QL ADVENTURE CREATION TOOL

SPECIAL EDITION
August 1988

Table of Contents

BACKING UP YOUR MASTER ACT DISK AND OTHER WARNINGS

0.0 THE DEMONSTRATION MINI_ADVENTURE

1.0 OVERVIEW OF ACT
- 1.0.1 ACT Notational Conventions
- 1.0.2 The QL CALL bug

1.1 Fixed Limits of the System
- 1.1.1 Initial Planning for Your Adventure

1.2 The Organisation of Messages
1.3 Object and Location data
1.4 The Vocabulary
1.5 The System Programs
1.6 The Component Parts
1.7 Adding Colour and Formatting Screen Text
- 1.7.1 Inserting Colour with the MSGedt Utility
- 1.7.2 Considerations for Changes of Text Colour
- 1.7.3 Split-MODE Screens
- 1.7.4 Formatting Messages

2.0 BEGINNERS GUIDE TO USING ACT
- 2.0.1 The CAPTAIN - ACT's Front End
- 2.0.2 The CAPTAIN and Memory
- 2.0.3 The CAPTAIN's Hot-Key Re-awakening

2.1 Adding a New Location to the Mini_Adventure
- 2.1.1 Editing the Location Message File
- 2.1.2 Editing the Location Data File

2.2 Adding a New Object to the Mini_Adventure
- 2.2.1 Describing Objects in the Text
- 2.2.2 The Object Data

2.3 Producing the Compressed Text File
2.4 Producing the Compiled Program File
2.5 Linking the Modules to Form a Completed Game
2.6 Tips on Using the Utility Programs

- 2.6.1 MSGedt_task
- 2.6.2 TXTcom_task
- 2.6.3 BASasm_task
- 2.6.4 VOCedt_task

2.7 Special Features to Watch Out For
- 2.7.1 Object 0, the Lamp
- 2.7.2 Object 3, the Lighter
- 2.7.3 Special Effects at Locations: Draughts and the Gas Leak
- 2.7.4 Other Special Effects

3.0 A DETAILED DESCRIPTION OF THE ACT SYSTEM

3.1 The ACT System Language

3.2 The ACT System Commands
- 3.2.1 Commands with no Parameters
- 3.2.2 Commands with 1 parameter
- 3.2.3 Commands with 2 Parameters
- 3.2.4 Commands with 3 Parameters

3.3 The Data Structure of the Mini_Adventure
- 3.3.1 Location Messages
- 3.3.2 Object Messages
- 3.3.3 Location Data
- 3.3.4 Object Data

3.4 How the System Programs Work
- 3.4.1 The Player Program
- 3.4.2 The Event Program
- 3.4.3 Considerations for Split-MODE Screens
- 3.4.4 May We Reserve Some Space?

3.5 Interfacing Machine-Code Extensions to ACT

3.6 The ACT On-Line Game Debugger System
- 3.6.1 What Is the Debugger?
- 3.6.2 How Does the Debugger Operate?
- 3.6.3 The Debbugger Commands
- 3.6.4 An Example Editing Session Based on the Mini_Adventure
- 3.6.5 The Debug Session, Step by Step

4.0 ADDING NEW FEATURES TO THE SYSTEM

4.1 Adding a New Word Function to the System

4.2 Changing the Lighter to a Box of Matches
4.3 Using the Routine YES_or_NO_Response
4.4 "Magic Transform" Words
4.5 The Use of Flags to Control Particular Events or Commands
4.6 Arranging for a Time-Delayed Event
4.7 Who's Afraid of the Dark?
4.8 Etcetera, etcetera

5.0 SOME FINAL COMMENTS ABOUT ACT AND WRITING ADVENTURE GAMES

6.0 APPENDIX 1 - THE GRAPHICS DESIGNER
6.1 Overview
6.2 The Additional Drawing Routines QFILL1 and QFILL2
6.3 Commands
6.4 The PIC1 Utility

7.0 APPENDIX 2 - LINKING ILLUSTRATIONS INTO AN ACT GAME
7.1 The Basics: How to Add Pictures to ACT
7.1.1 Adding the Picture Extensions to an ACT Game
7.1.2 Producing the Composite Picture File
7.2 Details of the Utility Programs
7.2.1 SCNcom, the Screen Compressor
7.2.2 GDI_1_task, the ACT / Graphics Designer Interface
7.2.3 GDI_2_task, the Picture Compiler
7.3 Extra Graphics Extensions
7.4 More About ACT's Split-MODE Screens
7.5 Making Alterations
7.6 Additional Comments on Graphics Designer Pictures
7.7 Additional Information

8.0 APPENDIX 3 - ADDING SOUNDS AND FONTS
8.1 What Is SNDedt?
8.2 How Are Sounds Incorporated into an ACT Game?
8.3 Starting SNDedt
8.4 Finding Your Way Around the SNDedt Screen
8.5 How To Use the Editing Window
8.6 SNDedt Commands

8.7 Loading the Machine-Code Module into a Game and CALLing the Sounds
8.7.1 Adding the Sounds Module to the ACT Last Data File
8.7.2 CALLing the Sounds from ACTBASIC Programs
8.8 The Sounds Used in IMAGINE
8.9 Adding Fonts

9.0 APPENDIX 4 - QFILL1 AND QFILL2
9.1 Using QFILL1
9.2 Using QFILL2
9.3 Additional Information about QFILL1 and QFILL2

TABLE 1 - THE DEVELOPMENT TREE OF AN ACT ADVENTURE GAME
TABLE 2 - THE OBJECT PARAMETERS DESCRIBED
TABLE 3 - LOCATION 0 PARAMETERS DESCRIBED
TABLE 4 - THE OBJECTS IN THE MINI_ADVENTURE
TABLE 5 - WORD TERMINATING NUMBERS
TABLE 6 - THE MAIN TABLE AREA IN THE ACT BASE MODULE

BACKING UP YOUR MASTER ACT DISK AND OTHER WARNINGS

Thank you for purchasing the ACT system. We at Digital Precision believe that it is the most powerful system yet released for making adventures come to life on your QL.

It is very important that you create a working copy of the master ACT disk. When using ACT to develop your adventure game you should always use your copy, never the original. To make the copy you can use the WCOPY command. You should use this copy rather than the original for ALL stages of developing your adventure.

Integral portions of the code used in the operation of an ACT Adventure are and shall remain the intellectual property of Digital Precision Ltd and the Author. Users of the ACT system may distribute any games produced without liability of royalty payments of any kind on the understanding that both Digital Precision and ACT will be given credit in any documentation accompanying the game and their names displayed prominently on one or more loading screens included in the game.

Digital Precision would like to point out that many hours have gone into the production of the ACT system. For the convenience of ACT users, there is no requirement for having the original master present in any drive in order to get the system started. We are relying solely on our customers not distributing copies unlawfully. But should we become aware of an act (ACT?) of piracy, we will sue the individuals concerned to the full extent the law permits.

A reward is offered for any information leading to the successful prosecution of individuals involved in the piracy of program material produced by or for Digital Precision.

0.0 THE DEMONSTRATION MINI_ADVENTURE

ACT is supplied with two adventure games. One of these, IMAGINE, is a complete adventure that provides a demonstration of the sort of game it is possible to produce with ACT. The other is the Mini_Adventure. This is, as you would expect from its name, a very small adventure which serves two functions.

1. It provides an additional demonstration of some of the many features of ACT.
2. It serves as a tutorial and also provides a starting point from which you can develop your own adventure games.

You can play the demonstration adventure as soon as you have made the system backups. To start the game, put your copy of ACT in drive 1 and then enter the command:

EXEC_W FLP1_MINI_ADVENTURE

The notes that follow will guide you through the Mini_Adventure and also mention some of the less obvious points that help in solving the game. Figure 1 is a map of the Mini_Adventure to guide you through your first experience with an ACT adventure. While no attempt has been made to make the Mini_Adventure into a mega-game, it is fun and, as small as it is, it contains many of the classic game features and demonstrates the flexibility of the ACT system.

As soon as you move away from the small cave, you will need to find a source of light. Don't waste time with the broken torch, it's only there as a decoy to stop you from finding a working light source.

If you examine the lighter, in particular if you READ it as well, it will tell you how it is operated. The lighter will only get you as far as the location to the north of the small cave (the narrow passage) or to the south (the smelly cave). Elsewhere, the strong draught will blow it out or, if you go beyond the smelly cave, the resulting gas explosion will kill you.

However, all is not lost. You will see some other objects in the small cave. The tool box might contain something useful, but you cannot open the lid until you have found a way of freeing it.

There is a note in the waste bin that may give you a clue. In fact, it is not the message that is helpful but rather the note itself, because paper will burn! Although the draughts will blow out the lighter, they aren't strong enough to extinguish anything more substantial.

You will need the lighter to ignite the note, but don't try setting fire to anything you are holding, since you will go up in smoke, too. The way around this is to leave the note in the bin, which acts as a fire bucket and can safely be picked up while the note within it is burning. The note will only last a short time before it is all burned away, so you will have to be quick to complete the next task.

You will probably have found the block in the narrow passage, and you might have worked out that it is a sponge. You will need the sponge, so pick it up and put it into the bin. Don't worry, it doesn't burn.

Now you can go north (or east) from the narrow passage into the windy cavern where you will find a pool of liquid which you might realise is oil. The oil will help you open the tool box but you cannot pick it up directly; you must use the sponge as a tool.

Commands such as "SOAK UP the OIL in the SPONGE" will work, as will various other command constructions. As you will see, it is important that the sponge is in the bin, since it will not hold the oil for very long before it leaks out.

If the sponge is not in a watertight container when this happens, the oil will just end up back on the floor. Once you have the oil, you can go back to the narrow passage (to the east), where the lighter can be used to provide light as soon as the paper is consumed.

Back in the small cave, you can now free the tool-box by saying OIL the BOX. You will find an electric torch inside which is not broken. You should not have any trouble distinguishing the two torches, so take the good one, switch it on, get rid of any burning objects you are still carrying, and then you can go south, into the smelly cavern and then east to adventure's end.

It is worth mentioning a few of the features used in the Mini_Adventure which may not be quite so obvious. More than one noun, or more than one distinctive adjective, may be used to describe an Object: torches may also be referred to as lamps; one of the torches is ELECTRIC and also OFTEN-READY, while the other is BROKEN.

Less obvious is the size difference between the torches. The electric one is smaller than the broken torch, and so the command TAKE the SMALL TORCH will single out the electric torch. Because size is relative, if there were another torch even smaller than the electric one, the same command would result in that torch being picked up. TAKE the LARGEST torch would also acquire the broken one.

Object containment is maintained for any degree of 'nesting'. You can construct any number of objects, assign them volumes (and lids) and place them one inside another. In addition, each container can hold any number of objects separately, provided it has sufficient volume to contain them all. Objects can also be defined with surface area rather than volume. A table would fall into this category, although only the two notes are like this in the Mini_Adventure; try the command PUT the LAMP on the NOTE, for example.

Objects can be defined as liquid. This automatically alters the description of the object to suit. You would say you can see 'SOME water' rather than 'A water'. It also ensures that the object behaves in a way you might expect of a liquid; for example, you would need a tool to help you pick up a pool of water off the floor.

Objects may be flammable. They will only burn for a minute or so before being consumed (the exact time is random) and, in addition, if they are on, in, or themselves contain another inflammable object, they will also start that object burning after a short while.

The use of some words is optional. In the command, 'PUT the SMALL ELECTRIC LAMP into the TOOLBOX', only the words shown in upper case are used by ACT's "dictionary" (the system's parser). The same result would occur with the command 'PUT SMALL ELECTRIC LAMP TOOLBOX'.

Words that are unimportant for some commands are important in others: the command 'PUT the LAMP on the TOOLBOX' is translated to 'PUT the LAMP into the TOOLBOX' since both are equivalent to the command 'PUT LAMP

TOOLBOX'.

The command 'SWITCH LAMP ON' is not the same as 'SWITCH LAMP', though, since the word ON now describes which state the lamp should be switched to. If the lamp is already on then the first command will result in a message telling you so, and the second will result in the lamp being switched off.

```
                                    +--------+
                                    |        |
                                    |        |
                       ******************    |    *****************
                       * Narrow passage *    |    * Windy cavern  *
                       *                *    |    *               *
                       * Location 2     *----+----* Location 3    *
                       *                *         *               *
                       *  Object 5      *         *  Object 4     *
                       ******************         *****************
                              |                           |
                              |                           |
                              |                           |
?????????????????     +++++++++++++++++                   |
? This location ?     +  Small cave   +                   |
? can be added  ?     +               +                   |
? by following  ?-----+ Location 1    +                   |
? the example   ?     + Objects 0,1,2 +                   |
? in Section 2. ?     +  3,6 and 7    +                   |
?????????????????     +++++++++++++++++                   |
                             |         \                  |
                             |          \                 |
                             |           -------\         |
                             |                   \        |
                             |                    *****************
                             |                    * Small chamber *
                             |                    *               *
                             |                    * Location 4    *
                             |                    *               *
                             |                    *               *
                             |                    *****************
                             |
                             |
                     *****************          *******************
                     * Smelly cavern *          * Adventure's end *
                     *               *          *                 *
                     * Location 6    *----------*  Location 7     *
                     *               *          *                 *
                     *               *          *  Object 8       *
                     *****************          *******************
```

Figure 1. The map of the mini_adventure

The example adventure starts in location 1 (the small cave), which is shown above along with the other 5 locations included. The numbers refer to the internal values used to describe each location in the file LOCN_dta. Objects are marked wherever they are to be found; see Table 4 for a description of these.

Note that location 0 is unused, while location 5 is used as the 'object dump', that is a location with no paths that is used to 'contain' objects not currently in the game.

Here is a list of objects in the mini_adventure:

		Starting Locations	
NUMBER	OBJECT	LOCATION	CONTAINED IN
0	The GOOD Torch	1	2
1	Toolbox	1	0
2	The BROKEN Torch	1	0
3	The GAS Lighter	1	0
4	Oil	3	0
5	Sponge	2	0
6	Waste Bin	1	0
7	Note (Starting Location)	1	6
8	Note (End Location)	7	0

You can see the complete list of words that the Mini_Adventure knows by examining the file WORD_dta (using VOCedt_task). There are 148 words in its current vocabulary.

Here are the main command verbs, along with their word numbers:

QUIT	27	IGNITE	84
LOOK	28	BURN	85
SAVE	33	SET	86
RESTORE	34	FIRE	87
SCORE	35	EXTINGUISH	88
GET	36	SUICIDE	90
TAKE	37	INFORMATION	103
DROP	44	INF	104
LEAVE	45	HELP	105
PUT	46	HEALTH	106
PLACE	47	OIL	107 (Noun also)

PICK	54	SOAK	114
READ	59	WRING	117
INSPECT	64	SQUEEZE	118
DESCRIBE	65	FIX	121
EXAMINE	66	MEND	122
POSSESSIONS	67	REPAIR	123
INVENTORY	68	JUMP	124
BELONGINGS	69	EAT	135
OPEN	70	DRINK	136
CLOSE	71	FEEL	137
SWITCH	74	WAVE	138
PRESS	75	BREAK	139
TURN	76	SMASH	140
LIGHT	83	DESTROY	141
FIND	142		

The number listed refers to the word number in the file. In addition, each word has a numerical tag used to indicate what type of word it is. This tag is purely notional (in order to simplify the operation of some of the sections of the ACT player program). Thus it is possible for a word to be used as both verb and noun, e.g. oil.

The types used so far are:

1. Direction- or movement-related words.
2. "Throw-away" words.
3. Commands.
4. Adjectives.
5. Nouns.
6. Special word types used in "find objects".
7. Swear words.

1.0 OVERVIEW OF ACT

ACT, an acronym for "Adventure Creation Tool", consists of fourteen programs and several data files (the latter constituting the demo adventure, Mini_Adventure) that can be used and combined to form illustrated text adventure games to run on a Sinclair QL. ACT provides the potential adventure writer with a flexible game environment to which almost any number of new features may be added.

ACT is to adventure writing as LOGO is to graphics. ACT adventures are written in a language called ACTBASIC, which has been deliberately designed to complement SuperBASIC, and then compiled with the ACT compiler. Automatic text and picture compression techniques insure that maximum efficiency of memory usage is ensured for both RAM and magnetic storage.

The ACT system provides the ultimate in flexibility. Any number of messages, flags, special conditions or other features may be included in your adventure. The information that defines a game is contained in the various data files, and changes or additions to the game are made by using the utility programs provided.

Although intended mainly as a tool for producing professional-quality Adventure games, the ACT system can be used with equal ease for educational programs - in fact, for any type of program requiring QL interaction with a user's response.

Section 2 of this guide provides step-by step-instructions, partly in the form of a tutorial, that demonstrate each stage of development of an adventure game.

Section 3 is a reference section that includes descriptions of the data files and of the ACT programming language, enabling users who are familiar with the concepts involved to produce their own games immediately without completing the tutorial section.

Section 4 contains example program additions and modifications that illustrate how to add new features to a game.

You have complete control over the presentation of your game. Text can be displayed in a variety of ways, and the full range of standard INK and STRIP colours are supported. Entering colour changes into ACT's message editor shows you your final display as it will appear on the screen.

Included as a standard part of ACT's display system is the unique split-mode screen system, offering you the option of having both MODE 4 and MODE 8 displayed on the screen at the same time. This powerful and flexible feature will enable you to have eight-colour capability for your illustrations and four-colour resolution for your text.

ACT's graphic module, the Graphic Designer, includes sophisticated features, such as two efficient screen compression techniques which will enable you to produce complex screens in as little as 500 bytes or less. Other unique Graphic Designer features include fancy text, three different kinds of shape FILL and many more.

1.0.1 ACT Notational Conventions

In order to provide the greatest degree of flexibility and convenience, all of the ACT utilities include system defaults for drives and file extensions which can be edited to any sensible alternative.

The BOOT and CAPTAIN (see Section 2.0.1) are run from flp1_. The CAPTAIN in turn will expect to find the ACT utilities on flp1_. The various utility programs will, by default, expect to find data files on flp2_. However, all these defaults can be altered at run-time if required.

Various conventions for file extensions have been used in setting up the ACT system. These conventions aren't obligatory but strongly recommended, since they will probably avoid possible confusion while you are learning about ACT. The following table summarises the main utility programs and the various data file conventions that are normally used.

	Input	Output
Utility	Extension	Extension
MSGedt_task	_msg	_msg

TXTcom_task	_dta	_dta
LOCedt_task	_dta	_dta
VOCedt_task	_dta	_dta
LSTedt_task	_dta	_dta
BASasm_task	_prog	_dta
LINKER_task	_dta	User-defined

1.0.2 The QL CALL bug

If you have a JS or later QL, you can ignore this section. If, however, you have a JM or AH version of the QL, you may suffer from a potential problem - the infamous CALL bug. This bug can cause your QL to crash when large programs use the machine-code CALL command.

This problem can occur for any large SuperBASIC program that uses CALL, even when compiled by TURBO or other SuperBASIC compilers, unless a software "patch" is applied before it is run. The use of some toolkits, such as the Supercharge or TURBO extensions, will provide such a patch, as will some versions of the TK2 Toolkit.

We have supplied a small patch (172 bytes) for users who may not have a toolkit available.

Currently, the BASasm_task utility will cause problems with the CALL bug, as may future releases of other ACT utilities. The BOOT program will automatically load the CALL patch if your QL is either a JM or AH release. This routine can be used separately for other programs as well.

If you start ACT from the BOOT supplied, then the system front end, the CAPTAIN, is started and will provide information about the various system utilities. Alternatively, if you wish to use the ACT utilities independently, we suggest that the CALL patch is loaded first with the command:

a=RESPR(172) : LBYTES flp1_CALL_BUG_FIX,a : CALL a

The use of the CALL command in ACTBASIC (see Section 3.5) will not be affected by this bug, no matter how big your game is. Any ACT adventure game will run correctly on any QL version.

1.1 Fixed Limits of the System

The working limits of an ACT adventure are, in part, restricted to the available QL memory - whether that of the originator of an adventure or, where the game is sold commercially, the memory available to the player.

Generally, the fixed limits of the system, where memory is not the major consideration, are:

- A vocabulary of up to 4096 words, each up to 20 characters in length.
- 98301 messages.
- 255 Locations.
- 256 Objects.
- Each Object or Location may include up to 127 parameters.
- Each parameter may have up to eight flags.
- Real-time event timing (one per second to 4-1/2 hours - but can be increased).
- Either MODE 4 or MODE 8 displays are supported. In addition, graphics in MODE 8 and text in MODE 4 can be displayed at the same time. The compromise made for this feature is that the response time of the game to user's commands is noticeably increased.

Each object or location can have up to 127 parameters, each of which can be used either as a single 8-bit number (values 0 to 255) or as eight separate flags (numbered 0 to 7).

1.1.1 Initial Planning for Your Adventure

As for any computer endeavour, careful planning should be made on paper, prior to beginning the actual programming. All too often, an otherwise good story line fails to achieve its desired effect because of insufficient initial planning.

The first step for any adventure is to plan a general outline of the main features of the game - a story line. This will provide a check for story and character continuity. Lists should be made for messages, the vocabulary the adventure is expected to respond to, the objects and their various attributes, and the cast of the characters you will be using - their names, descriptions and special characteristics (strengths and weaknesses).

The next stage is to draw a map of the options available to the intended player. Notes should be included in each of the location boxes with regard to any objects available, any hazards present and where any objects are interactive with the player. Lines connecting the boxes should be indicated as direct or "hidden" routes.

Three-dimensional mazes should be planned on plastic sheets as overlays, with the UP or DOWN routes clearly marked. If "magic" is used to gain access or transport to another part of the "game board", this should also be clearly marked on the "layer" of the planning overlay to which it applies.

Since colour plays an important part in the visual impact on the player, ACT has provided all of the tools you will need to present text in an effective manner. INK and STRIP may be used to emphasise portions of text; ACT's sophisticated illustration tool, Graphics Designer, provides a memory-efficient way of including text-related illustrations in your game; and split-mode screen displays provide the maximum colour range for text/graphic adventures.

Once the plan for the game is completed, you are ready to begin building your game with ACT.

1.2 The Organisation of Messages

ACT stores text used in a game in separate 'messages'. These messages are divided into three groups:

- Location - used to describe each location in the game.
- Object - used to describe objects and any messages written on them, or any other required text information.
- General message data - used for any other messages.

Messages in each group are given numbers, from 0 up to a maximum of 32767. Each message may be of any length up to a limit of 1600 characters. It is worth noting that messages are not divided into individual lines until the completed game actually prints them on the screen.

Each message is a string of words and will be formatted to fit the display window as the game prints it, automatically adjusting to the viewing MODE. When a new message is created by the MSGedt_task program, don't attempt to divide the message up into lines; just type words sequentially, and ACT will sort the display out for you.

1.3 Object and Location data

Objects and locations in the game are characterised by the information in their respective data files.

For example, the way that individual locations are connected to the others in the game is controlled by the first 10 parameters for each location in the file called LOCN_dta.

Each parameter is used to control a particular direction, while the value of the parameter describes where the direction leads. If a value is 0, that direction is not a valid one. Any other value (1 to 255) represents the location that the particular direction leads to.

The object data works in a similar way, except that the various parameters are used to describe the properties of each object.

For example, parameters 0 to 9 are used to control which words in the vocabulary can be used to describe the object, while parameter 10 describes where the object is.

A full description of all the parameters is given in Section 3, while the tutorial in Section 2 also provides examples explaining the use of the parameters.

1.4 The Vocabulary

Words that a completed adventure game knows are contained in a data file which, for the "Mini_Adventure", is called WORD_dta. Words can be of any length up to 20 characters, and each may be characterised as a verb, a noun, etc. Each word has an associated number, which is the position of the word in the data file. The first word is number 0, the second number 1, and so on.

The WORD_dta file contains about 150 words and may be altered or added to by the use of the utility VOCedt_task.

1.5 The System Programs

There are two programs, the 'Player' and the 'Event', both written in the ACTBASIC language, controlling the operation of an adventure game. The language used is similar to an assembly-level language in some ways, but is considerably easier to use and incorporates many features normally found only in high-level languages.

The structure of the ACT language is similar to SuperBASIC, so that programs may be created and edited by the normal QL commands EDIT, LOAD, SAVE, RENUM and AUTO.

They will not run as SuperBASIC programs, however, and must be compiled by the ACT utility BASasm_task before they are combined with the other data files to form the completed adventure. 'Player' is used to control what happens when the player enters a command.

It is the task of this program to make sense of whatever the player says and to respond accordingly.

'Event' is used to control activities such as the movement and actions of creatures, or, as in the Mini_Adventure, the control of any time-dependent function such as burning objects.

The 'Event' program is called repeatedly at a rate of about once a second, while the game is running. Details of the ACT system language are provided in Section 3; Section 4 contains example additions to the two system programs.

1.6 The Component Parts

To see how an adventure is constructed, it is suggested that you initially work backwards from the completed game through each stage of its creation.

Table 1 is a summary of the process that creates a text adventure. Starting at the top of Table 1, the demo game, called Mini_Adventure in the examples, is created by the ACT linker utility program called LINKER_task.

When LINKER_task is run, it will prompt for seven input files in turn. These are shown in Table 1 and are required in the sequence given (1 to 7).

- The first file is called ACT, the base module of the system. An alternative file, called ACT_short, is included with the ACT system. This excludes on-line error messages and the system debugger, saving about 3900 bytes of memory in the completed game, and can be used when testing of the game has been completed.

- The second file contains the messages. In the Mini_Adventure it is called TEXT_dta. TEXT_dta is constructed from the three text source (location, object and general message data) files, by the text-compression utility program TXTcom_task.

  TXTcom_task will prompt for the three input files required. The output file contains the text included in the three input files, but in a compressed form.

  This serves two purposes: for one thing, the space used in the final game by messages is reduced considerably, and it is possible to include a great deal more text in a given amount of memory. This saving can make a large difference to the degree of game complexity which can be accommodated on a standard QL.

  The second advantage is that the compressed text is coded in a way that makes it virtually unreadable, unless the decoding routine in the ACT base module is used to convert it back its original form. It is almost impossible for anyone to cheat when playing an ACT adventure.

- The third file contains the compiled player and event programs.

- The fourth and fifth files are the location and object data. These two share a common editor program, LOCedt_task.

- The sixth file contains the vocabulary data.

- The seventh file, LAST_dta, contains additional data required by LINKER_task and will also contain any machine code additions you might add to the game. We actually provide three versions of this file, two of which include code to allow the inclusion of illustrations produced either by Graphics Designer or by the screen compression utility. In addition, sound effects produced by the SNDedt_task utility can also be incorporated. This file really provides the link for a whole array of features that go beyond even the comprehensive flexibility provided by the basic ACT system. LAST_dta may be edited by the utility LSTedt_task.

Here is a list of all the files to be found on the ACT master disk. Note that you will see the extra files, such as '14 System Utilities Follow' in the directory too. These don't actually contain anything, they are just there to divide the directory up into sensible chunks!

BOOT	This loads the various SuperBASIC extensions required by the ACT development utilities or for use independently from other SuperBASIC programs of your own. It also provides the option to start CAPTAIN_task or GDES_task (Graphics Designer). Don't be tempted to re-run BOOT in order to restart either the CAPTAIN or Graphics designer since this will result in multiple copies of the extensions being installed, LRUN either of the two files that follow instead.
CAPTAIN_boot	A short SuperBASIC program that starts CAPTAIN_task, the 'front end' utility that allows semi-automatic selection of several of the ACT utility programs. The use of this is not essential since ALL the ACT programs can be operated directly from SuperBASIC, however, it is recommended that the CAPTAIN is used when you are first learning your way

		around ACT.
GDES_boot		A short SuperBASIC program that starts GDES_task, the Graphics Designer program. Note that you can also start GDES_task directly from SuperBASIC by using the EXEC_W command (not EXEC for this program).

14 System Utilities Follow. Note that all the ACT utility programs can be started by either the EXEC or the EXEC_W command from SuperBASIC. In addition, several (marked by '***' below) may be selected from the 'front end' program, CAPTAIN_task.

BASasm_task	***	The ACT program compiler. This converts the two ACT game control programs, PLAYER_prog and EVENT_prog, into a compact form that is used in the adventure game.
CAPTAIN_task		The 'front end' program. This allows several of the ACT utility programs to be called from a simple 'menu' system.
GDES_task		The Graphics Designer. This is used to produce the illustrations for an ACT adventure. It is also able to produce drawings for other applications. Note that this program is the one exception to the rule, it MUST be started with the EXEC_W comman from SuperBASIC.
GDI_1_task		This is a convertor program that takes Graphics Designer _txt files and converts them into a much more compact format which is used by the ACT adventure games. The output format, _APIC files, can also be reproduced directly by the PIC1 SuperBASIC extension.
GDI_2_task		This is the composite picture compiler. It takes any number of files produced either by the GDI_1_task utility or by the SCNcom_task screen compressor and combines them into a single file that provides the illustrations for an ACT adventure.
LINKER_task	***	This program combines the various data files required to form an ACT adventure game.
LOCedt_task	***	This program is used to edit the data files that describe the locations and objects (or creatures) contained in an adventure.
LSTedt_task	***	This program is used to edit a special data file that contains

		some 'odd' pieces of information required by each ACT adventure. It also allows machine-code additions to be incorporated into a game.
MSGedt_task	***	This program is used to edit the various message (or text) files used in the ACT game. Note that it allows full control of the INK and STRIP colours of the messages.
SCNcom_task		The most efficient screen compressor yet produced for the QL (by quite a margin). Compressed screens can be included directly into the composite picture file (produced by the GDI_2_task utility) and may also be reproduced directly from SuperBASIC. Either the whole screen or any part may be compressed.
SNDedt_task		A flexible sound (rather BEEP) editor. This allows any BEEP (or combination of BEEPs) to be incorporated into an ACT adventure.
TXTcom_task	***	The text compression utility. This program will combine the three message files used by the ACT system into a single data file which contains the text in a condensed format.
VOCedt_task	***	This is used to edit the vocabulary of words that an ACT adventure will understand.
ACTfont_bas		Unlike all the other ACT utilities, this is a SuperBASIC program. This simple program allows you to incorporate alternative fonts into an ACT game. ACTfont_bas actually works by taking the data from any QL format font data file and combining it with a short machine code loader that will automatically incorporate the chosen font into the game.

38 Data Files Follow. These form the basic building blocks of an ACT adventure game. You can use the various system utility programs to edit these files and so alter or add to the Mini_Adventure framework. The games you develop can include any of the features already provided as well as incorporating additional features, either to the two system programs (PLAYER_prog and EVENT_prog) or by way of machine code additions such as the sound module (produced by SNDcom_task) or of your own making.

ACT	This is the 'root' module of an ACT adventure. It

	contains the parser, the code to support all the ACTBASIC commands and also the game 'debugger'. This (along with ACT_short) is the only data file that you cannot alter in any way.
ACT_short	This is identical to ACT except that the 'debugger' is not included. This module is used instead of ACT once a game is finished and fully tested. The resulting game is 4K shorter (the room taken up by the game debugger) and also, more importantly, doesn't include the opertunities to 'cheat' that the inclusion of the debugger system would provide for the player!
PLAYER_prog	The operation of an ACT adventure is controlled by two programs, written in ACTBASIC. This is the source code for the program that is responsible for responding to commands entered by the player, hence the name chosen. This source file isn't included directly into an ACT game, rather it is compiled along with the other program, EVENT_prog, by the system compiler, BASasm_task, and the resulting module (PROG_dta) is then incorporated into the game.
EVENT_prog	This is the ACTBASIC source program that controls the operation of real-time events within an ACT game. This includes the movement of creatures or objects within the game, burning objects, random events etc.
PLAYER_prog_additions	Both PLAYER_prog and EVENT_prog are provided in a form that doesn't include any support for illustrations. If you want to produce a game that is 'text only' then they should be used directly. However, if you want to illustrate your game (we advise you to have a go at playing IMAGINE before deciding, this will highlight the pros and cons associated with including illustrations) then you should use the SuperBASIC MERGE command to add the relevent _additions file to both the system programs. These additions contain all the extra code required to support illustrations.
EVENT_prog_additions	The illustration additions required for EVENT_prog.

EXAMPLES_prog	Several example additions are described in the manual that add a variety of extra features to those included with the supplied version of PLAYER and EVENT_prog. The source code for each of these additions is included in this file to save you having to type them in. Note that most of them will require modifications or additions before they are included though, please study the relevent parts of the manual (Section 4.) before attempting to use the contents of this file.
GEN_msg	All the text used in an ACT game is contained in three data files. This one contains all the 'general' messages, that is those not specifically associated with location or object description.
OBJT_msg	This text source file contains all the text used to describe objects. This also includes any messages that might be written on an object.
LOCN_msg	This text source file contains all the text used to describe the locations. Two descriptions are used for each location, one is a full description while the other provides just a few words of identification.
LASTpic_dta	These three files are just different versions of
LASTpic_QFILL_dta	the same thing. They basically contain some odd
LAST_dta	bits of information required by the ACT game that doesn't really belong in any of the other data files as well as incorporating any machine code additions. We supply the basic form, LAST_dta, which contains no extra machine code enhancements to the system and the two 'pic' versions that include several extras that provide the illustration support for a game. LAST_dta should be used for 'text only' games while the choice of either LASTpic_dta or LASTpic_QFILL_dta is governed by which Graphics Designer features you include in your illustrations.
LOCN_dta	This data file contains the information that describes the nature of each location and how all the locations connect together.

OBJT_dta	This data file contains the information that describes the properties of the objects and creatures.
PROGpic_dta	The two ACTBASIC programs, PLAYER_prog and EVENT_prog, are compiled by the system compiler BASasm_task before being incorporated into the game. Although you can always re-produce this compiled form of the programs whenever you need to (the process takes a a few minutes) we thought it would be helpful to provide the compiled form with the kit to save you time when you are first learning to use ACT. Note that BOTH source programs are compiled into a single file by the compiler and that this is the illustrated version, ie. it is formed by compiling the source programs after the two _additions files are MERGED with them.
SNDedt_dta	This contains a sample set of 'sounds' data. In fact this contains the sounds used in IMAGINE; use SNDedt_task to load and hear them.
TEXT_dta	The three text files (those with _msg extensions) aren't included directly into the game, rather they are combined and condensed by the text compressor program TXTcom_task. This is the resulting compressed file that contains all the original text in a form that the de-compression routines in the ACT root module can understand but which it is virtually impossible to read by simple inspection (try reading this file if you don't believe us!). As for the PROGpic_dta file we have included this module to save you time when you first start using ACT, you can always re-create it from the source files as will be necessary whenever you make changes to the text. This re-creation process takes a few minutes.
WORD_dta	This file contains all the words that the game is to know. The contents can be altered or added to by using the VOCedt_task utility.
BLANK_txt EXPLOSION_txt LOC1_txt	These files are the Graphics Designer picture files that were used to form the illustrations used in the

LOC2_txt LOC3_txt LOC4_txt LOC5_txt LOC6_txt LOC7_txt OBJ0_txt OBJ1_txt OBJ2_txt OBJ3_txt OBJ4_txt OBJ5_txt OBJ6_txt OBJ7_txt OBJ8_txt	Mini_Adventure starting frame game. We have deliberately kept these pictures very simple in order to encorage you to substitute your own for the initial locations and objects in the games you develop. If you want to get a better idea of what sort of pictures can be constructed then try playing IMAGINE, the average space taken up for the pictures in this game is less than 500 bytes which should give you a feel for the sort of detail it is possible to include. Note that these files are in the format that can be read directly by Graphics Designer, they must be converted by the GDI_1_task utility into the _APIC format before being combined by GDI_2_task to form the composite picture file used by the adventure game.
BUILD_picture_APIC	This is a command file that can be used to make GDI_2_task construct the composite picture file automatically. The file basically contains a list of filenames in the order required for the files to be included in the composite picture. This may not seem to important for the Mini_Adventure since there are only about 20 pictures to consider and GDI_2_task can accept the filenames directly from the keyboard. However, once your game grows you will quickly find the benefit of automating this process (IMAGINE uses about 200 separate pictures for example, it would be extremely tedious to have to enter the names of all these by hand!). Note that this file reconstructs the composite picture file that is used with the Mini_Adventure. You will have to edit in additional filenames as you incorporate more locations or objects into your game(s).

The Adventure Frame Ready To EXEC. These two files that follow are the program and composite picture file respectively for the Mini_Adventure starting frame. You have all the required components to reconstruct each of these files and of course in doing so you can add or modify to the features in order to develop a game of your own.

Mini_Adventure	The starting 'framework' program.
MINI_save_pic	The starting 'framework' illustrations.

7 System SuperBASIC Extensions

CALL_bug_fix	This file contains a patch for one of the more serious problems that is exhibited by early QL versions. This 'bug' would result in some of the ACT utility programs not functioning on 'JM' or earlier machines, this file contains a fix for this problem. It is incorporated automatically by the BOOT file and it's benefits will also apply to any interpreted or compiled SuperBASIC program that you might use.
GDES_bin	This file contains several support functions required by Graphics Designer. These are not intended for use by other programs (or programmers).
QREST	These four files contain machine code extensions
PIC1	that provide various extra features on the QL.
QFILL1	They are all described in some detail in the
QFILL2	manual and are all loaded automatically by the BOOT file. Note that none of these is ever needed by an ACT adventure game, only by the various system utility programs that YOU use to develop your adventure game(s).
CAPTAINS_mate	This is actually a short EXECable task that does absolutely nothing at all! This isn't quite as pointless as it sounds though, it is actually used by the CAPTAIN_task program in order to make better use of the memory on the QL. Note that it is only effective if the CAPTAIN is started by the special BOOT program CAPTAIN_boot (or by the main BOOT file).
QFILL2_demo_bas	SuperBASIC Tutorial For QFILL2. A full description of

	both QFILL1 and QFILL2 is provided in the manual, however, we thought that a 'live' demonstration of QFILL2 might be helpful. You can LRUN this file at any time AFTER the QFILL2 extension has been installed (by the BOOT file).
RIVER_txt	A Pretty Graphics Designer Picture. This isn't really part of the adventure system except that it does illustrate most of the features provided by the Graphics Designer and the two QFILL routines. You can view this picture either directly by Loading it into Graphics Designer or you can convert it to the _APIC format (using GDI_1_task) and then use the PIC1 SuperBASIC extension (which is installed automatically by the BOOT file) to reproduce it directly from SuperBASIC.
IMAGINE	The Example Adventure Game. These four files
IMAGINE_save_pic	constitute a complete adventure game that was
IMAGINE_BOOT	written entirely by ACT. This game will probably
IMAGINE_CLONE	take you some time to solve, even if you are quite a 'seasoned' adventurer and it's no use your looking for hints in the manual, there aren't any! If you do get stuck with it though then we can supply a 'hints' set for the game that should take you quite some way towards solving it. Please send us an S.A.E. if you do need to consult this. Note that the game can be started by issuing the command 'LRUN FLP1_IMAGINE_BOOT', with your copy of the ACT system in FLP1_ of course. Alternatively, if you want to copy the game to some other device then simply LRUN the IMAGINE_CLONE file, this will also allow you to configure the game to operate correctly from another device.

1.7 Adding Colour and Formatting Screen Text

ACT supports INK changes and use of STRIP for screen text for drawing the player's attention to particular instructions and responses. Often this can be used for emphasis or particular warnings. Another use of INK/STRIP colour changes is for games which respond to instructions such as "ASK FOR...", or "SAY....". The colour change can be used to indicate a response.

Careful planning must be made during the writing of an adventure, especially if the adventure is intended for subsequent sale on a commercial basis. Authors of adventure games have no way of predicting, let alone controlling, the circumstances under which their games are to be played.

Players may have a monitor OR a TV available; there is no way of predicting which mode the game will be played in. The ACT system will cope with all possible permutations of playing conditions, with text being displayed as required by the mode used.

Care must be exercised to ensure that chosen colours will always be visible in either mode. As can be seen from the following table, text displayed as blue in MODE 8 will appear as black (and, therefore, invisible) in MODE 4. If blue is required, this should be displayed on a lighter colour STRIP to ensure that it will be seen in either MODE.

NUMBER	MODE 4	MODE 8
0	Black	Black
1	Black	Blue
2	Red	Red
3	Red	Magenta
4	Green	Green
5	Green	Cyan
6	White	Yellow
7	White	White

1.7.1 Inserting Colour with the MSGedt Utility

INK and STRIP colours are always reset to their default values of 5 and 0, respectively, when ACT re-enters the pre-parser to get the player's next command. The player's commands will always be shown in these default colours.

After a command is entered, the INK is changed to 7 (white) BEFORE the player program starts. This change also occurs whenever the event program runs.

While most text editors (NOT Quill) can be used to produce the source files required for compiling with TXTcom_task, MSGedt includes the facility for indicating the colour changes as they will occur in an adventure. If any other text editor is used, the control codes will have to be entered manually, when and as required. See Section 1.7.2 for further information.

As you enter messages, you can enter command codes for colour changes which, like the embedded commands in programs like Quill, are indicated as direct changes to the text entered on screen; that is, the colour change is reflected in all text following the command. Changes of colour are made with:

- <CTRL> & <n> - INK change, where 'n' is a number 0 to 7.
- <ALT> & <n> - STRIP change, where 'n' is a number 0 to 7.
- <SHIFT> & <left> or <right> - Cursor repositioning in steps of 200 characters.

WARNING:

UNDER NO CIRCUMSTANCES SHOULD <CTRL> & <ALT> & <7> BE PRESSED AT THE SAME TIME. THIS WILL CRASH THE QL!

Incorrect colour control codes are edited in the same way as any other character with <CTRL> & <left> or <right>. The three characters used by the ACT system (not seen on screen, hence called embedded commands) are treated as a single character; when they are deleted, the characters following the code will revert to their original colour.

When using the left or right cursor key, the entire screen is re-drawn whenever the cursor moves past a control group (in either direction) or whenever a control group is deleted, thus acting as a reminder of the location of the control groups.

This also shows up the use of colour changes which have been input accidentally as a double re-draw.

The MSGedt utility can split long messages between successive display screens. If a very long message is edited, the parts not displayed may be selected by moving the cursor past either end of the screen display.

1.7.2 Considerations for Changes of Text Colour

The embedded (and therefore not normally visible when used in the MSGedt editor) colour commands are &#n and &$n for INK and STRIP changes, respectively. n represents the number of the colour, 0 to 7. If a standard text editor is used, the control codes must be typed in as required.

If any of these characters are used individually, they will appear in the same way other characters do. As far as possible, ACT will ignore the presence of valid control groups when formatting the screen layout. For example, if the line:

```
|This is a line which just fits the screen|
|width after the word screen.             |
```

fits the ACT display screen as shown here, the inclusion of some colour changes shouldn't alter this. If the colour control groups are put around the word "which", so as to provide an inverse effect of black INK on a white STRIP, internally, to the system, the command change would look like

```
&#0&$7which&#7&$0
```

If the control characters were counted, the screen might end up as:

```
|This is a line which just                |
|fits the screen width after the word     |
|screen.                                  |
```

since the 12 control characters would, if counted as part of the line length, leave no free room for the next word after the word "just". In fact, this line would actually be printed correctly by ACT, since it knows that it should ignore the 12 control codes.

There is a situation where ACT might not get it quite right. This can occur where the control groups are used in the same way as above, but the word happens to come near the end of a line. In the example, if the word "screen" were highlighted instead of "which", ACT would print the message as:

```
|This is a line which just fits the       |
|screen width after the word screen.      |
```

In this case, ACT is unable to determine that the word "screen" will fit into the space at the end of the first line, and the format is modified as shown. One way to minimise the chances of this occurring is to try to arrange for the colour changes to be staggered wherever possible. It is better to use:

```
this is an example &#3 colour &#7 change
```

rather than:

```
this is an example &#3colour&#7 change
```

Both will produce the same alterations of colour, but the first is less likely to suffer from a change of display format.

In the preceding examples, the control codes are shown as they appear internally to the ACT system or if used in a standard text editor. If used in the MSGedt utility, they are entered as detailed in Section 1.7.

1.7.3 Split-MODE Screens

This feature is added by the use of a machine-code routine built into the ACT system when you use either of the special versions of LAST_dta that incorporate the additional code required to support illustrations.

The use of split modes allows the illustrations of an adventure to be in mode 8, that is to use 8 display colours, while allowing the text to be in mode 4 and so benefit from smaller character size and increased amount of text on screen.

A problem can occur when using split-mode screens on some QLs with some types of memory expansion. Depending on the type of memory expansion installed, it is possible for the code that controls the mode switching to "switch" at the wrong place on the screen.

This is only likely to happen if you are using large areas from the Common

Heap. As one example, most types of RAM disks will certainly cause problems if used in a game.

If you find that the mode change occurs before the end of the picture window while testing your game, you should re-start the test after removing whatever program or facility is occupying low memory. If in doubt, re-boot and re-run your game without running any other jobs first. This problem will not normally occur when most other programs are running along with your game, and it should never happen on an unexpanded QL.

1.7.4 Formatting Messages

As explained in Section 1.2, messages are normally formatted automatically by the system when they are printed to the text window in the completed game. There will be occasions when you might want to force a message into a pre-determined format, and this is accommodated by ACT through the use of a special control character.

The character '<' is accepted by the message editor and text compressor but it isn't reproduced by the completed game. Instead, each such character is converted to a 'newline' before it is printed, so that a message may be forced to any required format.

For example, say the following message is required in exactly this format:

```
The wizard says,

  "Begone weakling, before I turn you into a toad"

 and waves his magic wand.
```

In this case, you would enter the message, using MSGedt_task as follows.

```
The wizard says,<<  "Begone weakling, before I turn you into a
toad"<<and waves his magic wand.
```

2.0 BEGINNERS GUIDE TO USING ACT

This section is an introduction to the ACT system and how it works. It also describes the use of the utility programs and shows how to add or modify the basic building blocks of the adventure game produced, the locations and objects.

The following Section 3 and Section 4, contain more detailed information and in particular show how to modify the two ACT programs that control the logic of the final game.

2.0.1 The CAPTAIN - ACT's Front End

The CAPTAIN provides a simple way of selecting the various utility programs in ACT. The CAPTAIN is started automatically by the BOOT program on the ACT disk. Alternatively, it may be invoked by using the EXEC command (if you are unfamiliar with this, then the notes in Section 2.1.1 will help you).

2.0.2 The CAPTAIN and Memory

The CAPTAIN uses up quite a lot of memory, although there will be enough left to allow you to make any modification to the Mini_Adventure, even with an unexpanded QL. However, should you ever find that there isn't enough memory free to allow some operation, then the CAPTAIN provides a simple way round this. If at any stage you need more space in order to use any of the utilities, you should select the option to "ABORT AFTER THIS STAGE". This will cause the CAPTAIN to stop as soon as it has started the next utility you select, thus freeing an extra 35K of memory for use.

Following this procedure will also mean that you will have to re-start the CAPTAIN after the utility has finished if you want to continue using the front end.

2.0.3 The CAPTAIN's Hot-Key Re-awakening

Normally, the CAPTAIN will "sleep" as soon as it starts a utility program and will only become re-activated when the job has finished. The use of the special "Hot-Key" will re-awaken the CAPTAIN at any time, however, giving you the option to abort the utility if you wish.

The CAPTAIN can also be operated with the automatic re-awake option disabled, in which case the use of the "Hot-Key" is the only way to re-activate it, either while the utility is still running or once it has finished.

2.1 Adding a New Location to the Mini_Adventure

Adding a new location to the game, or just modifying an existing one, involves making modifications to two files and then reconstructing the game from the component parts.

The files that must be changed are the location message and the location data files, which are called LOCN_msg and LOCN_dta respectively in the Mini_Adventure.

Once these files heve been edited by MSGedt_task and LOCedt_task as required, various stages have to be passed to produce the modified game in its normal running form. Table 1 shows everything necessary for the modification process, but the following 'dry run' may be of some assistance.

Before you start with the tutorial examples that follow, you should copy the various data files that you will need to use onto a scratch disk. For the examples it is assumed that the relevent files are contained on a disk in drive flp2_. You should copy the following files (after you have formatted the scratch medium):

ACT, LOCN_msg, OBJT_msg, GEN_msg, PLAYER_prog, EVENT_prog, LOCN_dta, OBJT_dta, WORD_dta, LAST_dta, LASTpic_dta, LASTpic_QFILL_dta, PROG_dta and finally TEXT_dta.

Don't worry if you aren't sure what all these files are used for; we'll come to that shortly (or maybe longly!).

2.1.1 Editing the Location Message File

As outlined in Section 1, ACT divides text messages into three separate groups. Each of the three message source files contains lines of text, and each line represents a single message unit.

Messages can be of any length up to a maximum of 1600 characters, and most of the likely QL character set is allowed.

The utility program used to edit message files is MSGedt_task. This is run in the same way as the other ACT utilities. For example, you can start MSGedt by entering the command: EXEC_W flp1_MSGedt_task with your working backup copy of the ACT disk in drive 1. There is an alternative way of running tasks on the QL, though.

If you use the command: EXEC flp1_MSGedt_task, the program will run as before, but with one important difference. The EXEC_W command, used in the earlier case, suspends the SuperBASIC interpreter, while the MSGedt program runs.

This can be useful, since it avoids either of the separate jobs from receiving input commands meant for the other. On the other hand, it can be more useful to retain control of SuperBASIC so that you can use it to inspect directories or even delete unwanted files while any of ACT's programs are running.

If you use the EXEC command, you will need to know how to direct the commands you enter to either SuperBASIC or to the alternative program(s). QDOS uses <CTRL> & <C> (hold down the <CTRL> key and then press <C>) to switch input between jobs. The job that input is currently directed to is indicated by showing its cursor flashing.

If this is new to you, then you might try running the Mini_Adventure using the EXEC command and experimenting with selecting either SuperBASIC or the game for your input commands.

Once it has started, MSGedt asks if you want to edit an existing file. Choose the default answer of <ENTER> (or Y for Yes). You will be asked for the name of the file to be edited. Respond with flp2_LOCN_msg.

MSGedt, in common with the other editing utilities in ACT, will automatically create a backup of the file you are editing. The backup will have the same name as the file you are editing, except that it will have the extension '_BAK' added to it.

This backup process will be performed every time you run an ACT editing utility, and any existing backup of the edited file will be deleted. If you make a mistake in editing any file, you can always recover your original file by replacing the file with the backup copy.

For example, if you make a mistake while editing the LOCN_msg file with MSGedt_task, the following two lines should be typed in (after quitting the editor):

DELETE flp2_LOCN_msg
COPY flp2_LOCN_msg_BAK TO flp2_LOCN_msg

in order to completely recover the original file. Alternatively, if you have a toolkit available which supports renaming files, you can RENAME the file as above.

Once MSGedt has read the file, it will prompt for the line number you wish to edit. Type in '3', and you will see the long description of the starting location in the Mini_Adventure displayed on the screen.

This displayed line is now OPEN for editing, which is done using the normal cursor controls in the same way that a line of a SuperBASIC program is edited by the interpreter's EDIT command.

To add another location to the Mini_Adventure, start by altering the starting description to indicate that there is a path available to the new location. The new room will be to the west of the small cave, so try modifying the message to be "This is a small cave. It looks.... .....passable routes to the southeast, south and west".

Once the line is altered, pressing the <ENTER> key will cause the line to be CLOSED and any changes incorporated.

MSGedt will next prompt for a new message number to edit. If you examine the map of the Mini_Adventure in Figure 1, you will see that the last location number used is 7. This means that the next location will be number 8.

Each location uses 2 messages in the location message file: one for the short location description and one for the long. Section 3.3 describes the meaning of the various messages in both the object and the location files.

You need to know that the descriptions of location number 'N' are contained in location messages N*2 (short) and N*2+1 (long). So the new location will use location messages number 16 and 17 for the short and long descriptions respectively.

OPEN message '16' for editing. You will find that it has no text in it; you have to enter the short description you want for the new location. Let's say you call this "The new room.". Once you have finished the short description, you can go on to the next message number (17) by pressing the <DOWN> cursor key.

Alternatively, you could have CLOSED message 16 by pressing the <ENTER> key, but this will subsequently require the specification of Message 17 in response to the resulting prompt. It is quicker to use the <UP> or <DOWN> keys when editing sequential messages.

Once you have message 17 OPEN, enter a suitable long description to match whatever your short description was and press <ENTER>. To stop the MSGedt program, select a negative message number for editing. This will cause the program to delete the original file and replace it with your modified data.

2.1.2 Editing the Location Data File

The data controlling the geographical structure of the Mini_Adventure is contained in the file LOCN_dta and is edited by the program LOCedt_task. EXEC the module in the same way as the other ACT utilities and specify the LOCN_dta file for editing.

The next prompt will ask which location to edit. It will be necessary to add the new location (8), but first a modification must be made to location '1', in order to introduce a path that will allow the player to move to the new location.

After you specify location '1', the program will display the contents of the various parameters. There will be 11 parameters (numbered 0 to 10) displayed, although you will see that there is room for up to 127.

Parameters 0 to 9 control the 'map'. They work as follows: Parameter 0 is for routes to the NORTH; it has a value of 2 and means that travelling NORTH from location '1' will lead to location 2. This can be confirmed by checking the map in Figure 1.

Parameter 1 represents the direction NORTHEAST and has a value of 0. This means that there is no route northeast from location 1, which can be verified from the map.

Each successive parameter represents a different direction.

0 = North or 0 degrees.
1 = Northeast or 45 degrees.
2 = East or 90 degrees.
3 = Southeast or 135 degrees.
4 = South or 180 degrees.
5 = Southwest or 225 degrees.
6 = West or 270 degrees.
7 = Northwest or 315 degrees.
8 = Up.
9 = Down.

The direction parameters of location 1 are zero except for 0 (value 2), 3 (value 4) and 4 (value 6). As a practical exercise, we want to add a new route to the west which will lead to location 8. Use the cursor keys to select parameter 6 and type in the new value for this direction (8).

Once this is done, you can press <ESC>, which will call up a menu of options. One of these is 'E' for exit. Select 'E', which will repeat the prompt for the next location number to edit. Now we can add the new location: respond with an input of 8.

For now we will just add a single route out of location 8 back to location 1 by changing parameter 2 to a value of '1'. Parameter 10 is used for flag data that allows extra information about each location to be included.

There are 8 flags available in each parameter, but only 2 (either 0 or 1) are used so far with location data parameter 10. The second flag (bit 1) is used to control whether a location is light or dark, a value of 0 meaning dark and 1 light.

To make the new location light, use the cursor keys to select parameter 10, then press <ESC>. Press 'B' to select the BINARY input option, which will allow you to alter the state of the 8 flags in parameter 10 individually.

Use the left or right cursor to select flag number 1 and then press 'SPACE' to alter the flag to '1'. Successive presses of the space key will toggle the selected flag between '1' and '0'.

This completes the definition of the new location. To complete the editing of the LOCN_dta file, use <ESC> to select the menu and respond with 'Q' (for QUIT).

2.2 Adding a New Object to the Mini_Adventure

This involves exactly the same steps as for adding a new location to the game, except that the OBJT_msg and OBJT_dta files must be altered instead of the location files.

The OBJT_msg file is edited in exactly the same way as the LOCN_msg file, using the MSGedt_task utility. From Table 4 you can see that the next object will be number '9'.

2.2.1 Describing Objects in the Text

Each object has 4 messages associated with it. These contain short and long descriptions, similar to location messages, as well as text used to service a game's 'EXAMINE' and 'READ' commands.

Descriptive text will have the word 'A' or 'SOME' prefaced at the beginning of game information, as well as a full stop at the end when it is printed out by the 'player' program.

For example, this means that you would not describe a black stick as 'A BLACK STICK.', but as 'BLACK STICK'. Use the MSGedt editor to examine some of the objects currently defined, in order to see the required formats.

Object N uses message numbers 4*N to 4*N+3 for the short, long 'EXAMINE' and 'READ' text, in that order.

You can make up any new description you want for your new object. Try creating a new note. Starting with object message 32 (4*8), enter the messages:

32. note
33. special message on a big sheet of paper
34. This is a large sheet of paper with a message printed on it.
35. The message reads 'Well done, you've added a new object and location.

and exit the editor.

2.2.2 The Object Data

The Object data file OBJT_dta is edited by the LOCedt utility in the same way as the Location data file LOCN_dta. There will be additional parameters to edit (numbers 0 to 18), and you can see from Table 2 what each one is used for. Edit the following values:

P0=0	P1=130	P2=0	P3=131	P4=0
P5=132	P6=0	P7=133	P8=0	P9=0

These define which words will be used to describe the object.

P10=8	The Location it will start out in
P11=0	Determines that it won't be contained in any other object
P12=4	The weight of the object
P13=4	The size of the object
P14=3	The surface area of the object
P15=11010000	(these flags control various functions, see Table 2)
P16=0	Used for restrictions; none apply to the new note
P17=0	This would only be non-zero if the object is to be on fire initially (when the game starts)
P18=00000000	(these flags control various functions, see Table 2))

When this is completed, exit the editor as before in order to update the object data file.

2.3 Producing the Compressed Text File

This completes the creation of the new location and object. The modified texts are next combined with the general message data, and the format is compressed by the utility TXTcom_task.

This program is run in the same way as the MSGedt program. The first question asked by the TXTcom program is explained in more detail in Section 2.6.2. Reply with <ENTER>, causing the program to use the default reply, and go on to the next prompt.

The program will now ask for the three input files in turn. Enter the file names (including the drive they are contained on), terminating each one by pressing <ENTER>.

If you make a mistake in the file description, the program will repeat the prompt. There may be a pause between successive file prompts and a slightly longer one after the third file is entered. After this final pause the program will ask for the output file name.

This can be any file name on any device, but for now specify flp2_TEXT_dta. The program will then read through each file in turn and produce the output data. At the end of the process the program will report the compression factor achieved and then stop.

2.4 Producing the Compiled Program File

If you made any changes to either of the two system programs, PLAYER_prog or EVENT_prog, you will need to re-compile them to form a modified version of PROG_dta. If no changes have been made, use the version of PROG_dta supplied and proceed from Section 2.5.

This will only have to be done again after changes or additions are made to either of the programs. Changes may be made by following the examples in Section 4 or may be of your own design. You may also have incorporated changes if you add one of the ACT "add-on" kits, such as the Video Interface kit (this is included as part of ACT Special Edition). Section 3 gives details of the ACT system language and of the structure of the Mini_Adventure programs that will help you with this. To compile the two programs, use the utility BASasm_task. Run the program in the same way as the other ACT utility programs.

The compiler will prompt for two program file names. These should be flp2_PLAYER_prog and flp2_EVENT_prog. Two more questions will follow. The first question asks for the listing file name. This could be a disk or microdrive file, but for now specify SCR, to print any messages to the screen. The following question refers to including SOURCE line numbers in the compiled code. Selecting this option allows a detailed report of where, and in which program, any errors that might occur when your game is running are located. You should normally select to INCLUDE source line numbers; only exclude them in order to reduce the length of your game once it is completed and FULLY tested.

At the end of this phase, the compiler will ask for the output file name. Respond with flp2_PROG_dta. The program will proceed with pass 2. At the end of the second pass, the compiled form of the two ACT programs will be on your scratch disk.

2.5 Linking the Modules to Form a Completed Game

Once the steps outlined in Sections 2.1 to 2.4 have been taken, it only remains to use the LINKER_task utility to automatically link together the 7 files that make up a completed ACT adventure program.

Run LINKER_task in the same way as the other utilities. The linker will prompt for the input files in turn and, in addition, will assume the names specified by the defaults indicated.

All that is necessary for most of the input specifications is to press <ENTER>. The indicated file will then be read from flp2.

After all of the input files have been read, LINKER will ask for the output device and filename. Specify any file name you wish on your scratch disk: this file will be the revised version of the game.

Once the output file is completed (when the drive stops running), you will find the new version of the Mini_Adventure ready to run. Provided no errors have been made, you will find the new room and the extra note that you have included.

2.6 Tips on Using the Utility Programs

The operation of the utility programs is mostly self-explanatory, but the following notes may help with any points that are less clear and also provide an insight into some of the operational principles that should be of help, once you are completely familiar with ACT.

2.6.1 MSGedt_task

The use of MSGedt to update the three message files is not essential. Any text editing program could be used; however, MSGedt includes some features to make it ideally suited.

The text compression process supports most of the normal text characters available on the QL. However, there are some characters that you might try to use that are not included. If you attempt to use one of the 'illegal' characters, MSGedt will report an error when you try to close the line. The message will not be incorporated but instead re-presented for editing with the illegal character(s) replaced by question marks (which are allowed).

You then have the option of either re-closing the line, which will then be accepted with the '?' substitutions, or of inserting alternative characters.

If there is not enough room on the device containing the file to be edited for the automatically generated backup, this will be reported and the program will continue with editing.

At this point there will be no free space on the destination device, since the partially complete backup will have completely filled it. You may still continue with the editing session, but you must be aware of a potential problem that could subsequently occur.

Provided the editing session does not increase the size of the data file to an extent that it requires more Sectors to store it, all will be OK. When you exit the program, the file handling sequence is: the original file is deleted (leaving room for the edited replacement) and subsequently updated by the new file. If the edited version is longer, there will not be enough room for it and you will be left with a partially complete backup and a partially complete original as well!

Unless you have a separate backup elsewhere, you will now be left without a complete version of your file. There are two possible ways to avoid this problem. If you have run the editor with the EXEC_W command, the best solution is to replace the source disk (or microdrive) with another scratch medium which has enough room to contain the new file before you exit the program. In this case the original file will remain intact on the first medium (although the backup is incomplete).

If you used the EXEC command, you also have another solution. This is to switch to SuperBASIC and delete the backup before you exit the editor. This will usually leave enough room to contain the edited file, but there is a small risk of a problem with either the medium or the QL itself, which could still leave you without a valid copy of the file. To be safe, the first solution is best.

MSGedt has one limitation that you might find puzzling. This is that once a given message slot is used, it cannot be completely removed. Also, you can only add new messages sequentially.

For example, if the last message number in one of the source files is '10', you will find that the next number you try to add must be '11'. If you try to select message 12 without first adding number 11, you will get an error.

Once you have added message 11, you can add 12. Also, once a message has been created for the first time, you can reduce it so that it contains no characters, but this will not actually remove the message completely. For example, once message 11 and 12 have been added, you could edit message 11 to contain no characters by deleting all of the text in it. Message 12 will still be intact, however, and message 11 will still be there; you could even print it out in the completed game.

The reason for doing things in this way is to ensure that the numbering of the messages is never altered after they are created. If the 'removal' of message 11 had been complete, message number 12 would then have occupied the 11th slot. This would require that any reference to it in either of the two ACT system programs be altered to match. It was simpler to include the restriction.

2.6.2 TXTcom_task

The text compressor will ask a question that requires some explanation.

The question is:

Change the default maximum word length (Y/N, default = N)?

To understand what this means you must first have some idea how the text compressor works. The process of reducing the size of the text is done in two stages.

The primary stage involves re-coding all the characters to a more compact format. This format uses control codes to achieve switching between upper and lower case characters as well as between the two character sets that ACT uses (internaly in a completed game).

The first character set includes all the alphabetical characters and most of the common punctuation symbols. The second set includes numbers and some of the less common characters. The application of this process reduces the text size to a maximum of 67% of the original text, although changes between upper and lower case and occurrences of characters that are in the second set will usually reduce the improvement to about 75%. TXTcom improves on this by means of a second stage. This involves the replacement of words that are commonly used in the three text files by special codes that typically use only 2 bytes of memory. This process is quite complex, and TXTcom will take time deliberating which words to substitute in order to obtain the greatest improvement.

In making this decision, the program must be told a maximum length for words that it will consider replacing. Any word longer than this value cannot be included in the process. The default value is 26 characters maximum, which should be adequate for most situations.

There are two situations when altering the default might be beneficial. The most likely is when the three message files are so large that the working space available to TXTcom becomes full.

When this happens, a message will be printed to the screen while the compressor is working. The compression process will not stop, and it is not necessary to make any changes. The only penalty is that some words are simply never considered for substitution, since there is no room left for TXTcom to 'remember' them in order to see if it will be worth while tokenising them.

If you re-run the compressor but reduce the maximum word length, the program will be able to 'remember' more of the words it finds, which may result in an improved compression ratio, if more suitable words for substituting are found. In specifying the new word length, you should add 6 to the new maximum length before entering the value (for a maximum length of 14 characters, you would enter 20). The second situation is the reverse of this. If you have several words in your text that are longer than 26 characters, it might be beneficial to increase the maximum.

From TXTcom's point of view, a word is any string of characters. If you used a sequence of '-' symbols to underline a particular message, this would be a legal 'word' for TXTcom to consider replacing. To be replaced, however, a word must occur at least twice; upper and lower case are not the same.

In order to achieve the most efficient compression ratio, TXTcom works best with long alphabetical messages which are predominantly either upper or lower case. The use of frequent case changes, or of numerical characters, or of punctuation marks (other than ",.'!()- or ?") will reduce its performance.

The ratio that the program specifies at the completion of the compression process is the ratio of the length of the output file and the sum of the lengths of the three input files. The output file contains additional data used by the ACT base module to identify the location of each message in the file. This pointer data uses up 4 bytes per message, and if this extra length is taken into account, you will find that the true performance of the compressor is actually better than reported.

The reported ratio for the three files supplied with the Mini_Adventure is 68%. The actual text compression is 62% for these files if the extra data is counted separately. With longer text files it is quite possible for the utility to achieve better than 50% real reduction in text length (IMAGINE achieves a true compression ratio of 52%, for example).

2.6.3 BASasm_task

The example in Section 2.4 suggests that the listing from the compiler be sent to the screen. If many errors are reported while compiling, it might be an advantage to direct the listing to a disk file or possibly to a printer, so that you have a permanent copy to remind you which errors occurred where in the programs.

You may also get error messages when you run the completed game if you have made any errors with any program modifications. In this case the reported error will state which line of the program the problem occurred in, provided you select the option to include SOURCE line numbers when you use BASasm to compile your programs. If this option is not selected, your compiled programs will be much shorter but any errors will be reported as occurring in line number 0. You should only exclude SOURCE line numbers in the compiled programs after they are fully debugged and tested.

2.6.4 VOCedt_task

VOCedt_task includes a facility for word searching, and we suggest that you use this feature before adding a new word to the vocabulary, in case the word already exists in the vocabulary file. If the new word isn't found, the next free word slot is selected ready for you to make the addition.

Words are terminated by a number. This can be from 0 to 9, and particular values are used for different types of word. Table 5 provides a list of what each number is used for.

2.7 Special Features to Watch Out For

Sooner or later in any adventure game, you will want to build in special features that will be different for each game you write.

The Mini_Adventure includes some features of this kind and, when you adapt it to form your own game, you will either have to develop it with them in mind or remove them. These features are all coded in the system programs, and notes are included below to show you how to remove them. If you prefer to keep any, then be sure that your game fits each feature you keep.

If you do remove any feature, the messages associated with it may not be needed. In this case you can either reduce the message to zero length to save space or change it for use by other features you may wish to add. You can examine the message files by using the MSGedt utility, or you can simply print the contents of each message file to a printer, if you prefer a hard copy.

2.7.1 Object 0, the Lamp

The lamp is very special in two ways: firstly, it must not be an object which can contain another object. Referring to Table 2, this condition requires that object 0's parameter 14 must be zero.

This condition is imposed by the way that object containment works. A contained object has parameter 11 set to a value which indicates which object contains it. If object 3 (the lighter) has parameter 11 set to 1 (the tool box), this indicates that the lighter is inside the tool box.

If an object is not contained, it must have parameter 11 set to zero. Since object zero is the torch, it follows that nothing can ever be contained in it. This restriction cannot be removed unless you re-write several sections of the 'player' program.

The second special feature of the lamp is that it can be a light source without being on fire (it is an electric lamp, after all). Normally object parameter 15 bit 2 is used to indicate that an object is alight. If it is on fire, it will also give off light. The lamp is not flammable, however (this being controlled by parameter 15 bit

7), and the 'burning' bit is used to control whether it is switched on or off.

The switching of the lamp is controlled by the routine 'check_switch_torch', which you will find in the player program (lines 6370 to 6700). If you remove this routine (you must also remove the call to it by deleting line 420), the lamp will no longer respond to commands to switch it on or off.

If you do delete the above special feature of the lamp, you can also make object 0 flammable. To do this, set parameter 15 bit 7, which will flag the object as one that can burn.

In addition you must make another small change to the player program. Attempts to pick up burning objects are not allowed, this restriction being applied by lines 4420 and 4430 of the player program. The previous line (4410) is a test that stops object 0 from being affected by this restriction. If this test were not included, you would not be allowed to pick up the lamp if it was switched on! So if you delete line 4410, object 0 will have to pass the test in the same way as the other objects in the game.

There is also a single change that must be made to the event program as well. This is in line 1320, where the command 'if_eq 4' should be changed to 'if_lt 4'. This line is in the routine 'check_burning_objects', which controls how long objects will burn before they are consumed. This test currently stops checking with object 1, and the change will extend this to include object 0.

2.7.2 Object 3, the Lighter

There are two special features in object 3. It is the only object that can burn and also be picked up while it is burning. Secondly, once it has finished burning, it simply goes out rather than being consumed and removed from the game.

You can change either of these special features as follows:

To prevent the lighter from being picked up if it is burning, delete line 4420 from the player program. This will still allow you to light it (by the use of the 'PRESS LIGHTER' command) without coming to any harm. To completely remove the special feature, you would have to remove lines 6860 to 6910 in the routine 'check_press' in the player program. If this is done, the lighter will no longer function as a primary source of fire and you would have to provide an alternative way of starting something burning.

To make the lighter behave in the same way as other objects when it burns, that is to completely burn away after a short while, you need to make two changes to the event program:

The first is to delete the routine 'check_lighter' (lines 1110 to 1210) and the call to that routine (line 200). Secondly, delete line 1330, which is there to prevent the routine 'check_burning_objects' from servicing the lighter.

This change might usefully be implemented if you wanted to change the lighter to a match for example, which could be struck once but would be consumed after it burns. An example in Section 4 converts the lighter to a box of matches.

2.7.3 Special Effects at Locations: Draughts and the Gas Leak

Locations 3 and 4 both have the special property of blowing out the lighter if it happens to be alight. Location 7 is sensitive to any object that is alight (except the lamp) and will cause the player's death in an explosion.

The action at location 7 is controlled by the routine 'check_for_bangs' in the player program (lines 9150 to 9320). To remove this feature, delete these lines and also the calls to 'check_for_bangs' made in lines 1400 and 1450.

The blowing out of the lighter is controlled by lines 1330 to 1380 of the player program. Deleting these lines will prevent this from happening.

2.7.4 Other Special Effects

It is often difficult to find a clear distinction between 'special effects' and 'general' effects. The examples above are most definitely special, while features like an object's classification as 'solid' or 'liquid' are general.

There are a great many features in the Mini_Adventure that fall in neither classification. The examples in Section 4 show how many of the features work and how to add new or modified features to the system.

This, along with the description of each routine in the program source files, should enable you to find how any of the features work and modify or expand them to your own specifications.

3.0 A DETAILED DESCRIPTION OF THE ACT SYSTEM

This section provides details of both the ACT system and the structure of the 'framework' provided by the two supplied programs PLAYER_prog and EVENT_prog.

3.1 The ACT System Language

ACT shares similarities with assembler-level languages. From Section 1 and Section 2 you may have noticed that most of the characteristics of a game and the objects in it are dictated by the various parameters for each location and object as well as the various text messages.

The two programs control a game from within the framework defined by these data sources and interact with the data by way of a number of variables and a variety of system commands.

The system variables are similar to conventional variables used in most high-level languages, such as SuperBASIC. They are 16 bits in length and work with signed arithmetic, which enables them to represent numbers in the range -32768 to +32767.

There can be up to 256 system variables; the actual number being controlled by a setting in the LAST_dta data file. They are referred to in an ACT program simply by the use of a number. For example, one common command used is one that causes a message to be printed:

```
    100 PRINT_GEN 4
```

The actual message printed will depend on the value in variable 4. To print out a specific message:

```
    100 LOAD_VAR 4,25 : PRINT_GEN 4
```

In this case, the first command 'load variable' will put the value 25 into the variable number 4. Use of the colon to separate several commands on the same line is supported in the same way as in SuperBASIC. The remainder of the line will now result in general message number 25 being printed. The program is built up in much the same way as a SuperBASIC program. Each line of program has a number, and lines are executed in turn, unless something occurs to change the flow.

There are some commands in ACT that have the same meaning as in SuperBASIC. The two most common are the 'REMark' statement and the 'GO

TO' command. ACT also supports the use of named subroutines. The command to set up a subroutine is not the same as in SuperBASIC; instead the keyword NAME is used, followed by the name of the subroutine.

The program line:

100 NAME print_hello : LOAD_VAR 0,7 : PRINT_GEN 0 : RETurn

defines a single-line subroutine called 'print_hello', which will cause general message 7 to be printed whenever it is called. The RETurn statement performs the same function as in SuperBASIC. There is one addition to the use of subroutines in ACT.

ACT has two programs, the player program and the event program. These share the same variables and can also share the same subroutines. In the previous example, a call made to 'print_hello' will correctly execute that routine, no matter which program it is defined in or from which program the call is made.

At the end of the subroutine, 'RETurn' will cause the program flow to go back to the correct place in whichever program made the call. ACT does not support conditional structures of the form IF THEN ELSE. This means that the use of the GO TO instruction is inevitable. GO TO has one restriction on the way it is used. This is that you may not have a branch to a line number that only has a REMark statement.

This is because the assembler BASasm optimizes for space, and if it encounters a line with no executable instruction on it, that line is not included at all. If you happpen to include a GO TO that points to such a line, BASasm will report an error when you try to compile it. The use of a REMark at the end of any line with another command imposes no restrictions.

Conditional tests are made by using one of four commands, as in:

```
    100 LOAD_VAR 2,0 : IF_EQ 2 : LOAD_VAR 2,1 : PRINT_GEN 2
    110 LOAD_VAR 2,1 : IF_EQ 2 : LOAD_VAR 2,1 : PRINT_GEN 2
    120 ...
```

In line 100, the second command is IF_EQ 2. This command says 'test the value in variable 2 and, if it is zero, execute the rest of the line'. In this case, variable 2 will be zero and general message 1 will be printed. Line 110 is the same, except that variable 2 will not be zero. The test will fail and the remainder of the line is aborted so that, after the test instruction, the next command will be taken from line 120.

Other commands similar in meaning to SuperBASIC are the CALL command, which ACT uses to execute machine-code additions to the system, and the STOP command, which stops the program and sends control back to the ACT pre-parser.

A complete list of the KEYWORDS available to ACT is given in the following section.

3.2 The ACT System Commands

A number of conventions are used in this manual:

```
  Vn  refers to an ACT system variable.
  #n  refers to a number in the range -32768 to +32767
```

NOTE: When coding a command, only the relevant number should be included.

The command LOAD_VAR is described below as LOAD_VAR Vn,#m; in the program you would code this with the desired variable number and constant directly, LOAD_VAR 3,1000, for example.

The valid range of variables is 0 to 255, but the top limit is likely to be less than this, since not all of the possible variables need be allocated (the actual number is set by LSTedt_task).

3.2.1 Commands with no Parameters

REMark	Everything following on the same line is treated as comment.
RETurn	Return to the calling routine after a call has been made to a procedure.
STOP	The current program is stopped, and control goes back to the ACT pre-parser. The STOP can be within a procedure; there is no need to return to the calling routine.
QUIT	Stops the game completely.

3.2.2 Commands with 1 parameter

GO TO #n	Go to line number #n. Note this line must not be one that ONLY contains a REMark statement.
IF_EQ Vn	Test the value in variable Vn. The remainder of the line is executed if the value is zero, othewise execution continues from the following line. The value in Vn is unaltered.
IF_GE Vn	Test the value in variable Vn. The remainder of the line is executed if the value is greater than or equal to zero, otherwise execution continues from the following line. The

	value in Vn is unaltered.
IF_LT Vn	Test the value in variable Vn. The remainder of the line is executed if the value is less than zero, otherwise execution continues from the following line. The value in Vn is not altered.
IF_NE Vn	Test the value in variable Vn. The remainder of the line is executed if the value is NOT zero, otherwise execution continues from the following line. The value in Vn is unaltered.
NAME 'name_string'	This command defines the starting point of an ACT procedure. The name_string can be any group of symbols terminated by the first space or colon. The maximum length allowed is 30 characters. The NAME command must be the first on a line. Procedures can be re-entrant entry points into a common section of code. The maximum nesting level supported is 14.
NUMB_LOC Vn	The number of locations defined in the game is loaded into Vn.
NUMB_OBJ Vn	The number of objects defined in the game is loaded into Vn.
NUMB_WORDS V	The number of words entered in the last player command is loaded into Vn.
PRINT_GEN Vn	Print the general message indicated by variable Vn. The message will be terminated by a newline character and appear on the screen immediately.
PRINT_LOC Vn	Print the location message indicated by variable Vn. The message will be terminated by a newline character and appear on the screen immediately.
PRINT_OBJ Vn	Print the object message indicated by variable Vn. The message will be terminated by a newline character and appear on the screen immediately.
PRINTS_GEN Vn	Print the general message indicated by variable Vn. This command differs from the other form in that no newline is added to the end of the message. If the message is less than a line of characters (the size of which will depend on which mode the QL is running in - 4 OR 8), no output will appear until another message is added to the end of it. The

	use of the normal PRINT and the PRINTS forms of output allows text to be built up on the screen from a variety of separate parts.
PRINTS_LOC Vn	Print the location message indicated by Vn. This is the alternative form of output and operates in the same way as PRINTS_GEN.
PRINTS_OBJ Vn	Print the object message indicated by Vn. This is the alternative form of output and operates in the same way as PRINTS_GEN.
PRINT_VAR Vn	This will print the number contained in Vn. It operates in the alternative form as described for PRINTS_GEN.
PR_UKN_WRD Vn	This is the command to print any words that the pre- parser does not know. Any unrecognised words are placed in a special store by the pre-parser, and this routine can be used to print them out. The word to be printed must be contained in Vn, where a value of 1 stands for the first unknown word etc.
RANDOM Vn	A random number (range -32768 to +32767) is loaded into Vn.
RESTORE Vn	Restore the game from a previous SAVE command. Vn will be set to zero if the operation is a success or to 1 if there is a problem. The file name used is defined by the file LAST_dta and may be modified by the LSTedt_task utility.
SAVE_GAME Vn	Save the current game position. Vn will be set to zero if the operation is a success or to 1 if there is a problem. The file name used is defined by the file LAST_dta and may be modified by the LSTedt_task utility.

3.2.3 Commands with 2 Parameters

ADD Vn,#m	The value #m is added to the contents of Vn.
CALL Vn,Vp	Call the external routine indicated by the contents of Vn. External routines start with number 1. The value in Vp is loaded into the lower 16 bits of D0, and this value is passed to the routine. Vp is loaded from D0 on return from the routine.
DIVIDE Vn,#m	The value in Vn is divided by #m.

LOAD_VAR Vn,#m	The value #m is loaded into the variable Vn.
MOVE Vn,Vm	The contents of Vn are copied to Vm.
MULTIPLY Vn,#m	The value in Vn is multiplied by #m.
SUBTRACT Vn,#m	The value #m is subtracted from those of Vn.
VAR_ADD Va,Vb	The contents of Va are added to those of Vb.
VAR_DIV Va,Vb	The contents of Vb are divided by those of Va.
VAR_MUL Va,Vb	The contents of Vb are multiplied by those of Va.
VAR_SUB Va,Vb	The contents of Va are subtracted from those of Vb.

3.2.4 Commands with 3 Parameters

BITCLR_LOC Vn,Vp,Vb	This command is used to set the value of a single bit in one of the location parameters. Vn indicates the location number, Vp the parameter number and Vb the bit number. Locations and parameters are numbered from 0 upwards, while the bit number must be from 0 to 7. None of the variables are altered.
BITCLR_OBJ Vn,Vp,Vb	This command is identical to BITCLR_LOC, except that it operates on the object data.
BITSET_LOC Vn,Vp,Vb	A similar function to BITCLR_LOC, except that the indicated bit is set.
BITSET_OBJ Vn,Vp,Vb	This command is identical to BITSET_LOC, except that it operates on the object data.
BITTST_LOC Vn,Vp,Vb	This command is similar to the BITCLR or BITSET commands above, except that it does not change the value of the selected bit, instead it reads the value (either 0 or 1) and sets Vb accordingly.
BITTST_OBJ Vn,Vp,Vb	This command is identical to BITTST_LOC, except that it checks the bit in the selected object data.
COMPARE Va,Vb,Vr	This command is used to compare the contents of two variables and set the contents of a third variable accordingly. If Va=Vb, then Vr will be set to zero; if Va<Vb, then Vr will be set to -1; and if Va>Vb, Vr will be set to +1. The values in Va and Vb are not

	altered.
GET_WORD Vn,Vd,Vt	The Vnth word number entered in the player's last command is put into Vd. The word type number (0 to 9) is put into Vt. Vd will be negative if the requested word is not found. In this case Vt will indicate the reason with a value of 0 if the requested word is beyond the number actually entered, -1 if the word is unknown, -2 if too many words are entered, and -3 if the word is too abbreviated to be resolved.
READ_LOC Vn,Vp,Vv	The value of location number Vn parameter Vp is entered into Vv. The result will be a number in the range 0 to 255. Vn and Vp are unaltered.
READ_OBJ Vn,Vp,Vv	A similar command to READ_LOC, except that it operates on the object data.
UPDATE_LOC Vn,Vp,Vv	This routine updates the location data in location Vn, parameter Vp, with the value in Vv. The contents of Vv must be in the range 0 to 255.
UPDATE_OBJ Vn,Vp,Vv	A similar command to UPDATE_LOC, except that it operates on the object data.

3.3 The Data Structure of the Mini_Adventure

The following notes describe the Location and Object data structures (LOCN_dta and OBJT_dta files) and also the meaning of each of the Location and Object messages.

3.3.1 Location Messages

There are two types of description used in ACT games: short and long. The short description is used for brief player prompts, while the long description is used for detailed descriptions of surroundings, Objects and other points related to a story. For Location N, the short description is message number N*2, while the long description is message N*2+1.

For example, the Mini_Adventure's Location 7 uses the short description in Location message number 14 and long description number 15. Note that Location number 0 is not used as a Location, and so Location messages numbered 0 and 1 are free for other use.

3.3.2 Object Messages

There are four types of object message: the short and long descriptions (the text printed when the player issues a 'LOOK' command, for example), the 'EXAMINE' text and the 'READ' text.

An examination of the OBJT_msg file with the MSGedt_task utility details the expected formats in each case. The sequence of the messages is 'short' followed by 'long' descriptions, then 'EXAMINE', and lastly 'READ' text. The message numbers for Object N are therefore, N*4, N*4+1, N*4+2 and N*4+3 respectively.

A typical example from the Mini_Adventure for the box, Object number 1:

4. SHORT :	box
5. LONG :	metal tool box with rusty lid
6. EXAMINE :	The toolbox has a hinged lid with provision for a padlock, although there isn't one on it at the moment.

7. READ : There is no writing on the box.

3.3.3 Location Data

The data in Location 0 is different from the normal Location data and is described separately. There are 11 parameters for each Location in the Mini_Adventure; parameters 0 to 9 define the logical 'map' of the game, while parameter 10 is used to contain extra information about each Location.

The 'map' parameters are numbers that define which directions at a given Location lead somewhere else:

Each parameter corresponds to specific directions, arranged clockwise, starting with parameter number 0, the value for North, in 45-degree steps, to Northeast (parameter 1), to East (parameter 2) and so on, as detailed in Section 2.1.2. After Northwest (parameter 7), parameters 8 and 9 are used for the vertical directions, Up and Down.

If a parameter value is greater than '0', this indicates which Location the corresponding path will lead to.

If a parameter value is '0', the corresponding direction has no path. Parameter 10 is reserved for flags. Only two are used with the Mini_Adventure, and parameter 10 has 6 remaining flags which may be used for additional features. It is simple to add extra parameters if more are required - an option to do so is given by the LOCedt utility.

The flags currently defined for the Mini_Adventure (bits 0 and 1) contain the following information: bit 0 is set by the 'Player' program once a Location has been described for the first time. This is the mechanism by which the long description is given on first visits and the short one thereafter. Bit 1 indicates that the Location is naturally lighted if it is set to '1'. In the Mini_Adventure Location 1 (the starting point) has this bit set, while the other Locations do not.

3.3.4 Object Data

Objects have 19 parameters in the Mini_Adventure (0 to 18). The meaning of each is indicated in Table 2. Table 3 lists the meanings of the parameters in

Location 0, which is used to contain various items of information about the game. Again it is a simple matter to add extra parameters by using the LOCedt utility.

3.4 How the System Programs Work

The broad principles of how each program works are described in this section. Detailed descriptions of how to make various modifications and additions to the programs are provided in Section 4, along with examples and comments for each of the program source files.

3.4.1 The Player Program

You can examine either program with a text editor or by LOADing it as a SuperBASIC program and then LISTing it in the normal way. The main part of the 'Player' program is contained in lines 100 to 990.

The program consists of a series of subroutine calls. Line 360, for example, consists of the subroutine call 'save_the_game'. Each call to a subroutine is followed by a description of what it does, and the subroutine name, consistent with good programming practice in any language, is usually a description of its function.

'Save_the_game' will examine the first word of the command line entered by the player. If this is word number 31 (SAVE), the routine will perform a game save and return control to the ACT pre-parser, so that the next command may be accepted from the player.

If the word is not number 31 (SAVE), the routine returns control to the program and the next subroutine will be called, which is 'restore_the_game' (called from line 380).

The subroutines themselves are defined later in the program. Each has a detailed description of its operation at the beginning, but generally comments have been kept to a minimum within each routine. It is not likely that you will need to make many changes to the routines provided, since additions will generally be made by adding new routines. Some of the examples in Section 4 do modify some of the routines, however, and in these cases the way that the appropriate routine works is described in detail.

3.4.2 The Event Program

This works in a similar way to the 'Player' program, but its function is quite different. Most of the routines called will return control to the 'Event' program once they have executed, rather than back to the ACT pre-parser.

This means that all the routines are executed in turn, rather than just the one which successfully decodes the last command as in the 'Player' program. The 'Event' program will normally return to the pre-parser (at line 300) after all the subroutines have been called.

Each routine is described at its beginning in the same way as in the 'Player' program.

3.4.3 Considerations for Split-MODE Screens

The two program-merge files supplied with the Video Interface Kit contain the required code to support the Split-Mode feature. In addition, a switch is included to enable the feature to be switched ON or OFF when the game is first run.

The control of this is achieved through the spare Location 0 parameter number 2 bit flags 2 to 4. Bit 2 will directly control the operation of all of the CALLS (described in Section 7.3 and Section 7.4) to the Split routine: the flag must be set to enable the CALLs. Bits 3 and 4 are used by the new routine check_for_mode in the player program, telling it when to operate. A few additional lines in the EVENT program are also involved.

You can use the program startup feature as it stands or permanently enable or disable the split feature by: a) editing the LOCN_dta file (with the LOCedt utility) to set Location #0 parameter #2 bit #2 as required; and b) removing the following lines from the two merge files before using them to create their new PROG_dta file.

```
    Player_Prog_Additions: Delete 155, 156 and 30620 to 30760.
    Event_Prog_Additions : Delete 1063 and 1064.
```

You should be aware that the split screen puts a great demand on processor time. The use of half-screen pictures noticeably reduces the command response speed of the adventure program produced. This time penalty will be increased for larger pictures; for this reason the use of pictures larger than half-screen is not recommended.

3.4.4 May We Reserve Some Space?

In order to provide you with maximum flexibility, we have deliberately avoided including restrictions in the ACT system. However, in order to allow us the facility to add new features to ACT, we might need to use some of the unused parameters and flags.

For this reason we recommend that you do not use the remaining bits of Location 0 parameter #2, nor parameters 9 and 10 of Location 0. This will leave these data areas free for us to use for future additions to the system and ensure that compatability with existing games and new add-on modules is maintained.

Generally we recommend that you assign new variables/parameters for use with any new features that you incorporate into your games. This has the additional advantage that it will avoid any possible corruption of information required by the existing programs.

3.5 Interfacing Machine-Code Extensions to ACT

The ACT 'CALL' command (see Section 3.2.3) allows machine-code extensions to be called directly from either of the two ACT system programs. A call made in this way executes the user's machine-code routine as a subroutine, and control should be returned with an RTS instruction. On entry, the value in the lower 16 bits of D0 will be the number contained in the second variable of the CALL instruction. For example, the command

```
    LOAD_VAR 1,1 : LOAD_VAR 2,55 : CALL 1,2
```

will call the first machine-code routine and pass the number 55 in D0. The user's routine need not preserve any registers, and the value in D0 will be passed back to the appropriate variable on return. The machine-code routine can access information about the game by referring to the main table area maintained by ACT. This table is pointed to by A6; a list of some of the values in it is given in Table 6.

3.6 The ACT On-Line Game Debugger System

To help you sort out any problems with developing the PLAYER or EVENT programs for your adventure game, a powerful debugging system is provided within the ACT base module. You should, of course, not allow a copy of your game to be circulated with the debugger included; it would provide players with about the most effective cheating facility they could possibly ask for! The alternative base module, ACT_short, excludes the debugger as well as on-line messages; as explained in Section 1.6, you should use this module when linking the final version of your game(s).

3.6.1 What Is the Debugger?

The debug facility provides a means to do all of the following while the development version of your game is actually running:

1. Examine or alter the parameters of any object or location defined in your game.

2. Examine or alter the value of any of the system variables.

3. The debugger can be explicitly entered at any time by simply pressing the up cursor key.

4. Break points can be set in either the player or the event program or both. They will suspend program execution and pass control to the debugger.

5. The debugger allows you to step through either program, either line by line or instruction by instruction. Parameter or variable values may be examined and altered at any time during this process.

6. A complete symbolic representation of each instruction is provided along with an indication of the current source program line. Armed with a listing of a program, you can use the debugger to 'step' through problem sections and quickly isolate and identify any errors.

3.6.2 How Does the Debugger Operate?

The debugger is called initially by pressing the up cursor key to complete any command to the game. When this is done, the debugger is called before the player program is entered and your game command interpreted in the normal way.

The debugger maintains a pop-up window in the lower half of the screen. As soon as you exit from the debugger, the original screen contents are replaced and the game will continue as though nothing had happened. While the debugger is active, however, the game is completely suspended, not even the event program being allowed to run.

The debugger presents a simple '>' prompt and will accept a variety of commands as described below. You can use the dubugger to examine the values of any of the parameters or variables used in your game. Also, by altering values, you can dramatically alter the set-up of your game. For example, since Location 0 parameter 0 defines the current location of the player, it is a simple matter to 'move about' within your game simply by changing this quantity. As you will realise, this facility is useful in general testing of a game, as well as in isolating specific faults.

3.6.3 The Debbugger Commands

Whenever the debugger is entered, it will indicate which program is currently running and what the current line number is. It also indicates what the next instruction on this line is. You should realise that whenever the debugger is called using the up cursor, it is entered before the first line of the player program is actually reached. In this case it will indicate a line number of 0 and the next instruction indicated is not then relevant.

Another thing to remember is that the debugger can only know about the line numbers in the player or event source code if these are included by the BASasm compiler. If you attempt to use the debugger in a game where you have excluded these, then not all of the features are available to you. In particular, neither breakpoints nor the single-line execute mode will work, although you will still be able to execute the player program instruction by instruction, if you want. If you do operate the debugger in this way, it will always indicate a current line number of 0.

The following commands are accepted by the debugger. Please note that either upper-case or lower-case letters will be accepted, but you cannot use the delete function to correct a mistake. If you wish to abandon a command, simply press ESCAPE as many times as necessary to get back to the debugger's '>' prompt.

1. Right cursor key. This causes the next instruction of whichever program is running to be executed. Control is then returned to the debugger. This command can be used to 'single- step' through either program. You are free to use the other facilities available under the debugger to examine parameters and variables, and so the operation of selected parts of the program code can be followed in complete detail.

2. Down cursor key. This causes the the program to run until a new program line is reached. In effect, this allows the program to be executed 'line by line'. The use of this command requires that the programs are compiled with SOURCE line numbers included, this being controlled by an option in the BASasm compiler. Normally you should only exclude line numbers when a program is completely debugged; this then considerably reduces the size of the code but at the expense of limiting the flexibility of the debugger. If you do use this command when line numbers have been excluded, the program will execute in the normal way until control is returned to the ACT pre-parser.

3. Escape. This terminates the debugging session and passes control back to the program. The game will then continue in the normal way, unless you have set breakpoints.

4. 'C' key. This is short for 'COMMAND'. It causes the debugger to re-print the information that describes which program is running, the current line number and the next command.

5. 'B?' keys. Any command starting with the 'B' key is connected with the program breakpoints. There are four options, each selected by the next key pressed. In this case, the command asks for a list of the current settings of

each breakpoint.

There are eight breakpoints, numbered 1 to 8. These can refer to any line number in either the player or the event program. A breakpoint is inactive when the line number is set to zero, or to any line that doesn't actually occur in the program. In response to the B? command the debugger will print a list as follows:

```
P00000   E00120
E00200   P01980
P00440   E00000
P00000   P00000
```

These refer to the eight breakpoints, the initial letter indicating which program the break is set in. In this example you will see that the settings are:

```
Break No.1, set to PLAYER program, line 0 (not active)
  "   "  2,  "   " EVENT     "       "  120
  "   "  3,  "   " EVENT     "       "  200
  "   "  4,  "   " PLAYER    "       "  1980
  "   "  5,  "   " PLAYER    "       "  440
  "   "  6,  "   " EVENT     "       "  0 (not active)
  "   "  7,  "   " PLAYER    "       "  0 ( "     " )
  "   "  8,  "   " PLAYER    "       "  0 ( "     " )
```

6. 'BC' keys. This clears all the breakpoints.

7. 'BRn.'. This is used to clear a selected breakpoint. The character 'n' should be a digit, 1 to 8, to indicate which breakpoint is to be cleared. The '.' is required to complete the command, but you will find that other characters can be used as well, for example the space key.

8. 'BSn(P/E)nnnnn.'. This is the command used to set a breakpoint or to change the set-up of a breakpoint that is already set. The value 'n' indicates which breakpoint number is to be set: this should be a digit, 1 to 8. It should be immediately followed by either a 'P' or an 'E' to indicate which program the break is to be set in. Finally, the group 'nnnnn' is a number that indicates the required line number for the breakpoint. The final '.' is used to complete the command. The following examples show some valid formats that are accepted.

```
>BS3E550.       Breakpoint number 3 set to EVENT
                line number 550.
>BS6P01000.     Breakpoint number 6 set to PLAYER
                line number 1000.
>BS1P0.         Breakpoint number 1 set to PLAYER
```

```
                                    line number 0; this is exactly the
                                    same as issuing the 'BR1.' command.
```

9. 'Vnnn.'. Used to examine a variable. 'nnn' is the required variable number, which must be in the range 0 to 255, although the maximum variable number is likely to be less than this limit, depending on how many variables are actually assigned in the ACT 'LAST_dta' file. Once you complete the command with the '.' character, the debugger will output the appropriate variable's current value as follows:

   ```
   >V7./+00002/ <<<(cursor will be left at the end of this
    ---             string of characters)
   ```

   Here the parts typed by the user are underlined. Once the current value of a variable is examined in this way, you have several options. The simplest is to press the ENTER key, which will return the debugger to its prompt without making any alteration.

   Alternatively you can automatically select another variable by pressing either the left or right cursor keys. In this example this would result in variable number 6 or variable number 8 being examined, respectively.

   If you want to change the value of a variable, simply type the new number before pressing the ENTER or cursor keys. In this case control will pass back to the debugger prompt or another variable will be selected, but the current variable will be changed to the new number first. The following example should make this clear:

   ```
   >V7./+00002/      //examine V7, no change, on to V8
    ---    -
   >V008 /+00000/    //no change, on to V9
          -
   >V009 /-00001/50  //change V9 to 50, on to V10
          ---
   >V010 /+00017/    //no change, back to V9
          -
   >V009 /+00050/<ENTER> //re-examine, new value displayed
                 -------
   >                    //
   ```

10. '(L/O)nnnPmmm.'. Used to examine object or location parameters. This command works in a very similar way to the 'V' command described above (i). The first character of the command, either 'L' or 'O', selects either a LOCATION or an OBJECT to be examined. The group 'nnn' is the required location/object number; it is followed immediately by the letter 'P', which

marks the end of the first number, and then the second number, 'mmm', which indicates which parameter is to be examined. Valid examples are:

```
>L6P10. /000, flags 00000000/  //location 6, parameter 10
------
>L029p001. /028, flags 00011100/ //location 29,parameter 1
---------
>O0P10. /005, flags 00000101/  //object 0, parameter 10
------
```

Once a selection has been made, you have similar options to those available using the 'V' command. Thus ENTER will return to the debugger prompt without making any alterations, left or right cursor will examine the previous or next parameter of the same object/location, and typing a new number before any of these options will cause the value of the displayed parameter to be updated. In addition, the use of the up or down cursor keys will select the previous or next object/location, selecting the same parameter as in the current, displayed, one.

A simple editing sequence that should make the various options clear is shown below:

```
>L0P0. /001, flags 00000001/29\BD       //The player's current
 -----                         ---      //location in the game.
                                        //This moves him to loc
                                        //29.
>L000P001 /001, flags 00000001/28    //Player's previous
                                        //location.
                                  ---   //This is updated to 28.
>L001P001 /028, flags 00011100/0<ENTER> //This clears location 1
                                --------//parameter 1 effectively
                                        //removing the NE path
                                        //from loc 1 to loc 28.
>O8p10. /005, flags 00000101/0       //Obj 8 parameter 10 was
 ------                         --      //5, indicating that the
                                        //object is at location 5
                                        //the object dump. This
                                        //updates it to 0, the
                                        //object is now 'held'.
>O008P011 /013, flags 00001101/0<ENTER> //Parameter 11 was 13,
                               -------- //meaning that object 8
                                        //is contained inside obj
                                        //13. This parameter is
                                        //updated to 0, so that
                                        //object 8 is no longer
                                        //contained.
```

11. 'H'. This is the 'help' request: it will produce a short page of text in the debugger window that provides a brief reminder of the formats of the debugger commands.

3.6.4 An Example Editing Session Based on the Mini_Adventure

Following through this simple example should help you to understand how to use the debugger and will illustrate some of the possible 'tricks' it can be made to perform in order to help in testing a game.

The example uses the Mini_Adventure program supplied with your ACT kit. Run the 'debug' version of the Mini_Adventure and then follow the example below. You will find it helpful to have the Mini_Adventure map to hand and also a listing of both the player and event programs.

3.6.5 The Debug Session, Step by Step

Allow the game to start, unfreezing the screen as necessary in order to get to the first prompt that the game prints. Then follow the instructions below. Note that the parts you have to input are underlined; everything else is produced by the game or the debugger. Also note that the ENTER key is represented in the examples by <ENTER> and the escape key by <ESCAPE>. Comments are added after a // symbol. Note also that in reality the debugger will open a pop-up window while it is active. Interaction with the debugger is indicated in the example by indenting the appropriate text.

```
>PRESS LIGHTER<ENTER>                      //we are at location 1
--------------------
With a click, the lighter bursts into flame.
>                                          //this calls the debugger
-
 Break in PLAYER line 00000                //debugger entered prior
 Next instruction: LOAD_VAR 157,+00000     //to entering PLAYER prog
                                           //- LOAD_VAR 157,0 isn't
                                           //part of the player prog
 >L0P0./001, flags 00000001/7<ENTER>       //Move player to loc 7.
  -----                    --------
 >O0P10./001, flags 00000001/0             //This picks up the torch
  ------                    --
 >O000P011 /001, flags 00000001/0<ENTER>   //This removes the lamp
                               --------    //from the toolbox.
 >O0P15./032, flags 00100000/36            //This sets bit 2 of par
  ------                    ---            //15 for the torch, it is
                                           //now switched on.
 >O000P016 /000, flags 00000000/           //No change to par 16
                                -
 >O000P015 /036, flags 00100100/<ENTER>    //Re-looking at par 15 to
                              -------      //check we have flags set
                                           //as we want.
 >BS1E1190.                                //Breakpoint 1, EVENT line 1190
  ---------
 >BS2P1310.                                //Breakpoint 2, PLAYER line 1310
```

```
  ---------
 >B?                                       //Examine the breakpoints
  --
 E01190   P01310                           //Breakpoints 1 and 2 as set,
 P00000   P00000                           //all the others are 'clear'
 P00000   P00000
 P00000   P00000
 ><ESCAPE>                                 //Returns us to the game.
  --------
>LOOK<ENTER>                               //Let's see: where we are now...
-----------
This is the end of the road....            //...location 7, courtesy of the
                                           //debugger.
>INV<ENTER>                                //What are we carrying?
----------
You are carrying:-
A small electric torch (Often-Ready type) (switched on).
>
 //OK, this demonstrates how the debugger can manipulate the game.
 //Remember that we left the lighter burning, back in the small
 //cave. The breakpoint that we have set in EVENT line 1190 will
 //interrupt the program when the lighter goes out; just wait a
 //minute or so...
 Break in EVENT line 01190             //...the breakpoint passes
 Next instruction: New line +01190     //control to the debugger.
 >BR1.                                 //Clear this breakpoint.
  ----
 ><ESCAPE>                             //Back to the game.
  --------
>NORTH<ENTER>                          //We shouldn't be able to go
                                       //north from loc 7;
------------                           //the breakpoint in PLAYER
                                       // line 1310 will intercept
                                       //the attempt to move, though.
 Break in PLAYER line 01310
 Next instruction: New line +01310
 >\BD                                   //execute the next instruction
                                        //in line 1310
  -
 Break in PLAYER line 01310
 Next instruction: DIVIDE 001,+00002     //Refer to a listing of
 >\BD                                    //PLAYER to follow the
  -                                      //progress of the
 Break in PLAYER line 01310              //step-by-step execution.
 Next instruction: LOAD_VAR 000,+00000   //
 >\BD                                    //We want to step through
  -                                      //to the end of this line,
 Break in PLAYER line 01310              //we will then be able to
 Next instruction: READ_LOC 000,000,000  //change the value that the
 >\BD                                    //program reads from the
  -                                      //location data and fool
 Break in PLAYER line 01310              //the game into thinking
 Next instruction: READ_LOC 000,001,002  //that the NORTH route from
 >\BD                                    //location 7 really leads
  -                                      //to location 1.
 Break in PLAYER line 01310              //
 Next instruction: End line              //
 >V2./+00000/1<ENTER>                    //Variable 2 contains the
  ---         --------
 ><ESCAPE>                               //path destination; we
  --------                               //change this from 0,
                                         //indicating no valid path,
                                         //to 1, indicating a route
                                         //back to the small cave.
```

```
This is a small cave.
Here, I can see:-
A sheet...
 .
 .
>
```

This example illustrates something of what the debugger will allow you to do. Once you have mastered the simple principles that it works by, you can use it to help sort out any problem you might encounter with your additions to either the player or the event program.

4.0 ADDING NEW FEATURES TO THE SYSTEM

The file EXAMPLES_prog, included on the ACT master medium, contains the ACTBASIC source code for the example modifications described in this section.

You can MERGE this file with either of the two ACT system programs and delete any sections you do not want to use. Note that only subroutines are contained in the file; you must add the appropriate calls in the main body of the system program(s). You should also be aware that most routines will require some small changes (substituting particular numbers where necessary).

4.1 Adding a New Word Function to the System

The following example checks the first word of the command line and, if it is a certain value, causes a message to be printed. The name 'check_for_word' is used here, but you can change this as appropriate for the word you actually check for.

11000 name check_for_word

This line defines the starting point of the routine 'check_for_word'.

11010 load_var 0,1 : get_word 0,0,1 : subtract 0,#n : if_ne 0 : RETurn

This line gets the first word number into variable 0 and the word type into variable 1. The type information is not actually needed, since the routine only has to check whether the word is a particular number. 'Subtract 0,#n' will subtract the number #n from the word number contained in variable 0. Note that #n refers to whatever word you want to check for; you must substitute the appropriate number. The 'if_ne 0' command will check the value in variable 0 and only allow the remainder of line 11010 to be executed if the value is not zero. The routine will return to the calling program for all words except number #n . If the word is number #n, as is the case here, the test will cause the rest of the line to be skipped and the next line to be executed.

11020 load_var 0,#m : print_gen 0 : stop

This line prints general message #m and returns to the ACT pre-parser, ready for the next player command. You must substitute the message number you want printed for #m.

The routine works in exactly the same way as 'check_look' in the player program (lines 1720 to 1800), except that, instead of outputting a singlOP

In this case the three words #n1, #n2 and #n3 will each be tested for and any of them will cause message #m to be printed. Extra tests can be added between line 11100 and 11110, to allow more words to be checked. To perform something more interesting than simply printing a message in response to a word, the following example shows how the routine 'find_objects', included in the player program (lines 3480 to 4200), is used to find what objects the player is referring to in a command line.

```
11130 name check_requiring_an_object
11140 load_var 0,1 : get_word 0,0,1 : subtract 0,#n : if_ne 0 : RETurn
11150 load_var 7,2 : load_var 13,0 : find_objects
11160 load_var 0,-1 : compare 0,4,0 : if_eq 0 : check_first_obj
11170 check_object_found : load_var 0,#m : print_gen 0 : STOP
```

Lines 11130 and 11140 are the same as the first example. Line 11150 contains a call to 'find_objects' which will scan the other command words entered and identify up to two objects. Before the call is made, it is necessary to set up variable 7 with the word number in the command line that find_objects is to start searching from. In this example, variable 7 is set to 2, indicating that the search should start with the second word. It is also necessary to set up variable 13 with the value 0 to indicate that 'find_objects' should find an object if it is either in the same location as the player or held. If the value is set to 2, find_objects will only find an object specified if it is held.

The find_objects routine returns information about what objects (if any) it finds in variables 4 and 5. Variable 4 refers to the first object in the command line, and 5 refers to the second. The values in variables 4 and 5 will be any of the following after find_objects returns control:

-1	if there was no object specified in the command
-2	if an object is specified but cannot be found at the current location (or not held if V13=2)
-3	if there are more than one of the specified objects
N	any non-negative number indicates that the appropriate object (number N) was specified and found

Line 11160 checks the value in variable 4 (the first object specified); if it is equal to -1 (indicating that no object was specified), it calls the routine 'check_first_obj', which prints a message and then returns control to the ACT pre-parser.

Line 11170 calls the routine 'check_object_found'. This routine will check variable 4 and, if it is negative, print a suitable message, after which it returns to the ACT pre-parser. If the object has been found, it returns control to the program, where the remainder of this line causes general message #m to be printed.

This last example is very similar to the routine 'check_feel' (player program lines 9710 to 9810), which parses commands to FEEL an object. The player program includes several subroutines that use the 'find_objects' routine. An example of one that can make use of both objects is 'check_take' (lines 4210 to 4510). This routine is further complicated because it will allow a form of the command that involves two 'verb' words, 'PICK UP'.

You may use existing routines as 'patterns' to model new ones on, but be careful. Some of the routines include extra built-in features that may, if copied blindly to new functions, result in undesired effects. It is always a good idea to base your new routines on the examples provided and add extra features as you need them.

4.2 Changing the Lighter to a Box of Matches

It is simple to make cosmetic changes to the lighter so that it is described as 'MATCHES' or a 'MATCH' for example. The object description for the lighter (object number 3) will need to be changed in the same way as described in Section 2.1.1, by using the MSGedt utility and re-compressing the three text files, to form a revised version of the TEXT_dta file.

A number of minor changes are required:

- The list of words that the object will be recognised by, as described in Section 2.2.2, must be altered.

- Any new words must be added to the vocabulary file WORD_dta by using the utility VOCedt, we will need 'MATCH' and 'MATCHES' for this example (each should be given the 'tag' number 5, to flag them as nouns) and also 'STRIKE' (tag 3, a verb).

- Change some of the general messages that refer to the lighter.

The messages to alter are 91, 92, 93, 97 and 98. The new messages should be consistent with the alterations made to the object(s). An obvious example would be to change message 91 to read:

```
'One of your matches is now burning away with a nice bright flame.'
```

All that is left to do is to modify the player program routine 'check_press', so that PRESSING a match has no effect, and adding a new routine that will allow the command, 'STRIKE A MATCH', to work. Deleting lines 6860 to 6910 of the 'Player' program will stop the command 'PRESS LIGHTER' (becoming PRESS MATCHES) from doing anything. The 'check_press' command will still respond to this command, but only with the standard 'NOTHING HAPPENS' message.

All that is required to complete the modification is to add an extra routine that will respond to the command STRIKE MATCHES. Now use the VOCedt utility to add the word STRIKE to the vocabulary. The next free word 'slot' in the supplied vocabulary is 149, but you might have added other words, so let's say that the new command word is number #n. The following routine will add the command:

```
  11180 name check_strike
  11190 word1 : subtract 1,#n : if_ne 1 : RETurn
  11200 check_for_light_sources : if_eq 0 : read_loc 0,0,0 :
        load_var 1,10 : load_var 2,1 : bittst_loc 0,1,2 :
        if_eq 2 : load_var 13,2
  11210 load_var 7,2 : find_objects : load_var 0,-1 : compare 0,4,0 :
        if_eq 0 : RETurn
  11220 check_object_found
  11230 load_var 0,3 : compare 0,4,0 : if_ne 0 : GO TO 11290
  11240 load_var 0,15 : load_var 1,2 : bittst_obj 4,0,1 : if_ne 1 :
        load_var 0,90 : print_gen 0 : STOP
  11250 load_var 0,0 : read_loc 0,0,0 : subtract 0,3 : if_lt 0 :
        GO TO 11280
  11260 subtract 0,2 : if_ge 0 : GO TO 11280
  11270 load_var 0,92 : print_gen 0 : STOP
  11280 load_var 0,3 : load_var 1,15 : load_var 2,2 : bitset_obj 0,1,2 :
        load_var 0,93:print_gen 0:load_var 15,100:check_for_bangs :
        STOP
  11290 load_var 1,#m : print_gen 1 : STOP
```

This routine is almost identical in format to 'check_press' and it works in the following way.

The routine 'word1', in the first line of the routine (11190), is a call to another routine in the player program (line 10030). This gets the number of the first word of the player's command into variable 1.

The rest of this line is similar to the other examples and checks to see if the command is 'STRIKE'. Line 11200 sets up the value in variable 13 according to whether it is light (or not) at the current location. This is necessary since it allows the subsequent call to find_objects to exclude objects that are not held from being found if it is dark.

Lines 11210 and 11220 are similar to the previous examples; after each has been executed, the number of the specified object will be in variable 4. Line 11230 checks to see is the object specified is number 3. If it is, the program will continue with line 11240; if not, a jump is made to line 11290, which prints out general message number #m. This should be a suitable response when nothing special happens.

For example, you might make message #m "Striking that isn't very useful.". The equivalent line in 'check_press' allows one of three messages to be output. You could also do this here by modifying line 11290 so that it is similar to line 6930.

In this case, the use of the routine 'random_message3' requires variable 1 to contain the number of the middle message of a group of three. Lines 11240 to 11280 deal with actually setting fire to object number 3. The required action and/or message to be output depends on whether the match is already alight and also on where the player is.

The matches will be blown out by the draughts in some locations in the same way as the lighter is. Removing this feature is explained in Section 2.7.3.

Finally, you must add a call to the new routine 'check_strike' in the player program. Since the routine should work in the dark as well as in lighted locations, it is necessary to position the call before line 500, which returns control to the ACT pre-parser if it is dark. A suitable location would be:

```
  441 check_strike : REMark Special actions if word number 1 is STRIKE
```

4.3 Using the Routine YES_or_NO_Response

A special feature is built into the player program, allowing suitable responses to be made to certain messages. The feature works through the routine 'yes_or_no_response', which is included in the player program lines 1810 to 2090.

This routine is designed to detect the words YES or NO in response to a message previously output by the program. It works in the following way: Location 0 parameter 3 is used as a flag that initiates yes_or_no_response. If this flag is zero, line 1900 returns control to the player program.

In this way, if the player types the command YES or NO when no special condition is primed, one of the *"I don't understand"* messages will eventually be output once all the other routines in the decode chain have had their turn to try to respond to the command. If the flag is not zero, this represents a given special condition. In this case one of two things can happen:

- If the player doesn't enter a YES or NO command, 'yes_or_no_response' will clear the special condition and return control to the player program. This is dealt with by lines 1910 and 1920.

- If the command is YES or NO, several things happen. Line 1930 sets up variable 0 with a negative number if the command is YES and a positive number if NO. Subsequent lines test for each possible value of the special condition flag.

For example, lines 1940 to 1970 will respond if the flag is 1, which corresponds to the special condition to QUIT the game. This flag value is set by the *'check_quit'* command. Note that the command QUIT doesn't directly stop the game. Instead the player must respond YES to the message printed by the 'check_quit' routine.

In this case the game is actually stopped by line 1970 in the *'yes_or_no_response'* routine. If the player responded NO to the quit message, line 1960 would return control to the ACT pre-parser ready for the next command.

If the flag wasn't set to a value of 1, then lines 1980 to 2000 will check for and respond to a value of 2. In this case both YES and NO will produce a particular message that forms a suitable response to the original situation that set the condition flag.

Other condition flag values are tested for, and acted on if found, in turn. If the condition flag number is not catered for within 'yes_or_no_response', control is returned to the player program with no action being taken.

Flag values 1 to 4 may be set by the player program. You may add any number of extra values up to the maximum flag value of 255. For example, you might construct a routine that responds to the command HIDE with a message "Do you think that will do any good?".

If you make this routine set the special condition flag to a value of 5, you could add three extra lines of code to 'yes_or_no_response' that will respond with suitable messages for either a YES or a NO.

The extra lines should have the same form as lines 2040 to 2060, but with the message numbers altered according to which new messages you want to output in response.

Notice that the GO TO target line number in line 2040 will have to be altered to point at the first of your new lines, which, in turn, should pass control to line 2070, if the flag isn't 5.

How about the messages "Then you must be a bit stupid.

" if the response is YES and "Well, you have some sense at least!" if the response is NO in this simple example?

4.4 "Magic Transform" Words

A feature often used in adventure games is that of a "magic" word which has a special effect. Often this is used to allow the player to be transported from one place to another without having to use the paths that actually exist. The inclusion of such a system in ACT is achieved by the following routine.

```
11300 name magic_transform
11310 word1 : subtract 1,#n : if_ne 1 : RETurn
11320 load_var 0,0 : read_loc 0,0,0 : subtract 0,#m :
      if_ne 0 : load_var 0,#p : print_gen 0 : STOP
11330 load_var 1,#d : update_loc 0,0,1 : load_var 3,1 :
      load_var 2,#m : update_loc 0,3,2 : load_var 0,10 :
      bittst_loc 1,0,2 : if_ne 2 : GO TO 11350
11340 check_if_light : check_for_bangs : describe_location :
      load_var 0,0 : read_loc 0,0,0 : load_var 1,0 :
      describe_objects : STOP
11350 check_if_light : check_for_bangs : load_var 0,#d :
      multiply 0,2 : print_loc 0 : load_var 0,#d :
      load_var 1,0 : describe_objects : STOP
```

Line 11310 looks for word number #n, which should be the "magic" word the routine is to respond to.

Line 11320 checks the current player location. If this is equal to #m, the program passes to line 11330. If the player isn't at location #m, this line prints out general message #p and returns control to the ACT pre-parser. Message #p should indicate to the player that nothing special happens.

Line 11330 updates the player's current location parameter (location 0, parameter 0) to #d, which is the "destination". Location 0 parameter 1, the player's previous location, is also updated to #m.

The last commands on this line examine parameter 10 bit 0 of the "destination". If this is set (ie the player has visited the destination before), control is passed to line 11350. If the bit is clear (player's first visit to #d), control goes on to line 11340.

The two lines 11340 and 11350 output the long or the short form of the location description respectively, as well as describing objects etc. The routine 'magic_transform' will respond to a given magic word only if the player is in the correct "starting" location, and move him to the "destination". If he is not at the starting location, the routine will simply output a message saying that 'nothing interesting has happened', or whatever else you choose to say.

Possible improvements to this routine would be to arrange for a message to be output in lines 11340 and 11350, prior to the location description output. This message could be simply "Done.", or maybe something more spectacular such as "There is a blinding flash of light as you utter the magic word, and you are mystically transported to a new location.".

It might also be useful to allow the same routine to transport the player back from the "destination" to the original location (reversible transform).

If you want more than one transform in your game, you could either duplicate several different copies of this example routine, each with suitably different locations and magic words defined, or you might save space by modifying the example to allow multiple location/word use.

Finally, routine 'check_for_bangs' is included in this example, but it is only required if the destination is a location where you need to check for an explosion (which would only be location 7 in the Mini_Adventure). Do not include the call if the feature isn't required.

4.5 The Use of Flags to Control Particular Events or Commands

By now you will be familiar with most of the features of ACT and the ACT programming language. The use of flags to act as switches is very common in both the event and the player programs; several of the previous Sections make use of flags in this way.

If you are still a bit doubtful as to how a "flag" operates, you may find the following explanation useful.

A flag is just a way of remembering a single piece of information. You could use the system variables as flags, or any parameter of any location or object, provided it isn't being used for some other purpose. Since Location 5 parameter 0 is not used for anything, for example, this is available for use as a flag.

The use of variables or parameters as flags is rather wasteful of space, though. All a flag has to do is remember one of two states: either "true" or "false" (a logical 1 or 0).

This can be done quite adequately by a single bit in any of the object or location parameters; and the ACT system language has several commands built in to set, clear or examine single bits as flags.

Illustrating this feature, parameter 10 of each location is dedicated for use as flags, and currently two of the eight possible flags are actually used.

The flags are numbered from 0 to 7 and, as described in Section 3.3.3, bit 0 of each location parameter 10 is used to indicate if a location has been visited before, while bit 1 is used to indicate if the location is lit (independently of a light-emitting object, such as a torch).

The ACT program commands used to manipulate a flag are the 'bitset', 'bitclr' and 'bittst' commands. Look at 'Player' program line 1390.

The third from last command in this line is bittst_loc 2,1,0. The values in the three variables are set up by the preceding parts of the routine to be the current location, 10 and 0 respectively. From the description of this command (Section 3.2.4) you will see that this will result in the value of bit number 0 in parameter

10 of the current player location being loaded into variable 0.

In this way the move_command routine can determine if a particular location has been visited before.

The value of this flag is normally set to zero initially (by the LOCedt utility program when the location data is set up). The value is updated, once a location has been visited, by line 1600 in the player program (the routine describe_location). You will see the command bitset_loc 1,2,0 in this line, which changes the value of the flag to 1.

You can use any number of flags in your additions to the ACT system. You can use the remaining bits (number 2 to 7) in location parameters 10 or bits 5 to 7 of object parameters 18.

If you need more flags, use the LOCedt utility to add more parameters to either the objects or the locations. Doing so will not affect the operation of the ACT programs, although it will increase the size of the adventure game produced slightly and also the size of the SAVE file.

4.6 Arranging for a Time-Delayed Event

It is easy to introduce time delays for particular events by making use of the 'Event' program. As an example, consider the routine 'check_lighter' (lines 1110 to 1210 of the Event program).

It is the job of this routine to extinguish the lighter, after a suitable delay, if the player doesn't bother to do so by using a command such as 'EXTINGUISH LIGHTER'. The control of the time that the lighter is allowed to burn is governed by variable 15, which is dedicated to this purpose in the Mini_Adventure.

Line 1170 checks whether variable 15 is zero. If it is, the routine returns to the calling program, since this means that the lighter isn't alight.

Note that it would have been possible to find out if the lighter was burning by examining object 3 (the lighter) parameter 15 bit 2, the normal way to see if an object is alight. However, since variable 15 is dedicated to the lighter, it is simpler to make a check based on this variable.

Line 1180 subtracts 1 from the value in variable 15. If the contents are still non-zero, the routine will return to the calling program at this point. If the contents are zero, control passes to line 1190. Line 1190 does two things. First it switches off bit 2 of parameter 15 for the lighter (object 3), the bitclr_obj 0,1,2 instruction actually clears the bit flag. Secondly, if the lighter is accessible, that is if it is in a place where the player can see it, a message is output that tells the player that the lighter has just gone out. The routine object_accessible (lines 2750 to 2960 of the 'Player' program) is used to see if the lighter is visible or not.

The routine 'check_lighter' performs several functions: if the lighter is burning, the counter (variable 15) is decremented by 1 each time the event program runs. When the value reaches 0, the lighter is extinguished and an appropriate message is output.

You can make up your own routines using variables as counters in this way. In this case you should use the LSTedt utility to modify the file LAST_dta to add the extra variables you want to use in the system.

The Mini_Adventure uses variables 0 to 16 (17 variables in total), but you can allocate up to 255, if you need to. For purposes of compatibility of existing ACT games with future additions to the ACT system, we recommend that you do NOT use variables 17 through 20 because these are reserved for interfacing our new modules to.

The 'Event' program is called at a rate of about twice a second. If you load a variable with a large number (the largest possible is 32767), you would get a possible delay of up to about 4.5 hours. If you need longer delays than this, you could arrange to 'cascade' two counters. The following example illustrates the use of location parameters as counters and uses cascading of two separate counts to achieve a longer maximum time delay.

Object or location parameters are limited to the range of 0 to 255; thus only very short time delays are possible with a single counter. If you use two or more variables in place of the parameters, as in this example, you could get very long time delays if you require them.

```
11360 name delayed_event
11370 load_var 0,0 : load_var 1,9 : read_loc 0,1,2 : if_eq 2 :
      RETurn
11380 subtract 2,1 : if_ne 2 : update_loc 0,1,2 : RETurn
11390 load_var 2,#n : update_loc 0,1,2 : load_var 2,10 : read_loc
      0,2,3 : subtract 3,1 : if_ne 3 : update_loc 0,2,3 : RETurn
11400 update_loc 0,2,3 : update_loc 0,1,3
11410 REMark Put the instructions you want execured after the time
11420 REMark delay here.
11430 REMark If you want the delayed event to be repeated,
11440 REMark update location 0 parameters 9 and 10 to give a
11450 REMark suitable delay for the next event.
11460 RETurn
```

The value #n determines the period of the time delay. To use this routine you must update the values of location 0 parameters 9 and 10 with numbers in the range 1 to 255.

The delayed_event routine will decrement the two counters in turn, giving a total delay of #m*#n, where #m is the value you initially load into parameter 10, and #n is the value that you loaded into parameter 9 (and also the same number as in the example). The way the code operates should be apparent: add the lines of code you want to have executed in place of the REMark statements, and install the routine into the EVENT program.

The routine will execute as part of the event program sequence, and in this way a delay of roughly #m*#n/2 seconds will occur before execution of your added code.

4.7 Who's Afraid of the Dark?

The Mini_Adventure doesn't have any hidden traps or problems in the event of the player being in a dark location without a light source. One common feature of many adventure games is to arrange for the player to be killed (by some fiend lurking in dark places), if he stays in the dark too long. The following example shows how such a feature might be built into your own ACT game (IMAGINE includes a similar feature).

```
11470 name dark_monster
11480 load_var 0,0 : read_loc 0,0,0 : load_var 1,10 : load_var 2,1 :
      bittst_loc 0,1,2:if_ne 2:RETurn
11490 check_for_light_sources : if_ne 0 : RETurn
11500 random 0 : load_var 1,25000 : compare 0,1,2 : if_lt 2 : RETurn
11510 load_var 0,#m : print_gen 0 : load_var 0,0 : load_var 1,6 :
      update_loc 0,1,0 : RETurn
```

This routine should be called from some point near the beginning of the player program; line number 442 would be a suitable place. You will need to add an extra general message (number #m) which should be something along the lines of "You have just been attacked and killed by a monster lurking in the dark."

Lines 11480 and 11490 are used to see if the player is in the dark. Control is sent back to the calling program if there is light at the current location.

Line 11500 is used to provide a random chance of the player being killed in the dark. With the number 25000, as in the example, there is about a 1-in-6 chance of being killed each time a new command is entered when the player is in the dark.

Line 11510 prints out the message and updates the player's health rating to 0. This is all that is required to kill the player; the routine check_if_dead will do the remaining work required.

You can improve on this example, of course. It might be interesting to add a variety of different death messages. It might also be interesting to arrange for the routine to be called from the event program, either instead of the player program, or maybe as well.

In this case you should reduce the probability of death each time, or possibly make the call in the event routine part of a repeated delayed event, so that it is called once a minute or so.

4.8 Etcetera, etcetera

There are bound to be numerous examples that you will think of that are not included in this guide. ACT is so versatile that virtually anything you might want to arrange can be done in a game. The main problem is likely to be to decide the most logical way to achieve a particular feature.

The completed game, IMAGINE (sorry, we don't provide the source files for this), includes numerous features not directly covered in this guide and might provide ideas of additions that you will want to include in your own games. Note that everything included in IMAGINE is done using either the standard features (as in the Mini_Adventure starting framework) or by additions along the lines illustrated in the examples in this guide.

If you have any suggestions about additions to the ACT system, either the programs or the manual, we would be pleased to hear about them.

5.0 SOME FINAL COMMENTS ABOUT ACT AND WRITING ADVENTURE GAMES

This guide is intended to provide you with the information you will need to use ACT to write your own adventure game, whether for your own entertainment or for commercial applications. ACT is capable of producing complex games to a very high standard and with any number of features.

You will need to provide the ideas for your own games, of course, this being something we can't really help you with in the guide. However, there are some general tips on how to go about designing your game that you might find useful. Probably the most important point to remember is that you should check each step of game development as you go along.

It is all too easy to add numerous new locations, objects and lots of new routines to the developing system programs, all in one go, only to find that when you try to run the game with the additions installed, all sorts of illogical things happen which might be errors in any one of the additions you have made.

We recommend careful pre-planning, on paper, for a completely original game. The following sequence for building your game and adding new objects or features to it is the way we recommend you should proceed.

1. If you are going to start with the Mini_Adventure as the base for your game, make any alterations to this before you add anything new to the system. Once you are happy with the modified form of the Mini_Adventure, proceed with the following stages.

2. Add new locations and the appropriate location messages. Re-link the game and check that all the additions work. This can be quite a long job, since you will need to move around the game, checking each direction at each location.

3. Add any new objects along with the object messages required. Again you should re-link the game and check that all the objects are where you want (or expect) them to be and have all the correct properties.

4. You should add any new features you want to the two system programs; this step is the most likely to cause trouble.

Constructing the Mini_Adventure, the author of the ACT system had numerous "head-scratching" sessions, trying to understand why seemingly simple bits of code didn't do what was expected!

The best advice that we can offer is in line with that for any complex program: try to make your additions simple until you are quite familiar with the ACT system, and even then restrict yourself to adding your new features in small stages. The use of the named subroutine feature makes breaking up complicated pieces of code into simple Sections relatively easy.

ACT has been extensively tested, and you should not have any problems with errors in the system itself. However, as with any large programming system, it is not possible to guarantee that it is 100% bug-free.

If you do experience any problems, please write to Digital Precision

(DON'T EVEN THINK OF PHONING - THE ANSWERING SERVICE HAS BEEN TAUGHT TO DISCONNECT WHEN ACT IS MENTIONED);

we will do our best to determine the cause of any problem you encounter, and if a bug in the ACT system is the cause, we will fix the fault - almost as fast as LIGHTNING.

Happy adventuring!

6.0 APPENDIX 1 - THE GRAPHICS DESIGNER

6.1 Overview

The Graphics Designer is a sophisticated drawing program that is specially designed to produce illustrations for adventure games. The program provides the facility to incorporate all of the shapes that are 'native' to the QL graphics support, that is line drawing, circles, ellipses, blocks, arcs; and in addition the normal QL FILL facility is fully supported. However, additional features have also been included that allow much more advanced drawings to be produced. These additions are also available to you directly from SuperBASIC, so that you can use them for other applications as well as with ACT.

The Graphics Designer is also unusual in that it allows pictures to be stored as a 'command' file, that is as a set of drawing instructions that can be used to re-create the drawing. This has the advantage that the space required to store even quite detailed pictures is VERY much less than that required for a 'screen dump', even if a compressor (such as the SCNcom utility supplied with ACT which easily outperforms ALL other such utilities on the market) is used to produce the stored screen.

In fact three different ways of storing a picture are provided by Graphics Designer. These are:

1. Standard 32K screen dump -
   For single pictures or for loading screens for other programs.

2. A text co-ordinate/command (parameter) file -
   This is a very compact format that contains the 'native' instructions used by Graphics Designer to reconstruct a picture. This is the form that you must store illustrations in while they are still being developed and also the form that is subsequently processed into a 'composite' picture file for use by the ACT adventure game.

3. As a SuperBASIC program -
   This allows a picture to be incorporated as a procedure in your own programs, thus providing a convenient way to use the Graphics Designer

for applications other than the ACT games system.

Additional features that are provided by Graphics Designer are:

- The ZOOM option, used to allow selected magnification of areas of an illustration as well as allowing the program to effectively produce pictures that are much bigger than the QL screen area (which then acts as a 'window' into the much larger drawing).

- Area recolouring. A sophisticated and very fast routine is provided that allows any area of an illustration to be re-coloured. This routine works in either MODE and is able to cope completely with all stipple patterns.

6.2 The Additional Drawing Routines QFILL1 and QFILL2

Graphics Designer is supplied with two additional 'filling' routines, QFILL1 and QFILL2. Both of these can be multitasked to provide an animated filling effect. QFILL1 provides an alternative to the conventional DRAW and native QL FILL. A shape drawn with this option may be re-entrant and even have boundary lines crossing: QFILL1 will always produce a correctly filled shape. QFILL2 is a re-colour option. It allows any area of the screen to be re-coloured; plain colours or stipples are fully supported.

Both QFILLS are available independently of Graphics Designer; they may be used from SuperBASIC (interpreted or compiled) and are of course also supported by the ACT adventure writer system.

6.3 Commands

The following list provides a description of all the commands available with Graphics Designer. Note that some keys do different things depending on what the program is actually doing. For example, the F1 key will normally call up the main help menu, which provides a summary of the main options available within the program. However, if pressed while some other options are being executed (such as the CIRCLE option, for example), F1 provides a summary of information that relates to the currently selected option.

```
     Key or option          Function
    -------------           --------
     <F1>             Selects main Help menu - toggles off/on. The
                      options presented in this menu are selected
                      by pressing the first letter as shown in the
                      menu.

     <ESC>            Exits most options  without implementing
                      changes.

  Option Letters - Available with or without main menu visible:

     <M>ode           Toggles MODEs 4/8.

     <J>ump           Returns cursor to screen centre.

     <Z>oom           <1>, default, and <2> to <8>. Successively
                      changes scale to "magnify" at current cursor
                      position. Text can only be entered and will
                      only appear at Zoom setting "1".

     <W>ipe           If colour option is set to NORMAL, clears
                      drawing area to current paper colour.  If
                      colour option is set to COMPlementary or
                      HIGHlight, this option wipes the screen to
                      the current paper colour and redraws in
                      complement mode.

     <U>ndo           Successively  deletes  previous  drawing
                      actions.

     <R>eset          Sets picture to SCALE  1 and cursor to
                      starting position, then redraws the current
                      picture. Also resets all program parameters
                      to original defaults (except for file drive).

     <N>ew            This option will zap the current picture and
                      reset the SCALE setting to 1, INK to 7 and
                      PAPER to 0.
```

The following commands are for colour control (and related things).

```
     <O>ver (colour)
                       <N> NORMal colour representation.
```

	<C> COMPlement - XORs new colour on screen. <H> HIGHlight - this is similar to NORMal except when text is output. This option will then result in text being produced on a STRIP background (the strip colour will be the current PAPER colour).
<P>APER	<0> to <7> for first colour. <0> to <7> for second colour. <0> to <3> for stipple. If only one colour is required, the same number is entered twice and stipple choices are not offered. Eight colours are offered in both modes to insure that switching between modes supports the colours available to each mode. Current colour indicated next to "P" window.
<I>nk	As PAPER; current colour indicated next to "I" window.
<F>ill	BASIC FILL - toggles on/off. Indicates in window 2nd from right (top). This window is normally green in mode 4; cyan in mode 8. Also, during redraw for options requiring redraw, this window is red, indicating that drawing is taking place.

The following commands are the main 'drawing' options.

All drawing options are rubber-banded. With the exception of the DRAW option, shapes may be moved to their intended location and then set by pressing <SPACE>.

Cursor movement	8-way. The cursor can move and shapes may be drawn 'off screen'. This allows large shapes to enter into the screen from beyond borders.
<D>raw	1. 8-way <CURSOR> movement of free end of line. 2. Y/N option to select QFILL1 - offered only if the SuperBASIC FILL is NOT selected.

NOTE: If QFILL1 is selected (by pressing "Y"), the 'Job and 'repeat' values required by the QFILL1 routine must be specified. You should read the appendix 'QFILLs', which provides detailed information on what both QFILL1 and QFILL2 are and how they work.

'Job' values:

0	(this is the default obtained by simply pressing ENTER). This will result in the shape you subsequently enter being produced in the normal way once you have finished entering the lines that make it up. The prompt for 'repeats' is not made.
>0	The valid range is 1 to 255. This will set the fill, once specified, as an independent task on the QL. This means that control is handed back to the Graphics Designer (or the ACT

adventure game if the picture is used for it) immediately; the 'fill' simply carries on independantly. The actual value specified determines the priority that the QFILL1 task will run at.

'Repeats' values:

<=0 A value of zero or less will result in the specified shape being 'filled' repeatedly and indefinitely. Alternate fills will be in the current INK and PAPER colours, so that the shape will appear to 'flash' on the screen. Note that within Graphics Designer a special pause is included once a fill is started. This pause allows you to see the effect of a repeated QFILL on the screen, However, once you continue, it has been arranged that all active QFILL tasks are stopped. This allows you to continue with constructing a picture without the distraction and possible problems of the previously established QFILL shapes on the screen. Once a picture is included in an adventure game, however, any repeating QFILL tasks within it are only stopped when another picture is drawn. Please see the Section describing GDI_1_task.

>0 Repeats of 1 or greater work in a similar way as the indefinite repeat except that now the QFILL only repeats for the specified number of times. You could use this option to arrange some feature of a picture to flash when the drawing is first produced in a game, for example, and so draw the player's attention to it.

3. Press <SPACE> to set "free" end of line.

4. If the QL's native FILL is on, any closed shape drawn will be filled (subject to all the normal limitations of the native fill, i.e. shapes may NOT be re-entrant). If the shape is not closed, no fill will occur. If QL's FILL is selected, QFILL1 is not offered.

5. If QFILL1 is selected, the shape will be closed from the current cursor position to the original cursor position. In other words, QFILL1 will always assume that the first point of the shape is joined to the last point.

6. Press <ESC> to EXIT.

While the Draw option is active, the F1 and F2 keys do the following:

<F1>	Provides simple instructions.
<F2>	Provides information about the current line Section being drawn.
<Y> Outline	Outline for a filled polygon shape produced by the DRAW option (note, NOT the QFILL1 option). To use this facility, switch FILL on, select the DRAW option and draw the required shape. When the shape has been completed and filled, then without selecting any other option change to a new INK colour and press <Y>. The polygon will be outlined in the new

colour.

<C>ircle	The circle is drawn in outline and may be moved about the screen by use of the cursor keys. Once in the required place, it is set by pressing the space key. If the native QL fill is on, the circle will be filled; otherwise it will be drawn in outline only. The size of the circle may be altered by the use of the 'S' key (smaller) or the 'L' key (larger).
<F1>	Provides simple instructions.
<F2>	Provides some information about the current position and size of the circle.
<E>llipse	This is basically the same as a circle, except that the shape can also be rotated and the eccentricity may be altered. This is done using the keys 'C' (clockwise rotation), 'A' (anticlockwise rotation), 'I' (increase eccentricity) and 'D' (decrease eccentricity).
<F1>	Provides simple instructions.
<F2>	Provides some information about the current position, size, orientation and eccentricity of the ellipse.
<SHIFT><8> = <*>	This selects the QFILL2 area recolour routine. The action is to alter the colour of the area immediately under the cursor to the current INK colour. The actual area that is changed in this way is basically every pixel on the screen that can be reached without crossing a pixel of a different colour. See the appendix on QFILLs for additional information about this option. Several questions will be presented before the fill is started:

a. "Are you sure? (Y/N)"

b. Buffer size. Pressing <ENTER> accepts default buffer of 256 bytes. This parameter determines how complicated the fill area can be before QFILL2 is unable to fill it completely. A value of 10000 is likely to fill ANY shape, while the default of 256 will only cope with very simple shapes.

c. "Job value?" Default = 0.

   If 0 : program control returns when the fill is completed. Fill stops only when finished.

   If >0: Fills as an independent job. Program control returns immediately. Fill stops only when finished.

```
                    If <0: Infinitely repeating fill. This
                           works in much the same way as the
                           repeating  fill available  with
                           QFILL1. The alternative colour is
                           determined by the routine: it will
                           be the highest-numbered colour that
                           doesn't have any colour in common
                           with the current INK.

<B>ox               This works in much the same way as CIRCLE
                    and ELLIPSE. The box may be rotated ('C'
                    and 'A' keys) and may be made <T>aller,
                    <S>horter, <W>ider and <N>arrower.

<F1>                Provides simple instructions.

<F2>                Provides  some  information about  the
                    current position, size and orientation of
                    the box.

<A>rc               The current  end of the arc  may be
                    positioned by use of the cursor keys. The
                    'C' key will swap the current end of the
                    arc. The angle subtended by the arc can be
                    increased by use of  the 'I' key or
                    decreased by use of the 'D' key. If the
                    native fill is on,  the arc will be
                    automatically closed in and then filled
                    once it is fixed.

<F1>                Provides simple instructions.

<F2>                Provides  some  information about  the
                    current position, size and orientation of
                    the arc.

<T>riangle          The triangle is moved one corner at a time
                    with cursor keys; press 'C' to change
                    corners. An apex can be moved across the
                    triangle's base for effective inversion.

<X> Text            To add text to a picture, you should make
                    sure that the current ink is the required
                    text colour, then press the 'X' key. The
                    following information will be requested
                    and actions must be performed:

                 1. Required CSIZE.
                 2. Select (or not) surround writing.
                 3. Required text string (max 25 characters).
                 4. Position the string as required.
                 5. Press <SPACE> to set.
```

The following options are provided as 'aids' to help in drawing pictures.

```
<Q>uery             A review option, allows stepping back
                    through a drawing to view the x,y co-ords,
                    shape and option used for each element.

                    Directives: <P>rior <N>ext  <Q>uit

<G>rid              This provides a grid of lines, either
```

	solid or dotted, that can be used as a framework to help when drawing a picture. Note that the option is provided to Keep the grid in the picture for the Screen dump or SuperBASIC output; however, a completed ACT game will not reproduce a grid.
<H>old	This allows a time delay between drawing of successive elements of a picture when reproduced by Graphics Designer. This option has no effect when the picture is drawn by a completed game.

The following commands are associated with file handling.

<SHIFT><ESC>	Change default file device. Presently defaults to flp2_.
<V>iew	Directory from default drive. <CTRL><F5> to pause directory. Additional options: <L>oad <K>ill (delete) <S>ave
<L>oad	Load a previously saved Graphics designer picture (with the _txt extension).
<S>ave	A picture may be saved in three formats: as a 32K screen dump, as a coordinate/command file, or as a SuperBASIC program. The file will be given an extension to the name specified according to the save type chosen. This will be _scn for a screen dump, _txt for the coordinate file and _bas for the SuperBASIC dump. Note that only the _txt files can be read back into Graphics Designer or used to make a composite picture file for use by an ACT adventure.
<K>ill	File delete (from default drive).

6.4 The PIC1 Utility

The PIC1 utility is a SuperBASIC extension which is installed automatically by the ACT system BOOT file. This utility provides the fastest way of re-producing Graphics Designer drawings, combined with the smallest possible picture storage space.

In order to use a picture in an ACT game it is necessary to convert the picture _txt file to a more compact format by using the GDI_1_task utility. The resulting file (which has the extension _APIC by default) leaves out certain information included in the normal _txt files, which allows support for some of the features provided by Graphics Designer.

This means that the _APIC files cannot normally be inspected except by reproducing them in an ACT adventure game. However, since it might often prove useful to be able to reproduce pictures converted in this way, we have included the PIC1 utility (which is basically a copy of part of the code used in the machine-code module used in ACT). You can then open a picture file produced by the GDI_1_task utility to a SuperBASIC channel (#n) and use the command:

```
PIC1 #n
```

to reproduce the picture; you can also repeatedly re-draw the same picture, if you wish. If unspecfied, PIC1 will use channel #3 as the default.

PIC1 will support the QFILL1 and QFILL2 options. If one of these is used in a picture to produce a repeating fill, you can also use PIC1 to stop the fills by entering the command:

```
PIC1 -1
```

NOTE: This will only work on fill jobs created by PIC1. If you have your Graphics Designer extensions loaded and have used either QFILL1 or QFILL2 directly to draw a repeating fill, then either of these may be stopped only by the appropriate call to QFILL 1 or 2. Conversely, PIC1 cannot itself stop a fill job created by either of the QFILL routines.

You can also stop a repeating picture drawn by PIC1 by using PIC1 to draw another picture which has been prepared with the option to stop all current fill jobs, as described later.

There is more information about PIC1 in the appendix 'Linking Illustrations into an ACT Game' and also in the appendix 'QFILLs'.

7.0 APPENDIX 2 - LINKING ILLUSTRATIONS INTO AN ACT GAME

This Section describes how you can add illustrations to an ACT text adventure. Facilities are provided to allow drawings produced either by the Graphics Designer (included with the ACT kit) or by ANY other drawing utility to be included in your games. In addition, several of the facilities provided can be used 'externally' to the ACT system, directly from SuperBASIC for example, in order to provide illustrations for other applications.

It is recommended that you are familiar with both the text part of the ACT system and also the Graphics Designer before attempting to use the facilities described in this Section.

When used with the ACT adventure writing system, these modules provide a flexible means to add graphics to ACT. In its basic form the Video Interface Kit provides a sophisticated system allowing each location in an ACT adventure to be displayed; each object can also be incorporated into the Location pictures, whenever and wherever required.

In common with the flexibility inherent in ACT itself, this Video picture system can be easily extended to enable other pictures or features to be incorporated.

7.1 The Basics: How to Add Pictures to ACT

In order to add pictures to an existing ACT text game, there are two basic steps that must be taken:

1. Add the required machine-code picture extensions to the game and incorporate the required additions into the PLAYER and EVENT system programs.

2. Produce the picture data file ready for the modified game to extract pictures from as they are needed.

7.1.1 Adding the Picture Extensions to an ACT Game

This also involves two steps:

1. Load each ACT system program into your QL in turn, and then MERGE the appropriate merge file supplied with this kit (PLAYER_prog_additions and EVENT_prog_additions). Each modified version of the two system programs should then be separately saved on your working medium and given a suitable name to identify it as a 'picture' version of the system program.

   Once this is done, use the BASasm assembly program to assemble the two modified files. We use the name PROGPIC_dta for the compiled program file of the development version of Mini_Adventure supplied with ACT.

2. Re-link the adventure game using LINKER_task, but substitute the file 'PROGPIC_dta' (or whatever you decide to call it) for 'PROG_dta' and either 'LASTpic_dta' or 'LASTpic_QFILL_dta' for 'LAST_dta'.

If you are unsure about these steps, you should familiarise yourself with how each of the ACT system programs operates by reading the relevant Sections of the ACT User Guide.

The choice between 'LASTpic_dta' and 'LASTpic_QFILL_dta' depends on whether you intend to use either of the QFILL features available with Graphics

Designer. If you don't use either of these features in the pictures you intend to incorporate into your adventure game, use 'LASTpic_dta', which will save over 3K of memory. If you do use either of the QFILL's in any of the pictures to be included, you must use 'LASTpic_QFILL_dta'.

If you have incorporated changes into either of the two ACT system programs, you should check that none of your modifications will either affect or be affected by the additions in the two merge files. Each of these contains modifications to existing lines in both PLAYER_prog and EVENT_prog, as well as additional lines and new subroutines.

7.1.2 Producing the Composite Picture File

An ACT adventure which is solely text is ultimately assembled, using the various ACT utilities, into a single file. Where illustrations are to be used with an ACT adventure, two files are required; this Section gives the details for producing the composite picture file.

All the pictures that are to be produced by an illustrated ACT adventure are combined into a single data file. This process is performed by the utility program GDI_2_task.

This utility allows you to combine both Graphics Designer files (after they have been pre-processed by the GDI_1_task utility) and also files from the screen compressor SCNcom_task.

This flexibility allows you to make use of the very compact format file that the Graphics Designer/GDI_1_task combination produces for the majority of your illustrations, but also to include any number of pictures produced by other drawing programs.

The ACT picture system has been designed to expect individual pictures for each location and object in the game. These pictures are drawn automatically whenever they are needed and are linked to the logic of the text part of the adventure. Towards this end, the following rules have been adopted by the ACT Graphic System:

1. A picture will be drawn for a location ONLY if the location is lit. If there's

no illumination, the screen will go dark.

2. The picture is updated if it is lit AND whenever the player moves to another place or issues the LOOK command.

3. Objects are drawn only if they are:

    1. at the current location AND not held;
    2. not inside OR resting on another object.

4. If an object displayed in the picture is picked up, the whole picture (including any remaining objects) is re-drawn. If an object currently held is dropped, it is simply added to the picture without a re-draw. Objects are always superimposed, i.e. one on top of the other, if they happen to be drawn as overlapping.

5. The system automatically keeps a check on the current state of illumination. If the only light source is extinguished, or simply burns away, then the screen will go blank.

Location pictures start with location number 1 (location 0 isn't used as a Location in ACT), while object pictures start with object number 0. Each location and object must have a picture associated with it, although this can be a simple "null" (blank screen), as required.

The correct order for producing your composite picture file starts with Location 1, up to the total number of Locations defined in the game. The object pictures are added after the last Location picture, starting with object number 0.

After the Locations and Objects, the special picture BLANK_APIC which must be produced by the GDI_1_task utility from the file BLANK_txt, supplied with the kit, must be included. This is used by the system to clear and reset the window sizes at certain times during the game.

Any extra pictures are added after the BLANK_APIC file. One such picture, the explosion, is included with the Mini_Adventure demonstration. The procedures required to service it are also included in the two program merge files. Section 7.5 of this manual explains how you can add more pictures.

The name you choose for the composite picture file is important. The ACT adventure game must be told which file it is to read the illustrations from. Looking at the names used for the Mini_Adventure demonstration game will give you an idea how this is done.

You will see that the supplied composite picture file for Mini_Adventure, MINI_save_pic, has a name that is related to the ACT SAVE file name used to store the game position through the ACT "SAVE" command. The video system will always search for the picture file on the same device the SAVE file is created on and will use the filename formed by adding the extension _pic to the SAVE file name.

As explained in the ACT User Guide, you can change the SAVE file name or device by editing the approriate entry in the file LAST_dta. So, to alter the SAVE name and also the corresponding picture file name, use the LSTedt_task utility to edit LASTpic_dta or LASTpic_QFILL_dta, as appropriate.

Finally, you might wonder what the picture associated with Location 5, the Object dump, might be used for. Since this picture can never be reproduced as a normal Location during a game, it is convenient to arrange for it to be used as the game's "Title" or "loading' screen". This is done automatically by the additions in the merge files, and this picture will be drawn as soon as the game is loaded.

7.2 Details of the Utility Programs

The program utilities supplied with this kit are very simple to use. They are described in some detail in the following Sections.

7.2.1 SCNcom, the Screen Compressor

This program is run using the EXEC command; you shouldn't use the EXEC_W variation, since part of the design of the compressor assumes that the screen to be compacted will be loaded AFTER the utility is run. Once started, SCNcom can remain memory-resident although inactive, but be repeatedly re-activated whenever a picture is to be compressed.

When it is first started, and again after each screen is compressed, the utility will ask for an output file to store the next screen in. Once this file is specified, the utility will "sleep" but keep a check on the keyboard periodically while it waits for the special "activation" command. This activation is initiated by the keypresses <CTRL>&<N> (hold down <CTRL> and then press <N>). This should be done only when the picture you want to save is displayed on the screen. The compressor will take a "snapshot" of the screen at this point before displaying a prompt window. You are then given the opportunity to select which portion of the screen you want to save.

SCNcom can work on the whole screen or any smaller sub-Section of the screen: you select the appropriate area by the use of cursor keys. The appropriate directives are:

1. <SHIFT> & <cursor> - to change the size of the 'window'.

2. <cursor> keys only - to change the position of the 'window'.

3. For vertical height changes, the use of <CTRL> & <cursor>, or <CTRL> & <SHIFT> & <cursor> as appropriate, will alter the window in larger increments. Once you have selected the required window, press <ESC> to initiate the compression process. This can take a few minutes for very complicated screens, although simpler pictures - those that contain large areas of a single colour or stipple - will be processed in just a few seconds.

After it has completed its work, SCNcom will ask for another file name to store the next picture in. You can choose to stop the program at this point, if the current picture is the last one you want to compress. Once the compressed picture file is created, you can use it in either of two ways. Firstly, it can be incorporated directly by GDI_2_task into a composite picture file for your ACT adventure games.

You should be aware that pictures made in this way cannot re-scale the graphics or text windows in ACT, and it is important that you select the correct size of screen region when you use SCNcom for pictures intended for ACT. This is dealt with in more detail in Section 7.6.

You can also restore a compressed picture directly by using the QREST utility. This operates as a procedure from SuperBASIC and will restore a picture file by typing the following command:

```
QREST #n
```

where "n" must be a SuperBASIC channel number which has been opened to a file containing a compressed screen. Once you have opened a file in this way, you can re-issue the call to QREST as often as you want; there is no need to close and then re-open the channel in order to re-display the picture data.

QREST will assume a default channel number of #3, and the use of "#" preceding the channel number is optional. SCNcom needs to obtain working memory when it is first run. Assuming it finds enough (32K), it will operate as described above. If it can't find this much memory, it will still operate, but with some differences in presentation and also with very much reduced compression performance.

The main difference in presentation when it runs with insufficient memory is that you are not able to adjust the compression window while the appropriate picture is displayed. Instead this must be done at the same time as the output file is specified.

We recommend that you should always try to use the utility when enough free memory is available to allow it to work properly.

Very little extra memory overhead is required for the QREST decompression routine (or the appropriate routine included in the ACT system). This should work well even when nearly all the free memory on your QL is being used.

The effective improvement in storage space provided by SCNcom will be reduced if small windows are used. It is even possible that very small windows may actually produce larger files than if saved by conventional means. This is unlikely to be the case for "sensibly" sized portions of the screen, where the file compression factor of 50% or better should still apply.

Depending on the complexity of a window's contents, low, wide windows are likely to compress more efficiently than tall, narrow ones. This effect will be most noticeable for windows less than about 1/20th or so of the full screen width.

SCNcom presents very few limitations. It is, however, not restricted by the boundaries of the picture or text windows when used with an ACT adventure.

Any alteration of the window boundaries of the full-screen picture will not be replaced or corrected to the normal black PAPER if a screen prepared with this utility overwrites the area surrounding the boundaries of the text and normal picture windows. Text and normal illustration windows are 472 pixels wide. The effect described would be produced if a picture filling the full 512 x 256 pixel screen were used.

The following points are worth stressing:

1. The complexity of the screen will affect the achievable compression ratio as well as the time taken to compress the screen.

2. Very complex screens could take several minutes to compute the most efficient compression. The compromise we have taken is to provide a utility which is the most logical for a user of an ACT game: SCNcom is relatively slow in saving and fast in loading.

3. In order to provide you with the most flexible utility possible, we have ensured that you retain control of the keyboard at all times. But do be careful not to alter the contents of the screen during the compression process.

4. Decompression time is not especially dependent on complexity, although large areas of a single colour (or stipple) will reproduce faster than the more complicated areas of the picture.

7.2.2 GDI_1_task, the ACT / Graphics Designer Interface

GDI_1_task may be started with either the EXEC or the EXEC_W command. The notes in the ACT User Guide (Section 2.1.1) explain the advantages of both alternatives, if you are unsure.

In operation, GDI_1_task reads Graphics Designer Text/Coordinate files (saved with the _txt option, the first SAVE option in Graphics Designer) and produces a MORE compact version which can be used by the PIC1 utility or form input files for GDI_2_task to process into a composite ACT picture data file.

To simplify the specification of the numerous files you will need for an adventure game, GDI_1_task allows you to define default devices for both input and output files as well as default name extensions that it will automatically add to each input and output file name you specify.

When the program is first started these defaults are:

```
Input Device  =   flp2_
Output Device =   flp2_
Input suffix  =   _txt
Output suffix =   _APIC
```

and you are given the opportunity to alter these, if you wish, before you specify the first file for processing.

If you chose to accept the 'standard' defaults, and you specify the file "Loc1" for "Input" and "Loc1" for "output", the actual respective file names used by GDI_1_task would be:

```
Input file  = flp2_Loc1_txt
Output file = flp2_Loc1_APIC
```

In this way, you can quickly specify each file you need to process with a minimum of typed entry to the program.

If you need to change any of the defaults while the program is running, this is accommodated by an option given before you specify each new input file.

You can use GDI_1_task to provide some control over how a picture is drawn by ACT. For each picture you convert, GDI_1_task will ask two questions that allow you to "tailor" the picture for different purposes. The first of these questions is whether the previous picture should be erased before the current one is drawn.

This is controlled with the use of Graphics Designer's WIPE option. All pictures produced by Graphics Designer start by setting the PAPER to black, the INK to white, and the drawing mode to "Normal". A WIPE function is then executed, leaving a blank screen (as you normally get when you first start Graphics Designer). GDI_1_task allows this initial WIPE to be removed from the picture, so that the previous picture data is NOT erased.

This option allows pictures to be built up from separate files and is used in ACT to allow "Objects" to be drawn on top of "Locations".

You would normally include the initial wipe for a Location picture but exclude it for Object pictures. Any "null" picture (blank screen) that you want to include in your game should retain the initial "wipe".

This option is not available with the PIC1 viewing/testing utility, which will always clear the screen prior to drawing any picture.

The second question relates to the repeating fill options available with Graphics Designer's QFILL1 and QFILL2 options, both of which provide the options to draw pictures in which a picture element may be repeatedly filled, possibly with different combinations of colours.

If you include either of these fills in a drawing for ACT, you will need some way of switching off the fill when a new location picture is drawn. GDI_1_task gives you the option to stop any existing fill job before it is drawn. Normally this should be done along lines similar to the "wipe" option; location pictures should normally be arranged to stop all previous repeating fills, whilst objects should not stop them.

ACT has no built-in restrictions regarding the possible combinations of presentation you may use, but choosing among the many options available should be done with care.

A good general rule to follow is to consider, if you are going to use repeating fills, how having a split-mode screen (detailed later on in this Appendix) as well will affect your finished product.

The use of repeating fills can have a delaying effect on the split-mode feature under certain conditions. This is most likely to occur with QFILL1's repeating fills and will show up as a flickering region near the top of the text window if a FILL job is active AND the split-mode screen is in use.

The PIC1 utility works in the same way as QREST as described in Section 2.1; it provides the fastest way of re-producing Graphics Designer drawings, combined with the smallest possible picture storage space. Open a picture file (produced by GDI_1_task) to a SuperBASIC channel and use the command

```
PIC1 #n
```

to reproduce the picture. In the same way as QREST, you can repeatedly re-draw the same picture if you wish; PIC1 will use the same default channel number of #3.

PIC1 supports the QFILL1 and QFILL2 options. If one of these is used in a picture to produce a repeating fill, you can also use PIC1 to stop the fills by issuing the command:

```
PIC1 -1
```

Note that this will only work on fill jobs created by PIC1. If you have your Graphics Designer extensions loaded and have used either QFILL1 or QFILL2 directly to draw a repeating fill, then either of these may be stopped only by the appropriate call to QFILL 1 or 2. Conversely, PIC1 cannot itself stop a fill job created by either of the QFILL routines themselves.

You can also stop a repeating picture drawn by PIC1 by using PIC1 to draw another picture which has been prepared with the option to stop all current fill jobs, as described above.

7.2.3 GDI_2_task, the Picture Compiler

This program works in a way similar to GDI_1_task. It allows you to set up default input file types and to specify the default input device. The only requirement of this utility is that you must sequentially specify each file, modified by either the GDI_1_task (the default _APIC type) or SCNcom utilities, which you wish to include in the composite picture file. Please note: SCNcom_task format files will not alter the window sizes of the ACT game, and you should be careful to ensure that these files are only used within your game when the current window size is set up correctly.

The required sequence for compiling your pictures is:

```
  Location pictures - Starting with location 1.
  Object pictures   - Starting with object 0.
  BLANK_APIC        - Required by the system programs.
  Extra pictures    - Such as Mini_Adventure's "explosion".
```

You can include as many extra pictures as you wish; see Section 7.5 for more details.

To allow the large composite picture files to be produced without the need to manually specify each file in turn, GDI_2_task allows all files that are to be incorporated to be specified by a suitable 'command' file. This is a simple list of file names (exactly as you would otherwise input them to the utility by hand), and this option is simply selected when the first file (or any subsequent file) is specified.

You can prepare a 'command' file using any suitable text editor. QUILL can be used, for example, although you will need to PRINT the data to a file in order to produce a suitable format.

Alternatively you can use MSGedt_task. This does involve the restriction that you can't add new file names except at the end of the file. If, say, you add a new location to your game as you develop it, if you have also added the object file names to your command file, you will find that there is no way of including the new location file in the correct place between the current last 'location' filename and the first 'object' name.

The way round this is to use three command files, one each for locations, objects and extra pictures.

These three command files can then be specified in turn to the GDI_2 utility, and in this way all of the files specified in each will be added to the composite picture file as required. However, by keeping the main picture categories separate, it is now possible for MSGedt to add new locations/objects/extra pictures as you develop your game.

7.3 Extra Graphics Extensions

We now provide more detailed information about the graphics machine-code extensions as well as some extra tips concerning the utility programs. The LASTpic_dta and LASTpic_QFILL_dta files contain a set of eight machine-code additions that provide a variety of new facilities for the ACT system.

Since the two system program-merge files will make use of them for you, adding sophisticated graphics to your adventure games, there is no special requirement for an in-depth knowledge about how they work. If you are a more adventurous user, the following details will allow you to make use of these additions to add special features not provided by the supplied modifications, such as the addition of extra pictures to illustrate novel features in your game.

The eight machine-code routines are used in an ACT game by means of the CALL command. The use of this is detailed in the ACT User Guide, Section 3.5. The new routines are:

CALL #1. ACTPIC_bin. This is the main picture-drawing routine; it contains all of the required code to service both Graphics Designer and SCNcom Screen Compressor pictures. Note that this routine is different in the non-QFILL and the QFILL version of the file (LASTpic_dta and LASTpic_QFILL_dta, respectively). The exclusion of the code to support the two QFILL functions saves over 3K of file length. A typical line to draw picture number #n might be:

```
LOAD_VAR 0,1 : LOAD_VAR 1,#n : CALL 0,1
```

This line will call for the nth picture to be drawn. The return value in the call parameter (Variable 1, in this example) will be 0 if the picture was found and successfully drawn, or a negative number if there was some error.

CALL #2. CLRPIC_bin clears the picture window. The value in the call parameter is not used and will not be altered. There is a small difference between the versions of this routine in the QFILL and non-QFILL versions of the file.

CALL #3. CLRTXT_bin clears the text window and resets the freeze-screen counter to zero. The value of the call parameter is not used and will not be altered.

CALL #4. HOLDPIC_bin only returns to the program after a key is pressed. The character pressed is NOT lost but will be subsequently picked up by the ACT pre-parser.

For this reason, it is better for the game to prompt players with the instruction to "Press F5 to continue" when this routine is used. This will avoid unwanted characters from getting into the parser, which, like SuperBASIC, will ignore non-printing characters. The value of the call parameter is not used and will not be altered.

CALL #5. MODE4_bin is used in the same way as SuperBASIC's command MODE 4. The value in the call parameter is not used and will not be altered.

CALL #6. MODE8_bin is used in the same way as SuperBASIC's command MODE 8. The value in the call parameter is not used and will not be altered.

CALL #7. QMODE_bin will return a value in the call parameter that indicates which mode the computer is in. The value will be 0 for MODE 4, or 8 for MODE 8.

CALL #8. SPLIT_bin is used to provide the dual-mode screen option. The value in the call parameter is used to select one of four different functions:

0 - This will switch off the Split Mode feature. This must be done before the game stops, since otherwise, if the split-mode mode feature is left running, the computer will crash as soon as the QL makes use of the memory that the game occupied.

1 - switches on the Split Mode feature. The screen will be divided between the picture and text windows, with the picture in MODE 8 and the text in MODE 4. The routine will automatically adjust the switching point if different-size windows are used.

Note that the Split Mode feature uses up a lot of processor time; the larger the picture window, the slower the game response to the player's commands.

For this reason we do not recommend the use of Split Mode if your game uses pictures larger than half-screen.

2 - switches the computer to 'pseudo' mode 4. This is a software change only and doesn't affect the screen contents or the QL hardware in any way.

This must be done prior to the output of any text (in the lower window). It fools the QL into thinking that it is permanently in mode 4.

3 - switches the computer to 'pseudo' mode 8. This is a similar function to '2' (pseudo mode 4) but is used prior to drawing a picture.

7.4 More About ACT's Split-MODE Screens

In order to provide you with the maximum flexibility for the presentation of your game, we have left a number of options open to you, which, if all are used simultaneously, may produce unpredictable results.

If split mode and the QFILL routines are used at the same time, flickering can occur at the top of the text window.

The two program-merge files mentioned in the preceding two Sections contain the required code to support the screen's Split-Mode facility. In addition a switch is included to permit the feature to be switched ON or OFF when the game is first run.

The control of this is achieved through the spare Location 0 parameter number 2 bit flags 2 to 4. Bit 2 will directly control the operation of all of the CALLs (described in Section 1.7.3) to the Split routine; the flag must be set to enable the CALLs.

Bits 3 and 4 are used by the new routine check_for_mode in the player program and tell it when to operate. A few additional lines in the EVENT program are also involved.

You can use the program startup feature as it stands, or permanently enable or disable the split feature by:

1. editing the LOCN_dta file (with the LOCedt utility) to set Location #0 parameter #2 bit #2 as required; and

2. removing the following lines from the two merge files before using them to create their new PROG_dta file.

   Player_Prog_Additions: Delete 155, 156, and 30620 to 30760.
   Event_Prog_Additions : Delete 1063 and 1064.

As we have said many a time before, you should be aware that the split screen puts a great demand on processor time. The use of half-screen pictures noticeably delays the speed of the Mini_Adventure program. This time penalty is increased for larger pictures. For this reason the use of pictures larger than half-screen is not recommended.

The Split Screen mode available with ACT relies on a software timer that is set up to switch the QL's hardware between MODE 4 and MODE 8 at the correct point in each screen-refresh cycle.

This timer-and-switching function is set up as a polled routine within the QL and relies on being started at the correct point in the correct screen display cycle, so that the mode switch is in the desired place on the screen, between the text and the picture windows.

This normally works reliably, provided nothing happens to delay the starting of the polled routine. It is possible, however, for certain other jobs in the QL to delay the start of the polled routine. In particular, the use of microdrives has such an effect. For this reason, mode switching is disabled while a microdrive is running.

We do not generally recommend the use of repeating fills with an ACT adventure if the split-mode screen is also to be used. If you wish to combine both of these features in your game, we suggest that you keep the JOB priority of the repeating fills (set within Graphics Designer) to very low values AND have no more than one or two such fills active at any time. If you are planning your game for commercial release, we suggest that you make some comment about the possibility of any text-window flicker in the documentation accompanying your game.

Certain types of memory expansion can cause problems when using split modes in your display. This problem and some possible cures are detailed at the end of Section 1.7.3.

7.5 Making Alterations

If you want to add other pictures to your adventure, you should add them to the composite picture file after the BLANK_APIC picture. You can then draw additional picture number #n, using a line in the player or event program something like:

load_var 0,8 : load_var 1,3 : CALL 0,1 :
numb_obj 0 : numb_loc 1 : var_add 0,1 : add 1,#n : load_var 0,1 : CALL 0,1 :
load_var 0,8 : load_var 1,2 : CALL 0,1

This looks a bit complicated, but it is really quite simple. The first three commands call the external routine number 8 (SPLIT_bin) to put the QL into 'pseudo' mode 8. The next three establish how many Objects and Locations there are in the game and put the total into Variable 1.

The next command (add 1,#n) modifies this total to point to the required picture. It skips the explosion picture which is contained in the illustrated Mini_Adventure; this would be equivalent to a value of 0 for #n.

The next two commands CALL the first external routine (ACTPIC_bin), which will draw the appropriate picture number as contained in Variable 1.

Finally, the last three commands re-call external routine number 8 but this time to put the QL back to 'pseudo' mode 4, ready for any subsequent text output.

If you remove the split-mode option from your game, as described previously, then you don't need any of the commands associated with the pseudo mode changes; it won't matter if they are included, though.

7.6 Additional Comments on Graphics Designer Pictures

ACT will correctly re-produce all pictures produced by Graphics Designer; however, there are some restrictions that you should be aware of. ACT will accommodate any valid picture window size. Each time a new picture is drawn by ACT, a check is made on the required height. If it is the same as the last picture drawn, ACT will proceed to draw it immediately.

If the height is different, ACT will alter the window size of both the picture and text windows to suit. When this happens, BOTH windows will be cleared, and so any displayed text will also be lost. For this reason, we recommend that you try to keep changes in picture size to a minimum.

If you make the picture window too big, you will have a text window that is too small to allow any text to be output by ACT. If this happens, the game will seem to "hang"; unless you have toolkit utilities to give you direct job control, you may not be able to do anything besides press the reset button!

This problem can occur with picture windows higher than 220 pixels. It is posible to use pictures up to the maximum of 230 if you wish; just be sure that ACT doesn't try to output any text while a picture of this size is displayed. The Mini_Adventure title screen is an example of a large picture which won't cause the QL to hang up.

Making a "NULL" picture with Graphic Designer is simply a matter of selecting the correct screen size and then SAVE the blank Graphics Designer screen as a file, selecting the "txt" option.

When objects are added to a picture, remember that they might be drawn in front of some feature in the background, depending on which location picture they happen to be drawn with at any time.

For this reason, be careful about using the Complementary Colours option available with Graphics Designer. If parts of an object are drawn using this option, you will 'see' any background details through the object when it is drawn. This option can be useful for "ghosting" effects.

You might also bear in mind the order in which objects will be drawn. Whenever the player drops an object that is currently carried, this is added to the current picture contents. When a new location is drawn, either as a result of the LOOK command or when moving about the adventure, the objects are added to the picture in reverse numerical order.

Two features available within Graphics Designer which cannot be used within the ACT system are GRID and the delay (HOLD). If a drawing is produced using either of these options, when the pictures are converted by the GDI_1_task utilitiy, the internal file commands calling for these functions will be ignored.

7.7 Additional Information

Before preparing your drawings for an ACT adventure, it is a good idea to make up a "story board" for the game plan and required illustrations. Having a specification to work from will save considerable time and result in a more compact game.

You should remember that prior to making drawings it is good practice to pre-plan your drawing requirements on paper. Frequent changes of INK or PAPER will increase the size of the drawing file. Wherever possible, elements of the same INK or PAPER should be made together.

As stressed from the beginning of the ACT User Guide, you should plan carefully before either altering the Mini_Adventure or starting a completely new game of your own.

You should NOT try to remove any of the eight machine-code additions contained in either LASTpic_dta or LASTpic_QFILL_dta, unless you are careful to check that all references to the removed routines are also deleted from both the system programs. Failure to observe this precaution may produce unpredictable results.

Keep in mind that when LSTedt removes a machine-code addition from the data file, all subsequent routines are re-numbered. In practical terms this means that if you remove routine #3 (CLRTXT_bin), ALL references to routine numbers 4 to 8 would have to be renumbered (#3 to #7).

We recommend that none of the eight graphics routines be removed. You can add any of your own routines onto the end of the eight graphics ones (i.e. starting with CALL routine number #9).

Most of the graphics routines are quite short (except for the first, ACTPIC_bin). Many of these, such as CLRTXT_bin, MODE4_bin, MODE8_bin, and QMODE_bin, may be used in a text-only game. In this case you could remove the unwanted routines from the system and leave only the ones required. If this is done, we recommend that you DO NOT use the two supplied merge files but write your own additions to the original programs.

8.0 APPENDIX 3 - ADDING SOUNDS AND FONTS

If you are anything like me, you will never read documentation associated with computer software. Instead, you will immediately load each new program you get into your computer and expect to understand how to run it without ever needing to turn the pages of the user guide.

Well, provided you know a bit about the ACT system, you can probably get away with using SNDedt without ever reading beyond the first line of these notes.

If you need a little bit of help with actually putting your sound effects into an adventure game, you will probably need to glance at Section 8.7 of these notes. Hopefully the experience won't be too painful.

If, however, you are either one of those rare people who always read software documentation thoroughly (well done, do keep it up!) before ever attempting to use the program, or you simply can't figure out how to use SNDedt, then I'm afraid you will have no option except to read the boring blurb that follows. Hopefully that experience won't permanently damage your health!

8.1 What Is SNDedt?

SNDedt is an extra utility for the ACT system that allows sound effects to be added to any adventure game produced. The basic philosophy that SNDedt has been designed with is that ANY sound you can construct, using the SuperBASIC BEEP command, can also be incorporated into your game. In addition, SNDedt also allows multiple BEEP's to be combined, along with short pauses if required, and the resulting composite sound can be simply called by the game as though it were a single sound.

In order to maintain similarity with the Superbasic BEEP, SNDedt uses a similar set of parameters to characterise each sound. All of the sounds that are to be used in a game - up to 999 are allowed - are simply displayed in SNDedt's editing window. Changes to the sounds are made by the use of the QL's cursor keys to select or modify parameters, and sounds may be heard at any time, so that the effect of a change is immediately apparent.

To further simplify the task of creating the sounds you require, SNDedt can provide a page of text 'on line', which describes each parameter that controls how a given 'noise' sounds.

8.2 How Are Sounds Incorporated into an ACT Game?

While you are developing the sounds for a game, you can write and read the current repertoire to a storage file. When you have completed all the sounds you want to include, you simply select a different output option from the SNDedt menu, which produces a module in the format required by the ACT LSTedt utility. It is this module that is actually incorporated into your adventure game.

The module contains the information required to reproduce all of the sounds created by the SNDedt utility, along with a short Section of machine code that allows the ACT system to actually create the sounds.

Once this module is added to the ACT last data file by the LSTedt program, the sounds can be simply produced in the adventure game by using the ACTBASIC 'CALL' command. Only three ACTBASIC commands are needed to call a sound, and if a suitable 'subroutine' is 'NAMEd' in either the PLAYER or EVENT programs to call a particular sound, then only 3 bytes of space are taken up for each situation, or place in the programs where your game is to make a particular noise.

8.3 Starting SNDedt

Like all the other ACT utilities, SNDedt is designed to run as a separate job on your QL. To start it, use either the SuperBASIC EXEC_W or the EXEC command. If you are not familiar with running programs in this way, you should consult the ACT user guide. Section 2.2.1 describes both of these commands and their different advantages.

SNDedt uses quite a large amount of memory. If you don't have any extra RAM on your QL, then you might well find that you are unable to get SNDedt to run if you are also running other programs at the same time or if you have a large SuperBASIC program loaded. In this case you should RESET your QL and try again, remembering to remove any microdrives while you press the reset button!

If you start SNDedt using the EXEC command, you might have to use the CONTROL 'C' switch in order to get the program's cursor flashing. The relevant cursor is located on the left-hand side of the title window, at the top of the screen.

8.4 Finding Your Way Around the SNDedt Screen

There are four main 'window' areas presented on the screen, once SNDedt has started. The top of the screen is used for the program title window. Apart from the input cursor, as mentioned in Section 3, this window also contains the version number of SNDEDT.

The right-hand side of the screen is used to display a menu of commands or help text describing each of the parameters associated with the sounds. The program will start with the commands menu displayed, but you can select between this and the help text by pressing the 'H' key, as indicated in the menu. If the help text is selected, the appropriate text for the parameter currently selected in the main editing window will be displayed.

To the left of the screen is the main editing window. SNDedt can have up to 999 sounds defined, but the window is not large enough to display more than 10 at a time, so if there are more sounds than will fit into the window then the program will simply display a Section of the data. The actual sounds displayed may be selected either by the use of the up or down cursor keys, in order to browse up or down through the sounds data set, or by the use of three selecting 'commands' as described in Section 8.6. The currently selected sound and parameter is indicated by a colour change on the screen; you can select different parameters by the use of the left or right cursor keys.

The region at the bottom of the screen is used for prompts and extra information that might be required. Such use will only occur when certain options are requested, such as to read or write a data file.

8.5 How To Use the Editing Window

The data in the editing window consists of individual rows of 10 numbers, each such row defining a single 'sound'. The first of the parameters is simply the sound number. This will be in the range 1 to 999, and the numbering of the sounds will always be sequential.

Parameters 2 to 9 actually define the sound. These numbers are the same as the 8 parameters that the SuperBASIC BEEP command uses. If you want more information about them, you can consult the QL User Guide, or you can read the SNDedt help text for each one.

With the SuperBASIC BEEP command, the first parameter, the duration of the BEEP, can be in the range 0 to 32767. A value of 0 indicates that the sound is to last indefinitely. With SNDedt, a value of 0 isn't allowed, since this would lead to a sound that would effectively 'hang up' the adventure game, control only being passed back to the ACT system once a sound has stopped. Although a zero duration value is not allowed with SNDedt, the appropriate parameter can have negative numbers. These are used to select a pause of the length a 'sound' with the same positive number would have. If a pause is defined, the other BEEP parameters, 3 to 9 in the SNDedt window, do not have any effect.

The last parameter, for either sounds or pauses, is used to select what happens once the current sound has stopped. A value of 0 indicates that this is the end of the process: once the sound has finished, control is passed back to the adventure game. If this parameter is greater than 0, then at the end of the current sound a new sound will be started, as indicated by the actual value of the parameter.

When SNDedt is first started, there will not be any defined sounds. In this situation, or whenever the last defined sound is actually displayed in the editing window, SNDedt will display an extra line in white strip. This is the 'next available sound' slot; you will find that this is sound number 1 when the program is started.

In addition, if either the first or the last sound defined is currently displayed in the window, SNDedt will display a special line in green strip, indicating that this is the current top or bottom of the data set.

To change the value of a parameter, use the SHIFT up or SHIFT down cursor keys to increase or decrease the number. As soon as any parameter is changed in the 'next available sound', SNDedt will add this to the data set as a 'defined sound'. When this happens, the sound will be displayed normally, that is with black strip, and a new 'next available sound' will be added in the next slot. Once a sound has been defined, you cannot remove it, although you can Zap the entire sound table if you want, or you can re-read the data, if it was previously stored in a file.

If parameter 10 of a sound is not 0, indicating that this sound links on to some other sound once it has finished, then SNDedt will display the sound in green strip. In this way it is easy to see which sounds in a data set form part of a multiple set of sounds, since only sounds with black strip actually stop the BEEPing process.

When changing the value of a sound, using the SHIFT up or down keys, you can alter the values over a large range quickly by holding either of the SHIFT cursor keys down. When this is done, SNDedt will, after a few moments, start a more rapid alteration of the value than you obtain by single keypresses.

8.6 SNDedt Commands

Cursor keys.	These are used to select the 'current sound' and also the 'current parameter'. The current parameter in the current sound is indicated by having red strip, as opposed to black, green or white.
SHIFT up or down cursor.	These are used to alter the value of the current parameter in the current sound. Values can only be altered within the valid range for each. The help text, available for each parameter, describes what each one does, as well as the allowed range of values.
SPACE.	This will cause SNDedt to output the current sound. The current parameter position is not important; only the current sound is output, even if parameter 10 indicates that another sound will normally follow. This option allows individual sounds in a sequence to be heard on their own.
ENTER.	This will cause the sound sequence, starting with the current sound, to be output. This allows a composite set of BEEPs to be heard, exactly as they will be produced by the ACT system, once the machine-code module is dumped by SNDedt and installed into an adventure game by the LSTedt utility. To help keep track of which sounds are involved in a series as each sound is output, the appropriate entry in the editing window is indicated by a marker to the left of the first parameter column. Obviously, if the sounds link to a number not actually displayed within the window, no marker can be displayed for each such sound.
Zap.	This is selected by pressing the 'Z' key and will cause all sounds to be deleted from the current list, leaving the program in the start-up state, i.e. with no sounds defined.
Write.	Selected by pressing the 'W' key. This will write currently defined sounds to a file. You will be prompted for the required file device and name. The default uses the extension '_dta' on the file name,

and it is suggested that you continue to use this convention in order to differentiate the program 'save' files from the machine-code module output.

Read.	Selected by pressing the 'R' key. This is the opposite of the Write option. It is used to read the sounds back in from a previous write.
Go to.	Selected by pressing the 'G' key. This alters the current sound to any value. You will be prompted for the sound number you want to 'go to'. This is of most use when you have many sounds defined: it may then be much quicker than scrolling through the data set using the up or down cursor keys.
Top.	Selected by pressing the 'T' key. This selects sound number 1 as the current sound.
Bottom.	Selected by pressing the 'B' key. This selects the last defined sound as the current sound.
Quit.	Selected by pressing the 'Q' key. As you would expect, this stops the SNDedt program.
Dump.	Selected by pressing the 'D' key. This is used to write out the currently defined set of sounds as a machine-code module suitable for the ACT utility LSTedt to incorporate into your adventure game. Note that such a dumped module can't be read back into SNDedt, only the program 'save' files being suitable for reading by the Read command. The default 'dump' file name is given the extension '_bin', and as for the 'read' file extension, it is suggested that you maintain this simple extension convention in order to identify which of your data files are machine-code modules. Section 8.7 explains how you can incorporate the machine-code module into your adventure game and how to add calls to the sounds you have constructed in either the PLAYER or EVENT programs.

New.	Selected by pressing the 'N' key. This simply re-draws the entire SNDedt screen. The main use is if you are running SNDedt along with other programs: if another job changes the screen, you can simply use the New option to remove the corruptions.
Help.	Selected by pressing the 'H' key. This simply toggles the help/information window between a summary of these commands and the parameter help information.

8.7 Loading the Machine-Code Module into a Game and CALLing the Sounds

8.7.1 Adding the Sounds Module to the ACT Last Data File

Once you have completed all the sounds you want to incorporate into your game, you should use the Dump option in SNDedt to create a machine-code module. This module must be combined with the ACT last data file in order to make use of the sounds in your game.

The last data file is called LAST_dta in the basic (text-only) ACT system, or either LASTpic_dta or LASTpic_QFILL_dta if you have added the optional Graphics interface kit. This file contains some extra information required by the ACT linker when it combines the other data files to form a completed game. In addition it can also have up to 255 machine-code modules incorporated, which may be executed by the use of the ACTBASIC 'CALL' command from either the PLAYER or the EVENT program.

In the basic ACT system, i.e. the LAST_dta file, there are no machine-code modules included, while either of the appropriate files in the Graphics interface kit has 8 machine-code modules included already. To add your sound module, you simply run the LSTedt utility in the same way as all the other ACT utility programs are run, and then, after reading in the appropriate ACT last data file, select to alter option '2', the number of machine-code modules.

This done, you will be presented with a list of several options. You should choose 'A', to add a new module to the file. Simply specify the sound 'Dump' module that you have created with SNDedt, and LSTedt will read the data into the file.

You will notice that you can also use LSTedt to remove a module if you want. This option allows you to modify your sound module after you have added extra sounds, say. The procedure is to remove the old copy and then add the new one in its place. When removing a module, you must be careful. It is possible to remove any of the modules currently loaded, but if you remove one of the other modules already installed, the only way to re-establish it is to re-start from a backup copy of the last data file. You should be aware that you can ONLY add new machine-code modules to the end of the modules already installed; LSTedt doesn't allow you to insert a module in between existing ones. This restriction is deliberately included in the ACT system in order to avoid number changes for existing modules, which would require a similar change to any call to such modules from the ACTBASIC programs.

When you install your sound module, make a note of the module number, as you will need it to make calls to the sounds from the ACTBASIC programs. For the basic ACT file LAST_dta, the sound module will be number 1, while for either of the Graphics interface kit alternative last data files there are already 8 modules loaded, so that for them the sound module will be number 9. Of course, if you have added modules of your own or removed modules from the Graphics files, these numbers will be different. In this case LSTedt will tell you which number it has added the sound module as.

The sound module is quite short. It will add 132 bytes to the length of your game, plus 10 for each sound defined.

8.7.2 CALLing the Sounds from ACTBASIC Programs

In order to produce the sounds in your game, it only remains to add calls to the appropriate sound module by using the ACTBASIC 'CALL' command. This command is described in the ACT user guide, Section 3.2.3. As an example, suppose your sounds have been installed as module number 9 by the LSTedt utility. In this case the following three ACTBASIC commands will cause sound number 15, within the module, to be produced by the game.

```
LOAD_VAR 0,9 : LOAD_VAR 1,15 : CALL 0,1
```

More generally, the call to the sound module should be

```
LOAD_VAR 0,#n : LOAD_VAR 1,#m : CALL 0,1
```

where #n is the appropriate module number of the sounds, and #m is the required sound number. Obviously, any of the ACT system variables can be used for the call. For example, if variables 0 and 1 are being used for some other purpose at some point in the program where sound output is required, you should substitute some other variables. One simple way to avoid any possibility of changing the values of variables which need to be preserved is to allocate two extra variables for the exclusive use of the sound module.

As explained in the ACT User Guide, up to 256 variables may be used in a game, and the mini adventure demonstration game only uses 17, i.e. variables 0 to 16.

To allocate extra variables, simply alter the appropriate option in the ACT last data file, by using the LSTedt utility. You could actually do this at the same time as you add your sound module, if you want. If you do allocate extra variables in this way, you could make NAMEd subroutine calls to each of the sounds in the module in order to simplify the process of inserting the sounds into the logic of your game.

To do this, you might use the following example code lines. They can be added to either the PLAYER or the EVENT program, and the subroutines will be executable from BOTH programs, no matter which they are actually defined in.

```
32000 NAME sound_1:LOAD_VAR 17,#n:LOAD_VAR 18,#m:CALL
17,18:RETurn
32002 NAME sound_2:LOAD_VAR 17,#n:LOAD_VAR 18,#v:CALL
17,18:RETurn
```

Here sound_1 will call sound number #m from your sound module, while sound_2 will call number #v. You can obviously make the actual names of the subroutines a bit more descriptive than in this example. Here it is assumed that variables 17 and 18 are assigned to the sound module; if you are already using these in your programs, you should obviously substitute suitable alternative values.

Once your sounds are defined in this way, you can call them by simply referring to the appropriate subroutine name whenever you want a sound to be output by the game. For example, look at line 2720 of the distributed version of the PLAYER program. This line is:

```
2720 print_score : STOP,
```

and is executed whenever the command 'SCORE' is entered to the game. If you insert a call to sound_1,

2720 print_score : sound_1 : STOP,

the game will also output the appropriate sound whenever it responds to a request for the current 'score' by the player.

Clearly there are lots of places in the distributed programs where you might add sounds. You will also probably want to add sounds to some of your own additions to the game programs. No matter where you decide to add sound effects, the principle is the same. Once you have defined the sound subroutines, each call to a sound only uses up 3 bytes of extra space in your game. In this way it is possible to add many sounds to your game while only adding a few hundred bytes to the length of the game.

8.8 The Sounds Used in IMAGINE

You will find an example SNDedt sound 'save' file included with your copy of the utility. This file has the default SNDedt load name of 'SNDedt_dta'; so to read it in, simply select the Load command and press ENTER when the program prompts for a file name.

There are 31 separate sounds defined in this data file, although several are linked into multiple BEEPs. This set of data in fact constitutes the sounds used in IMAGINE.

The following list describes the 16 separate 'composite' sounds defined in this file, along with a short description of what each is used for in IMAGINE. You are welcome to use this data set as a starting point for the sounds to be included in your own games, if you want.

Sound number	Description
1	A short tune. This is used when the player gets a point, as well as when the game first starts.
12	This is used whenever the player is killed.
14	Re-incarnation.
15	Jumping, wherever there is a route down.
16	Jumping, this time where there isn't a path down.
17	Player's a dummy. Used frequently, whenever a command is not successful.
20	Explosion.
21	Done. Used frequently, whenever a command is successful.
22	The phantom's arrival.
24	The mouse.
25	Electric shock, caused by touching the stripe in the VAST room.
26	Nagging.
27	A special sound, reserved for a special situation involving the wife.
29	Touching the cube.
30	The alarm, in the perplexing room.

31 Effect whenever a 'magic' transportation word is used successfully.

8.9 Adding Fonts

If you want to incorporate alternative character fonts into your completed game, such as that used in IMAGINE for example, then you can use the simple program ACTfont_bas.

This program, which is supplied as SuperBASIC, should be LRUN in the normal way. It will prompt for the name of a data file that contains the font you wish to use in your game, this should be in the standard QL format. You can either use your own fonts, or alternatively you can use any of the numerous fonts available from various sources. Although not primarilly intended as a source of QL fonts the accelerator program, LIGHTNING, also available from Digital Precision, contains perhaps the most varied collection of alternative fonts currently available for the QL. Indeed, one of the fonts supplied with LIGHTNING is used in IMAGINE.

The output from ACTfont_bas is a machine code module which should be installed into the LAST_dta (or LASTpic_dta or LASTpic_QFILL_dta as appropriate) file prior to using LINKER_task to form an ACT game. The LSTedt_task utility is used to add machine code additions to the LAST_dta file, for more information on using this you should consult the relevant parts of the manual, in addition the Section on SNDedt_task also gives specific details about how to incorporate an extra machine code module.

It is also necessary to add a specific CALL to the FONT module in the EVENT_prog source file. The required addition is

LOAD_VAR 0,1 : CALL 0,1	appended to the END of line 1070 in the EVENT program if you are using the text-only form of the system programs. This also assumes that the font module produced by ACTfont_bas is incorporated as the FIRST module in the LAST_dta file.
LOAD_VAR 2,N : CALL 2,N	this should be inserted AT THE BEGINNING of the existing line 1063 if you are using the illustrated version of the system programs, that

is is the two _additions files have been MERGEd with PLAYER_prog and EVENT_prog. Note that the 'N' represents a number, this being the relevent number that the font module is loaded as by the LSTedt_task utility. N will most probably be 9, although it might be a larger number than this if you have also included other machine code additions. In any case LSTedt_task will always tell you the appropriate number whenever it installs a new module into any of the LAST_dta files.

9.0 APPENDIX 4 - QFILL1 AND QFILL2

QFILL1 and QFILL2 are two machine-code extensions that enhance the simple area-filling facilities of the QL and also allow simple 'animated' filling effects. The routines are both incorporated in Graphics Designer; in addition, each may also be used independently from SuperBASIC as well as under the ACT graphics system.

QFILL1 is an advanced shape-drawing routine. It can be used as an alternative to the native QL 'FILL' whenever an irregular shape made up of a sequence of DRAWn lines is required filled, rather than as a simple 'outline'. The main improvement offered by QFILL1 is that it allows re-entrant shapes of any complexity; it will even cope with shapes that have interSect_ing boundary lines. This allows it to draw shapes of any form. By the careful choice of outline it is possible to use a single call to QFILL1 that will produce such shapes as a 'star', a 'spiral' or even solid shapes, such as a box for example, but with embedded 'holes'.

QFILL2 is a powerful re-colouring routine. It may be used to change the colour of any area on the screen. It will cope with filling solid OR stipple colours in either screen MODE and can even re-colour an existing stipple with another stipple colour. Despite this flexibility, QFILL2 is faster than rival routines.

In addition to all these features, both QFILL routines have the facility to operate as independent jobs on the QL. At a simple level, this allows the application program to carry on as soon as the appropriate QFILL has been started. However, by including the options to allow such a job to repeat the fill a selected number of times, each successive time in an alternative colour, it is simple to produce a variety of animated effects. QFILLs operated in this way use up a minimum of memory since most of the 'work' is done by a common 'root' module that may be shared by any number of QFILL jobs. Among the effects possible are flames, flowing water, smoke, flying inSect_s, flapping birds: the list is only limited by your imagination. The example picture, RIVER_txt, demonstrates all of these examples. You can see this picture either by loading it directly into Graphics Designer or by converting it to the _APIC format (using the GDI_1_task utility) and then using the SuperBASIC extension PIC1 to reproduce it. Note that it should be drawn in MODE 8.

9.1 Using QFILL1

These notes are specifically designed to illustrate how QFILL1 may be used independently of the Graphics Designer. However, even if you don't ever intend to use the routine from SuperBASIC, you may still find this Section helpful in fully realising the capabilities QFILL1 can offer, whether as a part of Graphics Designer or stand-alone.

The routine is called as a FuNction, up to six parameters may be supplied.

These are:

```
      val=QFILL1([ #CHAN, ] INK1, STRING$ [ ,INK2, JOB, REPEATS ])
  normal defaults.... 1       --     --         7    0      1
```

Items in square brackets are optional, and the appropriate default is given below each. In detail, the parameters work as follows:

val	This will be set to a number that indicates what error, if any, occurred. If the call is successful, then val will be 0. Otherwise, val will contain the appropriate QDOS error code describing the problem.
CHAN	This is a number which indicates the appropriate SuperBASIC channel to draw the shape in (this should be a screen or console channel, of course). The default value of #1 is assumed if the first parameter in the list isn't preceded by a #.
INK1	This defines the colour that the shape will be filled with. The colour can be either solid or a stipple; in either case it is the normal composite value formed from the MAIN, CONTRAST and STIPPLE values by the relation:

```
INK1=MAIN + (MAIN^^CONTRAST)*8 + STIPPLE*64
```

In addition, negative values for INK1 cause other actions to be selected. INK1=-2 will always result in a return of the current QFILL1 version number. INK1=-1 will stop ALL current QFILL1 jobs.

STRING$ This is a string array of data that describes the shape to be filled. The array consists of co-ordinate pairs with each point using up 4 bytes (or characters) of STRING$. For example, suppose a simple triangle is to be drawn which has its corners at the points:

```
          20,30; 420,30; 220,200

+--------------------------------------------------------------------+
|Top left of window                                           (511,0)|
| (X,Y are 0,0)                                          max X is 511|
|                                                                    |
|  20,30                                        420,30               |
|   ***********************************************                  |
|     *******************************************                    |
|       ***************************************                      |
|         ***********************************                        |
|           *******************************                          |
|             ***************************                            |
|               **********************                               |
|                 ******************                                 |
|                   **************                                   |
|                     **********                                     |
|                       ******                                       |
|                         **                                         |
|                      220,200                                       |
|                                                                    |
|(0,255)                   max Y is 255                     (511,255)|
+--------------------------------------------------------------------+
```

NOTE: QFILL1 draws relative to the origin of the chosen SuperBASIC window. The top left corner is 0,0 and the maximum values of X and Y are 511 and 255 respectively. Usually, the actual limits will be less than these values, of course. Shapes will be reproduced correctly; if they are out of the window range, they will simply be 'chopped' at the appropriate boundary.

For this shape, STRING$ would be set up as follows:

```
STRING$=CHR$(0) & CHR$(20) & CHR$(0) & CHR$(30)
        <================>   <================>
            0+20=20 (X)          0+30=30 (Y)
```

then for the second point:

```
STRING$=STRING$ & CHR$(1) & CHR$(164) & CHR$(0) & CHR$(30)
                  <=================>   <================>
                    256+164=420 (X)         0+30=30 (Y)
```

and finally for the third point:

```
STRING$=STRING$ & CHR$(0) & CHR$(220) & CHR$(0) & CHR$(200)
                  <=================>   <=================>
                     0+220=220 (X)         0+200=200 (Y)
```

Note that QFILL1 will assume the shape to be solid with lines drawn between the points supplied in the order supplied. It will draw the final line between the first and last points specified automatically (from 220,200 to 20,30 in this simple example).

Shapes can be re-entrant and lines may cross: QFILL1 will still do a sensible fill.

There is a limit to the size of the STRING$ that may be passed to QFILL1. This is governed by the size of the work area that the routine takes from the common heap. Currently this is set to 1K, of which about 750 bytes are available for STRING$. This corresponds to a shape defined by up to 187 points.

It would be more convenient to define the shape passed to QFILL1 by an integer array, coords% for example. In this case the example above would be defined by:

```
coords%(1,1)=20   : coords%(1,2)=30
coords%(2,1)=420  : coords%(2,2)=30
coords%(3,1)=220  : coords%(3,2)=200
```

The following FuNction will draw any shape defined by coords%.

```
DEFine FuNction Qfill1_Interface%(length,chan,ink1,ink2,job,no_fills)
  LOCal string$(750),i
  string$=""
  FOR i=1 TO length
    string$=string$ & CHR$(coords%(i,1) DIV 256) &
      CHR$(coords%(i,1) MOD 256) & CHR$(coords%(i,2) DIV 256) &
      CHR$(coords%(i,2) MOD 256)
  END FOR i
  RETurn QFILL1(#chan,ink1,string$,ink2,job,no_fills)
END DEFine Qfill1_Interface%
```

INK2 This defines a second colour that will be used if JOB is set up to cause repetitive fills. It is formed in the same way as INK1 from the required MAIN, CONTRAST and STIPPLE values.

JOB The default value of 0 will cause a fill to operate as part of the calling program code. In this case the function only returns control when the fill is completed, and all other jobs will be prevented from using QFILL1 while the current fill is being done.

Values from 1 to 255 (actually, larger positive values will also work, they are taken as MOD 256) will set the fill up as an independent job. Control returns to the calling program as soon as the job is initiated.

The actual value will determine the priority of the job. Note that negative values of JOB are not allowed and will result in QFILL1 returning the BAD PARAMETER error code (-15).

Up to 10 concurrent fills are allowed. An attempt to start more than ten at one time will get the error code -2. Jobs may either stop automatically (when the appropriate number of REPEATS has occurred) or be stopped by the QFILL1(-1) call.

REPEATS This parameter only has an effect if JOB is >0. In this case, REPEATS controls how many times the fill will be performed before the created job stops itself.

Any valid 16-bit integer (positive) is allowed, i.e. from 1 to 32767. If REPEATS is 0 or negative, the fill is repeated indefinitely. In this case the special call to QFILL1(-1) can be used to stop ALL the current QFILL1 jobs.

Repetitive fills are done in alternate colours as defined by INK1 and INK2.

9.2 Using QFILL2

The demonstration program QFILL2_demo_bas illustrates most of the following notes about the routine. Simply LRUN the program after installing QFILL2 (both QFILL routines are loaded automatically by the Graphics Designer boot program). If you don't like demos, the following notes will reveal all!

A call to QFILL2 is made as follows:

```
                  val=QFILL2( X, Y, INK, [ ENABLE_STIPPLE, BUFFER, JOB ])
   normal defaults.........   -  -  -            0          2048    0
```

Items in square brackets are optional, and the appropriate default is given below each. In detail, the parameters work as follows:

X and Y	These define the position on the screen that the fill is to start from. QFILL2 doesn't use windows; rather, it regards the entire area of the screen as its working window. In operation, fills are only constrained by the boundary of the shape that is being re-coloured, or by the physical top, bottom or sides of the QL display. X and Y are in the range 0 to 511 (X, across the screen) and 0 to 255 (Y, down from the top of the screen); the top left corner of the screen is 0,0. If a call to QFILL2 is made with X negative, this is treated as a request to stop any active repeating QFILL2 jobs (created by a call to QFILL2 with JOB < 0).
INK	This is the colour that the area will be changed to. INK is the composite value formed from the MAIN, CONTRAST and STIPPLE values by the relation: INK=MAIN + (MAIN^^CONTRAST)*8 + STIPPLE*64 If the area being filled contains a colour in common with INK, QFILL2 will fill the area in a neutral (non-common) colour first before proceeding to complete the fill in the selected colour.

ENABLE_STIPPLE

This is a logical switch that is used to allow QFILL2 to re-colour an existing stipple. If ENABLE_STIPPLE is false (0), then QFILL2 will always assume that the area being filled is a solid colour. It determines what this colour is when it first starts the fill, by looking at the pixel element at the chosen starting point, X,Y. If you happen to choose a starting point that is actually in an area which is coloured with a stipple, then QFILL2 will only re-colour half of the stipple pattern, or possibly as little as a single pixel, depending on the actual stipple pattern.

If ENABLE_STIPPLE is true (not 0), then QFILL2 will always assume that the area being filled is a stipple. To this end, instead of simply looking at a single pixel at X,Y in order to determine the current area colour, it examines a small box 2 by 2 pixels in size. This allows the routine to work with any stipple pattern INCLUDING solid colours (which it treats simply as a stipple with both MAIN and CONTRAST colours the same).

There are a couple of small disadvantages with using QFILL2 with the ENABLE_STIPPLE option set to true, though. The first is that fills take a little longer than with the option set to false.

This is unlikely to be noticeable unless frequent fills involving large areas being re-coloured are attempted. The second is that you must be careful to set X and Y away from the boundary of the required fill area. If you do get too close, then it is possible for the routine to include pixel elements from outside the required area boundary within the 2 by 2 box. In this case, you will end up with QFILL2 doing rather peculiar things.

When used within Graphics Designer, QFILL2 is always called with ENABLE_STIPPLE set true.

BUFFER QFILL2 needs to 'remember' information as it operates. This is mostly to do with places in the area that it must 'go back to' after it has completed the current part of the fill. Complicated patterns may have many more such 'go back to' points than simple shapes; for example, a fill around an area containing text will require a large number of such points to be remembered.

If QFILL2 runs out of buffer space while doing a fill, it will indicate this by a destinctive BEEP. The fill will still carry on but there will be areas of the shape that are not completely filled. You can specify how much memory is to be made available for each fill by altering the BUFFER parameter.

The default BUFFER is 2048, which is large enough for most likely shapes. You can specify smaller buffers if you want. This might be necessary if you are using QFILL2 on an unexpanded QL, especially if other programs, such as Graphics Designer, are loaded. Larger buffers might be needed for fills around a large window filled with text, for example.

Note that the default value for BUFFER, when QFILL2 is used from within Graphics Designer, is 256 bytes. You will find that this is enough for simple shapes but will need to be increased if moderately complicated areas are to be filled.

JOB The default value of 0 will cause a fill to operate as part of the calling program code. In this case, the function only returns control when the fill is completed, and all other jobs will be prevented from using QFILL2 while the current fill is being done. Values from 1 to 255 (actually, larger positive values will also work; they are taken as MOD 256) will set the fill up as an independent job. Control returns to the calling program as soon as the job is initiated. The actual value will determine the priority of the job. Negative values of JOB are used to initiate repeating fills. The priority of the job is still determined by the specified value (taken as positive) but instead of the fill job simply stopping when the chosen area is completely filled, the job re-starts the fill from the beginning again. When this happens, since the fill area will now be completely filled with INK colour, QFILL2 will do the fill twice, the first time in a neutral colour, as described in the Section about the INK parameter above. The repeating fill, once started, will carry on, repeatedly re-filling the area in this way until it is stopped by the special call to QFILL2(-1), as described in the Section on X and Y above. Up to 10 repeating fills may be active at one time and as many 'once only' (JOB>0) fills as the QL will allow.

9.3 Additional Information about QFILL1 and QFILL2

Part of the protocol that allows either QFILL to multitask requires that no two tasks may actually try to initiate a job, i.e. be actually in the process of the QFILL call, at the same time. Once a job has been set up, another task may issue a new call (or another call from the same task, of course).

This restriction is automatic and is achieved by a flag within each QFILL routine that is set whenever it is active. If another call is made while this flag is set, the call is left waiting in a loop until the flag is cleared. The flag is ONLY cleared by the QFILL routine itself when the existing call returns.

There are several implications arising from this.

Firstly, if a task (a compiled SuperBASIC program for example) has issued a call to either QFILL without using the JOB feature, any other task that calls that QFILL will be blocked (not suspended, note; the code will be periodically looking to see if the existing QFILL job has finished). This will apply equally to a call from SuperBASIC, which will appear to 'hang up' until the existing fill finishes.

This is where a problem could arise. Say you set up a non-JOB fill from a task but then use a software toolkit to stop the task BEFORE the fill finishes. This will not directly cause any problems except that the QFILL routine will not have cleared its access flag. The result of this is that

ALL subsequent calls to the function (of any kind) will 'hang'. This is a very good way of locking up the entire machine!

Note that no errors or traps or any other nasties will occur; the computer will simply sit there waiting for some kind person to press the reset button, which is the only way to recover.

The solution to this potential problem is simple. DON'T use a toolkit to stop a fill that is only partly completed. Note that the independent JOB fills don't suffer from this problem. You are free to stop a partial fill of this kind whenever you want. The only time the restriction applies is when the QFILL base routine is active, that is when a non-JOB fill is in progress. Normally, it is simpler to use the appropriate call to either QFILL to stop a current fill, even if you do have a toolkit.

There is a simple way to find out if a current fill may be safely aborted by a toolkit. This is to issue the command PRINT QFILL1(-1) or PRINT QFILL2(0,-1) as appropriate. In fact this is an ideal check to make, since the function will only return a value when it IS safe to abort any active fills via the dreaded toolkit. If any of the current fills are not independent jobs, the version request will wait for the offending fill to finish before returning.

Some thought should be given to how you expect QFILL2 to work. It is possible for unexpected things to happen in some situations, unless you are aware of some of the peculiarities of the routine.

The first situation that might cause problems is when an area coloured with a stipple pattern is re-coloured. If the surrounding area is either a solid colour or another stipple that has a colour in common with the area to be filled, then, depending on the actual stipple patterns, you may get surprising results.

What can happen is that QFILL2 may assume that the area being filled has numerous single-pixel-width extensions into the surrounding area and re-colour these accordingly.

Another potential problem can arise if QFILL2 is forced to pre-fill the area in a neutral colour. If this happens, as it will when the area contains a colour in common with the specified INK, then the neutral colour used will be the highest numbered non-common colour. If you are unlucky, the area surrounding the chosen fill area may be the same as the neutral colour chosen by QFILL2. In this case, the fill will spill out of the chosen area.

TABLE 1 - THE DEVELOPMENT TREE OF AN ACT ADVENTURE GAME

```
                              The COMPLETED Text Adventure
                              ----------------------------
                                   (Mini_Adventure)
                                         |
                                  ****************
                                  * LINKER_task *
                                  ****************
                                         |
        +-----------------+--------------+-----------------+-----------+---------------+--
        |                 |              |                 |           |               |
        1                 2              3                 |           |               |
 ACT base module    Compressed text  The program  ****************************** **********
(ACT or ACT_SHORT)    (TEXT_com)     (PROG_dta)   *        4           5       * *     6
                          |              |        * Location data  Object data * *  Word da
                          |              |        *  (LOCN_dta)    (OBJT_dta)  * * (WORD_dt
                   **************        |        *                            * *
                   * TXTcom_task *       |        * Both edited by LOCedt_task * * VOCedt_t
                   ***************       |        ****************************** **********
                          |              |
      +---------------+---+-----------+  +---------------------------------+
      |               |               |                                    |
  ***********************************************                   ***************
  * Location text  Object text  General text   *                    * BASasm_task *
  *  (LOCN_msg)   (OBJT_msg)   (GEN_msg)       *                    ***************
  *                                            *                           |
  *  These files are all edited by MSGedt_task *         +-----------------+----------------
  ***********************************************        |
                                                ***************************************************
                                                * The player program                    The
                                                *  (PLAYER_prog)
                                                *
                                                *      These files can both be edited by Su
                                                ***************************************************
```

File names are included in upper-case characters, and the appropriate names used in the Mini_Adventure are in brackets.

Each box shows a particular ACT utility program and the data files it edits. Where the input files are from another program, the utility name only is included in the box and the input files are shown by the previous process or box below. All the utility programs have the _task extension.

TABLE 2 - THE OBJECT PARAMETERS DESCRIBED

Parameter	Meaning
0,1	Describing noun or adjective word. The word number is
2,3	divided between the first and second parameter in each
4,5	pair with the first containing (#word DIV 256) while
6,7	the second contains (#word MOD 256). For example,
8,9	values of 2 and 123 for parameters 0 and 1 represent word number 635 (2*256+123).
10	Object location. 0 indicates that it is held.
11	Containing or supporting object. 0 if not contained.
12	The weight of the object.
13	The size of the object.

14	The volume of the object (or surface area, see P15 B4)
15	Bit flags: 0 Lid: 0 if no lid, 1 if it has a lid. 1 Lid state: 0 if open, 1 if closed. 2 Burning: 0 if not, 1 if is. Note: The lamp (object 0) is special; this bit indicates if it is switched off (0) or on (1). 3 Set after an object has been described once. 4 For objects with volume or surface area. 0 for volume; 1 for surface area. 5 Used by find_objects routine. 6 Set if the object can support or contain other objects. 7 Set if the object will burn.
16	Restriction flags: 0 Open (Note: The restriction 1 Close applies if the bit is set) 2-7 Unused (yet)
17	A counter used for burning objects. It should be set to zero unless the object is to start out on fire. In this case (if parameter 15 bit 2 is set) this value will determine how long the object will burn. A value of 100 gives about 1 minute.
18	Bit flags 0 Solid (0) or liquid (1). 1 For containers this bit is set if the object can contain a liquid. 2 Object inedible (0) or edible (1). Note: All liquids are edible. 3 Set if an edible object is poisonous. 4 Set if the object can be used to extinguish a fire. 5-7 Unused.

TABLE 3 - LOCATION 0 PARAMETERS DESCRIBED

Parameter	Meaning
0	Current player location.
1	Previous player location.
2	Bit flags 0 Set after the initial startup message (general message 7) has been output. 1 Used by check_if_dead. Should be set to 0. 2-7 Reserved for future system expansion.
3	Set after a special reply is to be output by a Yes or No response. The value (1 to 255) indicates which special response is to be made. It is cleared after the next player command regardless of whether a Yes or No response is actually made.
4	The player's current score.
5	The player's total burden (weight of all objects currently carried).
6	Player's health rating. 0 is dead. To kill the player, update this value to zero. The normal maximum value is 20.
7	Player's maximum carrying capacity. The actual capacity is this value plus the health rating in parameter 6. The total burden (parameter 5) will not be allowed to exceed this via a command to pick up an object.
8	Number of remaining reincarnations allowed.
9-10	Reserved for future system expansion.

NOTE: Various special responses can be set up by simply setting the value of parameter 3. The responses are handled by the routine yes_or_no_response and apply when the player responds with Yes or No to particular questions.

The Mini_Adventure has three of these built in (corresponding to values of 1 to 3 of parameter 3).

These are:

Safety net on quitting game.

Response after a certain message about picking up liquids.
Response to certain SQUEEZE commands.

TABLE 4 - THE OBJECTS IN THE MINI_ADVENTURE

Number	Description	Location	Contained in
0	Electric torch	1	2
1	Tool box	1	-
2	Broken torch	1	-
3	Gas lighter	1	-
4	Oil	3	-
5	Sponge	2	-
6	Waste bin	1	-
7	Note	1	6
8	Note	7	-

TABLE 5 - WORD TERMINATING NUMBERS

Terminating number	Meaning
0	Unused in the Mini_Adventure
1	Direction- or movement-related
2	'Throw-away' words ("on", "in", "the", etc.)
3	'Command' words (usually verbs)
4	Adjectives
5	Nouns
6	Special word types used in "find_objects"
7	Swear words
8	Unused
9	Unused

TABLE 6 - THE MAIN TABLE AREA IN THE ACT BASE MODULE

Entry	Offset	Length	Description
VARLOC	$2E	4	Pointer to the variables - here
SRNLOC	$38	4	Pointer to the SAVE/RESTORE filename - here

CONID	$48	4	The text screen channel ID value
NEXT	$4C	1	Number of external routines
NVAR	$4D	1	" " variables
NLOC	$4E	1	" " locations
NOBJ	$4F	1	" " objects
NLOC_P	$50	1	" " location parameters
NOBJ_P	$51	1	" " object parameters
ENERMG	$55	1	Enable error messages flag
LOCLOC	$66	4	Pointer to location data - here
OBJLOC	$6A	4	" " object data - here

This table is pointed to by A6. For example: to read the value of the text screen channel ID, the assembler instruction MOVE.L $48(A6),A0 could be used. The entries where the description finishes with '- here' are relative pointers.

For example: to point A0 to the start of the object data, use the instruction LEA $6A(A6),A0 followed by the instruction ADDA.L (A0),A0.