certain commands to the disk filing system (Commodore DOS) e.g. to delete files, and display the directory.

In this Mode a cursor is displayed on the command line, and commands are entered on the keyboard. When the command is complete RETURN is pressed to cause its execution.

The commands include those such as RUN and TEST which cause entry into the appropriate mode. Other commands such as LOAD cause some action after which the system is still in Command Mode. The commands may be abbreviated for convenience (see Appendix B). In addition to the commands, a number of keys have special

significance in Command Mode. For example the RUN/STOP key causes entry into Edit Mode, while the cursor keys are used to locate frames in a module as described in chapter 6. All the commands and function keys are described in the following chapters of this manual. The main commands can also be listed by typing HELP.

The Microtext system can also be controlled by commands embedded in the text of frames (see below).

Edit Mode

Once a frame of text has been created or located in Command Mode, Edit Mode is entered to allow new text to be typed in or existing text to be edited. Full details of this are given in chapter 6. A frame can be edited by pressing RUN/STOP in Command Mode. To leave Edit Mode and return to Command Mode (for example to move to another frame or to test the module) RUN/STOP is pressed again.

Test and Run Modes

Test Mode is entered by the TEST command which enables a user to test a program or module. The system returns to Command Mode when the test run terminates. The run can also be interrupted by pressing RUN/STOP in response to an input request. Full details are given in chapter 10.

Run Mode is entered by he RUN command, which enables a user to run programs. It is similar to Test Mode, but the status line is removed, and any recoverable errors are ignored.

Run-time commands

Certain commands in the text can be used to control Microtext in Run and Test Modes. These include commands which control the Microtext environment, the use of the screen (including any special graphics), and other optional hardware. These commands start with a $, and are placed at the beginning of a line, or appear on the header line of a frame (if there is no assignment), e.g.:

```
*10 $JOIN
```

They can also be associated with a particular answer (see section 8.2)

Commands at other positions are treated as text.

The commands can also be typed in from the interrupt menu, primarily for debugging purposes (see chapter 10).

Commands can be spelled out in full, or shortened (the minimum

abbreviations are shown in Appendix B). Any mixture of upper an lower case can be used.

Invalid commands display as text. In Test Mode a warning messag is given; in Run mode the command is simply ignored.

CBM DOS commands (see section 10.5) can also be embedded in th text. These are preceded by a '>' character, for example:

```
$>S0:MODULE1
```

5.5] Storage of modules in files

As described earlier, each module in a Microtext program i stored as a separate tape or disk file with a name limited t sixteen characters. It is a Microtext convention to use the for PROGRAM-NAME.MODULE-NAME, for example 'MICROTEXT.START', o 'MIC.STA'.

Naming modules

A module can be named in one of three ways:

1. Using the NAME command, which has the form:

   ```
   NAME filename
   ```

   This is used to give a name to the module currently being created. The name is displayed on the status line.

2. Using the NEW command, which enables a new module to be created with the filename given, deleting the current module from memory:

   ```
   NEW filename
   ```

   Confirmation is requested if the current module has unsaved alterations. (NEW with no filename will both clear any filename display from the status line, and delete the current module from memory. Modules are deleted from disk by the CBM DOS >S0: command - see section 10.5.)

3. Using the SAVE command to store the module on tape or disk:

   ```
   SAVE file-name
   ```

   This saves the specified module on tape or disk (see section 10.3).

5.6] Size limitations

Frame size

In Microtext, the bottom two lines of the screen are used as status lines. This leaves an available text area of 23 lines, each of 40 characters.

The number of lines on a frame is limited by what can be edited on one screen. However one screenful to be displayed to the user can be split across two (or more) frames by terminating the text of the first frame with delimiting dots. The next frame can contain more text, or just prompt and branch instructions. This is similar to unconditional branching.

Module size

The maximum size of a module will depend on the memory available. In 64 Microtext, the memory available is currently always 24K, independent of the filing-system or screen-mode in use. In general each module should be a relatively small self- contained unit. A typical size would probably be 15-30 frames. (Note that since multiple characters are compacted, text can be indented with only a small effect on memory usage).

The FREE command can be used to determine how much memory remains (see section 6.2).

Program size

As a guide consider a program with an average frame containing 9 lines of text averaging 30 characters, so that with control information each frame occupies 330 characters (called bytes). With an average of 15 frames per module, each module will occupy 5000 bytes. This gives a measure of the amount of memory required when the module is loaded, and the amount of space required to store the module on disk or tape. If the program was stored on a floppy disk with a capacity of 200K bytes, then there would be room on the disk for about 40 modules.

5.7] Programming style

There are many ways of using Microtext from the very simple to the very complex. The remainder of this manual describes the wide range of features available, but whenever possible avoid using advanced facilities unless they simplify the application. In particular, try to design a simple branching structure. You will find this easier to do if you plan out the structure of the material in advance.

The advantages of writing simple material are:

1. You can concentrate your attention on the content and layout of the text, rather than being distracted by clever programming techniques.

2. You are less likely to make errors.

3. It is much easier if you need to make any changes later.

4. The material can be understood by other people who need not be programmers, and it is much easier if anyone else has to make changes.

Having initially produced simple material, you may subsequently decide to tailor it to special requirements. When you do this, try to introduce the specialised features in a consistent way, without unnecessarily increasing the complexity of the material. With careful planning and judicious use of variables and subroutines, you can often produce sophisticated results based on simply structured routines.

5.8] Exiting Microtext

To exit Microtext and return to BASIC, use the command mode command

```
EXIT
```

or the equivalent command

```
RESET
```

The screen will scramble breifly while the machine resets, following which you should get the familiar "COMMODORE BASIC" start-up message.

It is also possible to reset the machine and return to BASIC from Run mode, using

```
$EXIT
```

6] Text creation and editing

This chapter is concerned with the facilities for displaying text on the screen in Microtext. It explains text is typed in, and how it can be edited subsequently. Other features such as testing user input and the use of variables are covered in the following chapters.

6.1] Frame handling

This section describes how frames are created, located, and otherwise manipulated in Command Mode.

We are concerned with frames which cause text to be displayed, the user to be prompted for input, and so on. There are also non-executable frames called comment frames which are dealt with later in this section.

The main facilities for frame handling at this level are:

1. Creating new, empty frames
2. Locating existing frames for editing
3. Copying and erasing frames

Note: The current frame referred to in this chapter is the one displayed on the screen.

Frame creation

An empty frame is created using the FRAME command, which has the form:

```
FRAME n
```

where n is the required frame number. Frames are given numbers between 1 and 999 and are stored and executed in ascending order. When creating a new module, it is convenient to number frames in increments of 5 or 10, as this allows room for frames to be inserted later if needed.

Locating existing frames

To go directly to a frame whose number is known, simply use the FRAME command. For example

```
FRAME 100
```

will display frame 100. Should you give the number of a non-existent frame, an empty one will be created.

You can also move around the frames in a module using cursor keys as follows:

Cursor down steps forward to the next frame

Cursor up steps back to the previous frame

Home takes you to the first frame in the module

Clear takes you to the last frame

Frame number errors

If you omit the frame number in a FRAME command, or give one outside the range 1 to 999, you will get the error message

NO VALID FRAME NUMBER

An attempt to extend a module beyond the memory space available gives the error message

NO ROOM IN MEMORY

Guidance on the maximum sizes of frames and modules is given in section 5.6 above.

Copying frames

It is possible to copy one frame to another, new, frame using the COPY command, which has the form

COPY n

The current frame is copied to frame n. If frame n already exists, it will not be overwritten, but the error

FRAME EXISTS

is given and no copying will take place.

Erasing frames

A frame or series of frames can be erased from a module by use of the ERASE command.

Given without parameters, the command will erase the current frame. If there is nothing on the screen, the error

NO MODULE PRESENT

will occur.

You can erase other than the current frame by giving the frame number, for example:

```
ERASE 25
```

To erase a sequence of consecutive frames you give the numbers of the first and last in the range, separated by a hyphen or a comma. For example:

```
ERASE 10-55
ERASE 200,250
```

After the ERASE command there may be a short pause, and then the following frame (or the last frame of the module) will be displayed.

Comments

You may find it useful to insert comments at various points in a module for documentary purposes. A comment is a piece of text that can be displayed in command mode, but will not be presented to the user when the frame is executed.

Comments can be included in the text of a frame, prefaced by a comment command:

```
$COMMENT This frame is for advanced students
```

The comment will be ignored by Microtext when the frame is executed.

Alternatively, a whole frame can be designated a comment frame by creating it using the COMMENT command rather than FRAME, eg:

```
COMMENT 110
```

A comment frame is identified by having 'C' rather than '*' in front of its number, eg:

```
C110

The following set of frames are intended
for use by advanced students:
.....................................
```

During execution of the module a comment frame will be skipped over and the next frame in sequence executed. An ordinary frame can be converted into a comment frame and vice-versa. This can be useful during module testing.

6.2 Text input and editing

Once a frame has been created or located, Edit Mode is entered
pressing RUN/STOP. Material can then be typed in or edited. T
contents of a frame include text for presentation to the use
and instructions for branching to other frames.

This section describes the basic concepts of text creation a
presentation including relevant run-time and control commands.

Text on the Commodore 64

Text entry on the Commodore 64 is usually performed in "graphic
mode 0". This is a fast, convenient general-purpose mode, givi
sixteen selectable colours for text, border and screen, and
choice of two character sets, including characters which can b
used for simple graphics. To change character sets, enter Edi
mode, then hold down SHIFT and press the Commodore 'C=' key
Note that the case of the characters on the status and comman
lines change to show you which character set is selected. Th
character sets are as follows:

upper case/graphics

This is the Microtext default, in which only upper cas
characters are available and are produced without pressin
the shift key. Both SHIFT (right-hand symbols) and 'C=
(left-hand symbols) can be used to select the full range o
graphics characters. In this mode, the status and comman
lines are displayed in upper case.

lower/upper case characters

In this mode the keyboard behaves very like an ordinary
typewriter, and the status and command lines are displayed
in lower case. Only about half of the graphics characters
shown on the keys are available; these are generally the
left of the two graphics symbols shown on each key, and are
obtained by holding down the Commodore key while pressing
the key in question.

Note that in 64 Microtext, any character can be redefined,
allowing you to create any character set you want. This is
described in Chapter 12.

Cursor movement

The format of the screen is fixed, meaning that it is impossible
to type or move the cursor into the command area at the bottom,
off the top, or into the area reserved for the frame number at
the top left of the screen. Typing and cursor keys automatically
wrap round from the end of one line to the beginning of the next.

The position of the cursor on the screen shows where text will start to appear when you type. The following keys control the movement of the cursor.

Key	Effect
CURSOR	The cursor up/down and cursor left/right keys move the cursor around inside the allowed area, with wrap-around between lines.
RETURN	Moves the cursor to the start of the next line (except if already on the bottom line).
HOME	Moves the cursor the right of the area reserved for the frame number at the top of the screen.

Note that the CLR (clear) key is deliberately disabled in Microtext. To get rid of a frame, exit to Command Mode by pressing RUN/STOP, then ERASE the frame.

Text editing

A number of special keys provide facilities for editing the contents of a frame. These are as follows.

Key	Effect
DEL	Deletes the character to the left of the cursor and closes up the line. Repeated deletions stop when the cursor hits the far left of the screen.
INS	Creates space at the cursor position and moves the rest of the line one space to the right. There is no wrap-around and characters will be pushed off the right-hand edge of the screen.
F4	Inserts a blank line before the current line and moves the other lines on line down the screen. Control-I has the same effect.
CTRL-R	Inserts carriage return: moves cursor and characters to right of cursor to the start of the line below, and moves subsequent lines one line down the screen.
CTRL-D	Deletes the current line and closes up the text below.
F3	Deletes up the screen, moving the current and subsequent lines one line up the screen. Control-U has the same effect.
F5	Key repeat on/off. Press F5 to make all keys start repeating after they have been held down for a short time; press F5 again to stop this.

Text creation and editing

F6 Colour override on/off. Press F6 to switch colou override on - this makes it possible to change text o graphics to the current cursor colour by moving th cursor over the characters in question. Press F6 agai to switch this facility off.

F7 Screen colour change. Press F7 to "roll" the screer through the 16 available screen colours. Note that colours selected this way are NOT remembered permanently for use at run time - to do this use the $SCREEN command.

F8 Border colour change. Press F8 to "roll" the border colours in the same way. This setting is also temporary - run-time border colour is controlled using the $BORDER command.

CTRL-col Select primary text colour. Hold down control and press one of the number keys to select current cursor colour.

C= -col Select secondary text colour. Hold down the Commodore logo key and press one of the number keys to select one of the "secondary" text colours; this gives a choice of sixteen text colours in all.

CTRL-RVS Hold down control and press '8' to switch on printing in reverse field. Use control and '9' to switch reverse off.

shift-C= Hold down shift and press the Commodore logo key to change a frame between the default upper case and graphics mode, and the "alternative character set" which gives lower and upper case characters.

CTRL-P Cursor position report. Gives the current cursor line and column position on the command line.

Note: The control keys are operated by holding down CTRL while pressing the key in question.

Returning to Command Mode

When you have finished inputting or editing a frame, you normally return to Command Mode by pressing the RUN/STOP key.

If you have made a mistake such as unintentionally deleting or corrupting text, you can normally abandon the edit and return to Command Mode by holding down shift and pressing RUN/STOP. This will restore the frame as it was before editing.

Free space in memory

When modules are created using the editor the frames are stored in memory. If a module fills all available memory, the NO ROOM IN MEMORY message will be displayed. If this happens, it will be necessary to divide the material into two different modules.

The free space in memory can be checked with the FREE command:

```
FREE
```

This displays the number of free characters (bytes) in memory which can be used for text. Pressing any key returns to Command Mode.

Note that the free space will depend on:

1. The version of Microtext.
2. The size of the module (ie the number of characters).
3. The number and contents of any variables generated when running a module (see chapter 8).
4. The size of any summary generated when running a module (see chapter 9).

When creating or editing a module, you should leave enough free space for any variables and summary used when the module is run. However, with the current version of 64 Microtext, 24K of space is available to the user under all circumstances; this means that it should NOT be necessary to use special techniques to save space.

6.3] Control of presentation with run-time commands

A number of run-time commands are available to control the way in which text is presented to the user on the screen. These commands are most often placed on the header line of the frame, or at the beginning of a line at the required point in the text.

Page and Scroll Modes

There are two main modes of presentation: Page Mode and Scroll Mode.

Page Mode is the default (i.e. the mode assumed by Microtext in the absence of instructions to the contrary.) In Page Mode the screen is cleared after the user makes a response to a prompt and before a branch is made to the next frame.

This default state can be changed by various run-time commands.

$JOIN in a frame causes it to be joined to the next frame on the screen. Frames are automatically joined when they do not end with a prompt for user input. Any text displayed by the frame will remain after the branch to the next frame. Any text displayed by the next frame will thus appear below the existing display. If the text reaches the bottom of the text area as a result of joining the frames, the display will scroll up.

$CLEAR at any point in a frame will clear the screen completely before the next line is displayed.

To change from Page Mode to Scroll Mode the $SCROLL command is used. After this command, all subsequent frames are appended as if each contained a $JOIN command. The effect is like the rolling of credits at the end of a film.

To revert to Page Mode, insert the $SCROLL OFF command at the required point.

Note: It would be useful at this stage for you to load a module and edit it to try the effects of these commands. Remember to put them at the beginning of a line, or on the header line of a frame.

Fixing text on the screen *

Text can be fixed on the screen. The text must first be displayed. The $FIX command is then used to fix all or part of this text. The command takes two forms:

```
$FIX      Fixes all the existing text

$FIX n    Fixes the top n lines of the text
```

This creates a window on the screen below the fixed text. This window is then treated in exactly the same way as the entire screen is in the absence of any fixed text. That is, further text may be displayed in Page or Scroll Mode and $JOIN and $CLEAR commands may be used without affecting the fixed text area.

Fixing text in this way is particularly useful for form-filling and this is described in section 7.5. Some examples of using $FIX are given in that section.

To clear the fixed text and revert to the state before $FIX was given, use

```
$UNFIX or $FIX 0
```

Setting a margin *

In CBM Microtext, it is also possible to fix the left of the screen. This is done by setting a margin using the $MARGIN command:

```
$MARGIN n       Sets a margin of n lines at the left
```

For example, the command $MARGIN 5 will fix the first five characters of each screen line, and cause all subsequent text to be output indented by 5 characters. If you do this, make sure that subsequent lines are not too long (over 35 characters in this case) or they will "wrap" onto the next line on the screen.

Margin is set back to normal by $MARGIN 0, or just

```
$MARGIN         Resets margin to far left of screen
```

Obviously, you cannot set the margin less than 0 or greater than the screen width - if you try to, Microtext will give the fatal error

```
Bad margin
```

Text positioning *

You can specify the position at which the next line of text is to appear using the $LINE command. This can also specify a column within a line.

The forms of this command are:

```
$LINE line
```

or `$LINE line, column`

or `$LINE @number`

The effect is to move the cursor to the point specified. For example:

```
$LINE 11        specifies the beginning of line 11

$LINE 5, 15     specifies column 15 of line 5

$LINE @3        specifies form position 3 (see chapter 9)
```

Note that the top line of the screen is line 1 and the extreme left of a line is column 1.

Subsequent lines of text start at the specified position. This facility can be used to skip several lines forward or to move back and overwrite part of the screen, including text that has been fixed.

To determine the line and column numbers to use in a $LIN command, use the CTRL-P facility explained above. To calculat the line number for $LINE, subtract from the line number given b CTRL-P the number of control and command lines above it on th screen.

Speed of display *

The timing of the display of text can be controlled using th $PAUSE command. This has three forms.

The simplest form is

```
$PAUSE t
```

which causes an immediate pause in the execution of the module o t tenths of a second.

This can be useful, for example, to give a paced display of a series of frames, or to give the user time to think before the next question.

The form

```
$PAUSE INPUT t
```

can be used to create a pause of t tenths of a second after keyboard input (eg after the user has pressed RETURN). This prevents the user's answer disappearing immediately.

The speed at which text is displayed can also be controlled by the command in the form:

```
$PAUSE CHAR h
```

This causes a pause between the display of each character of h hundredths of a second. A display speed of between 10 and 15 characters per second can be very effective in holding the user's attention as material is presented.

The $PAUSE INPUT t and $PAUSE CHAR h commands remain effective until reset by $PAUSE INPUT 0 and $PAUSE CHAR 0.

6.4] CBM screen control

Mode 0 graphics

Four text and graphics modes are available in 64 Microte t, giving a very wide variety of possible screen displays. Modes 0 to 2 are "text modes" and can be used in Edit mode as well as Run and Test modes; mode 3 is for high-resolution graphics, and can be used in Run and Test modes only. Mode 0 is the system default and will be used by the majority of applications; for information about the other modes, see Chapters 11 and 13.

Colour of display

There are 16 text colours available on the 64, which can be grouped into 8 primary colours selected using CTRL and a number key, and 8 secondary colours, selected using the Commodore key and a number key. Any colour can be used for $ commands, but colour should not be changed in the middle of a command line. It is a good idea to use a standard colour for all command lines in a module - this helps them stand out from the surrounding text.

A number of special commands have been added to Microtext to control colour and graphics on the Commodore 64. Note that these are **not** part of standard or "core" Microtext, so modules that use them will probably need some adaptation to run on other computers. A number of these commands use "colour numbers" to select colours; these are as follows:

	primary		secondary
0	black	8	orange
1	white	9	brown
2	red	10	pink
3	cyan	11	dark grey
4	purple	12	middle grey
5	green	13	light green
6	blue	14	light blue
7	yellow	15	light grey

BORDER

In command mode, border colour can be selected by

```
BORDER n
```

where n is "colour number" (see above). Similarly, in Run or Test mode, border colour can be changed by

```
$BORDER n
```

In edit mode, it is possible to experiment with border colour using the F8 key to "roll" the colour.

SCREEN COLOUR

Screen colour can be selected in a similar way:

Command mode:	SCREEN n	(0 to 15)
Run or test mode:	$SCREEN n	(0 to 15)
Edit mode:	F7	

This is the simplest form of the SCREEN command - see Chapter 11 for the full version.

6.5] Printing frames

Frames can be listed to a CBM MPS801 (or equivalent) printer, using the PRINT command; printing can be abandoned by pressing RUN/STOP.

Specific frames or ranges of frames can be specified as follows:

Command	Frames listed
PRINT n	Frame n
PRINT n-m	Frames n to m
PRINT n-	Frame n and subsequent frames
PRINT -m	Frames up to and including m
PRINT	All frames

Note: A comma can be used in place of a hyphen; for example

PRINT 10-100 or PRINT 10,100

7] Response matching and branching

A frame may simply display text, or it may request input from the user. At the end of a frame a branch will usually be made to another frame. This branch may be **unconditional** - in other words it will always operate - or it may be **conditional** in which case it will operate under certain (specified) conditions. It is here that the power of Microtext lies, with its wide range of facilities for selecting the next frame to be executed on the basis of the user's response. Most of this chapter is concerned with these facilities, and chapter 8 describes the use of variables which can further extend the range of conditional facilities.

The prompt for user input is a line starting with '!' or '?'.

The '!' prompt normally specifies a single-character response, although it can also specify a fixed-format response of a set length - see below.

The '?' prompt specifies a **free-format** response of a word or phrase. The user is prompted by a line of dots and the response can be of any length that will fit on the line. The response can be corrected using the DELETE key and is terminated by pressing RETURN.

The line following the prompt should normally contain instructions to select a branch depending on the user's answer. The user answer is compared with the branch instructions until a match is found.

The instructions will normally be terminated with a **default branch** (e.g. =50 below). Microtext will branch to this frame if none of the other tests match the user input. If there is no default branch, an invalid answer from the user produces the error message:

```
Not understood - try again
```

The instructions can be continued on subsequent lines if required. This is particularly useful if long responses are being checked, and generally makes the text easier to read. For example:

```
*5
WHO KILLED COCK ROBIN?
?
FLY =10
SPARROW =20
=50
.....................................
```

Response matching and branching

7.1] Single character input

Some questions only need a single character answer, for examp
multiple choice or Yes/No answers. This is indicated by using
'!' for input:

```
*10
Which is the capital of the USA?
A: New York B: Washington C: Boston
!
A=20, B=30, C=40, =50
.....................................
```

Microtext prompts with a single dot and will branch as soon
one character has been typed. The RETURN key need not be press
and is, in fact, ignored.

If the **only** branch is a default, the branch will be tak
irrespective of the user's response:

```
*10
Press the Space Bar to continue
!
=20
.....................................
```

This technique gives a convenient way of letting the user pa
through a series of frames at his or her own pace by pressing
single key.

7.2 Keyword matching

Microtext normally searches the line typed by the user for
match with the specified keywords. For example:

```
*5
Do you live in London?
?
YES=2, NO=3, =4
.....................................
```

This will search the user's response for 'YES' or 'NO'.

Microtext will search for exactly what you specify, although t
case of the input is not usually significant (see below).

Microtext will normally search what the user types for t
specified keyword as a sequence of characters at any positio
The user's response will be checked for the occurrence of ea
keyword in turn. The first one to match will cause a branch.
there is no match, the default branch will be taken, if this
included.

To use a rather artificial case, if the user responded in t

above example with

```
NO I LIVE IN HAYES
```

Microtext would branch to frame 2 since the characters 'YES' would be the first keyword to match.

The analysis of user responses by Microtext can be extended or restricted in a number of ways to overcome this type of problem:

1. ignore blanks in user input
2. match keyword only with complete words
3. keyword must start a word typed by the user
4. keyword must end a word typed by user
5. keyword must be exactly what the user types
6. search for several keywords in any order
7. search for several keywords in specified order
8. test for numeric range

Upper and lower case input

Microtext does not usually take account of case when matching the user response against keywords, so it does not matter whether the response is typed in upper or lower case (or a mixture).

To make case significant when checking input the $CASE command is used. Then, for example

```
*5 $CASE
?
London=10
.....................................
```

will branch for 'London' but not for 'london' or 'LONDON'.

The effect of $CASE is cancelled by $CASE OFF.

Ignoring layout of response

Blanks are normally ignored when matching for keywords. For example, the question:

```
DO YOU WANT TO WORK FULL OR PART TIME?
```

might be answered

```
FULLTIME
```
or
```
FULL TIME
```

To accommodate such cases, the keywords are listed separated by blanks where it is anticipated they might be used. For example,

```
MICRO COMPUTER = 100
```

would accept a response of:

MICRO COMPUTER or MICROCOMPUTER

and similarly:

12 - 36 = 100

would accept:

12 - 36 or 12 -36 or 12-36

Matched words *

In the above example, the words MICRO and COMPUTER could appear anywhere in the response line provided that they are in the right order, so that

PSEUDOMICRO COMPUTERISED

would also be accepted.

In order to restrict the acceptable response the word or words should be delimited by a pound sign '[POUND]' to indicate the beginning and end. This causes other whole words and prefixes or suffixes to be rejected:

[POUND]MICRO COMPUTER[POUND] = 100

This would still accept 'MICRO COMPUTER' and 'MICROCOMPUTER', but would exclude 'PSEUDOMICRO COMPUTERISED'.

It is often convenient to check for the first few letters of long words, as this minimises problems with different endings and spellings. Thus if you wanted the program to accept 'COMPUTOR' or 'COMPUTER' as valid answers, the appropriate keyword would be:

[POUND]COMPUT = 120

which would accept any word starting with 'COMPUT'.

Similarly, you can check the endings of words by following the keyword with a pound sign. Thus you could check for any word ending in 'ise' rather than 'ize' with:

ISE[POUND] = 100

Note: When checking for the end of a word using [POUND], subsequent characters that are not letters or numbers will be ignored. Any keyword after must, therefore, start with a letter or number.

Exact matches *

If only an exact answer is acceptable, the text should be enclosed in single quotes:

```
Yes or No?
?
'YES'=2, 'NO'=3, =4
```

In this case 'YES PLEASE' would cause a branch to frame 4. Note that exact matches should be used as shown with no blanks or other characters following them. The text of an exact answer cannot contain a single quote character.

Special characters *

Double quotes allow the inclusion of special characters which might otherwise be ambiguous:

```
, = + & / < > [BACK ARROW] * ' ( ) [POUND]
```

For example:

```
"1 + 2" = 20 checks for answers containing '1 + 2', and

1 "+" 2 = 20 accepts answers including '1 + 2' or '1+2'.
```

Special characters can also be included in single quotes, when an exact test is used:

```
'1 + 2'= 20 checks for input '1 + 2'.
```

7.3] Logical tests

It is possible to test the user's input for a combination of 2 or more words or phrases in logical combination, using '/' for 'OR', '&' or '+' for 'AND', and '[BACK ARROW]' for 'NOT'.

The difference between the two forms of 'AND' is that '&' does not check for the order of words whilst '+' does.

An example of the use of '/' is:

```
*10
What is wrong with the wire in the plug?
?
CUT/SNAP/BROKEN = 20
.....................................
```

This means CUT or SNAP or BROKEN, and has the same effect as:

```
CUT=20, SNAP=20, BROKEN=20
```

Combinations of keywords can be checked in any order using '&', for example

```
*10
Who went up the hill to fetch a pail of
water?
?
JACK & JILL=20
JACK=30,
JILL=40
=25
.....................................
```

In this case there will be a branch to 20 if the answer contains both JACK and JILL (in any order).

The '/' operator is executed after the '+' or '&'. Brackets can be used to determine the order of execution (and can be nested):

```
*10
Name 2 cities, one starting with L and
one starting with B ?
(LONDON/LIVERPOOL)&(BRISTOL/BIRMINGHAM)=20
=30
.....................................
```

This is equivalent to:

```
LONDON&BRISTOL=20
LIVERPOOL&BRISTOL=20
LONDON&BIRMINGHAM=20
LIVERPOOL&BIRMINGHAM=20
```

Brackets can be used in this way to group combinations of keywords which can then be used in conjunction with one of the following operators: [BACK ARROW], +, &, /. The normal order of priority of execution is [BACK ARROW] first, then + or &, then /.

The NOT operator, [BACK ARROW], can be used to exclude particular words, and is executed first. It is useful to exclude negative answers, for example

```
What is the capital?
?
LONDON& [BACK ARROW]NO=20,=30
```

This would exclude answers containing a word starting with 'NO', such as 'THE ANSWER IS NOT LONDON'.

Note: Where a complex test is made more than once, it may be convenient to store it in a variable (see chapter 8).

Ordered tests *

There are several ways to check for keywords in order:

```
Which is the largest city in the United
States?
?
'NEW YORK'= 50
```

checks for the exact answer 'NEW YORK', while:

```
NEW YORK= 50
```

checks for the letters 'NEW' followed by optional spaces followed by 'YORK', and:

```
[POUND]NEW[POUND] [POUND]YORK[POUND] = 50
```

checks for the word NEW followed by at least one delimiter, followed by the word YORK. To check for an answer with the word 'NEW' followed by 'YORK' without necessarily being adjacent, the '+' operator is used:

```
[POUND]NEW[POUND] + [POUND]YORK[POUND] = 50
```

This would accept 'THIS IS NEW IN YORK'. The '&' operator does not check for order:

```
[POUND]NEW[POUND] & [POUND]YORK[POUND] = 50
```

and would also accept the answer 'YORK IS A NEW TOWN'.

As '+' means 'followed by' it can also be interpreted as a 'wild card' representing arbitrary characters, thus

```
[POUND]RUM+SKIN[POUND]
```

would allow for odd spellings of RUMPLESTILTSKIN.

7.4] Numeric tests *

In addition to checking for particular words, tests can be made for answers in a numeric range. A numeric answer is required if the prompt for input is followed by '=':

```
What is 2+2?
?=
<4 = 30, 4 = 40, >4 = 50
```

The '?=' specifies that a numeric response is required, and allows tests of numeric range to be made. Thus an answer of '3' would branch to frame 30, and '4' to frame 40. When Microtext is waiting for a ?= or != input, only the numeric keys are enabled on the keyboard.

Response matching and branching

The following types of test can be made:

<2	less than 2
[BACK ARROW]>2	not greater than 2 (i.e. less than or equal to 2)
2	equal to 2
[BACK ARROW] 2	not equal to 2
[BACK ARROW]<2	not less than 2 (i.e. greater than or equal to 2)
>2	greater than 2

Negative answers can also be used. Note that "level 2" Microtext is restricted to the use of integers only.

Do not confuse numeric and text prompts, for example,

```
?=
<4 = 30
```

will check for a numeric answer which is less than 4, but:

```
?
<4 = 30
```

will match with an answer containing the text '<4'.

7.5] Form filling *

Microtext normally prompts for the user's response below the text on the screen. Responses can, however, be prompted anywhere on the screen. This facility can be used to present a form on the screen for the user to complete with his responses.

Each point where a response is required is termed an input field, and is indicated by an '@' symbol followed by a field number in the range 0 to 9. For example, the frame

```
*100
Type in your weight in pounds @0
?=@0
>200 =150, =200
.....................................
```

would prompt

```
Type in your weight in pounds .........
```

and the cursor would be positioned at the first dot. The user would then type in his weight followed by RETURN in the usual way.

If several positions on the same form need to be checked, they should be labelled with '@0', '@1', '@2', etc, and the form

should be FIXed on the screen (with $FIX). Answers can then be checked by frames with prompts '?@0', '?@1', etc, which will move the cursor to the appropriate point.

For the above example, the input prompt is to the end of the screen, and this is termed a default field. It is also possible to define the length of the input field, by following the '@n' in the text by one or more dots. The user will then only be able to type in this field (cf the example in the next section). When the field is full ('!' mode) or RETURN is pressed ('?' mode) the answer will be checked.

Note that the '@' and field number do not appear on the screen when the user runs the material. If a field length is defined, '@n' will be replaced by 2 blanks to simplify alignment of forms, and the cursor moves to the position of the first dot. If there are no dots, the cursor moves to the position of the '@'.

'?' mode with default field *

In a default field prompt like the example above, the prompt is from the '@' to the end of the line. Any text following the form prompt will be overwritten. This allows sample answers to be included then overwritten by the user's actual response.

'?' mode with field length *

This will prompt with the required number of character positions. Input must be terminated with RETURN, and the cursor cannot be moved outside the field:

```
*10
Name the capitals of the following countries:

COUNTRY          CAPITAL
England        @0.........
France         @1.........
USA            @2.........

$FIX
?@0
LONDON=20,=50
.....................................
*20
Yes, try the next one.
?@1
PARIS=30, =50
.....................................
*30
Yes, and the last one.
?@2
WASHINGTON=40, =50
.....................................
```

'@n' will be replaced by blanks on the screen to maintain alignment of fields on the form.

'!' mode with default field *

This checks for a single character, for example:

```
*10
The Amiga computer was launched in 198@0
!=@0
<5 = 20, 5 = 30, >5 = 40
.....................................
```

'!' mode with field length *

In this mode '@n' is displayed as blanks to maintain alignment, and input is terminated when a character is typed into the last position in the field.

```
*10
  SERIAL NUMBER      DATE
                     Day  Month  Year
@0........          @1.. @2...  @3..
```

7.6] Unconditional branching

Microtext normally waits at the end of a frame of text for the user to make a response. It is also possible to branch directly to another frame of text without waiting for the response.

```
*10
Yes, well done
=20
.....................................
*20
```

Here '=20' means 'display any text then go straight to frame 20'. This can be used to prefix frame 20 with a comment appropriate to the answer to the previous frame. The automatic clearing of the screen is inhibited, and the frame after the unconditional branch is treated as if the previous frame had a $JOIN command. Unconditional branches can also be used to generate summary items (chapter 9).

If a frame is terminated with delimiting dots before the prompt for input, the next frame is treated as a continuation of the previous one, so that the example above could also be written as

```
*10
Yes, well done
.....................................
*20
```

7.7] Subroutines *

It may sometimes be necessary to repeat a series of questions at different points in the text. This can be achieved by having one set of frames which are used as a subroutine.

Subroutines are called by a branch in which a RETURN is specified:

```
*10
Now some questions about Germany
=100 RETURN
.....................................
*20
.
.
.....................................
*100
Is Berlin the capital of West Germany?
.
.
.....................................
*200
=RETURN
.....................................
```

The user will see the text in frame 10, immediately followed by that in frame 100. Branching then takes place normally until at 200 RETURN indicates that a return should be made to the frame following the original branch with RETURN.

It is essential that subroutines are always exited by a =RETURN. It is also possible to return to a point other than the original branch (although this is not generally recommended), for example:

```
=RETURN 30
```

returns from the subroutine to frame 30.

Or alternatively

```
= 100 RETURN 30
```

will cause a return to 30 when a = RETURN is encountered.

If no subroutine call has been made, a return branch will give the error message.

```
Bad return error
```

Subroutines must be in the same module as the frames branching to them. Subroutines may be nested to a depth of 10.

Applications of subroutines *

An important use of subroutines is in constructing HELP systems for users. This is described in section 7.7.

Subroutines can be called from several different places, with the exact function determined by variables (see chapter 8).

7.8] Branching to other modules

It is also possible to branch to a frame in another module, in which case it is necessary to prefix the frame number by the module name. Thus, to branch to frame 10 of MIC.INT:

```
MICRO = MIC.INT:10
```

Note that '= MIC.INT' will branch to the first frame of the INT module.

Loading modules

Complete modules are loaded into memory, so branches between frames in one module are almost instantaneous. New modules are loaded automatically when required, but this may take a significant time using floppy disks, or several minutes with cassette tapes. When branching from one module to another it may be appropriate to warn the user that there will be a delay.

On cassette-based systems it may be necessary for the user to press PLAY on the tape recorder, and the author should include appropriate instructions in the text; Microtext will also prompt appropriately on the bottom line. Backward jumps to previous modules are only possible if the user is instructed to rewind the tape.

8] Using variables

Microtext allows **variables** to be used to store text and provides a variety of facilities for manipulating them. If you are familiar with a programming language such as BASIC then the concept will not be new to you (although there are some important differences in the way Microtext variables work). If you are not familiar with the concept, think of a variable as a labelled box in which computers store things - the statement NAME = "FRED" causes the computer to put the word "FRED" in a box labelled "NAME", from which it can later retrieve it.

As a simple example, one way in which a variable can be used is on the response line of a frame to store a value that is determined by the user's input:

```
*10
Are you male (M) or female (F)?
!
F=30 (SEX="F")
M=40 (SEX="M")
.....................................
```

The variable SEX now contains the value "M" or "F" and this information can be used later in the module if you need to modify the dialogue depending on the user's sex. This idea is developed further in section 8.3.

8.1] Some basic facts

A variable can be created and named by the author; there are also a number of system variables which are created and updated by Microtext (see section 8.4).

Variable names

Only letters and digits may appear in variable names, and the name must start with a letter, eg:

```
NAME
AGE
REPLY1
REPLY2
```

Upper and lower case letters are treated as equivalent in variable names, so 'SCORE', 'Score' and 'score' all refer to the same variable.

Variable names may be up to 16 haracters in length.

Contents of variables *

A variable can hold up to 250 characters. No distinction is made between numbers and text: the context in which a variable is used determines how its contents are treated. This is fully explained in subsequent sections.

Initialisation *

A variable may be assigned a value when it is first used. If a variable is tested which has not been used previously it will be initialised by Microtext. The initial value assumed will depend on how the variable is being treated in the test. If treated as a numeric value (section 8.3) numeric zero will be assumed. Otherwise the variable will be treated as being empty. This is equivalent to the assignment:

```
VAR=""
```

8.2] Using variables

Naming the response

A variable name can be given at the top of a frame. This has the effect of creating a variable with the given name in which the user response to the frame will be stored.

Thus the example at the beginning of the chapter could be rewritten more succinctly like this:

```
*10 =SEX
Are you male (M) or female (F)?
!
F = 30, M = 40
.....................................
```

The variable name can be inserted after the frame number by positioning the cursor using the HOME key.

Note that the system variable ANS always holds the latest user response and can be used if the response need only be stored until the next frame (cf the example in section 5.2).

Matching responses contained in a variable *

Input can be taken from a variable instead of the keyboard by using a variable name after the input prompt. Branching can thus be done on the basis of the value of the variable. Thus, if the variable SEX had been set up as in the example above, a branch dependent on its value could be made later in the module as follows:

```
*50
We will now ask you a few questions on
your medical history.
!SEX
M=100, F=500
.....................................
```

All forms of prompt can be used. Thus the '?' prompt would be used if a word or phrase is required, and numeric prompts are employed when the variable contains a number e.g.:

```
    ?NAME     to  match with text in NAME
 or ?=SCORE   to test the value of a number in SCORE
```

Displaying variables

Variables can be displayed in the middle of text by enclosing them in angle brackets:

```
*1 =NAME
What is your name?
?
=2
.................................................
*2
Hello <NAME>, nice to meet you.
```

The variable will be expanded on the screen in the position indicated. Thus if NAME contained "Arabella", the text displayed would read:

```
Hello Arabella, nice to meet you
```

If the variable NAME had not been set up in frame 1, the text displayed would be:

```
Hello <NAME>, nice to meet you.
```

If the resulting line is over 40 characters long the text will wrap round on the screen as much as necessary, up to the maximum of 250 characters in one line.

Manipulating text variables *

Variables can be initialised to a particular value with '=', or concatenated with '&'. The instructions are given in brackets, either immediately following the frame number, or on subsequent lines, prefixed by '$', for example:

```
*1 (NAME="FRED ", LAST="JONES")
$(NAME&LAST)
Hello <NAME>
.....................................
```

NAME&LAST has the same effect as NAME=NAME&LAST, i.e. concatenate LAST on the end of NAME to give in this case "Hello FRED JONES".

More complex expressions are also permitted, for example,

```
$(INITIAL="F ")
$(NAME="MR "&INITIAL&LAST)
```

giving "MR F JONES".

The last user response is held in the system variable <ANS>, and can be added to a variable:

```
$(NAME&ANS)
```

or used to initialise a variable to the user response:

```
$(NAME=ANS)
```

Using variables on response lines *

Variables can be related to particular user responses by including them on the relevant lines as in the simple example at the beginning of this chapter. A more complex example is this:

```
*10 =NAME
What is your name?
?
=20
.....................................
*20
Are you male ?
!
Y=30 (TITLE="MR ")
N=30 (TITLE="MS ")
.....................................
*30 (NAME=TITLE&NAME)
```

Thus the answer to 10 could store "JONES" in NAME, and the answer to 20 could initialise TITLE to "MR". Frame 30 would then prefix NAME by TITLE to give "MR JONES".

It is also possible to put run-time commands on response lines, for example:

```
Y=30 $BEEP
```

would cause a beep (Chapter 15) if the answer was Y. A special case is $END which can be used without a branch address:

```
Do you want to stop now?
!
Y = $END, =RETURN
```

Note that commands on response lines should not contain commas as this can confuse Microtext - use spaces instead.

8.3] Numeric variables

Numeric variables can be used to keep a score, or check the number of times a question has been repeated. They can be set by the answer to a numeric question, or by explicitly assigning a number to the variable:

```
*10 (SCORE=0)
```

They can be incremented or decremented each time a branch is taken:

```
=20 $(SCORE+2)
```

(SCORE+2 can also be written as SCORE=SCORE+2).

Individual answers can be scored, e.g.:

```
LONDON=20 (SCORE-2)
BIRMINGHAM=30 (SCORE+1)
=40 (SCORE-1)
```

One variable can also be used to set another:

```
*20 (SCORE1=SCORE)
```

or can be added or subtracted from another:

```
*20 (SCORE=SCORE1-SCORE2)
```

It is also possible to do several things in one line:

```
*20 =NAME (TITLE="MR ",SCORE=0)
```

Note that in "level 2" Microtext only integer arithmetic is supported.

Since numeric variables are stored as a series of digits, they can also be manipulated as text (see section 8.2 and 8.9).

8.4] System variables

The system variable ANS, which contains the user's last response, has already been described above. There are two other system variables: TIME and RANDOM.

TIME

TIME is used as a clock to give the time elapsed since it was initialised. The variable is set using the run-time command $TIME. To initialise the variable to zero, the command:

```
$TIME=0
```

should be used. The command:

```
$TIME
```

will then set the variable TIME to the time interval elapsed since initialisation (in tenths of a second). The value can then be used as, for example, part of collecting statistics on the user's performance in tests.

RANDOM

The variable RANDOM is used to store a random number generated by the system. A random number can be generated in the range 1 to 255 using the $RANDOM command, which has the form:

```
$RANDOM n
```

and produces a number RANDOM in the range 1 to n.

8.5] General use of variables *

Variables can replace any part of a frame between the frame number and delimiting dots. This is an advanced feature which can significantly simplify many applications, but may need careful planning to give the required effect. A line containing variables will be expanded so that all variables are replaced by their contents before Microtext attempts to display or act on the contents of the line. Unknown variables used in the text remain unexpanded, so that the text could for instance refer to the <RUN/STOP> key.

A simple example is the construction of a list containing data:

```
*5 (COUNT=1)
=10
.....................................
*10 =NAME<COUNT> (COUNT+1)
Enter the next name, or END.
?
END = 20, = 10
.....................................
```

This would build up a list of names in NAME1, NAME2, and so on.

Variables can contain any part of a line, including text, summary

items, keywords to be checked, or frame numbers. They can be used to produce very sophisticated applications, but note that excessive use of variables can be an overhead on speed and work area, as they occupy space in memory until reinitialised (NAME="" etc, or $NEW).

8.6] Listing the variables

A display of all variables that have been used can be obtained in Command Mode by using the VARIABLES command (which has no parameters).

The variable names and their contents are displayed one at a time, and pressing any key moves from one variable to the next. If no variables have been used, a blank screen is presented.

The system returns to Command Mode when all variables have been displayed, or when RUN/STOP is pressed. This is useful during testing (see chapter 10).

8.7] Clearing variables

All variables can be cleared and re-initialised using the run-time command $NEW. The form:

```
$NEW VARIABLES
```

does this. If the parameter is omitted, both the variables and the summary are cleared. Further details are given in chapter 9.

8.8] Applications of variables

Variable values in run-time commands *

Variables can be used in place of values in $ commands. For example:

```
$LINE <ROW>,<COL>
$SUMMARY SAVE,<FNAME>
```

would use the numbers stored in the variables ROW and COL and the filename stored in FNAME.

Variables and summaries

A summary of the interaction with the user can be built up during a run, as described in chapter 9. Variables can be used in this process in the same way as described in this chapter.

Testing variables containing keywords *

Variables can be used to hold combinations of keywords used to test user input. This may be useful if the same combination of keywords is to be tested in several different places:

```
*10
Name 2 cities, one starting with D and
one starting with E.
$(CITIES="(DERBY/DURHAM)&(ELY/EXETER)")
?
<CITIES>=50,=20
.....................................
*20
No, try again.
?
<CITIES>=50,=30
.....................................
*30
No, you could have included Derby or
Durham, and Ely or Exeter.
=60
.....................................
```

Combinations of several variables, or of variables and literal keywords, can be tested if these are separated by appropriate operators:

```
LONDON+(<VAR1>/<VAR2>) = 20
```

Branching to variable locations *

A variable NEXT could be set to 1, 2 or 3 to determine a branch to 100, 200, or 300:

```
*10
Now for the next topic
?NEXT
1 = 100, 2 = 200, 3 = 300
.....................................
```

A more direct solution would be to set the variable NEXT to the value of the actual frame concerned, i.e. either 100, 200 or 300. The example above would then become:

```
*10
Now for the next topic
= <NEXT>
.....................................
```

Alternatively, if NEXT had been set to 1, 2 or 3, the example could be written as:

```
*10
Now for the next topic
= <NEXT>00
.....
```

Checking for repeated answers *

Variables can be used to count, or prevent, repeated answers.

The example below tests a count of the number of attempts at an answer. Numeric variables are tested using '?=':

```
*10 (COUNT=0)
What is the capital of the USA?
$FIX
.....
*20
?
WASHINGTON = 100
= 30
.....
*30 (COUNT+1)
?=COUNT
>2 = 50
= 40
.....
*40
No, try again.
= 20
.....
```

This could also be written using a general purpose subroutine:

```
*10
What is the capital of the USA?
$FIX
$(TEST="WASHINGTON",MAX=2)
= 100 RETURN 20
.....
*20 $UNFIX
.
.
etc ....
.
.
.....
*100 (COUNT=0)
.....
*110
?
<TEST> = 150
= 120
.....
*120 (COUNT+1)
```

```
?=COUNT
>MAX = 140, = 130
.....
*130
No, try again.
= 110
.....
*140
No, the answer is <TEST>.
= RETURN
.....
*150 Yes, well done.
= RETURN
.....
```

Variables can also be used to build up possible answers. Here is an example using a general technique to prevent the same answer being given twice to a question which is repeated several times. This uses a TEST variable which is initialised to the dummy answer "Z":

```
*10 (TEST="Z",ANS="Z")
=20
.....
*20 (TEST&"/"&ANS)
Which option would you like to try next?
A, B, C, D, or E
!
<TEST>=30
A=31,B=32,C=33,D=34,E=35
.....
*30
You have already tried that. Have another go.
=20
.....
*31
OPTION A ...
 = 20
.....
*32
OPTION B ...
= 40
.
.
etc ....
.
.
```

This builds up a variable containing the previous answers (e.g. Z/Z/A/B/C). New answers are tested against this variable to exclude repeated answers. Remember that a variable cannot contain more than 250 characters.

8.9] Rules for the evaluation of expressions *

This section explains the rules which apply when variables in angle brackets are used, default initialisation of variables is assumed, or numeric and text operations are mixed.

Use of variables in angle brackets *

As explained in section 8.5, variables enclosed in angle brackets are expanded into the text buffer before the line is interpreted by Microtext. They will not be affected by any manipulation of variables within the line. This will only become apparent if a variable is expanded which has been initialised by a previous expression on the same line. For instance, if COUNT=0, then:

```
$(COUNT=1, SCORE=COUNT)
```

sets SCORE to the new value of COUNT (i.e. 1), and:

```
$(COUNT=1) $(SCORE=ARRAY<COUNT>)
```

sets SCORE to the value of ARRAY1, but,

```
$(COUNT=0) $(COUNT=1, SCORE=ARRAY<COUNT>)
```

sets SCORE to the value of ARRAY0.

Evaluation of expressions *

Expressions containing operators are evaluated from left to right, one operation at a time. '+' and '-' operate on numbers, and '&' operates on text. If '&' is used with numbers, they will be concatenated. If a numeric operation is performed on text, the numeric part of the text will be used (as with the VAL function in BASIC).

For numeric prompts (i.e. ?= or !=, see section 7.4) the user's response is stored as a conventional number, so that the answer 0032 would be stored as "32", and just typing RETURN would store "0".

Previously undeclared variables in expressions will be created with null or zero contents. For example, assuming SCORE and NAME start as undefined variables:

```
SCORE+1          SCORE will be "1"
SCORE=5&SCORE    SCORE will be "51"
SCORE=1&5        SCORE will be "15"
NAME&"A"         NAME  will be "A"
SCORE=1+NAME     SCORE will be "1"
SCORE=1&NAME     SCORE will be "1A"
SCORE+5          SCORE will be "6"
```

9] Summary items

You may wish to build up a summary of the interaction which can be archived or printed out as a report at the end. For example, this could take the form of a detailed analysis of an individual student's performance in a test, rather than just the final score achieved. Items to be included in a summary are placed after response tests, prefixed by '*' or '**'.

9.1] General summary items

Items after response tests prefixed by '**' will be copied to the summary when the frame is used:

```
*1
What is the capital of Britain?
?
BIRMINGHAM=2, LONDON=3
**capital: <ANS>
.....
```

The inclusion of the system variable <ANS> indicates that the user response should be appended, for example,

```
capital: LONDON
```

The list of items and user responses can be printed out for the instructor when the student has finished.

If necessary, summary items can be more than one line long:

```
BIRMINGHAM=2, LONDON=3
**student said that <ANS> was
**the capital of Britain
```

User responses can also be stored without any other text, for example

```
*10
Please type your name
?
=20
**<ANS>
.....
```

9.2] Selected summary items

It may sometimes be more convenient to generate different summary items, dependent on the user response. In this case the summary items are prefixed by one '*', and follow the appropriate answers:

```
*1
Is Bonn the capital of West Germany ?
!
Y=2 *knows capital
N=3 *doesn't know capital
.....................................
```

The response given determines which summary item is used: this will be the first summary item if the user makes the first response ('Y'), the second if they make the second response ('N'). If there is no summary item for a particular response, then nothing is stored.

Selected summary items can start on the next line (in which case they apply to all tests on the previous line), and can be more than one line long, eg:

```
*10
2 + 2 is ..
?
4 = 20, FOUR = 20
*This time they got
*the answer right
= 20 *Wrong again
.....................................
```

9.3] Headings *

Summaries can be split up under several headings. For instance, un-conditional branches can be used to create summary items which are independent of the user's response. Typically these are headings used to divide up the summary:

```
*20
=21
**- WEST GERMANY -
.....................................
*21
Is Berlin the capital of West Germany?
?
Y=21,N=23
**capital Berlin - <ANS>
```

Summary items can contain blank lines, eg to include a blank line before and after the heading '- WEST GERMANY -':

```
*20
=21
**
**- WEST GERMANY -
**
.....................................
```

9.4] Frame number trace *

A trace can be made of the frames executed and a list of the frame numbers stored in the summary using the $TRACE SUMMARY command. This is useful during module testing and is described in chapter 10.

9.5] Displaying and saving summaries

Summaries can be displayed or printed at any point using:

```
$SUMMARY                  display summary on the screen
$SUMMARY PRINT            print summary
```

Summaries can be archived on tape or disk:

```
$SUMMARY SAVE             save with default name SUMMARY
$SUMMARY SAVE, filename   save as filename
```

The summary can be written to a variable filename by giving the name of a variable in angle brackets in place of the filename. For example

```
$SUMMARY SAVE,<NAME>
```

where the filename used will be the one currently stored in the variable NAME.

Summaries can also be displayed in Command Mode using the SUMMARY command, which has the same options as the $SUMMARY command.

9.6] Clearing the summary

The summary is cleared each time a run of the module is initiated by RUN or TEST. It can also be cleared at any point using the $NEW command:

```
$NEW SUMMARY              clear the summary

$NEW SUMMARY VARIABLES    clear summary and all variables
  or $NEW
```

(This should not be confused with using NEW in Command Mode, which also clears the module.)

9.7] Using a summary to store variables *

It may be necessary to save variables which can be recalled later in order to restart an interrupted presentation, increment a sequence number for each user, or hold a score, to name a few examples. The required effect can be achieved by using summary items to create a new Microtext module which contains the values

of the variables. These values can be recovered later by executing the new module, as illustrated in the examples below:

Module TES.STA:

```
*10
Restore variables?
!
Y = TES.VAR, N = 20
.....................................
*20
.
.
etc...
.
.
```

Module TES.END uses the summary to create the module TES.VAR:

```
*100 $NEW SUMMARY
$(SEQUENCE+1)
= 110
***10
**$(SEQUENCE=<SEQUENCE>)
**$(NAME="<NAME>")
**$(SCORE=<SCORE>)
**=TES.STA:20
**...
.....................................
*110
$SUMMARY SAVE TEST.VAR
$END
.....................................
```

TES.END creates a summary file called TES.VAR. This file is formatted as an ordinary module, as shown below:

```
*10
$(SEQUENCE=5)
$(NAME="SMITH")
$(SCORE=31)
= TES.STA:20
.....................................
```

This contains the necessary Microtext commands to restore the values of the variables.

A similar technique is used in the following module to store a newly generated summary in a series of summary files with an ascending sequence number:

Summary items

Module SEQ.END:

```
*10
= SEQ.SEQ
.....................................
*20 (FILENAME='SUM.'&FILENO)
$Comment First save the existing summary
$SUMMARY SAVE,<FILENAME>
$NEW SUMMARY
= 30
***10
**$(FILENO=<FILENO>+1)
** = SEQ.END:20
**...
.....................................
*30
$Comment Now save the new module
$SUMMARY SAVE,SEQ.SEQ
$END
.....................................
```

10] Testing and running modules

This chapter describes how Microtext modules are tested and run. The bulk of the chapter is concerned with running in Test Mode which provides facilities for the author to assist with developing error-free modules.

Run Mode is the normal mode of use of the system. It provides a subset of the facilities available in Test Mode that are appropriate for the ordinary user.

10.1] Module testing

A module is run in Test Mode using the TEST command. In this mode extra facilities are provided for 'debugging', such as warning messages, and complete protection against over-writing a newly edited module.

The TEST command

The TEST command initiates execution of a module in Test Mode. TEST without parameters runs the current module starting at the first frame. If there is no module current in memory then the error message

```
NO MODULE PRESENT
```

occurs.

You can also test particular bits of a program by starting its execution at a specified frame number. For example, the command

```
TEST 40
```

will run the current module from frame 40.

If you specify a filename, the module specified is loaded from tape or disk and run from its initial frame. Examples are:

```
TEST 100            runs the current module from frame 100
TEST NUM.ONE        runs module NUM.ONE from first frame
TEST NUM.ONE:40     runs module NUM.ONE from frame 40
```

The specified module will then be run, interpreting text and control information in each frame.

Note that in Test Mode the keys on the keyboard will repeat when held down; this 'auto-repeat' facility is suppressed in Run Mode however.

The system will return to Command Mode when $END is encountered or an attempt is made to branch past the last frame in the module.

You can interrupt the execution of a program when Microtext is waiting for user input by pressing RUN/STOP.

In this case an interrupt menu is displayed (see below). To resume the program run, use the CONTINUE command: the run will be resumed at the beginning of the last frame with the values of all variables intact.

You can also use the BACK command: this takes you back to the start of the frame containing the previous question.

Restarting from the interrupted frame using the TEST command will re-initialise the variables.

Holding down RUN/STOP and pressing RESTORE can be used at any time to return to Command Mode, but note that you cannot then use CONTINUE to resume the program run.

Interrupt menu

When execution is interrupted using RUN/STOP a list of options is displayed on the command line together with a prompt. The option required is selected by typing the first letter of the option name. The option list displayed in Test Mode is

```
Cont Stop Back Goto Edit Vars $...
```

Run Mode can be interrupted in the same way, but the range of options can be limited by use of the $ALLOW command.

One of the following letters should be typed:

C - Continues from the beginning of the current frame

S - Stops and returns to Command Mode. The module can be edited following which the CONTINUE command will continue from the start of the current frame.

B - Goes Back to the start of the frame containing the previous question. It will not function if the current frame was entered by a branch from another module, or by a previous 'back' function.

G - Goes to new frame number

E - Enters Edit mode. (This has the same effect as selecting 'Stop' then pressing RUN/STOP.)

V - dumps Variables to screen. Variable names and values are displayed one at a time; RUN/STOP can be used to escape from this, or press any other key to move from one variable to the next.

$ - prompts for a run-time command, including $PAUSE, $TRACE or $SUMMARY.

Execution trace *

Microtext can trace the execution of frames at run-time. Frame numbers can be stored in the summary, or displayed on the screen. In the summary, they are prefixed by '*', and the filename is also stored prefixed by '*' every time it changes:

$TRACE SUMMARY,ON/OFF frame numbers to summary ON/OFF

In Test Mode frame numbers are displayed on the screen

in the form 'program.module:frame'. The display can also be turned on in Run Mode by the command.

$TRACE SCREEN,ON/OFF frame number on screen ON/OFF

Comment frames

As explained in section 6.1, a comment frame is not executed but will be displayed when the module is listed or printed.

It may be useful during testing to temporarily disable a frame. For example, if an executable frame 50 exists, the command:

COMMENT 50

will turn it into a comment frame, and the command:

FRAME 50

will turn it into an executable frame when desired.

Listing variables and summaries

The contents of variables can be displayed on the screen by selecting the 'V' interrupt option.

The VARIABLES command can similarly be used in Command Mode as described in section 8.6.

It can often be useful to check the contents of the summary during testing. Again, this may be done in interrupt mode using $SUMMARY or in Command Mode using the SUMMARY command.

10.2] Run Mode

Run Mode is the mode in which tested and debugged modules are run by the user. It offers a subset of the facilities available to the author in Test Mode, and these are highlighted in the following sections.

Execution of a module in Run Mode is initiated by the RUN command which has the same formats as TEST.

Interrupting execution

A run may be interrupted using RUN/STOP as described on the last section, or by RUN-STOP/RESTORE.

If you press RUN/STOP while Microtext is prompting for user input in Run Mode, then by default you can do one of two things: either STOP the program, or CONTINUE. However, the author may extend or restrict this range of options for a module via the $ALLOW command.

Without parameters the $ALLOW command will completely disable the interrupt facility so that pressing RUN/STOP has no effect. Otherwise, the desired options are specified as parameters. For example:

```
$ALLOW C,S,B,G,E,V,$     allow all options
$ALLOW B,G               allow Continue, Back, and Goto
$ALLOW                   ignore RUN/STOP
```

Note that if any options are allowed, then C is automatically allowed.

The valid options will be listed on the command line when execution is interrupted.

A user can normally return to Command Mode at any time by using RUN-STOP/RESTORE.

Using function keys

In Run Mode the function keys normally generate the digits 1 to 8.

10.3 Saving altered modules

Modules are stored on tape or disk using the SAVE command:

```
SAVE filename
```

This saves the specified module to tape or disk. SAVE with no filename will use the current filename from the screen, or if no

name has been previously specified will give the error message NO VALID FILE NAME. If 'filename' already exists on disk, then Microtext will prompt 'REPLACE? ', which should be answered by a 'Y' or 'N' response.

Modules can be loaded with the LOAD command:

LOAD filename

This loads the specified module from tape or disk. Confirmation is requested if the current module has unsaved alterations. LOAD alone gives the error message 'NO VALID FILE NAME'.

If the author has edited a module without saving it, any command which would cause this module to be erased will prompt with the message:

Save edited module? Yes, No, Stop

Similarly in Test Mode, an attempt to branch from an unsaved module to another module will also give rise to this message.

Typing Y will save under the current file name, then continue with the specified operation; N will continue the operation without saving; and S will return to Command Mode.

Note: No warning is given when branching between modules in Run Mode.

10.4] Filename conventions

As explained previously, modules are stored in files with names of up to 16 characters. The Microtext convention adopted in this manual is for file names to consist of program and module names of up to 3 characters each (as in INT.STA).

10.5] File handling

Either disk or tape can be used for module storage; both can be used if available on the system.

Commands

The following commands provide file handling facilities.

CAT

Displays the catalogue (directory) for the current disk drive. The catalogue remains on the screen until another command has been typed.

CAT TES* performs pattern matching - ie it gives a list of all files starting with the letters TES. A '?' matches with

any single character in this context; '*' matches with any number of characters, up to end of filename.

TAPE
Selects tape filing system for all save and load operations.

DISK
Selects disk filing system.

DRIVE n
Selects drive number 0 or 1 on a twin-drive system. The Microtext default is to use drive 0.

>
Pass command to CBM DOS. Examples are as follows:

```
>$0                          Directory of drive 0
>$0:TES*                     Directory with pattern matching
>S0:TES.STA                  Scratch (delete) file 'TES.STA'.
>R0:TES.DEM=TES.STA          Rename TES.STA to TES.DEM
>C0:TES.BAK=0:TES.STA        Copy TES.STA to TES.BAK
>C0:NEW.MOD=0:M1,0:M2        Concatenate (join up) M1 and M2 to
                             form NEW.MOD
```

Other DOS commands can be used as described in CBM documentation. (These commands are also usable in run or test mode, by preceding them with a dollar sign, eg $>S0:TES.STA.)

10.6] Error handling

Two kinds of error may occur during Run or Test Mode: **fatal** and **recoverable**.

Fatal errors (such as NO ROOM IN MEMORY) are displayed on the command line; press any key to take you back to Command Mode, where the frame at which the error occurred will be identified.

Recoverable errors cause the system to react differently in Run and Test Modes. In Test Mode, the error message is again displayed on the command line, but the author can then press RUN/STOP to exit to Command Mode, with CONTINUE enabled. Pressing any key other than RUN/STOP causes the system to continue the run, making some assumption by default.

In Run Mode, no message is displayed for a recoverable error, nor does the system halt. Microtext will simply take the default action for the error.

When appropriate, Microtext will treat possible syntax errors as ordinary text (for example mis-spelt variables and commands will appear as text on the screen).

Details of the possible error messages and the defaults for each recoverable error are given in Appendix C.

10.7 Advice for users

The author will need to give the user some information about the Microtext system and the program. This may consist of an information sheet or other document that gives advice on such things as loading modules, program and module names, an outline of what the program does, available interrupt options and so on.

Microtext has two types of built-in help. Typing HELP in Command Mode supplies help for the author; this can be disabled using HELP OFF. Typing HELP in Run Mode can give the user help. The help system enables authors to build advice and assistance into their program.

The HELP system *

The author can provide subroutines (see section 7.7) to give users help at any point in a program. Help can be provided if the user types:

```
HELP
```

followed by RETURN in response to a ? prompt;

```
?
```

followed by RETURN in response to a ? prompt; and

```
?
```

in response to a !, or !@ prompt.

Help should be written as a subroutine in one or more frames (in the same module). The address of the HELP subroutine is given in a $HELP command, e.g.:

```
$HELP 900       HELP response calls subroutine at 900
$HELP           HELP response will give 'no help' message
$HELP OFF       HELP response is treated as normal input
```

The default response to HELP is the 'no help' message:

```
SORRY NO HELP AVAILABLE
```

The author may choose to provide one or more of the following types of help:

1. Help on this question
2. Help on this topic
3. Help on this module
4. Help on the use of the Microtext system

The help subroutine may include a menu to offer several choices, for example:

```
*900
Which would you like to do?

     1: Get help on this topic
     2: Restart this topic
     3: Branch to a known frame
     4: Try the question again
     5: Return to the main menu

!
1 = 950, 2 = RETURN 10, 3 = 910
4 = RETURN, 5 = RETURN 920
.....................................
*910
Which frame do you need?
?
= RETURN <ANS>
.....................................
*920
=MAIN.MENU
.....................................
*950

HELP FRAMES

... etc.
```

Note the following ways that help can be used:

1. To give access to special help frames, in this case starting at frame 950.

2. To branch back to a specific point, in this case frame 10.

3. To prompt the user to select his own frames (as an alternative to using the interrupt menu).

4. To return to the last question. Note that help subroutines return to the frame from which help was requested, while ordinary subroutines return to the following frame.

5. To give an easy way of interrupting a presentation to return to a main menu, in this case a module called 'MAIN.MENU'.

PART III - 64 GRAPHICS AND SOUND

In addition to the standard ("core") Microtext facilities described in Parts I and II of this document, Microtext for the Commodore 64 contains additional commands giving easy access to the full range of graphics and sound capabilities of this machine. However, it should be realised that most of these commands are specific to the 64 version of Microtext, and that modules that use them will not generally easilly be portable onto other implementations. Exceptions are the high-resolution $MOVE, $DRAW, $POINT and $TRIANGLE commands - these have been written in such a way as to achieve maximum possible portability of high-resolution graphics material between the 64, BBC, and Apple implementations of Microtext.

11] Extended text modes

A variety of different "modes" are available for screen display in 64 Microtext. This section discusses modes 0 to 2 which are primarilly used for text display and which may be selected both during run-time and for editing; mode 3 is for high resolution graphics and is available only in Run and Test modes (see Chapter 13). A further mode - mode 4 - is currently reserved for possible future use. To change mode, use

Command mode:	MODE n	(0 to 2)
Run or test mode:	$MODE n	(0 to 3)

11.1] Mode 0

This is the standard text mode, as described in Chapter 6. Mode 0 is always returned to on termination of execution, when the 'end of run' message is removed; mode 0 is also selected if the user performs an 'emergency exit' using RUN/STOP-RESTORE.

11.2] Mode 1 *

This is "multi-colour" text mode. In mode 0, only one text colour (selected using CTRL or C= plus a number key) can be displayed within any character position. "Multi-colour" mode makes two more colours available, at the cost of halving the screen resolution (number of screen points or "pixels") horizontally.

The two additional colours can be selected from any of the 16 available, using an extended version of the SCREEN command:

```
     SCREEN  background, colour 2, colour 3
or   $SCREEN background, colour 2, colour 3
```

where background, colour 2 and colour 3 are "colour numbers" in the range 0 (black) to 15 (light grey) - see Chapter 6.

When using MODE 1, characters written to the screen in any of the **primary** (CTRL-number key) colours are written as usual. However, characters written using the **secondary** (C= -number key) colours are interpreted in a different way, using the usual screen background and text colours, plus the extra colours specified. If you try this using normal text or graphics characters, you will find that the effects are very nasty. However, the point of the facility is that you can create your own **user-defined** multi-colour characters using a variation of the $DEFCHAR command - see Chapter 12.

11.3] Mode 2

This is **extended background mode.**

Modes 0 and 1 are limited to one "background" colour, selected by the simple form of the SCREEN command discussed in chapter 6, which provides background for the entire screen. Mode 2 gives you a choice of 4 different backgrounds in a screen, at the cost of reducing the number of characters available. The backgrounds are specified using another variant of the SCREEN or $SCREEN commands:

```
 SCREEN background1, background2, background3, background4
$SCREEN background1, background2, background3, background4
```

The different backgrounds can then be selected using combinations of SHIFT and RVS on and off:

unshifted	rvs off	background1
shifted	rvs off	background2
unshifted	rvs on	background3
shifted	rvs on	background4

This is a very easy mode to use (try it in Edit mode), and gives an effect comparable to using a highlighting pen. Its disadvantages are that ordinary reverse field and shifted characters (upper case or graphics) are not available.

11.4] More about the Mode and Screen commands *

You will have noticed that the SCREEN command does a lot of work. Note the following points:

1. The system sets various defaults which can be useful when experimenting. The default screen background (background1) is light grey - this allows the maximum possible number of text colours to show up reasonably well. The defaults for colour 2 and colour 3 in multi-colour mode are blue and yellow; the defaults for background colours 2, 3 and 4 in extended background mode are blue, yellow and white.

2. SCREEN parameters set in one mode carry over into the others. Thus entering a four parameter SCREEN command in mode 0 has no effect beyond setting the current background, but the additional values given will be used if another mode is selected.

3. Only those parameters explicitly given in a SCREEN or $SCREEN command will be altered. Thus SCREEN 1 will change the background (background 1) to white, without affecting any other settings made earlier.

4. Changing mode within a Microtext module using $MODE will cause the screen to clear; this does not happen when using MODE in command mode.

12] User defined characters *

The Commodore 64 knows about 512 possible characters; these are arranged as follows:

0 to 127	upper case/graphics, normal field
128 to 255	upper case/graphics, reverse field
256 to 383	lower/upper case, normal field
384 to 511	lower/upper case, reverse field

Values from 0 to 255 are identical to the "screen display codes" for upper case and graphics characters, given in the User Guide for the Commodore 64. Values from 256 to 511 are the "screen display codes" for the lower/upper case character set, plus 256.

In Microtext, the user has the power to define ANY or ALL of these characters using a variety of run-time commands, thus allowing a very large variety of possible character sets. Redefinitions remain in effect until changed again, so this facility should be used with caution; if the usual alphanumeric characters are changed into something sufficiently silly, then the status and command lines will be unreadable. Fortunately, a single command exists to restore the default character set; this should normally be done at the end of program execution.

The commands affecting user-defined characters are as follows:

$CHARACTER	- restore default character set
$CHARACTER n	- restore character number n
$CHARACTER n,p1...p8	- redefine character number n (BBC-style syntax)
$DEFCHAR n,option {matrix}	- redefine character number n (fairly sensible syntax)

$CHARACTER will cause a complete reset to the default state. This is the safety net.

$CHARACTER n will redefine a single character to its default state. Numbers n are in the range 0 to 511, as above.

$CHARACTER n,p1...p8 is used to redefine a character in a way desiged to make it reasonably easy to transfer definitions from the BBC Micro. p1...p8 represents a list of eight values all in the range 0 to 255, each of which represents one row of the character to be redefined. If you don't have a BBC Micro, don't worry about this.

$DEFCHAR n,option is used to redefine a character in a more convenient way. It is followed by a character number, then an option which must be zero for a normal character, or 1 for "multi-colour" character for use in mode 1. The eight lines following $DEFCHAR are used to define the character using a "character matrix" - this is an eight by eight grid, each element of which represents one "pixel" (dot on screen) of the character.

For a normal character, the matrix consists of an eight by eight grid of commas and stars, representing background and text colours respectively. A possible definition for the character '1' from the upper case/graphics character set is shown below:

```
$DEFCHAR 49,0
, , , , * , , ,
, , , * * , , ,
, , , , * , , ,
, , , , * , , ,
, , , , * , , ,
, , , , * , , ,
, , * * * * * ,
, , , , , , , ,
```

Note that for the redefinition to work, the grid must be entered correctly - spaces or full-stops are not valid. A non-valid grid will give the (fatal) error message

```
BAD CHARACTER MATRIX
```

and execution will terminate.

Multi-colour characters for use in mode 1 (see 11.2 above) are defined in a similar way, but on a 4 by 8 grid, and using comma for background, star for text colour, hash for colour 2, and plus for colour 3. Thus if you wanted to define character '1' in multi-coloured stripes for some perverse reason, you could use

```
$DEFCHAR 49,1
, , * ,
, * * ,
, , # ,
, , # ,
, , # ,
, , + ,
, + + +
, , , ,
```

This character could then be displayed in mode 1 only by selecting a secondary colour (Commodore key plus number), then typing a '1'. Note that only the upper case/graphics set 1 would be affected, and only the normal (not reverse) character.

Note the following further points:

1. The default set does not use pixels along the bottom and the left side of the character to allow for spacing between characters; user-defined character sets should usually do the same.

2. When branching between modules, the character set is left in whatever state of redefinition it was in. Thus modules which do nothing other that redefine characters can be built and

"chained" into other modules as and when needed.

3. In modes 0 to 2, if a character is redefined while text is being displayed on the screen, then ALL occurances of that character being displayed will immediately change to the new definition. This can be used to achieve interesting animation effects; however care is needed to stop it happening by accident, particularly when JOINing or SCROLLing frames into each other.

13] High resolution graphics

Modes 0 to 2 discussed above are **text modes** in which the basic unit of screen display is a **character**, usually consisting of an 8 by 8 matrix of "pixels" (screen points) as discussed above. Mode 3 is a **graphics mode** in which the basic unit of display is a **pixel** - ie a single point on the screen, which can be addressed directly.

A limitation of the 64 is that **colours** are always character-based - ie it is only possible to have one text colour within an 8 by 8 pixel character position, despite the fact that the screen is otherwise being addressed on a pixel-by-pixel basis. A "multi-colour" hi-resolution mode is supported by the 64 hardware; however this mode (mode 4) is not currently supported by Microtext, as it leads to problems with text.

The main disadvantage of mode 3 is that it is comparitively slow - the 64 has to work much harder to look after the display on a pixel by pixel basis, and this makes scrolling in particular painfully slow. In general, it is suggested that SCROLL mode is avoided when using high-resolution graphics, and that the display is handled on a page by page basis.

Assuming this can be lived with, high-resolution mode has many advantages:

1. ALL mode 0 Commodore-style graphics supported, including user-defined characters.

2. BBC-compatible high-resolution $MOVE, $POINT, $LINE and $TRIANGLE also supported. Can be used alone, or in conjunction with normal CBM graphics!

3. Supports 512 characters simultaneously, and has effectively unlimited user-defined character and extended background capabilities! Frames written in upper/lower case mode can be JOINed onto frames in upper case/graphics with no problems; $CHARACTER or $DEFCHAR can be used in the middle of a frame **without** affecting characters already printed; background colour can also be changed ($GLOGIC) without affecting the rest of the screen.

4. Runs with NO loss of memory - 24K of module space is still available.

13.1] Selecting high resolution mode

High resolution mode is selected in Run or Test mode only, by

```
$MODE 3
```

This command will cause the following actions to take place:

1. The current text screen is thrown away.

2. The high-resolution screen ("bitmap") is initialised, and cleared to the current background colour, as last specified using $SCREEN.

3. The origin for graphics plotting is set to the bottom left of the screen - see $ORIGIN below.

Once selected, high resolution mode remains active until another $MODE command, or until return to Command mode by some means, such as end-of-run, or RUN/STOP-RESTORE. The current high-resolution screen is then lost.

13.2] Hi-res points, lines and triangles

Impressive hi-resolution effects can be created by establishing an overall frame layout using text and low-res graphics characters (edit in mode 0), then displaying the frame in mode 3, and adding detail using the $MOVE, $POINT, $LINE and $TRIANGLE commands. These commands function only in mode 3; if encountered in other modes, they will be ignored.

These commands have been designed for maximum convenience transferring high-resolution material between CBM, BBC and Apple versions of Microtext. Points are plotted to a "graphics screen" of 1280 points horizontally by 800 vertically; the system then "divides down" approprately when plotting, to fit an actual screen resolution of 320 by 200 pixels in this case.

Hi-res screen positions are specified relative to a "graphics origin" corresponding to position (0,0). This starts at the bottom left of the screen, but can be moved elsewhere - see $ORIGIN below.

$Move and $Draw

Graphics plotting is performed by a "graphics cursor" or "pen"; this can be moved to a new position relative to the graphics origin using

```
$MOVE <X-coordinate>,<Y-coordinate>
```

or used to draw a line from its current position to a new position using

```
$DRAW <new-X>,<new-Y>
```

X and Y coordinates are always given relative to the graphics origin, and can be anywhere from -32768 to 32767. Note that this means the pen can be moved off the visible screen, which usually corresponds to X-values from 0 to 1279 and Y-value from 0 to 799. Points drawn off the visible screen obviously do not appear and

are said to be "clipped"; the ability to draw lines to a point off the visible screen is very useful in certain perspective effects.

As an example, the following 1-frame program prints a message to the screen, then draws a non-rectangular border round it:

```
*10
$MODE 3

              HELLO THERE!

$MOVE 448,784
$DRAW 864,784
$DRAW 832,720
$DRAW 416,720
$DRAW 448,784

=$END
.....................................
```

(To get this lined-up correctly, leave one blank line then 14 spaces before "HELLO THERE!".)

$Point

For fine detail, single points (1 pixel) can be plotted using the command

```
$POINT <X-coordinate>,<Y-coordinate>
```

This moves the pen to the point specified (relative to the graphics origin as usual) and plots a single point there. For example, to create a rather silly border of just 4 points:

```
*10
$MODE 3

              HELLO THERE!

$POINT 448,784
$POINT 864,784
$POINT 832,720
$POINT 416,720

=$END
.....................................
```

$Triangle

Any form of polygon may be created by successive use of $TRIANGLE, which has the form

```
$TRIANGLE <X-coordinate>,<Y-coordinate>
```

This moves the pen to the point specified, and draws a filled triangle between this point and the last two points visited. For example, to replace the border with a filled box, use

```
*10
$MODE 3
$GLOGIC 4

              HELLO THERE!

$MOVE 448,784
$MOVE 864,784
$TRIANGLE 832,720
$MOVE 416,720
$TRIANGLE 448,784

=$END
.....................................
```

Note the use of two $MOVES to establish the first two points before the first $TRIANGLE, and the use of two $TRIANGLEs to make another shape - in this case a trapezium.

Note that it is possible to plot to the screen without obscuring material already there - in this case the "HELLO THERE!" message. This is achieved by setting an "overlay" mode for plotting, which is done by the command $GLOGIC 4, explained fully below.

13.3] The Graphics Origin - $Origin *

When mode 3 is first selected, the "graphics origin" (0,0) is placed at the bottom left of the screen, i.e. at the bottom of the Microtext command line. If this is inconvenient, it can be moved using

```
$ORIGIN <X-coordinate>,<Y-coordinate>
```

This moves the origin to a new position, at a position (X,Y) relative to the bottom left of the screen. For example, to move the origin two text lines up (above the Microtext command and status lines) use

```
$ORIGIN 0,64
```

or to move it to the centre of the screen, use

```
$ORIGIN 640,400
```

You can even move the origin off the screen if you really want to! Points can then be plotted to the screen with positive or negative X and Y values relative to this new origin, provided all coordinates stay in the range -32768 to +32767; an attempt to plot outside this gives a fatal error

BAD ARGUMENTS

13.4] Plot options - $Glogic

The colours used for plotting and other plot options can be set using the command $GLOGIC (graphics logic):

$GLOGIC <option>,<ink>,<paper>,<ground>

The meanings of the various terms are as follows:

Option

Controls the way that points lines and triangles are plotted to the screen. Options are as follows:

0 Overwrite material already on screen.
4 Overlay existing material, so that it is still visible.

Values from 1 to 3 do nothing, and are currently reserved.

The "overlay" option is known to the initiated as "exclusive-ORing". If GLOGIC 4 is selected then an object can be plotted "on top of" existing objects; if the object is plotted again in the same position, it will be removed, leaving the screen as if nothing had happened. This can be used for (rather slow) animation effects.

The default option selected on $MODE 3 is mode 0 - overwrite.

Ink

This is the foreground colour used for drawing points, lines and triangles. To change colour, use

$GLOGIC <option>,n

where n is a "colour number" from 0 to 15. As mentioned above, note that only one ink colour can be used within an 8 by 8 pixel character position; if a character position is already in use and you plot into it using a new ink, the colour already there will change, so watch it!

The ink colour is initialised to the last used text colour.

Paper

This is the background colour used when plotting, and is also a "colour number" from 0 to 15. The same remarks apply to paper as were made above about ink; only one paper colour can be used within one character position, so be careful.

The paper colour is initialised to the current screen background.

Ground *

This is an optional parameter, used to change the background colour for text and low-res graphics output, without effecting the whole screen, giving an unlimited "extended background" effect. The ground is initialised to current screen background when $MODE 3 is selected but can then be changed thereafter:

```
$GLOGIC 0,6,15,15
```

This would specify overwrite mode, with pen colour 6 (blue) and paper colour 15 (light grey); it would also change the ground to 15, so that further text or low-res graphics would appear on a light-grey background.

(Having changed background like this, you may want to "pad out" new text with spaces so that the new background is displayed up to end-of-line. This can be done using shifted-spaces which are not stripped off the ends of lines like normal spaces. To make shifted-spaces show up differently, try editing in mode 2.)

Note that the background can still be changed across the entire screen using the simple 1-parameter form of the $SCREEN command - eg

```
$SCREEN 15
```

14] Animation and sprites

The Commodore 64 supports up to eight "sprites" - these are independent graphic objects maintained by the hardware, and movable in front of or behind the normal text and/or high resolution display. Microtext on the 64 allows sprites to be defined particularly easily, then moved to any point on the screen, set in motion or even animated using a small number of commands. Sprites can be used in any text or graphics mode.

14.1] Defining a sprite

Sprites are defined very like user-defined characters - see $DEFCHAR above - but on a much larger matrix of 24 by 21 pixels. Thus a sprite definition takes a complete frame of Microtext. For convenience, a command to create a frame of this nature has been added to the editor:

```
SPRITEFRAME frame-number
```

will create a frame consisting of a command $DEFSPRITE followed by a matrix of 24 by 21 commas. This frame can then be edited to create a sprite definition; the frame below defines sprite 0 to be a cube:

```
*10 $DEFSPRITE 0
,,,,,,,,**********,,,,
,,,,,,,*,,,,,,,,,,**,,,,
,,,,,,*,,,,,,,,,,*,*,,,,
,,,,,*,,,,,,,,,,*,,*,,,,
,,,,*,,,,,,,,,,*,,,*,,,,
,,,**********,,,,*,,,,
,,,*,,,,,,,,,,*,,,,*,,,,
,,,*,,,,,,,,,,*,,,,*,,,,
,,,*,,,,,,,,,,*,,,,*,,,,
,,,*,,,,,,,,,,*,,,,*,,,,
,,,*,,,,,,,,,,*,,,*,,,,,
,,,*,,,,,,,,,,*,,*,,,,,,
,,,*,,,,,,,,,,*,*,,,,,,,
,,,*,,,,,,,,,,**,,,,,,,,
,,,**********,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
........................................
```

This is the simple form of $DEFSPRITE, used to define an ordinary single colour sprite. $DEFSPRITE is followed by a sprite number from 0 to 7, followed by a 21 line definition matrix, each consisting of 24 stars or commas, the stars representing regions of colour, and the commas regions where the sprite is

transparent. This must be entered correctly; otherwise Microtext will give a fatal error when the frame is executed

```
BAD SPRITE MATRIX
```

Single-colour sprites

The full form of the $DEFSPRITE command when used to define ordinary single-colour ("hi-res") sprites is as follows:

```
$DEFSPRITE <number>,<colour>,<priority>,<X-exp>,<Y-exp>
{matrix}
```

Meanings of the terms are as follows:

<number>	sprite number from 0 to 7. Sprites of low number always take priority over those of higher number - eg if sprites 2 and 3 overlap, sprite 2 will appear in front.
<colour>	sprite colour number from 0 (black) to 15 (light grey). In the simple form of $DEFSPRITE with only one parameter, sprite colour is defaulted to be the same as sprite number, so sprite 0 is black and sprite 7 is yellow.
<priority>	set to 0 for data priority (sprite moves behind normal text and graphics etc), 1 for spr te priority (sprite moves in front of text). Default is 1 - sprite moves in front.
<X-exp>	X expansion. 0 for normal size (unexpanded); set to 1 to double the size of the sprite in the X-direction (horizontally). Default is 0 - unexpanded.
<Y-exp>	Y expansion. 0 normally; set to 1 to double the size of the sprite vertically. Default is 0.

Thus a sprite definition

```
$DEFSPRITE 0,15,0,1,1
{matrix}
```

defines sprite 0 to be light grey (colour 15), to move behind normal text or graphics, and to be doubled in size in both the X and Y directions.

Multi-colour sprites *

Multi-colour sprites are similar to multi-colour user-defined characters; they use twice as many colours, at the expense of half the horizontal resolution. Each multi-colour sprite has its own "primary colour" corresponding to stars in the grid used to define it; two further colours are also available, represented by hashes and pluses in the grid. These two further colours are the same for ALL multi-colour sprites, so if they are changed, they

may affect sprites already on the screen.

Multi-colour sprites are defined by a further extension of $DEFSPRITE:

```
$DEFSPRITE <n>,<c>,<p>,<X-x>,<Y-x>,<mode>,<col2>,<col3>
{matrix}
```

The first five parameters represent sprite number, colour, priority over text, X expansion and Y expansion as above. The three further parameters are as follows:

<mode>	0 for an ordinary sprite, 1 for multi-colour.
<col2>	second colour, represented by hashes - same for all multi-colour sprites.
<col3>	third colour, represented by pluses - same for all multi-colour sprites.

A multi-colour $DEFSPRITE is followed by a sprite definition matrix; this uses only the first 12 positions on each line (horizontal resolution halved), and uses ',' to represent background, '*' to represent principal colour, and '#' and '+' to represent colours 2 and 3. For example, to define a multi-colour cube:

```
*10 $DEFSPRITE 0,6,0,1,1,1,15,10
,,,,******,,
,,,*++++**,,
,,*++++*#*,,
,,*++++*#*,,
,*++++*##*,,
,******##*,,
,*#####*##*,,
,*#####*##*,,
,*#####*##*,,
,*#####*##*,,
,*#####*#*,,,
,*#####*#*,,,
,*######**,,,,
,*#####*,,,,,
,******,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
....................................
```

This defines sprite 0 to have principal colour blue (6), to move behind text and normal graphics, to be expanded both vertically and horizontally, and to be a multi-colour sprite; it also sets colours 2 and 3 to be light grey (15) and pink(10) for ALL multi-colour sprites. It then defines the sprite to be a cube with blue edges, grey sides and a

pink top.

Changing sprite parameters *

It is sometimes convenient to be able to change sprite parameters such as priority or colour, without affecting the definition matrix. This is achieved using the command **$REDEF** which takes the same parameters as $DEFSPRITE, but which is NOT followed by a definition matrix. Examples are as follows:

```
$REDEF 4,4          - changes colour of sprite 4 to purple (4)

$REDEF 0,10,1,1,1   - redefines sprite 0 to be pink, to take
                      priority over text, and to be fully
                      expanded in both directions.

$REDEF 1,10,1,1,1,1,14,15
                    - as above, but also says sprite is multi-
                      coloured, and sets colours 2 and 3 to 14
                      (light blue) and 15 (light grey) for ALL
                      multi-coloured sprites.
```

Two special forms of $REDEF are supported, to return a single sprite or all sprites to their default settings, which are single-colour unexpanded blocks, with priority over text. These special forms will also move sprites back to their default position at bottom left, and switch them off - see $SPRITE below.

```
$REDEF 4            - restore sprite 4 to default definition
                      and position

$REDEF              - restore all sprites to defaults.
```

Note that sprites are switched off (see below) at end of run, but are left in their current positions and definitions unless put back by $REDEF.

14.2] Positioning a sprite

Sprites can be placed anywhere on or off the screen; a sprite on the screen can be switched on (visible) or off (invisible).

Sprite positioning is performed relative to the same "graphics screen" used for hi-resolution plotting; the screen is considered as being 1280 points wide by 800 points high, with an origin (0,0) which by default is at the bottom left of the screen, but which can be moved using $ORIGIN - see above.

Sprite positions are specified by giving X and Y coordinates

in the range -32768 to +32767, relative to the current ORIGIN. Note that changing the origin using $ORIGIN effects sprite position even when not in high-resolution mode.

When specifying sprite positions, the coordinates given refer to a point at the top left of the sprite definition matrix. Sprites are initialised to graphics coordinates (0,0) - this means they are tucked under the lower left border of the screen - and are switched off.

Sprites can be moved to another position on or off the visible screen using the $SPRITE command:

```
$SPRITE <number>,<X-coordinate>,<Y-coordinate>
```

If the coordinates given correspond to a position where the sprite is at least partially visible, it will be switched on automatically. For example, the following 1-frame program creates a "staircase" effect using the default sprites, which are solid blocks of colour:

```
*10
$REDEF
$SCREEN 11
$SPRITE 0,100,100
$SPRITE 1,200,200
$SPRITE 2,300,300
$SPRITE 3,400,400
$SPRITE 4,500,500
$SPRITE 5,600,600
$SPRITE 6,700,700
$SPRITE 7,800,800
=$END
.....
```

Note that the sprites are switched off at end of run.

Special forms of $SPRITE can be used to make sprites appear (enabled) or disappear (disabled) at any time, without affecting their position. These are as follows:

```
$SPRITE <n>          - make sprite n visible if on screen
$SPRITE <n>, OFF     - make sprite n invisible
```

Note that if $SPRITE <n>, OFF is applied to a moving sprite (see below), it will not take effect until the sprite comes to rest.

14.3 Animation using sprites

64 Microtext allows the eight sprites to be set in motion or animated, WITHOUT holding up normal Microtext execution! Thus it is possible to maintain an animated sprite display, while at the same time performing other actions, such as prompting the user for input.

Moving sprites

Sprites can be made to move smoothly from their current position to a new position, using an extended form of $SPRITE:

```
$SPRITE <n>,<new X>,<new Y>,<speed>,<wait>
```

Sprite number, and the coordinates of the point to move to are specified as before. The new parameters are as follows:

<speed> - speed of movement in arbitary units, from 1 (slow) to 255 (fast). A speed parameter of zero is equivalent to the simple case of sprite positioning discussed above, and causes the sprite to shift immediately to the new position.

<wait> - This is an optional extra parameter, which can be used to determine what happens if the sprite in question is already moving. In the default case (wait = 0) the new command takes effect at once, and the sprite changes direction immediately. If a wait parameter of 1 is specified however, Microtext will wait for the sprite to finish its current movement before starting the new one; this will hold up Microtext execution.

The following 1 frame program causes sprite 0 to move fairly slowly across the screen:

```
*10
$SCREEN 1
$SPRITE 0,100,100
$SPRITE 0,800,800,5

PRESS ANY KEY

!
=$END
.....................................
```

Notice the way that Microtext continues with printing "PRESS ANY KEY" then prompting for an input, while the sprite is in motion.

Animating sprites *

Several sprites can be combined into a single animated object using the $ANIMATE command, which has the form

$ANIMATE <maximum sprite number>,<delay>

Animation works by rotating a number of similar sprite definitions in a regular manner. Maximum sprite number tells Microtext the number of the last sprite to use in the animation sequence (0 to 7); delay tells Microtext how long to wait between each change and the next, in units of 1/60th of a second (0 to 255). For example, the command

$ANIMATE 3,6

tells Microtext to animate the first four sprites (0 to 3), by rotating the sprite definitions every tenth of a second. An animated object can then be displayed by enabling any or all of sprites 0 to 3; at any given time, each sprite will show a diffenent stage in the animation process, giving an impression of phase. Note that a delay of 0 causes the sprite definitions to rotate once; animation then comes to a halt.

The following example moves a winking eye across the screen:

```
*10 $DEFSPRITE 0
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,*,,,,*,,*,,*,,*,,,*,,,
',,,*,,,*,,*,,*,,*,,*,,,,
',,,, **************,,,,,
',,,**,,,,,,***,,,,,,**,,,
',**,,,,,,*,,,,,*,,,,,,**,
*,,,,,,,,*,,*,,*,,,,,,,,*
',**,,,,,,*,,,,,*,,,,,,**,
',,,**,,,,,,***,,,,,**,,,
',,,,,**************,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
',,,,,,,,,,,,,,,,,,,,,,,,,
'''''''''''''''''''''''''''
```

```
*20 $DEFSPRITE 1
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,, **************,,,,,
,,,******************,,,
,**********************,
************************
,**********************,
,,,******************,,,
,,,,,**************,,,,,
,,,*,,,*,,*,,*,,*,,*,,,,
,,*,,,,*,,*,,*,,*,,,*,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,
.....
*30
$ANIMATE 1,30
$SPRITE 0,0,400
$SPRITE 0,1280,400,5

PRESS ANY KEY

!
=$END
.....
```

Note how easily animation can be combined with sprite movement, as in frame 30 above.

Two special forms of the $ANIMATE command are as follows:

```
$ANIMATE n
```

can be used to change the maximum sprite number in the animation sequence once animation has started. (If $ANIMATE is used in this form before animation has been started, a fatal BAD ARGUMENTS error will result.)

```
$ANIMATE OFF
```

will cause animation to stop immediately.

Advanced animation - $Status *

For advanced animation effects, it is useful to be able to get information about the current position, state-of-motion etc of a sprite. To get information about sprite <n> use

```
$STATUS <n>
```

This returns information as a string of characters in the system variables STATUS - a value of 'EVM' for example would indicate that the sprite in question was Enabled, Visible and Moving when the $STATUS command was processed.

Possible returned characters are as follows:

E - ENABLED: the sprite is currently switched on; for this to be the case, the sprite must either be the visible region of the screen, or very close to the border.

V - VISIBLE: the sprite is in a position where, if it is enabled, it will to some extent be visible.

B - BORDER: the sprite is on or partially under the border.

M - MOVING: the sprite is currently in motion.

D - DATA: the sprite has collided with text/graphics data since the last $STATUS.

S - SPRITE: the sprite has collided with another sprite since the last $STATUS.

The first use of $STATUS for any sprite usually returns 'S' indicating that it has collided with another sprite - this is because the sprites start on top of each other at (0,0). An extra $STATUS can be put in near the start of the program to clear this if necessary.

Examples of returned strings are as follows:

EVB - Enabled, stationary and visible at the border

VD - On screen but disabled, collision with data occured

EMV - Moving, visible and enabled

A typical use of the instruction might be:

```
*100
$STATUS 0              (get status of sprite 0)
?STATUS                (input from variable)
B=120                  (if border then frame 120)
S=130                  (if not border but sprite collision then 130
[BACK ARROW]V=200      (if NOT visible then 200)
                       (else fall through)
.....................................
```

14.4] Notes on animation and sprites *

The following further points should be noted:

1. Moving sprites around interferes with the normal timing of the Commodore 64. This can cause problems when loading from disk, eg when branching between modules. Microtext therefore temporarily switches all sprites off while this is done.

2. Completely changing sprite definitions using $DEFSPRITE, or changing sprite origin using $ORIGIN while sprites are switched on can cause flickering - this is best avoided.

15] Sound

The Commodore 64 contains a sophisticated three-voice sound synthesiser known as the "SID" chip. Microtext gives full access to the facilities of this chip, allowing a range of sound effects from the very simple ($BEEP), to the very sophisticated ($VOICE, $SOUND, $NOTE, $FILTER). This section contains a summary of facilities available but does NOT attempt to be a complete guide to musical theory or sound synthesis - for more information see Commodore documentation such as the 64 User Guide.

15.1] Simple sound - $Beep

In many cases, the only sound effect required is a simple beep, to draw the users attention to some important event such as a warning message. This can be achieved in 64 Microtext very simply:

```
$BEEP
```

15.2] Defining a waveform - $Voice *

Getting more sophisticated effects than $BEEP is usually a two stage process - first you define a "voice" using $VOICE, then you make noises with this voice using $SOUND.

There are three sound channels or "voices" in the SID chip, each of which can be set up independently with its own waveform, its own envelope and its own filters.

Waveform *

The **waveform** determines the overall sound quality or timbre. A choice of 4 waveforms is available for each voice:

0: TRIANGLE wave. Quiet, flute-like.

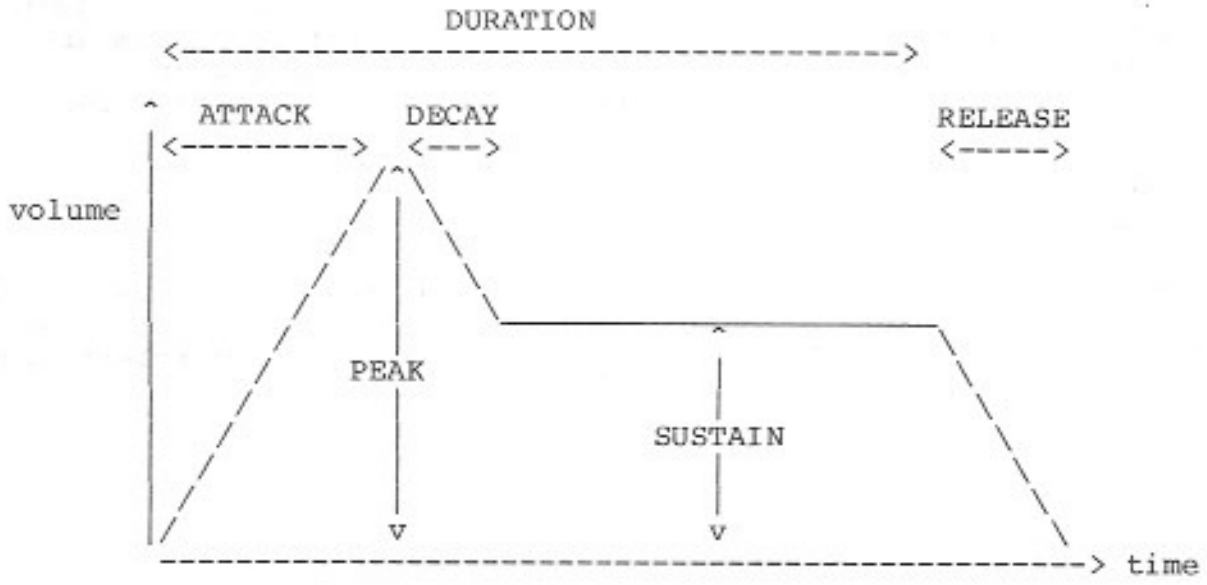
1: SAWTOOTH wave. Strident, saxaphone-like.

2: SQUARE wave. Bright, brass-like. Can be varied (see $FILTER) from a thin to a much fuller sound.

3: NOISE. A hissing type of sound, useful for special effects (ocean waves, pistol shots), and to add texture to other waveforms.

Envelope *

The way that a musical note (such as a piano note) varies with time can be represented approximately as follows:



When the pianist hits the keyboard, the sound first of all builds rapidly to a peak volume ("attack"). It then falls to a quieter level ("decay"). It then stays more-or-less constant until the pianist releases the key ("sustain") - this isn't really true, but in most cases it gives a reasonable approximation. Finally it dies away to nothing ("release"). Peak volume and duration are determined by how hard and for how long each note is played, and are specified on a note-by-note basis using $SOUND. The other parameters - attack, decay, sustain and release - can be taken as more-or-less constant for a given instrument, and are specified using $VOICE.

Filter *

Once a waveform and envelope have been established, further "shaping" can be applied to a voice by applying "filters", and other special effects. These are trickier facilities which tend to vary in effect from SID-chip to SID-chip; however, if you want to experiment, see $FILTER below.

Using $Voice

A voice is defined in CBM Microtext as follows:

```
$VOICE <number>,<waveform>,<attack>,<decay>,
       <sustain>,<release>,<filter>
```

Meanings of the terms are as follows:

number voice number from 0 to 2; invalid numbers produce a fatal error 'BAD ARGUMENTS'

waveform select 0 (triangle), 1 (sawtooth), 2 (square) or 3 (noise)

attack Time to reach peak volume specified in $SOUND command. Select 0 (immediate) to 15 (slow).

decay Time to reach sustain volume (see below). Select 0 (immediate) to 15 (slow).

sustain Volume of sound at sustain level from 0 (silent) to 15 (loud). Should usually be less than the peak volume specified in the $SOUND command.

release Time taken for note to die away, after completion of its duration as specified using $SOUND. Select 0 (immediate) to 15 (slow).

filter Use values from 0 to 7 to select various filter options and other special effects - see $FILTER below. Different settings are used to select different combinations of filter, "synchronisation" and "ring modulation" (see $FILTER):

VALUE	Filter	Sync	Ring mod
0	OFF	OFF	OFF
1	ON	OFF	OFF
2	OFF	ON	OFF
3	ON	ON	OFF
4	OFF	OFF	ON
5	ON	OFF	ON
6	OFF	ON	ON
7	ON	ON	ON

Straightforward values to try are 0 (filter & effects all off) and 1 (filter on). The default filter softens and "smooths out" the sound considerably.

Exact values for $VOICE parameters are best obtained by experimentation; however if you want precise timings, see Commodore documentation.

Note that $VOICE should normally be used to define a voice before notes are sounded; trying to use it to change the waveform of a sound currently in progress is not recommended.

Microtext sets all voices to a simple default at start of run. This is waveform 2 - square wave, attack 1 - fast, decay 3 - medium fast, sustain 8 - medium, release 3 - medium fast, filter 0 - off. The default voices can be used to experiment with

$SOUND without bothering with $VOICE.

15.3] Making a noise - $Sound *

Having defined a voice (or decided to use the Microtext defaults), it is possible to make noises using $SOUND:

$SOUND <voice number>,<peak volume>,<pitch>,<duration>

These values are as follows:

voice Which voice to use (0 to 2).

peak Peak volume from 0 (silence) to 15 (loud).

pitch Pitch (frequency) from 0 (very deep) to 65535 (very high). Pitch values for eight octaves of musical notes are as follows:

octave->	0	1	2	3	4	5	6	7
C	268	536	1072	2145	4291	8583	17167	34334
C#	284	568	1136	2273	4547	9094	18188	36376
D	301	602	1204	2408	4817	9634	19269	38539
D#	318	637	1275	2551	5103	10207	20415	40830
E	337	675	1351	2703	5407	10814	21629	43258
F	358	716	1432	2864	5728	11457	22915	45830
F#	379	758	1517	3034	6069	12139	24278	48556
G	401	803	1607	3215	6430	12860	25721	51443
G#	425	851	1703	3406	6812	13625	27251	54502
A	451	902	1804	3608	7217	14435	28871	57743
A#	477	955	1911	3823	7647	15294	30588	61176
B	506	1012	2025	4050	8101	16203	32407	64814

duration Duration of note from start of attack to start of release, in units of 1/60s. Range is from 0 to 255; a value of 0 causes the sound to continue forever.

$SOUND is processed "concurrently" by Microtext in a similar way to sprite movement and animation, i.e. the $SOUND command starts the sound off, but Microtext execution then continues while the note is being sounded. However, if another $SOUND is encountered while a note is playing, Microtext will wait for the first sound to finish before starting the next one - unless the first sound has been set to sound indefinitely (duration 0)! This does hold up Microtext execution. Note that currently playing sounds will be cut short at end of run, if you use $BEEP, or if Microtext needs to access disk or tape, eg when chaining between modules.

A simple example is as follows:

```
*10
$VOICE 2,0,0,3,11,3,0
$SOUND 2,15,2145,120
.....................................
```

The $VOICE command sets up voice 2 to be a triangle wave (flute-like), with instant attack, sharpish decay (3), quite high (level 11) sustain, rapid release, and filter off. The $SOUND command selects voice 2, sets the peak volume to 15 (maximum), sets the pitch to 2145 (MIDDLE C), and and the duration to 120 - 2 seconds. Further $SOUND commands would typically follow, using the same $VOICE parameters.

$VOICE and $SOUND are also used for sound effects, typically using waveform 3 (noise). Examples are as follows:

```
*10 $COMMENT sawing wood
$VOICE 0,3,9,1,0
$SOUND 0,15,2145,10
!
=20
.....................................
*20 $COMMENT dark & stormy night
$VOICE 0,3,15,15,15,15
$SOUND 0,12,15200,255
!
=30
.....................................
*30 $COMMENT hammer on metal
$VOICE 0,3,1,1,5,9,0
$SOUND 0,15,1000,10
!
=$END
.....................................
```

Note that some "background" noise continues to come from the SID chip even after completion of the release phase of a sound. This can be got rid of by setting pitch and volume settings for the voice n in question to zero, using

```
$SOUND n,0,0,1
```

15.4] Playing chords - $Note *

Voices can be set "pending" by using $NOTE; the note specified is then played together with the next sound output using $SOUND. This can be used to form chords of up to three notes, using the three voices. The syntax for $NOTE is as follows:

```
$NOTE <voice number>,<peak volume>,<pitch>
```

(Note the absence of a duration parameter - this is controlled by

the $SOUND command which "sets off" the pending note.) An example is as follows:

```
*10
$NOTE 0,15,5000
$NOTE 1,15,10000
$SOUND 2,15,20000,20
!
=$END
.....................................
```

This sounds a chord of three voices, for a duration of a third of a second (20/60), all voices at full volume.

15.5] Special effects - $Filter *

Special effects selectable using the SID chip are filters, ring modulation, and synchronisation. Filters allow a waveform to be "shaped" by suppressing or enhancing certain frequencies; ring modulation and synchronisation are special effects in which the output of voice 2 (which must be a triangle wave) is used to control or "modulate" another voice. These facilities are tricky, and not guaranteed to work the same on all SID chips, so for most authors they are probably best avoided. However, the following (rather technical) summary is given for the sake of completeness.

Filter and effect parameters set using $FILTER are applied to all voices which have opted to use them, using the <filter> parameter in the $VOICE command.

There are three main forms of filtration: high-pass, low-pass and band-pass. These may be combined; for example, combining a low and high pass filter forms a "notch" filter. Additionally, an option exists to suppress direct output of voice 2 if it is going to be used for ring modulation or synchronisation.

The $FILTER command is as follows:

```
$FILTER <Cut-off pitch>,<filter type>,<resonance>,<pulsewidth>
```

The meanings of the terms are as follows:

Cut-off — Determines the frequency at which the filter takes effect - range 0 to 2047. Exact values best found be experimentation; or see Commodore documentation.

Type Determines filter type - range 0 to 15 as below:

TYPE	Low	Band	High	Voice2
0	OFF	OFF	OFF	ON
1	ON	OFF	OFF	ON
2	OFF	ON	OFF	ON
3	ON	ON	OFF	ON
4	OFF	OFF	ON	ON
5	ON	OFF	ON	ON
6	OFF	ON	ON	ON
7	ON	ON	ON	ON

8-15 As 0 to 7, but with voice 2 off.

Complex filters are created by selecting appropriate combinations of low, band, and high-pass. Type values from 8 to 15 with voice 2 off are used for ring modulation etc, where voice 2 is not directly output, but is used instead to modulate or control other voices.

Resonance A value from 0 to 15 which controls the "sharpness" of the filter. Again, best set by experiment; or see Commodore documentation.

Pulsewidth A value from 0 to 4095 which controls the exact shape ("mark-space ratio") of the squarewave which is waveform 2. The middle value of 2048 will produce a true square wave in which the space and mark times are of equal duration. Lower values shorten the mark relative to the space and larger values vice-versa. This is also best programmed by experimentation.

Note that it is possible to change a note which is currently being sounded using $FILTER, provided you don't also try to change the waveform using $VOICE!

16] Miscellaneous facilities *

The following facilities are rather specialised, and will not be of interest to most Microtext authors.

16.1] User port control *

Microtext contains two commands for user-port control, giving a simple (if limited) way of controlling external devices like special keyboards, video players etc. These commands are mainly useful for experimentation; sophisticated control of external devices will usually make use of extension commands - see below.

The Microtext USET and UWAIT commands allow any of the eight data lines (0 to 7) on the 64 User Port to be used either as inputs or as outputs. The port also has two "control" lines; these can be accessed directly if necessary using $PEEK and $POKE - see below.

Microtext configures all lines as inputs by default. To use a line for input, use the UWAIT command:

```
UWAIT n        wait for line n (0 to 7) to go high (+5V)
UWAIT n,OFF    wait for line n (0 to 7) to go low  (0V)
```

A line can be reconfigured for output and set high or low by using the USET command. DON'T experiment with outputting to external devices using this command unless you are sure you know what you are doing; if you try to use a line for output while an external device is trying to input to it you can blow a chip on the 64, so watch it!

```
USET n        configure line n for output & set it high (+5V)
USET n,OFF    configure line n for output & set it low  (0V)
```

16.2] Extension commands *

Some versions of Microtext allow additional facilities to be added to the system by "fetching" extension commands; this effectively adds new command mode and run-time commands to Microtext. A set of extension commands stored on disk or tape in an appropriate format can be added to Microtext by the command

```
FETCH filename
```

The extra commands added can then be listed by typing

```
EXTRA
```

For more information on extension commands, including file formats etc, see separate technical documentation.

16.3 Peek and Poke *

As a development aid for experienced programmers, some versions of 64 Microtext contain commands allowing the contents of 64 memory to be examined or altered directly; in Microtext (unlike BASIC) a very wide range of effects can be obtained WITHOUT doing this, so unless you are absolutely sure you know what you are doing, avoid these commands completely!

In command mode, to examine the contents of a machine address in the range 0 to 65535, type

```
PEEK address
```

This will return the address contents on the command line. Alternatively, it is possible to examine the contents of machine addresses (such as VIC or SID chip registers) in run mode by

```
$PEEK address
```

This will return the value in a specially-created variable PEEK.

In command mode, to change the value of a machine address, use

```
POKE address,value
```

where value is a number from 0 to 255; in run mode use

```
$POKE address,value
```

Indiscriminate use of this command will mean you have to switch the machine off and back on again, then reload Microtext. Don't say we didn't warn you.

Appendix A - Module portability

Microtext modules can be transferred between Microtext and a word processor, or between versions of Microtext running on different computers.

AA.1] Using modules created on other systems *

It is possible to use other systems such as word processors to create modules for use with Microtext, provided that the files are correctly set up and that the frames are in the correct format. Note that it is not possible to use normal word processors to create or edit modules containing graphics characters. A word processor may be useful for producing a first draft, which should be output as a text file with an appropriate name (e.g. NEW.STA) for use with Microtext.

Frame structure *

The following rules concerning the structure of frames must be observed:

1. A valid executable frame must start with a * character followed by a valid frame number, and be terminated by a row of two or more dots. A frame which does not start with a * will be treated as a comment and ignored by the interpreter; a frame which starts with a * not followed by a valid frame number may prevent the module from being used.

2. The first five characters of the first line of the frame are reserved for * and frame number, and should not be used for any other purpose.

3. The delimiting dots at the end of the last frame of the module must be followed by a return character. There should be no further return characters following this.

Module names *

Other versions of Microtext may use longer program and module names. Long filenames are accepted on the Commodore 64, but only the first sixteen characters are significant.

Module storage *

Microtext modules are stored in Commodore's version of ASCII (PETSCII), including special characters for graphics, colour changes etc; see Commodore documentation.

The Microtext authoring system does not employ any special characters to mark End-Of-File. However, the authoring system

will ignore any number of special characters at the end of a module, provided these do not include carriage return, $0D.

The authoring system is able to load a module written in straightforward PETSCII. However, normally when modules are typed into the authoring system, simple text compression is applied in order to save space in RAM and on disk. The text compression is as follows:

1. Three or more spaces are replaced by a byte $01, followed by a byte giving the number of spaces minus one, in the range 2-255.

2. Three or more other repeated characters are replaced by a byte $02, followed by the PETSCII code for the character, followed by a byte giving the number of these characters minus one, in the range 2-255.

If a Microtext module has been created on another system (e.g. by a word processor) and it is desired to compress it, then this can be done as follows:

1. Load the module into the authoring system.

2. Use the cursor down key in command mode to step through the module frame by frame. For each frame, press RUN/STOP once to enter Edit mode, and again to return to Command mode. This will cause the text to be compressed.

3. Save the module back onto tape or disk.

If it is desired to de-compress a Microtext module in order to work on it with a word processor, then this can be done by the following short BASIC program:

```
10 REM FILE DECOMPRESSOR FOR CBM 64
20 REM
30 PRINT "MICROTEXT FILE DECOMPRESSOR"
40 PRINT:INPUT "FILE NAME",F$
50 PRINT:PRINT "DECOMPRESSING ";F$
60 OPEN 7,8,7,F$:OPEN 8,8,8,F$+"/D,S,W"
70 GET#7,A$:S1 = ST:IF A$ = "" THEN A$=CHR$(0)
80 A=ASC(A$):IF A>2 THEN PRINT#8,A$;:GOTO 120
90 B=32:IF A=2 THEN GET#7,A$:B=ASC(A$)
100 GET#7,A$:A=CHR$(A):S1=ST
110 FOR X=0 TO A:PRINT#8,CHR$(B);:NEXT
120 IF S1=0 THEN 70
130 CLOSE 7:CLOSE 8:PRINT "DECOMPRESSED TO ";F$;"/D"
```

AA.2] Writing portable modules *

If it is likely that modules created on the Commodore 64 may be run on other systems, then certain precautions should be observed to simplify transfer.

Text area *

The text area in each frame should be limited to 20 lines of 40 characters, which is compatible with a wide range of other machines.

Graphics *

Care should be taken with the use of graphics if a module is to be converted to run on another system. In general, you can only rely on text being transferred. However, efforts have been made to make high resolution graphics using $MOVE, $LINE, $POINT and $TRIANGLE as portable as possible between CBM, BBC and Apple versions of Microtext; modules using these facilities can usually be transferred with only a small amount of adjustment.

(Users transferring material from the BBC to the 64 should note that the 64 supports fewer lines, with the result that the CBM "graphics screen" is 1280 points wide by 800 points high, as opposed to 1280 by 1024 on the BBC. This means that points at the top of the BBC screen will tend to be "clipped" off the top of the 64 screen. This is unfortunate, but necessary to keep the "aspect ratio" the same, so that objects stay more or less the same shape!)

Module names *

Certain module names are reserved for specific purposes in some implementations of Microtext and should be avoided when creating portable modules:

LIBrary	list of packages
CATalogue	list of programs in a package
CONtents	program contents
INDex	topic index (alphabetical)
HELp	help module (optional)
DIRectory	list of modules in a program
SUMmary	summary for current user
SYS, BAD	reserved names (for PDP-11 compatibility)

Abbreviation of run-time commands *

Although it is possible to abbreviate some run-mode commands to one or two characters, abbreviations of less than three characters should be avoided as the minimum abbreviations may differ on other implementations of Microtext.

Appendix B - CBM Microtext summary

The following keywords/commands/instructions are recognised by Microtext 64. Commands identified as Microtext core commands function on this machine as they would on any other Microtext implementation. Commands identified as specific are unique in effect and/or syntax to the Commodore 64. The use of the pling (!) after a core command signifies that a marginal difference in effect may occur. An example of this is $BEEP which beeps very nicely, but sounds different to a BBC computer beeping.

Angle brackets around the command name signify that the entity desribed is a single or multiple key press. Other conventions used are as follows:

(frame)	frame number	(1-999)
(frame range)	frame number range (frame) - (frame)	({1-998} - {2-999})
(module name)	valid file name	(see note * below)
(module reference)		
	frame number	(same module)
	module name	(new module)
	module name:frame number	
(n)	single valid numeric	(command dependant)
(,x,y,z..)	multiple numerics	(command dependant)
(list)	complex parameter list	(command dependant)
(TEXT)	explicit TEXT as parameter	

AB.1] Microtext frame definition

A Microtext frame consists of four zones:

ZONE NAME	Position	Function
HEADER ZONE	1st line only	Frame number/preliminary command
TEXT ZONE	Variable	Text / Run-time commands
PROMPT ZONE	End of text	Input from console or variable
RESPONSE ZONE	After input	Direction of control within module

All frames have at least a header zone.

Frames with no direction in the respose zone will fall through into the frame immediately following.

A prompt is optional.

Output to the summary is always performed in the response zone.

AB.2] System break

System specific

<RUN/STOP> & <RESTORE>	terminate execution to command level

AB.3] Editor facilities

Editor core

<Cursor keys>	cursor motion within window
<RETURN>	new line
<CTRL-D>	line delete
<CTRL-U>	line delete upwards
<CTRL-R>	insert carriage return
<CTRL-I>	insert newline and scroll
<CTRL-P>	display text cursor co-ordinates

Editor Specific

<RUN/STOP>	enter/exit editor
<HOME>	cursor to header line
<DEL>	delete character
<INST>	insert character space
<f3>	line delete upwards
<f4>	insert newline and scroll
<f5>	auto-repeat toggle
<f6>	colour over-ride toggle
<f7>	screen colour roll
<f8>	border colour roll
<CTRL> & <1 to 8>	set primary text colour
<C=> & <1 to 8>	set secondary text colour
<CTRL-9>	reverse field text ON
<CTRL-0>	reverse field text OFF
<C=> & <SHIFT>	toggle character set
<other>	text characters / console controls

AB.4] Command facilities

Command {mandatory parameter} (optional parameter)	brief description
Command core	
Back	previous frame after interrupt
COMment {frame}	comment {frame}
Continue	continue after execution interrupt
COPy {frame}	copy current frame to {frame}
ERase (frame)	delete (frame) from module
EXIT (!)	terminate system
EXTra	display extra commands (if any)
FEtch {module name}	load and link extension module
Frame {frame}	create or display (frame)
FREe	display character space remaining
Help	display help summary
Help OFF (!)	disable help summary
Load {module name} (*)	load Microtext module
Name {name} (*)	set module filing name
NEw (name)	create new module (name)
Print (frame range)	print (frame range)
Run (module reference)	load and run Microtext module
SAve (module name) (*)	save Microtext module
SUmmary	display summary if any
SUmmary PRINT	print summary if any
SUmmary SAVE (*)	save summary to current media
Test (module reference)	test run module from (frame)
Variables	display all held variables if any

(*) 64 specific parameter formats

File directories pathnames are not supported.
File name can be up to 16 characters in length.
Filenames MUST begin with a letter.
Wildcards in filenames are '?' - any character, and '*' - multiple characters to end of filename.
QUOTES are always ignored.
Summaries are saved to file **SUMMARY** unless another filename is specified.

Command specific

<HOME>	display first frame in module
<CLR>	display last frame in module
<Cursor DOWN>	display next frame
<Cursor UP>	display previous frame

BOrder {n}	set screen border colour
CAT (filename)	catalogue filing media
DISC	set disk filing system
DISK	set disk filing system
DRive {n}	select disk drive n
Mode {n}	set screen mode n
PEek {addr}	display machine address contents
POke {addr,value}	set machine address contents
RESET	alternative EXIT command
SCreen {n}(,x,y,z)	set screen colour parameters
SPRiteframe {frame}	create sprite matrix template
TAPE	set tape filing system
> {DOS command}	DOS-support prefix

DOS support examples:

```
>$   catalogue current media

>S0:filename/matching string
     delete file(s) from current drive 0

>C0:target=0:first,0:second
     copy two files with merge into target file
```

AB.5] Run-time facilities

NOTE: When run-time commands appear on the header line, the $ prefix is not mandatory.

Run-time core

$Allow (options)	interrupt options on run
$CAse (ON/OFF)	case matching toggle
$CLear	clear current text window
$Comment	comment
$End	end of execution
$Fix (line)	fix all lines above (line)
$Help (OFF/frame)	help vector
$Join	join frame to next frame
$Line (line(,column))	position text cursor
$MArgin (n)	set left hand margin to column n
$New (VARIABLES/SUMMARY)	new variables and/or summary
$Pause (CHAR/INPUT) {n}	wait for n centi-seconds
$Random (max)	create variable RANDOM (maximum value)
$Scroll (ON/OFF)	scroll toggle
$SUmmary (SCREEN/PRINT)	summary to screen or printer
$SUmmary SAVE (filename)	summary save to media
$TIme (=0)	create variable TIME (and set to 0)
$Trace (OFF/SCREEN/SUMMARY)	frame trace vector
$Unfix	unfix all lines [= $Fix 0]
$USet {n} (,OFF)	set user port line n high (low)
$UWait {n} (,OFF)	wait for user port line n high (low)

Frame control core

* {frame}	start of frame
..<CARRIAGE RETURN>	end of frame (minimum requirement)

Frame vector core — in first column

= {module reference}	goto frame in module referenced
{match}= {module reference}	goto reference on match after input
= {frame} RETURN (override)	goto frame subroutine
= RETURN (override)	return from subroutine

NOTE: A $command or *summary append may be placed AFTER a frame vector command and will be executed before transition of control to the nominated frame reference.

System variable core

TIME	contains time after $TIME
RANDOM	contains value after $RANDOM

Variable input core - on header line

={variable}	assign input to variable on entry

Variable input core - in first column

!	single character input from console
?	multi character input from console
!=	numeric input from console
?=	multi numeric input from console
! {variable}	single-char input from literal variable
? {variable}	multiple-char input from literal variable
!= {variable}	single-char input from numeric variable
?= {variable}	multiple-char input from numeric variable
!@{n}	input at fill point (single)
?@{n}	input at fill point (multiple)

NOTE: some multiple constructs are permissible, e.g. ?=@1

Variable input core - anywhere on screen

@ {n} {.} (....) fill point (...no of characters)

Text indirection (variable expansion) core

<variable> use variable as parameter or command

String matching core

/	OR
&	AND
+	followed by..
[POUND]	word boundary
[BACK ARROW]	NOT
"	literal
'	exact match only

Numeric matching core

=	equals
>	greater than
<	less than
BACK ARROW	NOT BACK ARROW

Variable assignment core

$({assignment} {,assignment..})

Literal assignors:

= {literal}	equals
&	concatenation

Numeric assignors

= {numeric}	equals
+ {numeric}	plus
- {numeric}	minus
+ {number}	unary plus
- {number}	unary minus

NOTE: Assignment must be performed within a valid assignment command, e.g.

```
$(fred="BANANAS! OH YES")
$(bert="WHAT A LOVELY BUNCH OF "&fred)
$(thing=-100,otherthing=1280)
```

Assignment does not require specific equation:

```
$(thing+2)          will add 2 to the numeric variable 'thing'
$(thing=thing+2)    is also valid
```

Summary control core

```
** {text}  write {text} to summary - unconditional
* {text}   write to summary - conditional on last branch
```

Run-time specific

```
$ANimate {n} (,speed)/(OFF)   sprite animation options
$Beep                         beep
$BOrder {n}                   set border colour n
$CHaracter (param list)       (re)define character
$Draw {x,y}                   draw line
$DEFChar {n,col}{matrix}      define character with matrix
$DEFSprite {n} (,list) {matrix}    define sprite with matrix
$FIlter {param list}          set sound filter
$GLogic {param list}          set graphics colour and logic
$Mode {n}                     set screen mode
$MOVe {x,y}                   move graphics cursor
$NOte (param list)            set sound channel pending
$Origin {x,y}                 set graphics/sprite origin
$PEek {addr}                  read machine address into PEEK
$POInt {x,y}                  draw point
$POKe {addr,val}              set machine address to value
$REdef (param list)           redefine sprite
$SCreen {n} (,a,b,c)          set screen colour values
$SOund {param list}           initiate sound execution
$SPrite {param list}          sprite control options
$STatus {n}                   read sprite condition into STATUS
$TRIangle {x,y}               draw triangle
$Voice {param list}           set sound channel characteristics
$> {DOS command}              DOS-support prefix
```

System variables specific

```
PEEK                          contains value after $PEEK
STATUS                        contains string after $STATUS
```

Appendix C - Microtext error messages

AC.1] Command Mode error messages

CAN'T CONTINUE Caused by CONTINUE without having exited from Run or Test Mode. 'CAN'T CONTINUE' will also be given if in the course of adding new frames, variables have been overwritten.

FRAME EXISTS Caused by an attempt to COPY to an existing frame.

INVALID RANGE Caused by invalid frame number range in ERASE, PRINT or LIST.

LONG FRAME Caused by displaying a frame with more lines than can be edited - probably transferred from another system.

NO MODULE PRESENT Caused by trying to SAVE, RUN or TEST with no module in memory.

NO ROOM IN MEMORY There is no more room in memory for a user-defined command module, a particular screen MODE, or text in the current module.

NO VALID FILE NAME Caused by LOAD etc. without file name.

NO VALID FRAME NUMBER Caused by missing or out of range frame number.

NOT UNDERSTOOD Caused by typing an invalid Microtext command.

NOTHING TO EDIT Caused by pressing RUN/STOP in Command Mode, with nothing to edit on the screen (the FRAME command must first be used to create or display a frame).

Other standard error messages can be obtained from CBM DOS - eg 'FILE NOT FOUND'.

The messages are displayed on the bottom line; pressing any key gets rid of them and causes a return to Command Mode.

AC.2] Run/Test Mode error messages

Fatal errors

The following errors are fatal and can occur in Run or Test Mode. They are displayed on the command line, following which pressing any key will give the 'end of run' message, and return to Command Mode.

BAD ARGUMENTS Caused by missing or out-of-range arguments in a command line - eg trying to define a voice greater than three using $VOICE.

BAD CHARACTER MATRIX Caused by missing or invalid characters in

a character definition matrix following $DEFCHAR. Check especially for rogue spaces or full stops.

BAD MARGIN Caused by trying to set a margin less than zero or greater than the screen width using $MARGIN.

BAD RETURN Caused by missing or out of range return frame number as in FRED->1000 RETURN, or by FRED -> RETURN with nothing to return to.

BAD SPRITE MATRIX Caused by missing or illegal characters in a sprite definition matrix following $DEFSPRITE. Check especially for rogue spaces or full stops.

LINE TOO COMPLEX A line contains more than 100 variables. Normally caused by a variable referencing itself, for example: $(A="<A>"), followed by <A> in the text.

LINE TOO LONG The length of the line when all variables have been expanded exceeds 256 characters. Check the contents of the variables.

NO ROOM IN MEMORY The combination of text, variables, and summary items have filled available memory. Could be caused by running a long program of several modules using different variable names, without $NEW, or by a very long summary.

NUMERIC OVERFLOW A numeric variable has gone outside the allowed range of approximately 5E11 to -5E11.

MATCH TOO COMPLEX The combination of keywords to be matched contains too complicated a combination of operators.

STRING TOO LONG A string has gone beyond the allowed maximum length of 250 characters. Could be caused by repeated string concatenation in a loop.

TOO MANY SUBROUTINES Caused by trying to nest subroutines more than 10 deep.

In addition to the above, errors may be given by CBM DOS - eg FILE NOT FOUND.

Recoverable errors

The following errors are non-fatal, and cause the system to react differently in Run or Test Mode. If the error is encountered in Test Mode, the appropriate error message is displayed on the bottom command line. Pressing RUN/STOP will then cause an exit to the editor at the appropriate frame; the CONTINUE command will then cause execution to resume from the start of the offending frame. Pressing any other key will cause the system to carry on with the appropriate default, as detailed below. In Run Mode, no error message is displayed, and the system immediately defaults

as specified.

BAD ASSIGNMENT Caused by incorrect syntax in an assignment statement. The default is to treat the line as normal text.

BAD COMMAND Caused by incorrect syntax in a command line - for example an incorrectly spelled command. The default is to treat the line as normal text.

BAD FILE NAME Caused by a bad or missing file name, usually in the response zone. The default is to ignore the line.

BAD PROMPT Caused by invalid syntax in the prompt zone (for example ?***). The default is to perform a normal input.

BAD RESPONSE TEST Caused by incorrect syntax in the response zone - for example no characters after the goto sign. The default is to ignore the line.

MISSING FIELD NUMBER Caused by a missing fill-point number in the prompt zone, for example ?@ The default is to treat it as a non fill-point input.

NO SUCH VARIABLE The contents of a pair of angle brackets is not a variable name, for example <ESCAPE> when there is no variable ESCAPE. The default is to leave the line unchanged.

NON-EXISTENT FRAME This message is given when branching to a non-existent frame, or branching past the end of module. The default is to carry on to the next frame in sequence after the non-existent frame, or to end the run.

PROMPT FIELD NOT ON SCREEN A fill-point specified in the prompt zone was either not included in the text zone, or else has scrolled or been cleared off the screen. The default is a non fill-point input.