# SAM ADVENTURE SYSTEM

An adventure creating utility for the SAM Coupe

(C) - 1992 AXXENT SOFTWARE

# CONTENTS

# INTRODUCTION

Thank you for purchasing **SAS**. This program enables you to design and create professional adventure games on your SAM Coupe, enabling you to use SAM's impressive capabilities to the full. **SAS** is designed to give the user a sense of freedom from many of the usual coding chores associated with writing a game from scratch. It allows you to instead, concentrate on the puzzles and overall design of your adventure, whilst still allowing you full flexibility over the final "feel" and ambience that you wish to create.

I have tried my best to make **SAS** as flexible as possible. The user is able to create his own BASIC or Machine code routines and combine them with the adventure source, giving the final game a greater degree of individuality than was possible with other adventure creators. I have devoted one section of this manual to "customising" **SAS** to suit the user's own personal preferences.

The system consists of three main parts, namely :

| | |
|---|---|
| The Editor | - To create your adventure "source" (the component parts that make up the adventure such as locations, vocabulary etc). |
| The Compiler | - To convert your adventure "source" into executable code. |
| The Interpreter | - This is a set of code routines which combine with your compiled adventure to form a complete stand alone game which you can play. |

**SAS** requires 512k of internal memory, at least one disk drive fitted and a ROM 2.1 or later present. If you're unsure of which ROM version you have, type PRINT PEEK 15 from BASIC. If the number printed on the screen is less than 21, then you will need a replacement ROM chip for your SAM. Contact SAMCo for details on how to obtain this. **SAS** will also recognise and use the 1Mb memory interface (allowing even larger adventures to be created), an extra disk drive and the SAM mouse if they are present.

If you bought your copy of **SAS** directly from AXXENT Software, then you will already have been registered as a user, and entitled to free upgrades to any future versions of SAS that may be produced, along with discounts on any future AXXENT software products. If you bought your copy of **SAS** elsewhere, then it would probably be a good idea to fill in and send me the registration form printed at the end of this manual.

Many thanks to the various people who cajoled me into writing **SAS** in the first place, particularly, Alan Miles (who will probably want me to re-write 99% of this manual!!), Phil Glover, Dave Whitmore, and Dave Ledbury.

If you have any suggestions or problems with **SAS** or its manual, then please let me know, as I am always interested in hearing from budding adventure writers. Who knows, I may even be interested in publishing your finished adventures!

I look foward with enormous interest to seeing **SAS**'s great potential realised, and more adventures appearing for the SAM Coupe, which with its large memory and fast processor, I have always felt was ideally suited to adventure playing.

Colin Jordan. 8th May 1992.

(C) 1992 Copyright Colin Jordan/AXXENT Software.

First edition May 1992. All rights reserved.

# SETTING UP SAS

**SAS** is supplied as a master disk without a disk operating system. To make a working copy of **SAS**, you will need two FORMATted disks with either SAMDOS 2.0 or MASTERDOS (you will need MASTERDOS if you wish to use an external 1Mb memory interface for increased adventure source space) saved as the first file on the disk.

Next, copy all the files from your master disk onto one of the disks

If you have two disk drives fitted, type `COPY "d1:*" TO "d2:*"`

If you only have one disk drive, type `COPY "*" TO "*"` and swop disks as prompted.

You have now created a "Utilities" disk. It might be a good idea at this stage to label the disk as such.

Now you will need to save some files onto the second FORMATted disk. Insert the master disk back into drive 1.

If you have two drives fitted, type `COPY "d1:????EDITOR" TO "d2:*"`

If you only have one disk drive, type `COPY "????EDITOR" TO "*"` and swop disks as prompted.

DÁVAT POZOR NÁSLEDUJÍCÍ

Take care to ensure that you type exactly four question marks in the filename.

The second disk which you have just written to is your "Editor" disk. Make sure that this disk is NOT left write protected (you should not be able to see through the small square hole in the corner of the disk), as the editor will need to use "spare" sectors of the disk as a temporary storage space. Again, it might be a good idea to label this disk as the "Editor" disk at this stage.

You should now remove your master disk and keep it safe. Please do not use your master disk as a working disk or for data storage. ULOŽIT

You will now have two **SAS** working disks. The "Editor" disk contains the adventure "source" editor and should have around 714k of unused space left on the disk. Don't be tempted to use this space for extra files or data, as the editor requires this space as a temporary storage area.

The second "Utilities" disk basically contains "everything else". It contains files such as the compiler and graphic extension programs and the "START" and "SPAMCO" source files (more about these later).

---

This program took many months of hard work to write (I hope it shows!), and the price is very reasonable. Please do not infringe my copyright by making illegal copies of this system for your friends. Instead, ask them to buy their own copy. Software piracy is ILLEGAL and only results in fewer software companies and individuals being willing to develop new software products for the SAM.

Of course, you are perfectly free to market and distribute your completed adventures as you wish, **PROVIDED** your software clearly states that it was written using **SAS AND** no part of the editor or compiler is included with your software.

# PLAYING THE DEMO ADVENTURE

By now, you're probably wanting to see some of the impressive results that can be achieved by using **SAS**, so insert your "Utilities" disk into drive 1 and type BOOT 1 to install the DOS, and LOAD "DEMO" to load the demo adventure.

When playing, you'll notice that **SAS** will allow you to enter multiple commands. From the initial location (outside the SPAMCO building), try entering the following command line.

```
GO SOUTH. EXAMINE CAR. GO NORTH. NW. EXAMINE BINS.
```

**SAS** is now able to deal with each separate part of your input in turn.

You may enter multiple commands by using either commas or full stops, but remember to include a space before the following word.

The words "**AND**", "**THEN**" and the ampersand character ("&") can be used in a similar way to normal everyday English.

If you type the input

```
EXAMINE THE CHAIR AND THE BINS
```

or

```
TAKE THE COFFEE THEN DRINK IT
```

**SAS** will know exactly what you mean and act accordingly.

While typing your commands, **DELETE** can be used in the normal way to delete the character to the left of the cursor. Pressing the **EDIT** key will bring back the last command(s) you typed in. This can be useful if you made a minor mistake last time that you wish to correct, or if you wish to repeat your last input by pressing **EDIT** then **RETURN**.

The graphics in the adventure are actually captured screens from SAMCo's Video Digitiser interface. **SAS** will allow you to include MODE 3 or MODE 4 SCREEN$ files or FLASH screens for use as location graphics in your own adventure.

---

The adventure source file for this demo adventure is also included on the "Utilities" disk under the filename of "SPAMCO" and is fully commented. Later on, you may find it interesting to load this source into the editor (once you are familiar with how the editor works) and see how the adventure was constructed.

# PLANNING YOUR ADVENTURE

Before we look at how adventures are put together using **SAS**, it's worth mentioning how to plan your adventure.

If you're new to adventure playing, you should know that adventures are in essence "interactive novels", allowing you as the "player" to take part in the plot through your typed instructions on the keyboard. It's very much like reading a book in which you are actually playing the part of one of the characters and are able to shape events in the adventure world as the story unfolds.

The computer prints on the screen details of your current location, what objects (if any) that you're carrying and responses to any commands that you've typed in.

In the adventure OKOLÍ, PROSTŘEDÍ environment, the player will often come across objects which might come in handy to solve puzzles which the player will have to overcome later on in the game. An example would be a key to open a locked door.

The BEZPROSTŘEDNÍ immediate and most important task facing an adventure writer is ROZHODNUTÍ deciding upon the plot or storyline behind the game. It is **ESSENTIAL** that you do this before you start writing the game, otherwise you'll probably just end up with an incoherrent mess! (which is why I've placed this chapter here at the beginning of the manual). Where is the adventure to take place ? Perhaps in the days of King Arthur, or on an alien planet or maybe even in your own city or the Wild West. These are just a few ideas, try to think of an original one for yourself. (If I had a pound for every adventure based on a dungeons & dragons theme.....)

Once you've decided where to set the scene, you must decide what the object of the game is (otherwise the player will just end up wandering around not knowing what he's supposed to do!). It could be to find some hidden treasure, solve a murder or even save the whole of mankind! Try to keep the aim of the game consistent with the setting you've decided. (You wouldn't expect to find Long John Silver looking for treasure on an alien planet for example!)

You can't think of a good story ? Then read. There are THOUSANDS of good ideas just perfect for adventures in books. And if the story is out of copyright (if the book was first published over 75 years ago you'll be safe), then you'll be free to use it or adapt the story for your own adventure. If the book is not quite that old, then you'll have to HLEDAT seek permission from the relevant publisher - be warned, this can be very expensive !!!!

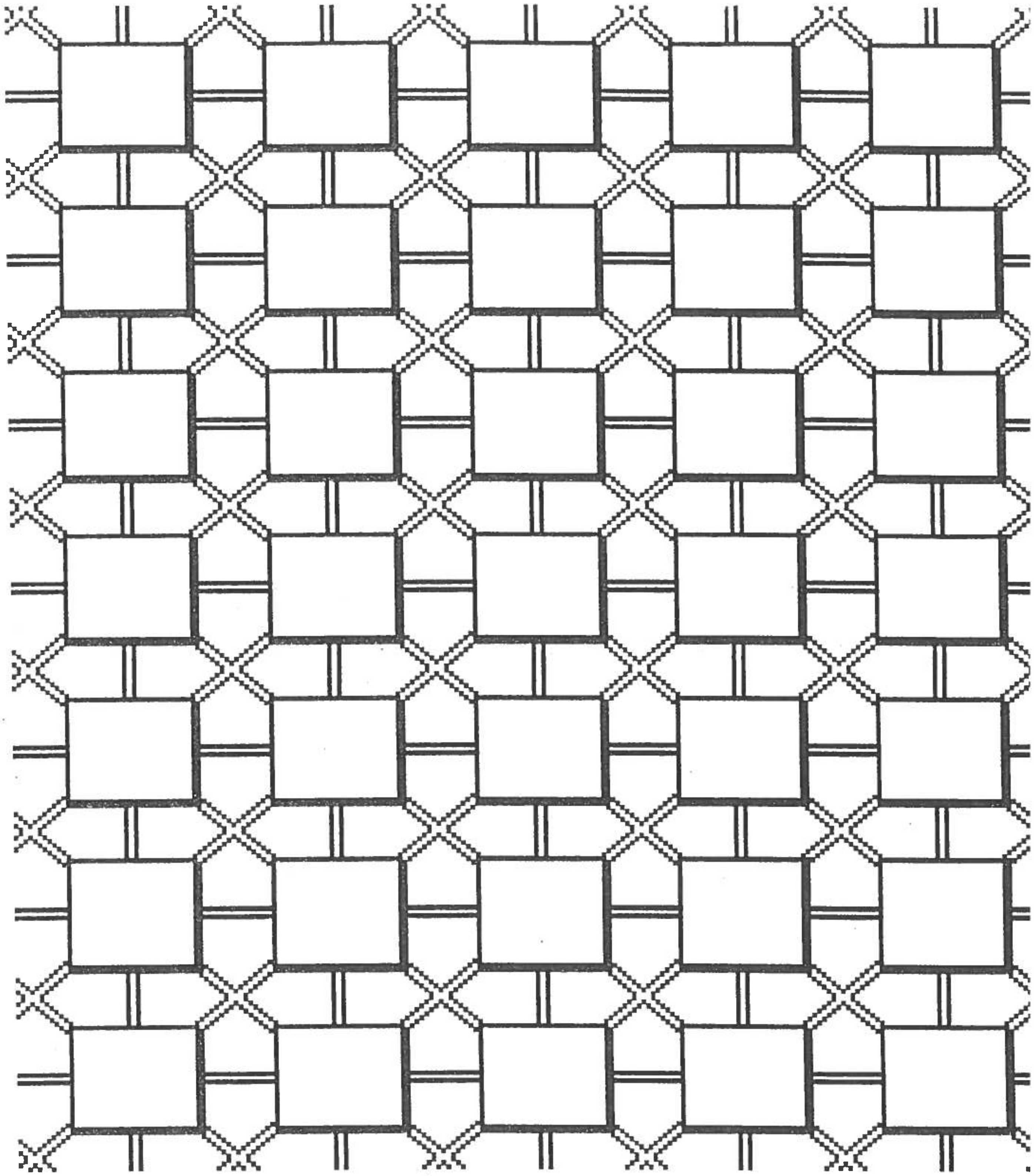
The next step to take is to actually map out the locations where you want your action to take place. Locations could be rooms in a building, caves or even a space outdoors. It's entirely up to you. **SAS** allows you to define up to 255 different locations (which would make a very big adventure indeed!), but if you're just starting out, it's probably best to begin with just a TUCET dozen or so.

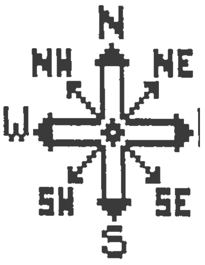
On the next page is chart which you can use to help map out and connect your locations. I suggest you remove and photo-copy this page. If your adventure has a large number of locations, then you can tape several photo-copies of the page together to make a large map. Each location on the chart has possible connections for all eight major compass points. If you want to invent your own directions or use UP or DOWN, you'll have to draw these in yourself.

Often when deciding upon the individual locations, ideas for puzzles and starting locations for useful objects will come to mind. Make a note of these and jot them down on the map as well so that you can incorporate these into your game later on.

When designing puzzles, think of how the player will solve them. Will he need to use any objects ? If so, will he be able to pick up the objects and carry them around, or will they be set in one place ? (This distinction is important, as you'll see later when using the editor). Where will the objects be found ? Are you going to give the player any clues on how to solve the puzzles ? If so how ?, perhaps by reading a book found in a location for example.

Above all, what VERBS will the player need to type in, in order to solve the puzzles ? Make a list of any VHODNÝ suitable verbs (along with all the alternatives that you can think of) that you think the player should use. Also make a list of all of the objects that you wish to include in your game.





ADVENTURE_________________________ Map No. ____

# THE SOURCE EDITOR

vynaložit

The source editor program is where you'll spend most of the time developing your adventure. If you have a SAM Mouse or a 1Mb memory interface, then connect these to your SAM. Make sure you have a blank formatted disk handy (with no DOS), and load up the "Editor" disk which you prepared earlier by pressing the F9 function key as normal.

After a short while, you will see the main menu as shown in Figure 1.

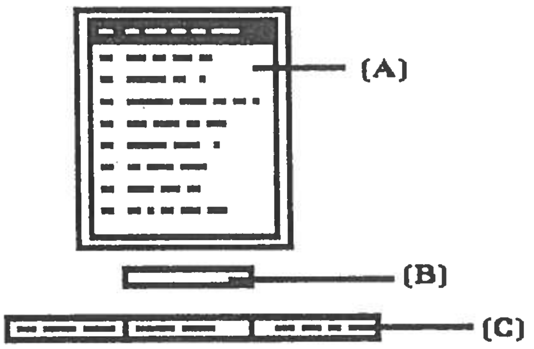


Fig 1.- The Main Menu screen

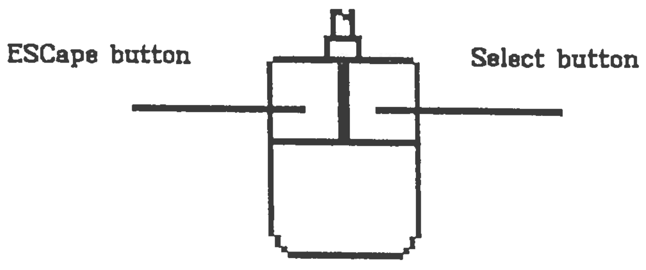


Fig 2. - The Mouse controls

The section of the screen marked (A) in the figure is a bar menu, from where you may select which component of the adventure you wish to edit. Try highlighting the various menu options by moving the dark choice bar up and down using the **CURSOR UP & DOWN** keys. If you have a SAM Mouse connected, you may also highlight the various options by moving the mouse up and down across a flat surface.

Bar menu choices are selected (don't select one yet!) by pressing the **RETURN** key, or pressing the mouse "select" button as shown in Figure 2. To the far left of each bar menu option is a single letter. You may also directly select the option you require by pressing this key on the keyboard. Later on, you will encounter other bar menus in various other sections of the editor. All of them work on this same principle.

The section of the screen marked (B) in Figure 1. gives an indication of the amount of memory available for your adventure source. The number displayed is actually the number of bytes free. As you define the various components of your adventure, this figure will steadily decrease.

Section (C) is used to show which version of **SAS** you are using, and which section of the editor you are currently using.

Because you've just loaded the editor, most of the bar menu options will not work if you try to select them. This is because at this stage, there is no adventure source defined in memory. The editor needs to know whether you wish to create a completely new adventure, or load one from disk which you are already working on.

Select the bar menu option marked A Load Adventure Source. The screen will clear, and you will be prompted to insert your source disk into the drive. Insert your "Utilities" disk. Note that if you have two disk drives fitted, the Source loading and saving options **ALWAYS** requires you to place the source disk into drive 2.

Press a key. You will now see a directory on-screen with the filenames "SPAMCO.HDD" and "START .HDD" displayed. These are adventure source files which you can load into the editor and manipulate if you wish. "SPAMCO" is the source for the demo adventure on the "Utilities" disk, and the "START" file is a starter file containing a basic adventure framework which you can develop your own adventure from.

Type the name START in the entry box at the bottom of the screen (no need to add the ".HDD" file extension) and press **RETURN**. Once the file has loaded, you will be prompted to insert your "Editor" disk again (unless you have 1Mb interface connected or two disk drives) and you will return again to the main menu screen.

You will notice that the first four options on the bar menu are "banks" numbered from 1 to 4. It is in these banks that you will define how the adventure actually behaves as it is being played. In these four source banks you will be using a simple programming language specifically designed to make adventure writing as easy as possible. Don't worry too much about these first four options for now, as you will learn how to use this programming language later on.

The options available from the Main Menu are as follows :

| | |
|---|---|
| **1 Source Bank (1)** STARTOVNÍ POZICE | This section is used to define the STAV state of the adventure at the start of the game. It is here that you would set up which location you start out in, where any objects are located etc. |
| **2 Source Bank (2)** REAKCE NA POKYNY | It is in this section that you will specify how the adventure reacts to the commands entered by the player. It is in this section, that you will be defining most of the puzzles in the game. In effect, it is in this section that "high priority" conditions (conditions that will be acted upon as soon as the player has typed in a command) will be performed. |
| **3 Source Bank (3)** | This section is used for "low priority" conditions. (Low priority conditions are conditions that will be acted upon, once the "high prioity" have been dealt with). |
| **4 Source Bank (4)** MÍSTNÍ PODMÍNKY | This is used for "local conditions". Local conditions will be acted upon when the player moves to a new location. |
| **V Vocabulary** | In this section, you will define a vocabulary of words which the computer will recognise when the player types in his commands. |
| **M Messages** HLÁŠENÍ | This section is used to define all messages that will be used BĚHEM during the game. Messages can be thought of as any text that will be printed on the screen. |
| **L Locations** POPIS MÍSTNOSTÍ | It is here that you will define the descriptions of the various locations in your adventure, along with the various exits that connect the locations together. |
| **B Verb Definitions** SLOVESA | This section is used to define the names of the verbs you wish to use in your adventure, in case you wish to include the name of a specific verb in a message. |
| **O Movable Object Definitions** | This section is used to define the names of all POHYBLIVÝ movable objects that you wish to include in your adventure. (Movable objects are objects that you can actually pick up and carry around with you). |
| **U Unmovable Object Definitions** | It is here that you will define the names of all unmovable objects that you wish to include in your adventure. Unmovable objects are objects such as trees and doors, which obviously cannot be picked up and carried around, but which the player might wish to interact with in some way, for example climbing a tree or opening a door. |
| **D Direction Definitions** | This section is used to define the actual names of the directions which you wish to use to connect your locations together. |

| | |
|---|---|
| C Create New Adventure Source | This option allows you to create a new completely new adventure from scratch. If you're new to SAS, you'll probably want to adapt your adventure source from the "START" starter file instead. |
| A Load Adventure Source | This option allows you to load an adventure source into the editor. We used this option a short while ago to load in the "START" starter file. |
| S Save Adventure Source | This option is used to save to disk the adventure source you're currently working on. You can use this option to save a half-finished adventure that you wish to complete later, or to save your source for the compiler to convert into a runnable game that you wish to play or test. |

## Creating An Adventure

To help guide you through using the editor step-by-step, we'll create a very simple adventure whicn is mapped out in Figure 3.

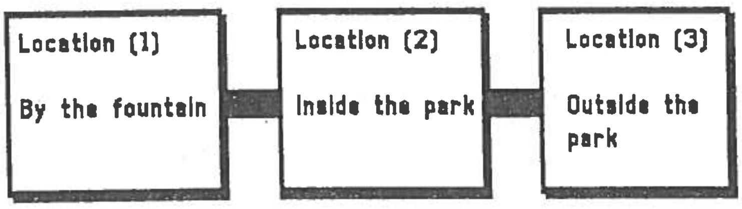


Fig 3. - "Park" Adventure map.

The aim of the game will be to escape from the park to location number (3). Initially the player will not be able to go East from location (2) because the way will be barred by the park gates which are closed and locked. To get past the gates, the player will have to examine a fountain in location (1), find a key hidden there and unlock the gates with it.

The "START" starter file which we have loaded, already contains the following verbs:

| | | |
|---|---|---|
| (1) LOAD | (5) QUIT | (9) TAKE |
| (2) SAVE | (6) SCORE | (10) DROP |
| (3) RAMSAVE | (7) INVENTORY | (11) EXAMINE |
| (4) RAMLOAD | (8) LOOK | |

In addition, we will need to define the following new verbs :

| | |
|---|---|
| (12) UNLOCK | (to unlock the gates with the key). |
| (13) OPEN | (to open the gates once they are unlocked). |

Also included in the "START" file are the following directions :

|  |  |  |
|---|---|---|
| (1) NORTH | (5) NORTHEAST | (9) UP |
| (2) SOUTH | (6) NORTHWEST | (10) DOWN |
| (3) EAST | (7) SOUTHEAST |  |
| (4) WEST | (8) SOUTHWEST |  |

As we'll only be actually using EAST and WEST in our short adventure, there's obviously no need to add any extra direction names.

The "START" file contains no definitions for any movable or unmovable objects.

We'll need to define the single movable object :

|  |  |
|---|---|
| (1) GOLDEN KEY | (to unlock the gate with) |

And the following unmovable objects :

|  |  |
|---|---|
| (1) FOUNTAIN | (where the key is to be hidden) |
| (2) GATES | (Which will have to be opened and unlocked for the player to escape) |

## Entering Locations

To begin with, we'll define the three locations that will be used in our short "Park" adventure.

Select the option L Locations from the main menu.

A smaller bar menu will be displayed with the following options :

|  |  |
|---|---|
| B Browse Locations | This option is used to look through the locations that have already been defined. |
| E Edit Locations | This option is used to alter locations that have aready been created, or to create new locations. |
| P Print Locations | This option can be used to send to a printer (if connected), the decriptions and exit details of any locations you have defined. |
| M Return To Main Menu | As you've probably already guessed, this option will leave the locations section and return you to the editor main menu. |

Select option E Edit Locations from the menu. You will now be prompted to enter the number of the location you wish to alter. You can enter any number between 1 and 255 (the maximum number of locations allowed). Type 1 to edit location number 1 and **RETURN**. The screen will clear, and after a short moment, you will see the location editing screen as shown in Figure 4.

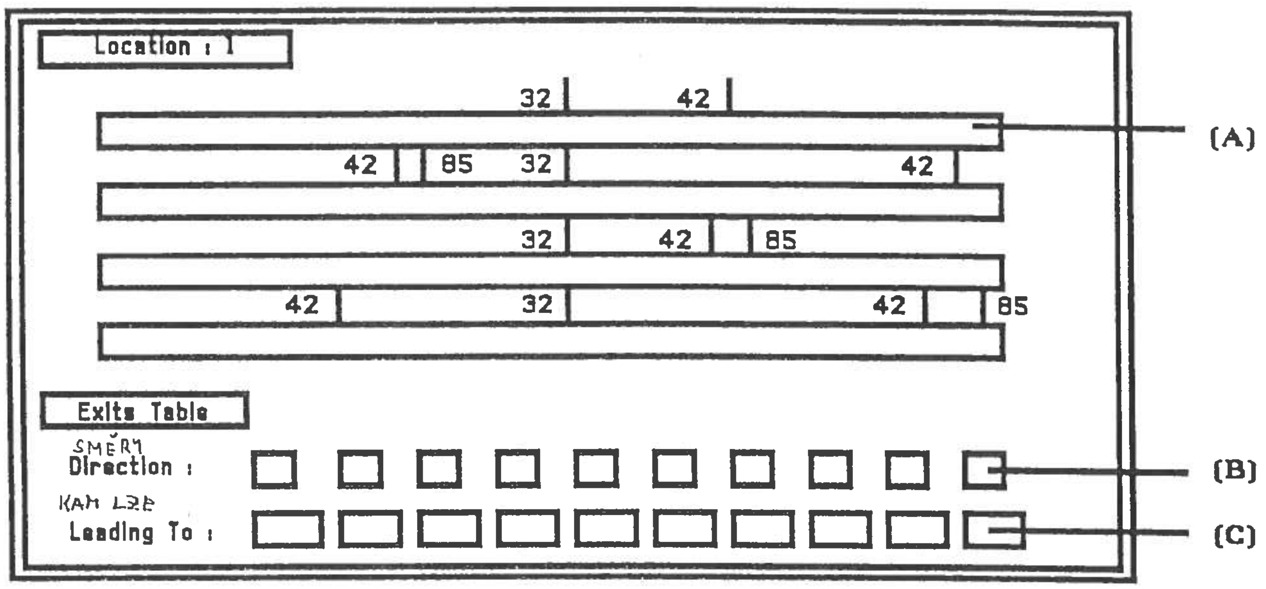


Fig 4. - The location editing screen.

The section of the screen labelled **(A)** in Figure 4 is used to hold the actual description of the location. There are four entry fields used to hold the description text, each 64 characters long. In the first field, you will notice the text "This is the initial location.". This is a default description of location 1 which is supplied in the START starter file.

Directly above each description field, you will notice numbers marked off at various points along the field's length. These are used to indicate the end of the text row when the description is printed on the screen, depending upon the column mode the adventure is using. **SAS** supports four different column settings : 32, 42, 64 and 85 columns. For our short "Park" adventure, we will be using 64 column mode, so each of the four description fields will represent exactly one row of text printed on the screen.

A cursor will be positioned at the first character of the top description field. You may move the cursor around the text using the **CURSOR LEFT** and **CURSOR RIGHT** keys. Now move the cursor to the middle of the text and press **F2**. This key can be used to insert a space in the field. Move the cursor to just past the end of the text and press **DELETE** until all the text has been deleted and the cursor is back at the first character position of the first field.

We are now ready to enter the description of location number 1. Type in the following text :

You are in a quiet corner of the park. Nearby is a fountain.

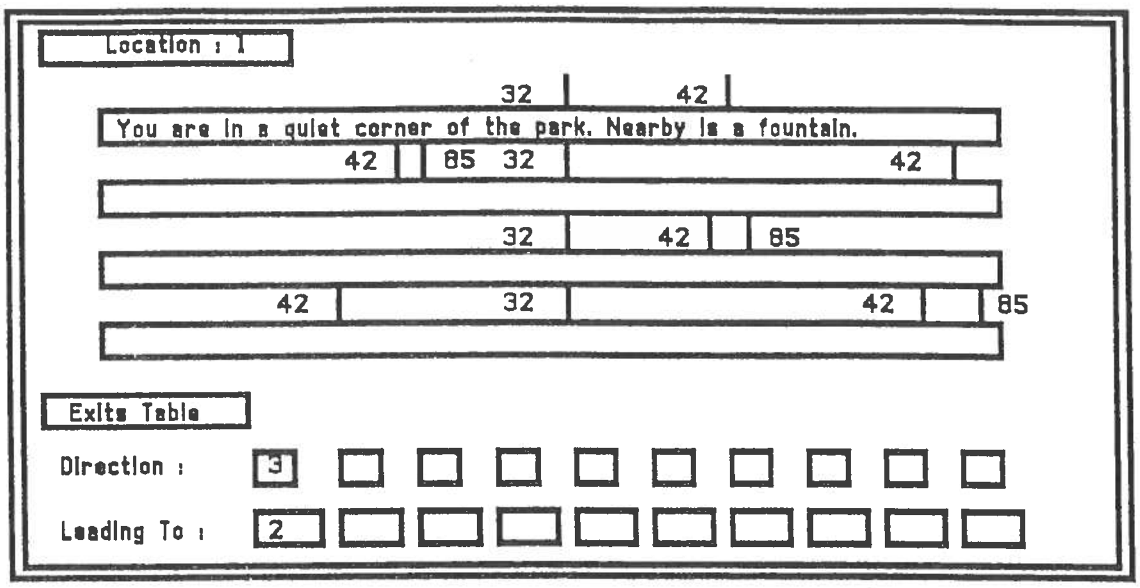
Press **RETURN** four times to skip past the remaining description fields. You should now be in the section of the screen marked **(B)** in Figure 4.

The fields located in sections **(B)** and **(C)** of the screen are used to define the exits (if any) available from the location currently being edited.

The fields in section **(B)** are used to define the direction numbers of any directions leading away from the location. There is only one exit from location 1, so type **3** (for direction number 3 - East) and press **RETURN**. Your cursor should now be in a field directly below, in section **(C)** of the screen.

The fields in section **(C)** are used to specify which locations the exits actually lead to. Location number 2 is East from location 1, so type **2** and press **RETURN**.

The screen should now look like the one shown in Figure 5.



You are in a quiet corner of the park. Nearby is a fountain.

3

2

Fig 5. - Location data for location 1.

Keep the **RETURN** key pressed down to skip past the remaining exit table fields. You will now be asked

Is this correct ? ( Y / N )

If all's well, Type **Y**, otherwise type **N** then **ESC** to return to the locations menu and try entering the location data again.

Now press **CURSOR RIGHT** to step to location number 2. Enter the following location description :

You are inside the city park. It is very quiet here.

and the following exits :

3 leading to 3
4 leading to 1

Once you have confirmed all is correct, step to location 3 (using **CURSOR RIGHT** again) and enter the following location description :

You are outside the park gates.

and the exit :

4 leading to 2

Once you have confirmed all is correct, press **ESC** (or press the **ESC** mouse button) to return to the locations menu.

We can now inspect all three locations we have just entered by selecting the option B Browse Locations from the locations menu. Use **CURSOR LEFT** and **CURSOR RIGHT** to step backwards and forwards through the location definitions (although locations numbered 4 upwards should be blank). When you've seen enough, press **ESC** to return to the locations menu, and select the option M Return To Main Menu to return once again to the editor Main Menu.

Next, we will define the various messages which contain the text that will be printed on the screen at various points during the "PARK" adventure. The actual messages can be whatever you like - a response to something that the player has just done, a welcome message at the start of the game, or even a detailed description of one of the objects in the adventure.

Select the option M **Messages** from the editor main menu.
A smaller bar menu will be displayed with very similar options to the menu we saw when defining the locations ie., Browse, Edit, Print and Return To Main Menu.

The "START" starter file already contains 16 messages which are used for various purposes common to most adventures. To look through these, select the option B **Browse Messages** from the messeges bar menu. You will now be prompted to enter the number of the message which you wish to start looking from. Type 1 and press **RETURN**.

you will now see the defined text of message number 1 on screen. You will notice that the message is displayed in four entry fields, each 64 characters long, in exactly the same way as the location descriptions when we were defining locations earlier. Of course, there is no exit table information needed for messages!

we can now inspect all of the other defined messages by using the **CURSOR LEFT** and **CURSOR RIGHT** keys to step backwards and forwards through the other messages (although messages numbered 17 upwards will be blank). When you've seen enough, press the **ESC** key (or the mouse **ESC** button) to return once more to the messages menu.

we will now add the extra messages that will be required by the "PARK" adventure. Select the option E Edit **Messages** from the bar menu. You will now be prompted to enter the number of the message that you wish to alter. You can enter any number between 1 and 1024 (the maximum number of messages allowed). Since we'll be adding our new messages to the end of the ones supplied in the START starter file, type **17** (to start editing from message number 17) and press **RETURN**.

As when editing location descriptions, a cursor (displayed as an underline character) will be positioned at the first character position of the first of the four entry fields. Now type in the following text for message number 17 :

```
Hidden in the fountain, you discover a golden key !
```

Again, as when typing in the location description text, the **CURSOR LEFT** and **CURSOR RIGHT** keys can be used for moving the cursor around the field, the **DELETE** key can be used to delete the character to the left of the cursor, and the **F2** key will insert a space at the current cursor position.

As you will have probably guessed, this is the message that will be printed once the player has EXAMINEd the fountain and found the golden key hidden there.

Now press **RETURN** (four times) until the cursor has gone past the last text field. You will now be asked

```
Is this correct ? ( Y / N )
```

Provided you have typed everything in correctly, press the **Y** key (no need to press **RETURN**), otherwise if you see you have made a mistake somewhere, type **N** then press the **ESC** key to return to the messages bar menu, and try entering the message data again.

Now press the **CURSOR RIGHT** key to step to the next message - message number 18. For this message, type in the following text :

```
The gates are currently closed and locked.
```

This message will be printed if the player EXAMINEs the park gates before they have yet been unlocked and opened.

As before, once you have typed in the text, press the **RETURN** key until the confirmation prompt appears asking if all is correct. Again, press Y followed by the **CURSOR RIGHT** key to move on to the next message (message number 19). For this message type in the following text :

The gates are closed.

This message will be printed if the gates are examined after they are unlocked, but before they are open.

Again, confirm all is correct and move on to message number 20 :

The gates are now open.

As you've almost certainly guessed, this message will be printed if the park gates are examined after they have both been unlocked and opened.

Now define message number 21 as :

You have no key !

This is an "error" message which will be printed if the player tries to unlock the park gates without actually carrying the golden key with him.

Message number 22 is defined as :

The gates are already unlocked !

This is another "error" message that will be printed if the player tries to unlock the gates after he has already unlocked them!

Message number 23 is defined as :

You unlock the park gates with the golden key.

This is a message that will be printed when the player has successfully unlocked the gates.

Message number 24 is defined as :

In vain you try to open the gates, but they are securely locked.

This message is printed if the player attempts to open the park gates without unlocking them first.

Message number 25 is defined as :

The gates are already open !

this is another "error" message which will be printed if the player attempts to open the park gates after he has already successfully opened them!

Message number 26 is defined as :

With a loud creak, the gates slowly swing open.

This is a message which is printed when the player has been successful in opening the park gates after they have been previously unlocked.

Message number 27 is defined as :

```
CONGRATULATIONS you have escaped from the park -
Press any key for a new game.
```

This message will be printed at the end of the game when the player has escaped from the park.

This message is different from the others we have defined so far - it will be printed on two consecutive text lines. To define this, type the text

```
CONGRATULATIONS you have now escaped from the park -
```

and press **RETURN** once. The cursor will now be positioned at the start of the second entry field (remember each entry field is 64 characters long, and we have already decided that the PARK adventure will use the 64 column text mode - if we were using 42 column text mode instead, we would have moved the cursor to the next "42" column marker). Now type in the second text line of the message :

```
Press any key for a new game.
```

Now press **RETURN** until you see the normal confirmation prompt, and move on to the next message in the normal way.

Message number 28 is defined as :

```
You try to go East, but your way is barred by a pair of
rusty looking park gates.
```

This is another message printed on two lines, and will be printed if the player attempts to go through the park gates without first unlocking AND opening them.

This is the last message needed for the adventure, so after you have typed **Y**, instead of moving on to message 29 by pressing **CURSOR RIGHT**, press **ESC** instead to return back to the messages bar menu (if you have already moved on to message number 29 by mistake, don't worry - simply press **RETURN** until you reach the confirmation prompt, type **N** and then press the **ESC** key).

As we have now finished defining the messages, select the option M  Return To Main Menu. The messages that we have just defined will be stored, and we will be returned back to the editor main menu.

Next, we'll define the single movable object we'll be using in our "Park" adventure, object number 1, the golden key.

Select the option O Movable Object Definitions from the editor main menu.

A smaller bar menu will be displayed with very similar options to the menu we saw when defining the locations ie., Browse, Edit, Print and Return To Main Menu.

Select the option E Edit Movable Objects from the menu. You will again be prompted to enter the number of the movable object you wish to alter. You can enter any number between 1 and 255 (the maximum number of movable objects allowed), although since we only have a single movable object in our "Park" adventure, type 1 (to edit movable object number 1) and press **RETURN**. The screen will clear, and you will now see the movable object editing screen.

There are only two entry fields on this screen. The cursor will be positioned in the first field near the top of the screen. This field is 15 characters long and is used to hold the name of the movable object.

Type

golden key

and press **RETURN**. The cursor will now be positioned in the single character field directly below. This field is used to define a prefix which will be added to the beginning of the movable object's name when printed on screen, enabling longer movable object names to be defined. There are four prefixes available, depending on which character is entered in this field -

A adds the prefix "A " to the movable object name
N adds the prefix "An " to the movable object name
S adds the prefix "Some " to the movable object name
T adds the prefix "The " to the movable object name
A space entered in this field will result in no prefix being added to the movable object name

The golden key will have a prefix of "A ", so type **A** (no need to press **RETURN** in this field). We have now ensured that whenever the name of movable object number 1 is printed in our adventure, the computer will print "A golden key". As when editing locations, you will now be asked

Is this correct ? ( Y / N )

If you're happy with what you've just entered, type **Y**, otherwise, type **N** then press **ESC** to return to the movable objects menu and try entering the movable object data again.

If we had more movable objects to define, we could now use **CURSOR RIGHT** to step to the definition of movable object number 2 in the same way as we did when defining our locations earlier. However, as we've now finished our movable object definitions, press **ESC** (or press the **ESC** mouse button) to return to the movable objects bar menu.

As before, we can now inspect our movable objects by selecting the option B Browse Movable Objects from the movable objects menu. Again, **CURSOR LEFT** and **CURSOR RIGHT** can be used to step backwards and forwards through the movable object definitions (although movable objects numbered 2 upwards should be blank). When you've seen enough, press **ESC** to return to the movable objects menu, and select the option M Return To Main Menu to return once again to the editor main menu.

## Defining Unmovable Objects

Now we'll define the two unmovable objects in our "Park" adventure : unmovable object number 1, the fountain and unmovable object number 2, the park gates.

Select the option U Unmovable Object Definitions from the editor main menu.

A smaller bar menu will be displayed with the usual four options : Browse, Edit, Print and Return To Main Menu. (All this should seem quite familiar by now!)

Select the option E Edit Unmovable Objects from the menu and select unmovable object number 1 for editing.

There is only one entry field on this editing screen. This as you might expect, is used to hold the name of the unmovable object.

Type

```
stone fountain
```

and press **RETURN**. The normal prompt asking you whether the data is OK will appear. Type **Y** and move on to define the next unmovable object definition :

```
rusty gates
```

and return once again to the editor main menu.

## Defining Verbs

As mentioned previously, the START starter file we have loaded contains 11 verbs already defined. However, for our "Park" adventure, we will need to add two extra verbs - verb number 12, UNLOCK and verb number 13, OPEN.

Select the option B Verb Definitions from the main menu. Another bar menu with the usual four options will be displayed.

Select the option E Edit Verbs from the menu. As we will start adding verbs from verb number 12, type 12 and press **RETURN**.

Again, there is only one field on this entry screen. This is used to hold the "name" of the verb.

Type

```
unlock
```

and press **RETURN**. The normal prompt asking you whether the data is OK will appear. Type **Y** and move on to define the next verb :

```
open
```

and return once again to the editor main menu.

## Defining Directions

The START starter file already contains most direction names that you're likely to want to use in your own adventures. You probably won't want to define any new directions yourself unless you are inventing your own completely new direction names, or if you're developing an adventure source from scratch without adapting it from the START starter file.

Direction names are defined in exactly the same way as objects and verbs (select the option D Direction Definitions from the editor main menu), although the maximum number of direction names that can be defined is 99 (movable objects, unmovable objects and verbs may have a maximum of 255 definitions) - It is most unlikely that you will want more than 99 different direction names in your adventures!

Our "Park" demonstration adventure will not require us to define any new direction names, as the START starter file already contains definitions for the two directions that we will be using ("EAST" and "WEST"). Feel free to look through any of the direction definitions using the "Browse" option from the directions menu if you wish.

---

It is important to realise that we have so far only defined the NAMES of the objects, verbs and directions that will be used in our "Park" adventure. The interpreter would still not understand the words "FOUNTAIN" or "KEY" if we included them in our typed-in commands when playing the adventure game.

We still need to give the interpreter a rudimentary knowledge of all the important words that a player is likely to use when playing our "Park" adventure. To do this, we will need to define a VOCABULARY for our game...

## Defining A Vocabulary

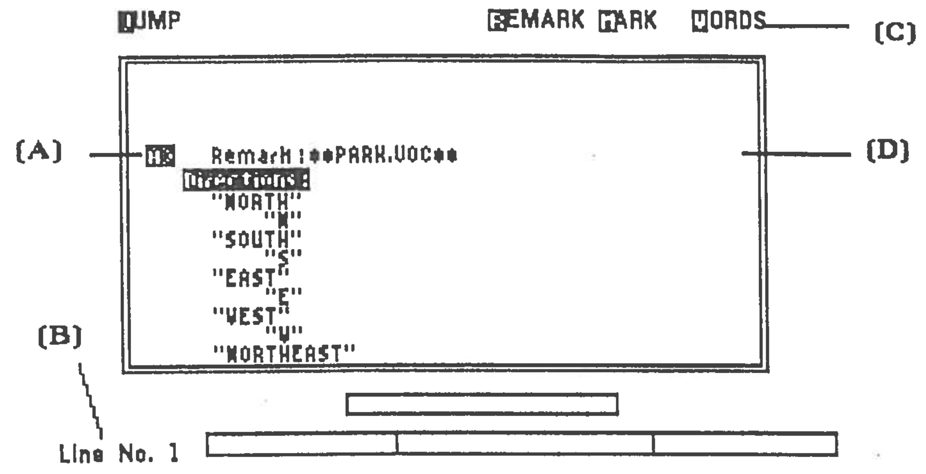
The VOCABULARY section of the editor is where we define all the words that the interpreter will recognise when instructions are typed in by the player when playing the game. It is by defining words in this section, that we can make sure that the interpreter actually "understands" what the player has typed in!

In the vocabulary, all words are categorised into five separate "classes" : Directions, Verbs, Movable Objects, Unmovable Objects and Prepositions.

| | |
|---|---|
| Directions | These words are the directions that we are using to connect locations together, eg. NORTH, SW etc. |
| Verbs | These words are the verbs used in our adventure such as TAKE, DROP etc. Game commands such as SAVE, RAMLOAD etc are also classed as verbs. |
| Movable Objects | These words correspond to any movable objects defined in the adventure. |
| Unmovable Objects | These words correspond to any unmovable objects defined in the adventure. |
| Prepositions | These are words that can be used to alter the meanings of verbs such as IN, OUT, ON, OFF etc. Consider as an example the inputs GET IN BOAT and GET OUT BOAT, which although contain the same verb ("GET") and unmovable object ("BOAT") obviously have different meanings. |

Each "class" may contain up to 255 words, and each word may have up to 255 synonyms (synoyms are different words with the same meaning).

Select the option V Vocabulary from the editor main menu. You will see the vocabulary editing screen as shown in Figure 6.



```
Remark :**PARK.VOC**
Directions:
"NORTH"
      "N"
"SOUTH"
      "S"
"EAST"
      "E"
"WEST"
      "W"
"NORTHEAST"
```

Fig 6. - The vocabulary editing screen.

The entire vocabulary is stored as long list of words each on its own line. This list is always shown and manipulated in the area of screen marked (D) in Fig. 6. The cursor marked (A) in Fig. 6. is used to point at the place in this list where we may want to delete, add or alter lines.

Move the list down one line by using the **CURSOR DOWN** key. You will notice that the area of screen marked (B) in Fig. 6. now shows that the cursor is pointing to the second line in the vocabulary listing. Press the **CURSOR UP** key to move back up to line number 1. If we want to move up or down the listing more quickly, we can use the **F1** and **F0** function keys which will move the listing up and down 10 lines respectively.

If you have a SAM Mouse connected, you will notice a small pointer somewhere on the screen. By moving this pointer to a line in the listing and pressing the SELECT button, we can bring any line we wish directly to the cursor. This can be handy for moving up and down the listing quickly.

As the vocabulary is stored as one long list, we obviously need to mark in some way where one "class" of words ends and the next begins. Line number 2 is currently a marker indicating the start of the direction words. You will notice that this marker is displayed in INVERSE (ie. PAPER on PEN) so that it stands out quite clearly.

The first word after the marker is "NORTH". As it is the first word in the directions section, it will be recognised as direction number 1 if the player types this word in his commands when playing the adventure. The next line down is a SYNONYM of direction number 1 - "N". We can tell it's a synonym because it is justified to the right. Remember, synoyms are alternative words that we wish to have exactly the same meaning. So in this case, if the player typed in **EITHER** "NORTH" or "N" in his commands, the interpreter would recognise the word as being direction number 1. As mentioned previously, each word can have up to 255 synoyms, so it's probably best to enter as many as you can think of when designing your own adventures, so that they are as friendly as possible. Nothing is worse than having to search for an obscure word in order to solve a problem when playing a game!

The next line down showing the word "SOUTH" (line number 5), is justified back to the left again. This indicates that it is a new word rather than another synoynm of the word "**NORTH**". This word would be recognised as direction number 2 if the player included it in his commands when playing the adventure.

The START starter file already has a small vocabulary supplied which corresponds with the existing definitions for directions and verbs, along with a few common prepositions. At the moment, there are no words defined in either the movable or unmovable object "classes".

At the top of the screen in the section marked (C) in Fig. 6, you will notice the words JUMP, REMARK, MARK and WORDS. These are titles of several bar menus which can be accessed in order to edit the vocabulary.

Press the **CURSOR RIGHT** key. You should now see the "JUMP" bar menu on-screen. By using the **CURSOR RIGHT** and **CURSOR LEFT**" keys, we can move across the screen highlighting the various bar menus which are available. press the **ESC** key (or **ESC** mouse button) to remove the currently selected bar menu from the screen.

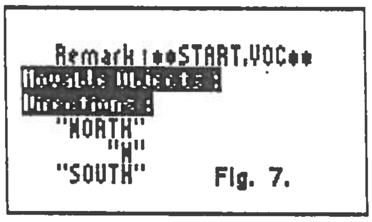
If you're lucky enough to own a SAM Mouse, you may select a bar menu by moving the mouse pointer up to the title of the bar menu you require, and pressing the mouse SELECT button.

Bar menus may also be directly selected by pressing the key which corresponds to the letter highlighted in the bar menu title. Press the **J** key. The "JUMP" bar menu will again be selected. As you have probably guessed, this menu allows us to jump directly to various points in the vocabulary list. select the option V  Jump To Verbs. The cursor will now be pointing to a marker (in inverse) which defines the start of the verb words in the listing. The options D  Jump To Directions, or P  Jump To Prepositions can also be used in the same way to jump to the appropriate markers in the vocabulary listing. However, trying to jump to either the movable or unmovable markers will result in an error message being displayed because at the moment these markers do not yet exist!

Select the "JUMP" bar menu again, and this time select the option L  Jump To Line Number You will now be prompted to enter the number of the line you wish to jump to. Type 1 and press **RETURN**. The cursor will now be positioned back at line number 1 in the vocabulary list.

We will now enter the vocabulary entries corresponding to the movable object (the golden key) which we defined earlier. The first step to take is to insert a movable objects marker in our listing. Select the "MARK" bar menu and select the option M  Mark Movable Objects. You will now be asked

Is this correct ? ( Y / N )

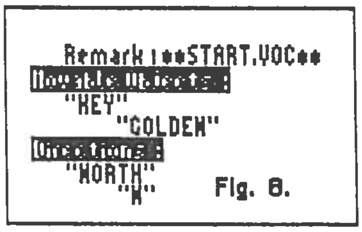


```
Remark :**START.VOC**
Movable Objects :
Directions :
  "NORTH"
    "N"
  "SOUTH"
```

Fig. 7.

Type **Y** (no need to press **RETURN**), and the marker should be inserted as line 2 in the vocabulary. Your listing should now look like the one shown in Figure 7.

Next, select the "WORDS" bar menu, and choose the option N  New Word. you will now see an entry field 15 characters long. Type in the word **KEY** (it will be forced into upper-case, don't worry about this) and press **RETURN** and the **Y** key to confirm all is correct. We have now defined movable object number 1. We will now add a synonym to this word. Select the "WORDS" bar menu again, and this time select the option S  Synonym Word. Now type in the word **GOLDEN** and press **RETURN** and **Y** as before.



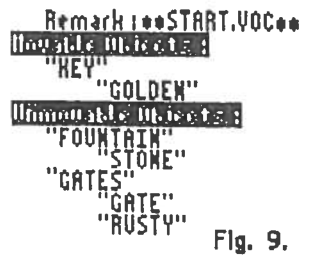
```
Remark :**START.VOC**
Movable Objects :
  "KEY"
    "GOLDEN"
Directions :
  "NORTH"
    "N"
```

Fig. 8.

The listing on screen should now look like the one shown in Figure 8.

Next, we will insert the unmovable objects in a similar way. Firstly, select the "MARK" bar menu and choose the U  Mark Unmovable Objects option. confirm all is correct, and the unmovable objects marker should be inserted as line number 5 in the listing.

Now enter the word **FOUNTAIN** as the first unmovable object word with the synonym **STONE**. The next unmovable object word is to be **GATES** with the synonyms **GATE** and **RUSTY**. The start of our vocabulary listing should now look like Figure 9.



```
  Remark :**START.VOC**
Movable Objects :
  "KEY"
      "GOLDEN"
Unmovable Objects :
  "FOUNTAIN"
      "STONE"
  "GATES"
      "GATE"
      "RUSTY"
```

Fig. 9.

Note that we have entered the unmovable objects in the same order as we defined their names earlier. **IT IS ESSENTIAL THAT THIS IS DONE.** If we had entered the rusty gates in the vocabulary listing before the stone fountain, we would have had very funny results when playing the finished adventure!

Now all we need to do, is to add the extra verbs "UNLOCK" and "OPEN". We could jump directly to the verbs marker in the vocabulary list by using the option V  Jump To Verbs in the "JUMP" bar menu, but since we know that the last verb defined so far is "EXAMINE" (verb number 11), we can jump directly to this word instead. Select the option W  Jump To Word from the "JUMP" bar menu. You will now be prompted to enter the word which you wish to jump to. Type the word **EXAMINE** and press **RETURN**. The cursor should now be pointing at the appropriate word in the listing. This feature can be very handy indeed for moving around the vocabulary quickly if we know of a word located near the point where we wish to start editing.

Move the cursor two lines further down the list by pressing the **CURSOR DOWN** key twice. The cursor should now be pointing to the the second synonym of the word "EXAMINE" - "EXAM". We are now at the correct position to enter our two new verb words, "UNLOCK" and "OPEN". Enter both of these as "new" words (neither need have any synonyms). Again, take care to ensure that the word "UNLOCK" is entered first, as we defined this as the name of verb number 12 earlier, and "OPEN" as verb number 13.

This is all very well, but how do we delete or change a line if we have made a mistake ?

Press the **F0** function key to move the cursor to the very end of the vocabulary. It should now be pointing to the last preposition defined - "AT". Now select the option R  Remark from the "REMARK" bar menu. You will be prompted to enter some text into an entry field on the screen. Type the text **SAS** and press **RETURN**. The remark will now be inserted at the very end of the vocabulary list. (Remarks are used to simply comment a section of the vocabulary listing for your own reference - they have no effect at all on the vocabulary as it is compiled).

Now press the **ESC key** (or the **ESC button on the SAM Mouse**). A new bar menu will appear in the middle of the screen with the following options available :

| | |
|---|---|
| A  Add Line | This option forces the cursor into "Add line" mode. While in this mode, a letter "A" will be positioned by the cursor (at (A) in Figure 6.), and all lines entered will be INSERTED at the current cursor position. The cursor is automatically set to "Add line" mode when you first enter the vocabulary section from the editor main menu. |
| C  Change Line | This option forces the cursor into "Change line" mode. While in this mode, a letter "C" will be positioned by the cursor, and all lines entered will OVER-WRITE the current line rather than insert a new line at the current line position. The first time that this option is selected from the bar menu, the current line will be displayed, allowing you to edit or amend it. The cursor remains in "Change line" mode until the option A  Add Line is selected from this same bar menu. |
| D  Delete Line | This option will delete the current line pointed to by the cursor. Before deleting the line, you will be prompted for confirmation - just in case! It is impossible to delete line number 1 in the vocabulary list (a REMARK), as this is used by the editor for reference purposes. |

| | |
|---|---|
| O Count Current Word | This option can be used to count how far the current word pointed to by the cursor is in its "class" (very handy!). A suitable error message is displayed if the cursor is currently pointing to a "class" marker or a REMARK line. |
| L List Vocabulary | This option allows you to list a section of the vocabulary onto the screen. When selecting this option, you will be prompted to enter the line numbers where you wish the listing to start and end. While listing, the number displayed at the far left is the line number, while numbers shown in brackets are the word's item number in the current "class". |
| P Print Vocabulary | Works exactly the same way as above, but the listing is sent to a printer instead of the screen. |
| M Return To Main Menu | This option returns you back to the editor Main Menu. |

To alter the text in the REMARK line we've just added, press **ESC** to bring up the options bar menu (if it isn't already displayed) and select the option C Change Line. The remark entry field will be shown on the screen containing the text "SAS" which we entered earlier. To change the remark text simply type something different over the existing text and press **RETURN**. You may move through the field by using the **CURSOR LEFT** and **CURSOR RIGHT** keys if you wish.

Type **Y** to confirm all is correct, and you will now see your amended REMARK line in the vocabulary list. Note that the cursor is still in "Change line" mode - there is a letter "C" displayed by the cursor. If we were now to add a new line, for example by adding a "new" preposition, it would over-write our REMARK line rather than being inserted just after it. If we wished now to insert a new vocabulary line, we would have to bring up the options bar menu (by pressing **ESC**) and select the option A Add Line to put the cursor into "Add line" mode first.

Now let's delete our REMARK line, by pressing **ESC** to bring up the options bar menu, and selecting the option D Delete Line. Type **Y** and press **RETURN** to confirm that we do indeed wish to delete the line. The vocabulary listing will now be displayed on the screen with our deleted line removed.

Finally, to leave the vocabulary section of the editor, press **ESC** to bring up the options bar menu again, and select the option M Return To Main Menu to return once more to the editor main menu.

---

When a player types in a command when playing an adventure, the interpreter firstly splits up the input into separate words and searches through the vocabulary looking for a match for each word. If no match is found, then the word is in effect completely ignored. This saves us the rather tedious chore of having to define silly words such as "a" or "the" in the vocabulary!

Depending upon in which "class" the match was found, the interpreter knows whether for example, verb number 5, or direction number 2 was mentioned by the player. Later on, you will learn how to use this information to allow the player to perform various actions and solve the adventure's puzzles.

Note that it doesn't matter in which order the various "classes" are marked in the vocabulary listing. The compiler automatically sorts out the classes for you when it compiles an adventure source.

## Saving Adventure Source To Disk

Now might be a good time to save our uncompleted "Park" adventure source to disk. It is generally a good idea to save your progress regularly when creating an adventure source - you never know when you're likely to have a power-cut !!

Make sure you have handy the blank formatted disk mentioned at the start of this chapter - (this will be your "source" disk, used to store your "Park" adventure source), and select the option S Save Adventure Source from the editor main menu. You will now be prompted to enter a filename in a field near the bottom of the screen. Note that this field already contains the filename "START" because that was the name of the file we loaded earlier. Type **PARK** over the existing filename (using a space to over-write the old "T"). **UNLESS YOU HAVE A 1MB MEMORY INTERFACE CONNECTED, MAKE SURE YOU STILL HAVE THE "EDITOR" DISK INSERTED IN DRIVE 1.** Now press the **RETURN** key. The editor will load in some files from the drive, and after a short while will prompt you to insert your "source" disk. Remember, if you have two drives fitted to your SAM, the editor always expects the "source" disk to be in the right-hand drive - drive 2.

Note that when saving very large source files, if you only have a single disk drive and no 1Mb memory interface, you may be prompted to swop between your "source" and "editor" disks several times. Just follow the on-screen prompts and insert the disks as directed.

After the file has been saved, you will be prompted to insert your editor disk again, and you will return back to the editor main menu.

As a general rule, it's not a good idea to store more than one adventure source file on the same disk (or anything else for that matter!). This is because the editor ALWAYS expects the full 780k on the disk to be available for data storage.

---

You might now wish to compile the "Park" source as it now stands, and see how the adventure plays so far. If you wish to do this, refer to the next chapter **"The Source Compiler"** on how to compile the adventure source and play the adventure. When playing, you will notice that you will not be able to "do" very much apart from wander around the various locations. To make our adventure interact with the player's commands, we will have to use **SAS**'s special programming language to write some simple routines which will evaluate and act upon any instructions given by the player, and also to control the various events and puzzles which will be found in our adventure world.

When you've finished wandering around the park (and not doing anything in particular!) re-load the **SAS** editor disk and load back in the "PARK" adventure source from your "source" disk.

## The source banks

You will have noticed from the editor main menu, that there are four source "banks" available. The **SAS** programming language is used to write routines in each of these banks which control how the adventure behaves as it is being played.

Each of these four banks has a distinct purpose :

| | |
|---|---|
| Source Bank 1 | This bank countains routines which define the initial state of the adventure world at the start of each new game. The routines located in this bank are ONLY executed when a new game is begun. It is therefore, in this bank that you would define the location number which the player starts out in, where any movable or unmovable objects are to be found in the adventure world and the initial values of any flags (more about flags later) that you may wish to use. |

| | |
|---|---|
| Source Bank 2 | This bank is used mostly for routines which evalute what the player has typed in his input, and act accordingly. The routines in this bank are executed as soon as the player has finished typing in his commands, and has pressed the **RETURN** key. At this point, the interpreter knows what directions, verbs, movable objects, unmovable objects and prepositions (if any) that the player has mentioned in his input. The routines in this source bank can be thought of as "high priority" conditions. |
| Source Bank 3 | This bank is used for routines which will be executed regardless of whatever commands the player has just typed in. As an example, this would be the ideal place to locate a routine which would make the player feel "hungry" after a certain number of turns. The routines in this source bank will be executed after those in source bank 2 (assuming that a **JUMP TO END** source command has been used - more about this later). They can therefore be thought of as "low priority" routines. |
| Source Bank 4 | This bank is used to contain routines which will be executed ONLY when the player has moved to a new location. These routines will be executed after those in bank 2, and before those in bank 3 (provided a **JUMP TO END** source command is used). It is in this bank that routines will be located to describe the "new" location that the player has just moved to, for example. |

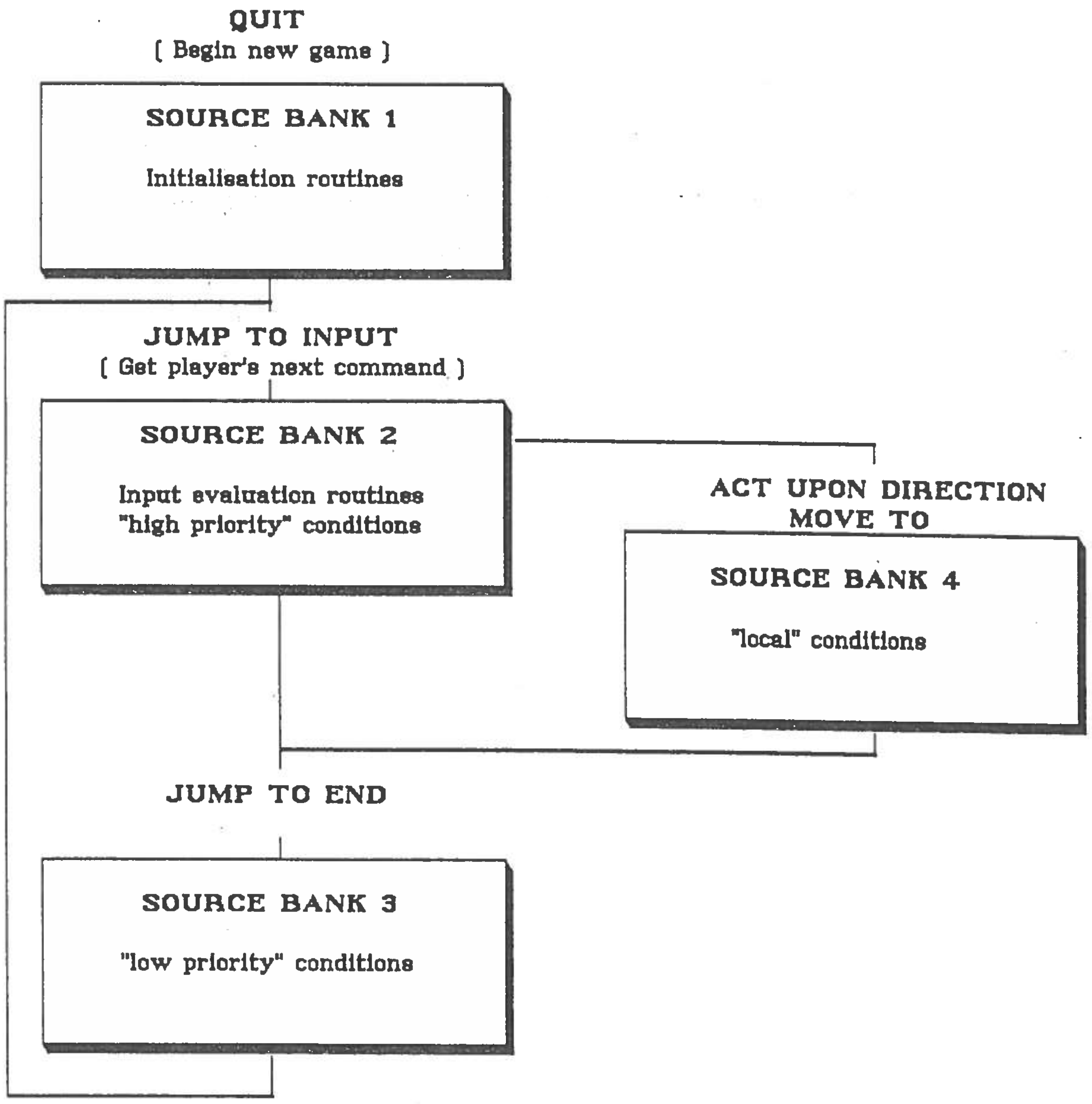


Fig. 10. - Source banks flow-chart.

Routines in each of the four source banks are stored and edited as a list of commands, with each command having its own new line. Each of the four banks may contain up to 4096 command lines (available memory permitting) which should be more than adequate for most of the routines which you're likely to want to write.

Altogether, there are about 70 different commands (these are all listed in detail in the appendix near the end of this manual), although you will probably not use some of them very often (if at all!).

As an example, consider the command **PRINT MESSAGE** which we would use if we wanted to print some text onto the screen during the game. The text could be anything - a response to something the player has just done, a welcome message or even a detailed description of one of the objects in the adventure.

As there are up to 1024 different messages available, we obviously have to specify which one we want to use. we do this by adding to the command what is known as a **PARAMETER**. In this case, the parameter is a numeric expression which gives the value of the message number we wish to use. The numeric expression could be just a simple number, or a more complex numeric expression. **SAS** allows the use of addition, subtraction and various numeric functions (you'll learn about these later) to be included in any numerical parameter required by a command. This flexibility makes the **SAS** programming language very powerful indeed.

To illustrate this, suppose we wanted to print message number 10. If we used a simple number, the command would appear as :

```
  Print Message [10]
```

(Note how the parameter always appears within square brackets). Alternatively, we could have printed the same message (message number 10) by using the command like this :

```
  Print Message [9+1]
```

Or if we were feeling particularly obtuse :

```
  Print Message [12-10+6+2+1-1]
```

Which (believe it or not) would have had the same effect!

If we wanted to print several messages, one after the other, we would simply use the **PRINT MESSAGE** commands consecutively in the listing. For example :

```
  Print Message [11]
  Print Message [12]
  Print Message [13]
```

Which as you would expect, would result in message number 11 being printed on the screen, followed by messages 12 and 13.

If you're used to programming in BASIC, you might have noticed that the **SAS** language does not use line numbers for each line. If we wish, we can use what is known as a **LINE LABEL** to give a name to a certain point in our "program" listing. Line labels can have names up to 8 characters long and would appear in a listing like this :

```
LINE.LAB:
  Print Message [10]
```

In this case, the line label's name is "LINE.LAB". Line labels always have their names displayed in upper case and have a colon character (:) added to the name. Note that the line label itself is justified to the left of the listing. This makes it stand out and easy to spot when looking at a listing with a large number of control lines.

We can force a jump to a different point in the program by using the **GOTO** command. Note that unlike BASIC's GOTO command, **GOTO** requires you to specify a line label instead of a line number to jump to.

Example

```
  Print Message [10]
  Goto JUMP
  Print Message [11]
JUMP:
  Print Message [12]
```

If we were to use the above routine, message 10 would be printed, but the **GOTO** command would force a jump past the **PRINT MESSAGE** command printing message 11 (which would not be printed) to the line label "JUMP", and message 12 would then be printed.

Sometimes, we will only want a command executed in certain circumstances. For commands to be executed conditionally, we can use one of the various **IF..** commands available in **SAS**.

Example

```
  If Verb > [10] THEN
    Print Message [80]
  Print Message [81]
```

In the above example, message number 80 would only have been printed if the player had included a verb with a number greater than 10 in his input (if the player had not mentioned any verb at all, then the verb number would have been zero). In all cases, no matter what the player had typed, message 81 would still have been printed. This is because an **IF..** command with a **THEN** "action" only affects the following single command line.

Similar **IF..** commands exist for checking which directions, movable objects, unmovable objects or prepositions (if any) that the player has included in his input. The comparisons which can be made are as follows :

| | |
|---|---|
| < | Is less than |
| > | Is greater than |
| = | Is equal to |
| <> | Is not equal to |
| <= | Is less than or equal to |
| >= | Is greater than or equal to |

Other "actions" are available with all of the **IF..** commands. Using a **THEN** action is fine if we only want to execute a single command line conditionally, but if we want to execute a whole series of commands conditionally, we have to use a **DO** action with the **IF..** command instead.

Example

```
  If Preposition = [1] DO
    {
    Print Message [1]
    Print Message [2]
    }
```

In this example, messages 1 and 2 will only be printed if the player mentioned preposition number 1 in his last input.

Note that IF.. commands with a DO action ALWAYS require the conditional commands to be placed within curly brackets. The commands between these brackets are known as a **STRUCTURE**. The first bracket (the "{" character) is in fact a **STRUCTURE START** command and must always be placed immediately after the IF.. command on the next command line. The closing bracket (the "}" character) is a **STRUCTURE END** command.

Structures may contain any number of commands. It is even quite possible to place one structure inside another, in fact structures may be "nested" in this way up to 255 levels deep!

Example

```
 If Verb = [5] DO
   {
   Print Message [10]
   If Movable Object = [1] DO
     {
     Print Message [11]
     }
   }
```

In this example, message 10 would only be printed if the player had mentioned verb number 5 in his last input. If the player had mentioned verb number 5 AND movable object number 1 in his input, then message number 11 would have been printed as well. It should be apparent that by using structures, quite sophisticated logic paths can be defined in the **SAS** programming language.

Several IF.. commands may be chained together by using the actions **AND** or **OR**.

Example

```
 If Unmovable Object = [9] OR
 If Unmovable Object = [10] THEN
   Print Message [15]
```

In this case, message 15 would only be printed if the player had mentioned EITHER unmovable object number 9 or unmovable object number 10 in his last input.

The **AND** action is used in a similar way :

```
 If Verb = [8] AND
 If Movable Object = [5] AND
 If Preposition = [2] THEN
   Print Message [1]
```

In this example, message 1 would only be printed if the player had mentioned verb number 8 AND movable object number 5 AND preposition number 2 in his last input.

Any number of IF.. commands may be chained together in this way, but the **AND** and **OR** actions may not both be present in the same chain. A routine such as :

```
 If Verb <> [2] AND
  If Verb > [18] OR
  If Preposition = [3] THEN
    Goto JUMP
```

would NOT be permitted. This would cause the compiler to display a suitable error message when the source was being compiled.

GOTO can also be used as an action for an IF.. command. The command line :

```
 If Unmovable Object >= [3] GOTO UOBJ.JMP
```

would have exactly the same effect as

```
 If Unmovable Object >= [3] THEN
   Goto UOBJ.JMP
```

but is much more memory efficient as it only uses one source command line instead of two.

Sometimes, there will be certain sections of our program which we will want to execute several times. we could duplicate the same routine in our program for each time it was used, but this would be rather wasteful of adventure source space. What can be done instead, is to store the routine once in what as known as a **SUBROUTINE.** We can execute the command(s) in our subroutine by using the **GOSUB** command. Like the **GOTO** command, **GOSUB** uses a line label as a parameter instead of a line number like SAM BASIC's GOSUB.

The subroutine which we are calling must ALWAYS begin a line label so that **GOSUB** knows where the subroutine is located. There can be as many commands in the subroutine as you wish, but the last command in the subroutine must ALWAYS be a **RETURN** command. When a **RETURN** command is encountered. The interpreter knows that the subroutine is completed, and control is then passed back the command line AFTER the one which originally called the subroutine by using **GOSUB.**

It is quite possible for one subroutine to call another one, though each subroutine must of course end with its own **RETURN** command. Subroutines can be nested in this way up to a maximum of 255 levels deep.

Example

```
  Gosub MESS1
  Gosub MESS1
  Goto SKIPPAST
MESS1:
  Print Message [1]
  Return
SKIPPAST:
```

In the above example, we have used a subroutine called "MESS1" to print out message number 1 twice (once for each of the **GOSUB** commands). Note that we have taken care so that the program flow bypasses the actual subroutine itself by using **GOTO** to jump forwards to the line label "SKIPPAST". This is because if the interpreter encounters a **RETURN** command without a **GOSUB** being previously used, an error message is displayed, as the interpreter has no idea where in the source bank the **RETURN** command is supposed to "return" to !

**GOSUB** may also be used as an action for any of the **IF..** commands in a similar way to **GOTO.** For example :

```
  If At Location [3] GOSUB LOC.3
```

which again, is more memory efficient than the equivalent

```
  If At Location [3] THEN
    Gosub LOC.3
```

This example would execute the subroutine located at the line label "LOC.3" if the player was currently at location number 3.

Note that **GOSUB** can ONLY call subroutines which are located in the same source bank. It is impossible to use **GOSUB** in one source bank to call a subroutine located in a different bank.

It is possible to loop around a certain section of a program by using a **FOR / NEXT** command. When using this command, START, END and STEP numeric paramaters must ALWAYS be defined.

**FOR / NEXT** ALWAYS requires to be followed immediately by a structure (similar to **IF..** commands with a **DO** action) in which the actual looping is to take place.

Example

```
For / Next : Start [1] : End [10] : Step [1]
  {
  Print Message [30]
  }
```

This example would result in message number 30 being printed onto the screen no less than 10 times!

It is possible to access the current looping counter by using the numeric function **CNT**. Numeric functions may form part (or all) of any numeric parameter required by a command. In the example below, **CNT** is used to specify the number of the message which is printed onto the screen within the loop.

```
For / Next : Start [2] : End [6] : Step [2]
  {
  Print Message [CNT]
  }
```

By using the **CNT** numeric function, this example would result in message number 2 being printed on the screen, followed by messages 4 and 6.

**FOR/NEXT** loops may NOT be nested (ie. it is not possible to palce one **FOR / NEXT** loop inside another one).

There are many other numeric functions which can be used in a similar way to **CNT**. These are all listed in detail at the start of the glossary of source commands located towards the end of this manual.

## Using Flags In An Adventure

Most people think that flags are nicely coloured pieces of material which are hung from flagpoles on special occasions, or are used to decorate sandcastles.

**SAS** uses what are known as "flags" to indicate anything that has happened during the adventure. It's probably best to think of these flags as the flags used by linesmen in football games, where a linesman holds up a flag to indicate when a player is "offside" or keeps his flag held down when he's not. By looking at the linesman's flag, the football referee therefore knows if the player is "offside" in the game.

**SAS**'s flags are used in a similar way, although these flags could be used to indicate whatever you like! They could indicate for example, whether a door is open or not, or whether a hidden object in the game has been discovered yet. **SAS** has 255 flags available for your use, which in a well planned adventure should be more than adequate (my first adventure for the SAM - "**FAMOUS FIVE**" only used about 80 flags).

**SAS**'s flags can be thought of as a series of 255 little boxes which can be inspected throughout the game in order to know the flag's current "status".

Although in most cases, you will only want a flag to contain one of two values - "true" (in which case the flag will contain a value of one) or "false" (in which case the flag will contain a value of zero), flags may actually contain any number between zero and 255. This would allow us to also use flags as counters or we could for example, use a flag to indicate how many coins a player was carrying at a given time.

As mentioned above, it is entirely up to you to decide what flags are used for which purpose in your adventure. IT IS ESSENTIAL THAT YOU KEEP A NOTE OF WHICH FLAGS YOU HAVE USED. You may have no trouble remembering which flags you are using while writing your game, but when you are play-testing your adventure and are trying to trace any annoying "bugs", such a list of flags and their uses will prove invaluable.

At the start of each new game, ALL flags ALWAYS have their contents automatically set to zero.

Flags can have their contents altered by using the **SET FLAG** command.

Example

```
  Set Flag [18] To [1]
```

This example would place a value of 1 in flag number 18.

The contents of flags can be checked by using the **IF FLAG** command, which is used in exactly the same way as the other **IF..** commands discussed earlier.

Example

```
  If Flag [3] = [1] THEN
    Print Message [12]
```

This example would print message number 12 if flag number 3 contained the number 1.

Flags can also have their contents inspected by using the numeric function **FLG(n)**, where **n** represents a number or another numeric expression corresponding to the number of the flag which we wish to inspect.

Remember, numeric functions can only be used in commands which expect numeric parameters.

Example

```
  Print Message [FLG(8)]
```

In this example, flag number 8 would contain the number of the message which would be printed by the **PRINT MESSAGE** command.

There are several other commands which can be used to alter the contents of flags such as the **INCREMENT FLAG**, **DECREMENT FLAG** and **NOT FLAG** commands. These are all detailed in the glossary of source commands near the end of this manual.

In our demonstration "PARK" adventure, only three flags are needed :

| | |
|---|---|
| **FLAG 1** | This flag is used to indicate whether the key has been found in the fountain. A value of zero indicates that the key is still "hidden", and a value of one indicates the key has been found. |
| **FLAG 2** | This flag is used to indicate whether the park gates are unlocked or not. A value of zero indicates that the gates are still locked, and a value of one indicates that they have been unlocked with the key. |
| **FLAG 3** | This flag is used to indicate whether the park gates are open or not. A value of zero indicates that the gates are still closed, and a value of one indicates that they have been opened. |

Note that for all three flags, we have used a value of zero to indicate the status required at the start of the game. This is because at the start of each new game, all flags automatically have their contents set to zero.

In addition to the 255 flags available for your own use, **SAS** also has 30 system flags which the interpreter uses for its own purposes. THESE SYSTEM FLAGS MUST NOT BE USED AS NORMAL FLAGS BY YOUR ADVENTURES.

Because these system flags are used by the interpreter, you should never alter their contents unless you are sure of what you're doing. An appendix near the end of this manual lists all of the system flags and explains their uses.

The system flags that you are most likely to want to alter are system flags 10 to 15 which are used to hold the numbers of messages which are printed when the **INVENTORY**, **SCORE** and **DESCRIBE** commands are used. The "START" starter file already contains commands which assign default values to these system flags.

The contents of system flags can be altered by using the **SET SYSTEM FLAG** command.

Example

```
  Set System Flag [10] To [1]
```

This example would result in system flag number 10 containing a value of 1.

The contents of system flags can be checked by using the **IF SYSTEM FLAG** command or **SFLG(n)** numeric function, where **n** represents the number of the system flag to be inspected. **n** can be either a number or another numeric expression

Example

```
  If System Flag [11] > [0] THEN
    Print Message [SFLG(11)]
```

This example would print the message whose number is held in system flag 11 if system flag 11 contained a value greater than zero.

Like normal flags, system flags ALWAYS have their contents set to zero when a new game is started (except for system flag 3 whose contents remain unchanged). Therefore, the best place to put any commands which assign values to these flags is source bank 1.

---

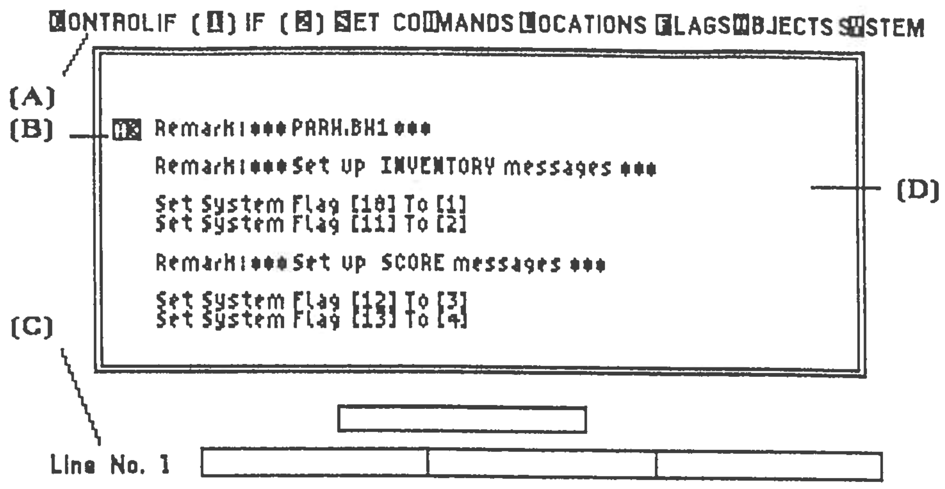
Now that we have a very basic understanding of the **SAS** programming language, we will start inserting the various routines which our PARK adventure will require...

## Entering Source Commands

If you do not have the "PARK" adventure source currently loaded, then load in the partially completed "PARK" adventure source which we saved previously by selecting the option A  Load Adventure Source from the editor main menu.

The commands stored in each of the four source banks are manipulated and edited in almost exactly the same way as the words were defined and edited in the vocabulary which we defined earlier. Each command is inserted on its own line, and this makes the editing of our routines very easy.

To begin with, we will write the routines which define the state of our adventure at the start of each game. As you should have realised, these routines ought to be located in source bank 1, so select the option 1  Source Bank (1) from the editor main menu.


CONTROLIF (1) IF (2) SET COMMANDS LOCATIONS FLAGS OBJECTS SYSTEM
(A)
(B)
(C)
(D)
Line No. 1

```
Remark!***PARK.BK1***

Remark!***Set up INVENTORY messages ***

Set System Flag [10] To [1]
Set System Flag [11] To [2]

Remark!***Set up SCORE messages ***

Set System Flag [12] To [3]
Set System Flag [13] To [4]
```

Fig 11. - The Source Bank Editing Screen.

The screen will clear and after a short while, you will see the source bank editing screen (in this instance for editing source bank 1) as shown in Figure 11.

The first thing you'll probably notice, is that the screen looks very similar to the editing screen which we used earlier for defining the vocabulary in our adventure, and indeed, the command lines forming our routines in each of the four banks are edited in almost exactly the same way as the vocabulary lines, with each command line being inserted, deleted or amended withing the long "list" of lines which form our program.

Like when editing the vocabulary, the "list" of command lines is always shown and manipulated in the area of screen marked (D) in Figure 11, and the cursor marked (B) is used to point at the place in the listing where we may want to delete, add or alter command lines. Again, like in the vocabulary editor, this cursor has two "modes" - ADD line mode (signified by the cursor displaying the letter "A") where new commands are INSERTED at the current cursor position in the listing, and CHANGE line mode (signified by the cursor displaying the letter "C") where new commands OVERWRITE the existing command line pointed to by the cursor.

We can move up and down the listing by using the **CURSOR UP**, **CURSOR DOWN**, **F1** and **F0** keys. The last two move the listing up and down 10 lines respectively.

If a SAM Mouse is connected, then it is possible to bring any desired line directly to the cursor by moving the mouse pointer to the required line, and pressing the mouse SELECT button.

Move the listing down one line by pressing the **CURSOR DOWN** key. The line counter located in the section of the screen marked (C) in Figure 11. should indicate that the cursor is now pointing to source line number 2 in the listing.

The routines in source bank 1 are used to define the initial status of the adventure at the start of each new game, and probably the first thing to consider, is where all of the movable and movable objects used in the game are to be located (ie. in which location they all "start out" in).

To place an unmovable object in a specific location, we can use the **PUT UNMOVABLE OBJECT** command (this is a command you will not have encountered before).

Example

```
Put Unmovable Object [10] At [25]
```

This example would place unmovable object number 10 at location number 25.

All of the commands available in SAS's programming language are accessed by selecting the various bar menus, whose titles are displayed at the top of the screen in the area marked (A) in Figure 11.

You will notice that a letter in each bar menu title is highlighted. By pressing the key corresponding to the highlighted letter, we can directly select the bar menu we require. SAM Mouse owners may also select a bar menu by moving the mouse pointer over the required bar menu title, and pressing the mouse SELECT button.

Press the **CURSOR LEFT** key. The SYSTEM bar menu will be selected. You may view all of the commands available by using the **CURSOR LEFT** and **CURSOR RIGHT** keys to move across the top of the screen, selecting the various bar menus which are available. A selected bar menu can be removed by pressing the **ESC** key. This can be handy for removing a bar menu which we selected by mistake. Press the **ESC** key now. The currently selected bar menu will disappear from the screen.

The **PUT UNMOVABLE OBJECT** command is available on the OBJECTS bar menu. Select the OBJECTS bar menu (by pressing the O key) and select the bar menu option U  Put Unmovable Object. You will now be prompted to enter the number of the unmovable object being used in an entry field. In our "PARK" adventure, we expect to find unmovable object number 1 (the fountain) at location number 1. (refer to the map of the "PARK" adventure as shown in Figure 3. near the start of this chapter), so type 1 (for unmovable object number 1 - the fountain) and press **RETURN**. In the next entry field, we define which location the unmovable object is to be placed in, so type 1 (for location number 1) and press **RETURN**. You will now see the familiar prompt :

```
Is this correct ? ( Y / N )
```

Provided you have not made any mistakes, type Y to confirm all is correct (no need to press the **RETURN** key here), and we will now see the new **PUT UNMOVABLE OBJECT** command line inserted as line number 3 in our listing.

We will also be placing unmovable object number 2 (the park gates) at location number 2 by using another **PUT UNMOVABLE OBJECT** command. So without altering the current editing cursor position (it should still be pointing at line number 3), insert another **PUT UNMOVABLE OBJECT** command placing unmovable object number 2 at location 2. This new command line will be inserted as command line number 4. After you have inserted the command, the start of the listing in source bank 1 should look like the one shown in Figure 12.

```
Remark : *** PARK.BK1 ***

Put Unmovable Object [1] At [1]
Put Unmovable Object [2] At [2]
Remark : *** Set up INVENTORY messages ***

Set System Flag [10] To [1]
Set System Flag [11] To [2]
```

Fig 12.

If we had wanted to place any movable objects at any specific locations where they could be "found" at the start of the game, then we could have used the **PUT MOVABLE OBJECT** command, which is used in exactly the same way as the **PUT UNMOVABLE OBJECT** command, to place any movable objects in their respective initial locations. However, since the only movable object which is to be used in our "PARK" adventure (movable object number 1 - the golden key) is in effect "hidden" (it will only be "discovered" when the fountain is EXAMINEd), there is no need to define an initial location for it.

At the start of each new game, before the first command line of bank 1 is executed, ALL movable and unmovable objects have their locations set to zero. Therefore, location zero can be thought of as a special location which cannot be visited by the player, but where all "unborn" and "destroyed" objects in the game are stored.

As you will have noticed in Figure 12., the second **PUT UNMOVABLE OBJECT** command looks rather cluttered right next to the following **REMARK**. In order to make our listing a lot more neat and readable, it is possible to insert blank lines at various points. The editing cursor should now be positioned at line number 4 in the listing. Insert a blank line by selecting the B  Blank Line option from the CONTROL bar menu. Type Y at the usual confirmation prompt to confirm all is correct, and the blank line will be inserted as line number 5.

It is often a good idea to label certain sections of our routines with **REMARK** commands, which will make the logic of the routines easier to follow and help in tracing any "bugs" which might crop up when we are playtesting.

Various **REMARKs** have been placed at certain points throughout the START Starter file to explain the purpose of the following commands, and it would be nice to insert a new **REMARK** at the start of the command lines we have just entered, explaining their purpose.

**REMARKs** are there simply to aid the adventure writer (in a similar way to SAM BASIC's REM statement) - they have no effect at all on the final adventure when it is compiled.

To insert our **REMARK**, we will first have to move the editing cursor back to just before our first **PUT UNMOVABLE OBJECT** command. We could of course, simply move our editing cursor back to the correct point in our listing by repeatedly pressing the **CURSOR UP** key, but the editing options bar menu also gives several other useful ways of moving around the source banks quickly.

Press the **ESC** key (or the mouse **ESC** button) to bring up the editing options bar menu. The menu will appear in the middle of the screen. The options available from the editing options bar menu are as follows :

| | |
|---|---|
| A Add Program Line | This option puts the editing cursor into "Add Line" mode. While in this mode, a letter "A" will be positioned by the cursor (at (B) in Figure 11.), and all command lines entered will be INSERTED at the current currsor position. The cursor is automatically set to "Add line" mode when you first enter a source bank section from the editor main menu. |
| C Change Program Line | This option forces the editing cursor into "Change line" mode. While in this mode, a letter "C" will be positioned by the cursor, and all command lines entered will OVER-WRITE the existing line rather than insert a new line at the current cursor position in the listing. When this option is selected from the options bar menu, the existing line at the current postion will be displayed, allowing you to edit or amend it if you wish (type "N" at the confirmation prompt if you do not). The editing cursor remains in "Change line" mode until the option A Add Program Line is selected from this same bar menu. |
| D Delete Program Line | This option will delete the current command line pointed to by the cursor. Before deleting the line, you will be prompted for confirmation - just in case! It is impossible to delete the first command line in each source bank (always a **REMARK**), as this is used by the editor for its own reference purposes. |
| G Go To Line Label | This option will allow the editing cursor to jump directly to an existing line label located anywhere within the current source bank. You will prompted to enter the name of the line label that you wish to jump to. This can be very handy for moving around the listing very quickly, provided of course, you know of a line label located near the point in the listing where you wish to jump to. It is NOT possible to jump to a line label located within another source bank. |
| N Go To Line Number | This option allows the editing curor to jump to any specific command line located within the current source bank. You will be prompted to enter the number of the line that you wish to jump to. |
| L List Program | This option displays a listing of the current source bank onto the screen. Unlike the display shown in the section of the screen marked (D) in Figure 11., this listing will be justified with all structures and line labels indented, making the program logic a lot easier to follow and understand. All program listing examples in this manuals are justified in this way. When selecting this option, you will be prompted to define the line numbers where you wish the listing to start and end. |

| | |
|---|---|
| P Print Program Listing | Works exactly the same way as above, but the justified listing is sent to a printer (if connected) instead. |
| M Return To Main Menu | This option returns you back to the editor main menu. |

To jump to the correct place in the listing for our **REMARK** command, select the option N Go To Line Number, and type 2 to bring command line number 2 to the editing cursor.

To actually insert the **REMARK** itself, select the option R Remark from the CONTROL bar menu. In the **REMARK** entry field, type in the following text :

```
*** Define all unmovable object's initial locations ***
```

Press **RETURN** and type **Y** at the usual confirmation prompt to insert the line as line number 3 in the listing.

```
Remark : *** PARK.BK1 ***

Remark : *** Define all unmovable object's initial locations ***

Put Unmovable Object [1] At [1]
Put Unmovable Object [2] At [2]

Remark : *** Set up INVENTORY messages ***
```

Fig 13.

Now insert another blank line immediately on the next command line to keep things looking nice and neat. If you have entered everything correctly, the start of the listing in bank 1 should now look like the one shown in Figure 13.

Now we need to define which location the player begins the game in. This is done by the **SET LOCATION** which is now located at line number 28 in the adventure source. Bring up the editing options bar menu again by pressing **ESC**, and use the option N Go To Line Number to jump to command line 28 in the listing.

The **SET LOCATION** command has a single numeric parameter which is used to define the number of the desired location. It is immediately followed by a **DESCRIBE** command which makes available any defined exits, and prints a full description of the "new" location detailing any exits and objects which are present You will notice that the START starter file has given the **SET LOCATION** a default location number of 1. If we had wanted the game to begin with the player starting at location number 1, then this command would have been fine, but since the player is to start out in location number 2 instead, we will need to amend this command line.

Press **ESC** to bring up the editing options again, and select the option C Change Program Line. You will immediately see an entry field for the command's numeric parameter. It will already contain its current expression (in this case the number 1). To alter the contents of the field, you simply type over any existing information, so type the number 2 (for location number 2) and press **RETURN**. Type **Y** at the confirmation prompt. The end of the listing should now look like the one shown in Figure 14.

```
Set System Flag [15] To [6]

Remark : *** Now define the initial location and describe it ***

Set Location To [2]
Describe

Jump To Input
```

Fig 14.

The **SET LOCATION** command should ONLY be used in source bank 1 to define the player's initial location. If you wish the player to move directly to other locations in other source banks, you should use the **MOVE TO** or **ACT UPON DIRECTION** commands instead.

Once the initial location has been described (by using the **DESCRIBE** command which prints the location description text, and gives details of any available exits or movable objects present), the following **JUMP TO INPUT** command is used to get the player's first input.

At the start of each new game, all flags (both user and system) have their contents set to zero (except for system flag 3 - see appendix on system flags). This means that if our adventure had required any flags to have a non-zero initial value, then appropriate **SET FLAG** commands would have had to have been inserted in source bank 1 as well. This does not apply however, with the PARK demo adventure.

The remaining commands in source bank 1, are all **REMARK**s and **SET SYSTEM FLAG** commands - used to define the numbers of various messages which are printed on the screen whenever the **SCORE, INVENTORY, DESCRIBE, DESCRIBE LOCATION, LIST EXITS** or **LIST MOVABLE OBJECTS PRESENT** commands are used. You could of course alter the relevant **SET SYSTEM FLAG** commands if you wished to use a different message (or none at all) instead of the default values assigned in the START starter file.

We have now completed all of the routines required for source bank 1. Note that the editing cursor is STILL in "Change line" mode (the letter "C" will be displayed next to the editing cursor). If we had wanted to insert any more additional commands before leaving leaving source bank 1, WE WOULD HAVE TO HAVE PUT THE CURSOR INTO "ADD LINE" MODE FIRST, by selecting the option A  Add Program Line from the editing options bar menu.

To leave source bank 1, select the option M  Return To Main Menu from the editing options bar menu. The changes you have just made in bank 1 will be stored, and you will return to the editor main menu.

---

Next, we will add the routines required for our PARK adventure in source bank 2. Remember, the routines in source bank 2 are ALWAYS executed as soon as the player has hit the **RETURN** key after typing in his latest instructions, so this is where we will add the routines which deal with the fountain and park gates being EXAMINEd, and the new UNLOCK and OPEN verbs.

Select the option 2  Source Bank ( 2 ) from the editor main menu. After a short while, you will see a source bank editing screen identical to the one used when editing source bank 1.

The START starter file already consists of a "skeleton" of various routines in souce bank 2, among which are routines already dealing with the SAVE, LOAD, RAMSAVE, RAMLOAD, QUIT, SCORE, INVENTORY, LOOK, TAKE, DROP and EXAMINE verbs. Also included are routines which handle any directions mentioned by the player, and routines which actually check (and display suitable error messages if nescessary) that any objects mentioned by the player are actually present or being carried by the player - as appropriate. The entire listing of START starter file, including the routines in source bank 2, are included in an appendix near the end of this manual, and might make interesting reading.

Firstly, we will deal with the fountain being EXAMINEd. We will need to insert some lines in the routine dealing with the EXAMINE verb, which control the golden key being "discovered" once the fountain has been inspected. Select the option G  Go To Line Label from the editing options bar menu, and enter **EXAMINE** as the name of the line label to jump to. All verb routines in the START starter file have line labels located at their beginning, and as you might expect, "EXAMINE" is the line label located at the start of the commands dealing with the EXAMINE verb.

The editing cursor should now be pointing at the "EXAMINE" line label currently located at command line number 165. Use the **CURSOR DOWN** key to move the editing cursor down to the blank line located at line number 169. This is where the extra commands dealing with the fountain and gates being examined will be inserted.

Now insert at this point the following command lines which deal with the fountain being examined :

```
If Unmovable Object = [1] AND
If Flag [1] = [0] DO
  {
  Print Message [17]
  Set Flag [1] To [1]
  Put Movable Object [1] At [1]
  Jump To End
  }
```

The **IF UNMOVABLE OBJECT** and **IF FLAG** commands are to be found from the "IF (1)" bar menu, the **STRUCTURE START** ({), **JUMP TO END** and **STRUCTURE END** (}) commands from the "CONTROL" bar menu, the **PRINT MESSAGE** command from the "SYSTEM" bar menu, and the **PUT MOVABLE OBJECT** command from the "OBJECTS" bar menu.

In the above routine, the commands located within the structure would ONLY be executed if the player had typed EXAMINE FOUNTAIN, AND flag number 1 contained the value of zero (its initial value). Notice that the routine itself does not actually detect whether the player has included the verb EXAMINE in his input. This is because this has already been detected earlier in the listing (at command line number 58), and the routines located at the line label "EXAMINE" will ONLY be executed if the player had included the verb EXAMINE in his input.

The **PRINT MESSAGE** command prints some text describing that the player had found the golden key in the fountain. The **SET FLAG** command is used to put a new value of 1 in flag 1 (used to indicate that the key has been found) - This prevents a "new" golden key being discovered every time the fountain is examined! The **PUT MOVABLE OBJECT** command then places the golden key in location 1, ready to be picked up by the player, and the **JUMP TO END** command is used to leave source bank 2 and jump to any "low priority" routines located in source bank 3.

Now insert a blank line at the current editing cursor position, and enter the following commands which deal with the park gates being examined :

```
If Unmovable Object = [2] DO
  {
  If Flag [2] = [0] DO
    {
    Print Message [18]
    Jump To End
    }
  Print Message [19+FLG(3)]
  Jump To End
  }
```

Followed by another final blank line.

In this routine, the commands within the first structure would ONLY be executed if the player had typed EXAMINE GATES. The commands within the second, "nested" structure will only be executed if flag 2 (used to indicate whether the gates are still locked) contains a value of zero. If so, then message number 18 will be printed. The following **JUMP TO END** command jumps directly to source bank 3, and prevents the second **PRINT MESSAGE** command being executed as well!

The second **PRINT MESSAGE** command in the above routine illustrates quite well just how useful **SAS**'s advanced expression interpreter can be. The command's numeric expression adds 18 to the contents of flag 3 (used to indicate whether the gates are open or not) by using the **FLG(n)** numeric function. Therefore, if the gates are closed (in which case, flag 3 will contain a value of zero), message number 19 will be printed, but if the gates are now open (in which case, flag 3 will contain a value of 1), then message number 20 will be printed instead. By using a numeric expression in this way, we have saved ourselves the need to use TWO separate **PRINT MESSAGE** commands - one for each different value of flag 3.

Next, we will insert two totally new routines which will deal with the verbs UNLOCK and OPEN. Select the editing options bar menu and jump to line number 59.

Insert the following command lines :

```
  If Verb = [12] GOTO UNLOCK

  If Verb = [13] GOTO OPEN
```

The **IF VERB** command is found in the "IF (1)" bar menu. Make sure that you have inserted a blank line after each **IF VERB** command in order to keep the listing nice and neat. These two lines check whether the player has mentioned the verbs UNLOCK or OPEN in his input. If so, then the program jumps to the as-yet undefined line labels "UNLOCK" or "OPEN" where our routines will be placed. Notice that there are similar **IF VERB** commands located in this section of the listing which jump to the various routines in this source bank dealing with each verb.

To insert the routine handling the UNLOCK verb, jump to command line 198 by selecting the option N Go To Line Number from the editing options bar menu. Now insert the following commands including the blank lines :

```
  Remark : *** UNLOCK command routine ***

UNLOCK:
  If Unmovable Object <> [2] GOTO CAN'T

  If Not Carried [1] DO
    {
    Print Lower Message [21]
    Jump To Input
    }

  If Flag [2] = [1] DO
    {
    Print Lower Message [22]
    Jump To Input
    }

  Print Message [23]
  Set Flag [2] To [1]
  Jump To End
```

The **LINE LABEL** is found from the "OPTIONS" bar menu (remember - the final colon character ":" shown in the listing is NOT part of the line label name), the **IF NOT CARRIED** command from the "IF (2)" bar menu, and the **PRINT LOWER MESSAGE** command is found from the "SYSTEM" bar menu.

Remember to again insert a blank line after the **JUMP TO END** command in order to keep the listing neat.

The above routine dealing with the UNLOCK verb actually consists of four separate parts :

It firstly makes sure that the player has ONLY tried to use the verb with unmovable object number 2 - the park gates (trying to UNLOCK the fountain would clearly be nonsense!). If the player did NOT mention unmovable object number 2, then a jump is made to the line label "CAN'T" which is followed by a routine which prints the error message You cannot do that !

Secondly, the **IF NOT CARRIED** command is used to check whether the player is NOT currently carrying the golden key which is needed to unlock the gates with. If he is indeed not carrying the key, then a **PRINT LOWER MESSAGE** command is used to print an "error" message in the lower input window, and a jump is made to immediately get the player's next input.

Thirdly, a check is made to see whether the gates have already been unlocked (you can't [illegible] unlock an already unlocked gate!). this is signified by flag 2 containing a value of 1. If this is the case, then another suitable "error" message is displayed and a direct jump is made to get the player's next input.

finally, providing all's well, message 23 is printed (confirming that the gates are now unlocked), flag 2 is set to contain the value 1 (used to indicate they are unlocked) and a jump is made to any "low priority" routines in source bank 3.

The editing cursor should now be pointing to the blank line entered after the **JUMP TO END** command - command line number 219. This is where the routine dealing with the OPEN verb will be located.

Type in the following commands, again including the blank lines :

```
  Remark : *** OPEN command routine ***

OPEN:
  If Flag [2] = [0] DO
    {
    Print Message [24]
    Jump To End
    }

  If Flag [3] = [1] DO
    {
    Print Lower Message [25]
    Jump To Input
    }

  Print Message [26]
  Set Flag [3] To [1]
  Jump To End
```

again, remembering to make the listing neat by including a final blank line after the **JUMP TO END** command.

This routine consists of three separate parts :

Firstly, flag 2 is inspected to see whether it contains a value of zero, indicating that the gates are still unlocked (obviously, you cannot open a locked gate!). If so, then message 24 is printed, and a jump is made to bank 3.

Secondly, flag 3 is inspected to see whether the gates are already open - indicated by flag 3 containing a value of 1. As it would be silly to attempt to open gates which are already open, a suitable "error" message is printed, and a direct jump is made to get the player's next input.

Finally, provided all's well, message 26 is printed indicating that the gates are now open, flag 3 is made to hold a value of 1 indicating that the gates are now open, and a jump is made to the "low priority" routines in source bank 3.

we have now entered all of the routines required in source bank 2. To leave source bank 2, select the option M  Return To Main Menu from the editing options bar menu. The changes you have just made in bank 2 will be stored, and you will return to the editor main menu.

---

Next, we will add the "low priority" routines required for our PARK adventure in source bank 3. Low priority routines are ONLY executed once the "high priotity" routines in bank 2 have been executed, and a **JUMP TO END** command has been encountered.

There is only one routine which needs to be placed in source bank 3 - One that detects whether the player has finally escaped from the park and is now in location number 3. Select the option 3 Source Bank ( 3 ) from the editor main menu. After a short while you will see the now familiar source bank editing screen.

As you can see, the START starter file does not presently contain any low priority routines at all! In fact, apart from the three **REMARKs**, the only command in this bank at the moment is the single **CONTINUE WITH INPUT** command. **CONTINUE WITH INPUT** works in a similar way to the familiar **JUMP TO INPUT** command, but before getting the player's next input, any remaining multiple commands in the player's previous input are dealt with first. It is this command placed at the end of source bank 3, that allows the player to enter many multiple commands separated by commas, full stops, or the words "THEN" or "AND". If a multiple commannd still exists, then it is processed by the interpreter in the normal way, and a jump is made to the first command line of bank 2.

Press the **CURSOR DOWN** key twice to move the editing cursor to command line number 3, and insert a blank line. Now immediately afterwards, insert the following commands :

```
If At Location [3] DO
   {
   Print Message [27]
   Pause
   Quit
   }
```

The **IF AT LOCATION** command is found on the "IF (1)" bar menu, and the **PAUSE** and **QUIT** commands from the "COMMANDS" bar menu.

What this routine does is to use the **IF AT LOCATION** command to check whether the player has escaped from the park and is presently in location number 3. If he is indeed in location 3, then an end-of-game congratulations message is printed, and the **PAUSE** command is used to wait for the player to press a single key before the **QUIT** command begins a completely new game by jumping to the first command line in source bank 1.

We have now finished with source bank 3, so return to the editor main menu in the normal way.

---

Select the option 4 Source Bank ( 4 ) to jump to the source editor for source bank 4.

All that now remains to be done in our PARK adventure is to insert two local conditions in source bank 4. The routines in source bank 4 are ONLY executed when the player immediately moves to another location. This can be either by the **ACT UPON DIRECTION** command, which checks the current location's exits table for any direction mentioned by the player in his last input - if a valid connection exists, then the player is moved to the relevant location, otherwise no action is taken, or the **MOVE TO** command which can be used to move a player directly to any specific location.

The most inportant command already present in this bank is the **DESCRIBE** command which is used to describe the "new" location as it is entered by the player.

The purpose of our first local condition will be to create a "barrier" preventing the player from simply walking through the park gates if they are closed or locked.

Leave the cursor at command line number 1, and insert a single blank line followed by the following commands :

```
If Flag [2] = [0] OR
If Flag [3] = [0] THEN
  If At Location [3] DO
    {
    Print Message [28]
    Move To [2]
    }
```

The **MOVE TO** command is found on the "LOCATIONS" bar menu.

Once a player has entered any new location, this routine firstly checks whether the park gates are still locked (signified by flag 2 containing a value of zero), or if they are still closed (signified by flag 3 containing a value of zero). If either is the case, then a check is made to see whether the player has just moved to location 3. Since the only way to location 3 is by travelling East from location 2, the player must therefore have "walked through" a locked or closed gate! If this is indeed the case, then a suitable message is printed telling the player that his way has been blocked by the gate, and he is moved back to location 2 - although in effect, the player is unaware that he was briefly in location 3 at all, as this routine is placed before the **DESCRIBE** command which would have described location 3 as he entered it!

Move the editing cursor to the **REMARK** at command line 15, and insert a blank line, followed by our second local routine which consists of the following commands :

```
If At Location [2] OR
If At Location [3] THEN
  Put Unmovable Object [2] At [LNO]
```

What this routine does, is to actually make the park gates (unmovable object number 2) "follow" the player as he moves through them. The **PUT UNMOVABLE OBJECT** command uses the numeric function **LNO** as its second parameter. **LNO** simply returns the number of the player's current location.

The idea of actually moving an UNMOVABLE object to a different location might seem a little strange, but not if you consider that the player might wish to go through the gates and EXAMINE them from the other side! Strictly speaking, this routine is not really nescessary for the PARK adventure, since the game ends as soon as the player has reached location 3, and he would simply not have time to examine the gates (or do anything else for that matter!), but this certainly would not be the case if the game was expanded later. The same procedure would apply to certain unmovable objects in your own adventures which the player would be permitted to travel "through" such as doors, windows, and possibly unmovable objects used as a mode of transport such as boats, cars, planes etc., which should ideally be present both on the INSIDE and OUTSIDE!

We have now completed the PARK adventure source, and all that remains to be done is to return to the main menu, and save the entire source onto your "PARK" data disk (the one which you were told to FORMAT and have ready at the start of this chapter). Select the option S  Save Adventure Source from the editor main menu. The compiler will now be able to convert the adventure source saved on this data disk into a complete adventure game which you will be able to either play straight away, or save as complete BOOTable game onto a new disk. How the compiler works is detailed in the next chapter...

---

Admittedly, the PARK adventure is a pretty pathetic game by anyone's standards (the well-seasoned adventure player would probably take about 30 seconds to complete it!), but it is only intended to serve as a guide to using and finding your way around the source editor.

Once you have successfully compiled and played the PARK adventure, why not try and improve or add to the game by for example, adding a few more puzzles and improving on the existing rather sparse location descriptions. As a challenge, try to implement the following :

Include LOCK and CLOSE verbs so that the gates can be closed and locked with the key as well. Remember, the verbs will have to be defined BOTH in the vocabulary and the verb definitions as well. Your routines will also have to make sure that the gates are closed before the player is allowed to lock them.

Try to include some more movable objects and locations. How about making sure that the golden key can only be reached by a magnet on a piece of string ? I'll leave it up to you to decide which verbs to use for this.

# SOURCE EDITOR CONTROLS SUMMARY

## Bar Menus

| | |
|---|---|
| CURSOR UP | Highlights the previous bar menu option. |
| CURSOR DOWN | Highlights the next bar menu option. |
| RETURN | Selects the currently highlighted bar menu option. |

Bar menu choices may also be selected by pressing the key corresponding to the first character displayed in the name of each bar menu option.

SAM Mouse owners may also highlight the option required by moving the mouse up and down, and selecting it by pressing the mouse SELECT button.

## Entry Fields

| | |
|---|---|
| CURSOR LEFT | Moves cursor one character to the left. |
| CURSOR RIGHT | Moves cursor one character to the right. |
| DELETE | Deletes the character to the left of the cursor position. |
| F2 | Inserts a space at the current cursor position. |
| RETURN | Accepts the data in the entry field. |

## Vocabulary Editor and Source Bank Editors

| | |
|---|---|
| CURSOR UP | Move up the listing by one line. |
| CURSOR DOWN | Move down the listing by one line. |
| F1 | Move up the listing by ten lines. |
| F0 | Move down the listing by ten lines |

SAM Mouse owners may bring any line to the cursor by moving the mouse pointer to the desired line, and pressing the mouse SELECT button.

| | |
|---|---|
| ESC | Select the "editing options" bar menu / remove a currently selected bar menu. |
| CURSOR LEFT | Select the next bar menu to the left. |
| CURSOR RIGHT | Select the next bar menu to the right. |

A bar menu may also be selected by pressing the letter higlighted in the bar menu titles at the top of the screen.

SAM Mouse owners may also select a bar menu by moving the mouse pointer to the title of the desired bar menu at the top of the screen, and pressing the mouse SELECT button.

# THE SOURCE COMPILER

The compiler program converts your adventure source code created in the source editor into a fully running machine code adventure which you can either play straight away, or save as an auto-running game onto a formatted disk.

The compiler itself is one of two main programs located on the "Utilities" disk which was prepared earlier in the chapter "Setting Up SAS".

To load the compiler, simply insert the "Utilities" disk into drive 1 and press the **F9** key to boot up the disk in the usual way. You will now see a simple menu screen prompting you to press the 1 key to load the compiler, or 2 to load the graphics extension program. The graphics extension program is used to add location graphics onto compiled adventures, and will be detailed later.

Press 1 to load the compiler, and after it has finished loading, you will be prompted to insert your adventure source data disk into drive 1. Note that unlike the source editor, the compiler ALWAYS requires you to insert your data disks into drive 1, even if your SAM has two disk drives fitted.

To compile our "PARK" adventure, insert your data disk containing the adventure source saved from the source editor, and press a key. The compiler will now read a directory of files from the disk, and display all adventure files with the filename extension of ".HDD" (the "header" file saved by the editor). On normal data disks, there will only be one ".HDD" file, as it is normally wise only to save one adventure source per disk, but there are two different adventure sources actually saved on the "Utilities" disk itself - the "START" starter file, and the "SPAMCO" source for the demonstration adventure which you played earlier.

Near the bottom of the screen, you will be prompted to type the name of the adventure source that you wish to compile. Type in the name PARK (no need to add the ".HDD" filename extension) and press **RETURN**. You will now be asked how many text columns per line your adventure will be using (you should have already decided this when defining the location descriptions and messages in the editor). Since the PARK adventure was designed to use 64 text columns, type 2 for choice number two (no need to press **RETURN**).

Next, you will be asked to specify how many characters in each word in the player's input will be recognised against the words defined in the adventure vocabulary. With **SAS**, you may specify any number of characters between 4 and 15 (in which case, nearly all words in the game would have to be spelled out in full !). The number of recognised characters is entirely up to you. This can be most useful in allowing the player to abbreviate most of the words in his input, although if you used 4 characters, an input such as

BREAK WINDOW

might not nescessarily be recognised as what the player meant by the interpreter - a case of flatulence perhaps ? !!!!

Normally, 5 characters will be sufficient for most adventures (in fact the compiler displays a value of 5 in the entry field as a default value), but you should realise that this could still cause confusion when the player types in the full name of one of the four diagonal compass directions. For example, the directions

NORTHEAST

and

NORTH

both have the same first 5 characters in their name. In this case, the interpreter would assume that the player meant whatever word appeared first in the vocabulary list. Of course, this problem would have been avoided if the player had typed NE instead!

Press **RETURN** to use the default value of 5 characters for vocabulary recognition.

You will now be asked whether you want your compiled adventure to use text compression. Adventures using text compression will result in the compiled adventure being a smaller size, with therefore more memory being available for location graphics (should you decide to have any), with the amount of space used for text in the adventure being reduced by around 30 to 60 percent (depending of course upon what words you have used, and your own personal literary style).

However, text compression can add several minutes to the amount of time taken by the compiler to compile your adventure source, especially if your adventure uses many messages or locations with lengthy descriptions. It is therefore a good idea NOT to use text compression while compiling your adventures for play-testing purposes, using it only when you are reasonably sure that your adventure is "bug-free" and you are ready to save your finished game to disk for others to play.

For the "PARK" adventure, type **N** (no need to press **RETURN**) for no text compression. The compiler will now load each separate section of the adventure source from the data disk and compile it in turn.

After about 20 seconds (**SAS**'s compiler works fairly quickly), the "PARK" adventure will have been compiled, and you will see a message on the screen along the lines of

```
Compilation OK.
===============
Compilation Length : 3349 Bytes - Suitable For 256k / 512k SAMs.
Bytes Free For Graphics : 144107 (256k) / 406251 (512k).
```

(Don't worry if the numbers shown on the screen are not exactly the same as the ones shown above)

Sometimes however, the compiler will be forced to stop compiling your adventure source, emit a loud BEEP and display an error message on the screen such as

```
Line Label Not Defined
```

This means that the compiler has detected an error you have made while writing your adventure source in the editor (the exact error message displayed will of course, depend on the type of error that was detected). The error will have been detected in whatever part of your adventure source was being compiled at the time the error occured, and the compiler will tell you which command line number has caused the trouble along with a copy of the command line itself in most cases.

To rectify the problem, you should first consult the appendix in this manual "Compiler Error Messages" which lists all possible compiler errors along with their likely causes, and return to the source editor in order to amend the offending commands/items before attempting to compile your adventure source again.

Provided your adventure has compiled with no problems, you will now be able to save your adventure to disk as an auto-running file or play your compiled game straight away.

Press **P** to play your adventure. There will be no way of returning back to the compiler while playing the game.

Press **S** to save your adventure as an auto-running game onto disk. Before selecting this option, you should first make sure that you have handy a formatted disk with a DOS saved as the first file. If you intend to sell your finished games, or pass them on to your friends, then it is probably best to use SAMDOS on the disk, as the inclusion of MASTERDOS (or MASTERBASIC for that matter) will infringe copyright.

Note that you MUST save your compiled adventure to disk if you intend to add location graphics to your adventure by using the graphics extension program on the "Utilities" disk. How this is done is detailed in the next chapter...

# THE GRAPHICS EXTENSION PROGRAM

This chapter will deal with how high quality location graphics can be added to your compiled adventures.

The Graphics Extension program on your "Utilities" disk is used to add graphic screens to your final compiled adventure disk. As each screen is added, it is automatically compresseed and combined with your adventure code.

## Designing Your Pictures

The graphics extension program can add location graphics either by using screens which have been designed with the art utility program FLASH, or normal SCREEN$ files. If SCREEN$ files are used, then any PALETTE changes in the picture using line interrupts will be ignored. If your adventure is using either 64 or 85 text column mode, then the picture will have to have been designed/saved in MODE 3. If your adventure uses 42 or 32 text column mode instead, then your pictures will have to have been designed/saved in MODE 4.

There are a few limitations to bear in mind however, when designing your pictures :

1) Only the first 128 rows of pixels in your picture will actually be saved and included in your adventure. This is roughly equal to the first two thirds of the screen, and represents 16 rows of text as printed by the interpreter. Therefore, you must make sure that your picture is concentrated towards the top of the screen. Remember, it is always possible to load your screen into FLASH, scroll it upwards within the program and re-save it.

2) Ideally, your picture should not use PALETTE postions 0 and 1. **SAS** uses PALETTE position 0 as its PAPER colour and PALETTE position 1 as its PEN colour (Normally these are assigned default colour values of 0 - "pitch black" and 120 - "turnip" respectively). **SAS** will NOT allow the colours in these two PALETTE positions be overwritten by the colours in your pictures which have been assigned to these PALETTES. Therefore, if your screen picture DOES use colours in these two PALETTE positions, you should make sure that they contain colours with match the same values that **SAS** is itself using, otherwise your picture will seem to contain two strange colours which were probably not intended when it was designed. It is possible to change the values in PALETTE positions 0 and 1 changing **SAS**'s normal PAPER and PEN colours. This is detailed in the chapter "Customising **SAS**".

3) The above limitation is especially important if your adventure is using 64 or 85 text columns, as then the picture will be represented in MODE 3 which only has 4 PALETTE positions available - and two of these (PALETTE positions 0 and 1) will be used by **SAS** itself! The remaining 12 PALETTE positions normally used in MODE 4 pictures will give various mixes of the colours contained in PALETTE positions 0 to 3. It is therefore recommended that the graphics included in 64 or 85 text column adventures should be designed in a more-or-less monochrome style, and that they are designed in the MODE 3 setting of the art program FLASH.

4) Your pictures should not be designed to use "flashing" colours (This is normally achieved by a PALETTE position in each of the two consecutive PALETTE tables holding a different colour value). Because **SAS** runs most of the time with the ROM interrupts disabled, such pictures will only appear to "flash" very intermittently (if at all!)

Once all of your pictures have been designed, you should make sure that they are all saved on a new single disk, as the graphics extension program will require this.

You can define at which point in your adventure a graphic is shown by using the **SHOW GRAPHIC** command. **SHOW GRAPHIC** is used with a numeric parameter which specifies the number of the graphic to be shown. **SAS** allows up to 255 different graphics present in the adventure, but in practice, you will probably find that there is not enough memory available for that many!

Graphic number 1 refers to the first graphic to be added to the compiled adventure by using the graphics extension program, graphic number 2, the second and so on.

To use the **SHOW GRAPHIC** command to display a location graphic for a specific location, we would write a routine in source bank 4 such as :

```
If At Location [10] THEN
  Show Graphic [1]

Describe
```

This example would show graphic number 1 every time the player moved to location number 10. Note that we have displayed the location picture BEFORE the location was described using the **DESCRIBE** command. This is because before **SHOW GRAPHIC** automatically scrolls any existing text out of the upper text window before displaying the picture, and if we had placed the **SHOW GRAPHIC** command AFTER the **DESCRIBE**, the location desription text would have scrolled off of the screen before the player had had a chance to read it!

Of course, there is no reason why **SHOW GRAPHIC** should be limited to just displaying location pictures. You could just as easily use **SHOW GRAPHIC** to display a picture in response to an object being examined, or an object such as a treasure map being read. You could even use **SHOW GRAPHIC** in source bank 1 to display a welcoming picture or logo every time a new game is started.

If **SHOW GRAPHIC** commands are present in a compiled adventure which has not yet had graphics added by the graphic extension program, then the **SHOW GRAPHIC** command is completely ignored by the interpreter. This is handy for play-testing adventures without having to go to the bother of adding graphics after compilation.

## Using The Graphics Extension Program

To illustrate how the graphic extension program works, we will add some graphics to the "SPAMCO" adventure source which is supplied on the "Utilities" disk. This is actually the source for the DEMO adventure which you will have played in the chapter "PLAYING THE DEMO ADVENTURE".

Firstly, prepare a blank formatted disk, and save a DOS file as the first file on the disk. This disk will be used for the compiled SPAMCO adventure, and will eventually contain a complete auto-running adventure game complete with graphics.

Load up the "Utilities" disk by pressing **F9**, and press **1** to load the compiler. Now load in the "SPAMCO" adventure source for compilation (this is also on the "Utilities" disk, so keep the disk inserted when prompted to insert your source data disk).

Compile the SPAMCO adventure source, using 42 text column mode, 6 character vocabulary recognition and no text compression. The source will take a little while longer to compile than the PARK adventure did, but then it's considerably larger!

After a short while, you will see a message indicating a successful compilation. At this point, press the **P** key to save an auto-running adventure onto the blank disk. Note that the graphics extension program requires the compiled adventure to be saved in this way before any graphics can be added.

At this point, you may play the SPAMCO adventure if you wish by loading up the disk, but of course, there will be no graphics included in the game.

Reset the computer and BOOT up the "Utilities" disk. This time, press 2 to load the grraphics extension program. Once the program has finished loading, you will be prompted to insert your compiled adventure disk. Insert the disk with the new auto-running SPAMCO game and press a key.

You will now be prompted to insert your SCREEN$ disk. This refers to a disk containing all of the pictures (saved as either FLASH screens or SCREEN$ files) that you wish to include in the adventure. In this case, all of the screen pictures to be added are to be found on the "Utilities" disk, so insert this disk again and press a key.

A directory of all available files on the disk (whether they are screen pictures or not) will be displayed, followed by some information indicating how much memory is still available for graphics. This information will look something like

```
124240 Bytes free for graphics ( 256k )
386384 Bytes free for graphics ( 512k )
```

(Don't worry if the numbers shown on the screen are not exactly the same as the ones shown above)

The numbers indicating how much free memory is available, will steadily decrease as each new picture is added. Exactly how much memory each picture will use up will depend upon how well it compresses. As a general rule, pictures with large areas left blank or areas filled with solid patterns or colours will compress better than more cluttered pictures, but almost all pictures will compress to some extent.

At the bottom of the screen, you will be prompted to enter the filename of the first picture to be included. Type in the filename FR and press **RETURN**. A picture of the famous "DREAD" Magazine publisher Colin McDoughnut will be loaded from disk. As the picture is loading, the colours may seem a little strange - don't worry about this. After a very short pause, the picture will have been compressed, and it will then be re-displayed in its proper colours.

The order in which you load in the various pictures is VERY IMPORTANT, as the FIRST picture you load in will be recognised as graphic number 1 by the interpreter, the SECOND as graphic number 2 and so on.

If you have made a mistake in the order you have loaded one of the pictures, don't worry. You can delete the last loaded picture. Do this now by typing D followed by **RETURN**. You will notice that as soon as you have done this, the amount of available memory immediately increases as the the memory taken up by the last loaded picture is reclaimed.

Now load up the seven pictures required by the SPAMCO adventure in the following order :

```
MI CB GR FR CH AL HA
```

Once you have loaded in the last graphic picture, type F followed by **RETURN** to finish. You will now be prompted to insert your compiled adventure disk again. Do this and press a key. After a short while, your previously saved compiled adventure will be over-written by a new version containing all of the graphics you have just added. Load it up and see!

---

If you intend to sell your adventures commercially, it is probably a good idea to make sure that your adventure and graphics do not use up enough memory to prevent them being run on a 256k SAM - a lot of people still have the 256k machines!

Of course, there is nothing to stop you from including TWO different versions of your game on the same disk. The 512k version could include extra features or graphics to the second 256k version.

# TESTING YOUR ADVENTURE

Playtesting is very important when writing adventures. All too often, adventures are released which contain "bugs", errors in logic or spelling mistakes. Such adventures when released will look unprofessional and are not likely to be very well reviewed (if at all).

When playtesting a game, it is very tempting to just play through following the correct solution. Try to play the game in the same way as someone would who was approaching the game for the very first time. Ideally, try to use a couple of friends to try the game out - if you can actually watch them playing, then so much the better. You'll be supriesd at the number of good suggestions and comments that they will come up with - valid things that you would never have noticed yourself.

Above all, always have a dictionary handy when writing and correcting your adventure text. Follow the simple rule **IF IN DOUBT ABOUT A WORD - LOOK IT UP!** Nothing is more effective in helping potential publishers reject your game than bad spelling or English grammar. Again, a couple of friends will prove most useful in spotting spelling mistakes that you would have otherwise missed.

Sometimes while playing an adventure, the interpreter will clear the screen, emit a loud BEEP and display an error message such as

```
Return Without Previous Gosub
```

This means that the interpreter has detected an error while executing a command in the compiled adventure (the exact error message displayed will of course, depend upon the type of error that was detected) The interpreter will display after the error message the line number of the command which generated the error, along with its source bank number.

To rectify the problem, you should first consult the appendix in this manual "Interpreter error Messages" which lists all possible interpreter errors along with their likely causes, and then return to the source editor in order to amend the offending command before attempting to compile your adventure source and subsequently play your adventure again.

Once an interpreter error message has been displayed, it will be possible in most cases to continue playing the adventure by pressing any key to return to the game. In some instances however, continuing play after an interpreter error will have unforseen consequences for the remander of the adventure, and may even result in a system "crash" or reset.

It is possible to make adventures suppress the printing of interpreter error messages altogether, and even detect whether such an error has occured. See the appendix on system flags for more details on this.

It goes without saying, that any known interpreter errors should be fully corrected and tested before any adventure is commercially published!

# MORE ABOUT MESSAGES

Earlier chapters have dealt with how the **PRINT MESSAGE** command can be used to print a message on the screen during an adventure, and **PRINT LOWER MESSAGE** which can be used to print "error" messages near the bottom of the screen if the player attempts anything foolish in his input. **SAS** normally imposes a maximum message length of 256 characters per message. In mosst cases, this is more than adequate, but there will be times when a longer message length is required

**SAS** allows you to "chain" any number of messages together by using the **PRINT MESSAGE SUPPRESSED** command.

Normally, whenever a message is printed, after the last row of text in the message has printed, the screen is scrolled an extra time. This makes sure that there is a blank row of text on-screen before the next message to be printed.

For example, if message number 1 contained the text "hello", then the commands

```
Print Message [1]
Print Message [1]
```

would result in the text

```
hello

hello
```

being printed on the screen.

**PRINT MESSAGE SUPPRESSED** works in a similar way to the **PRINT MESSAGE** command, except that the screen is NOT scrolled an extra time after the last row of text in the message has been printed. This means that commands such as

```
Print Message Suppressed [1]
Print Message [1]
```

would result in the text

```
hello
hello
```

being printed as if it was a single "paragraph".

**PRINT MESSAGE SUPPRESSED** can also be used to give enlarged location descriptions when used in conjunction with the **DESCRIBE** command.

Example

```
Print Message Suppressed [20]
Describe
```

In this example, messsage 20 would contain the first half of the location description text, and the normal location description definition would contain the second half of the location description text.

In addition to the 1024 messages that can be defined within **SAS**, it is also possible constantly re-define a special message called message 0. Unlike normal messages which are restricted to a maximum length of 256 characters, message 0 may have a maximum length of 1024 characters.

The text that message 0 contains is defined by the **SET MESSAGE ZERO** command. This command has a single parameter which is used to hold a string expression which defines the text which message 0 will contain.

Most parameters required by the majority of the other commands in the **SAS** programming language are numeric parameters. That is to say, they must contain either a number or a numeric expression. The parameter required by the **SET MESSAGE ZERO** command however, is different. It must contain a STRING expression made up of at least one of the following string functions which are used in a similar way to the familiar numeric functions.

**MSG(n)** This function returns the text contained in message number **n**. **n** needn't just be a number. It may be any valid numeric expression.

As an example, if message number 1 contained the text "hello", then the command

```
  Set Message Zero To [MSG(1)+MSG(1)]
```

would result in message 0 containing the text "hellohello".

Message 0 can be printed out in the same way as the other defined messages.

Example

```
  Print Message [0]
  Print Message Suppressed [0]
  Print Lower Message [0]
```

**SPC(n)** This string function can be used to insert **n** number of spaces into message 0.

Example

```
  Set Message Zero To [MSG(1)+SPC(1)+MSG(1)]
```

this would result in message 0 containing the text "hello hello".

**D@(n)** This function inserts the name of direction number **n** into message 0. The name of direction **n** must have been defined in the Direction Definitions section of the source editor, otherwise an interpreter error will be generated.

**V@(n)** This function inserts the name of verb number **n** into message 0. The name of verb **n** must have been defined in the Verb Definitions section of the source editor, otherwise an interpreter error will be generated.

**M@(n)** This function inserts the name of movable object number **n** into message 0. The name of movable object **n** must have been defined in the Movable Object Definitions section of the source editor, otherwise an interpreter error will be generated. Whether the name of the movable object includes its prefix depends upon the value held in system flag 17. See the appendix on system flags for more details on this.

**U@(n)** This function inserts the name of unmovable object number **n** into message 0. The name of unmovable object **n** must have been defined in the Unmovable Object Definitions section of the source editor, otherwise an interpreter error will be generated.

**STR@(n)** This function inserts a single space into message 0, followed by a string representation of the result of the numeric expression **n**.

As an example of how this function may be used, imagine a player has a number of coins (the exact number of coins being held in flag number 10) and wishes to count them. If Message 1 contains the text "You currently have" and message 2 contains the text " coins.". By using the command

```
  Set Message Zero To [MSG(1)+STR$(FLG(10))+MSG(2)]
```

the complete text detailing how many coins the player currently had could built up in message 0.

If for example, the player was currently carrying 200 coins, then after the above comand was executed, message 0 would contain the text

```
You currently have 200 coins.
```

As you can see, the flexibility of **SAS**'s expression handling can allow very complex messages to be built up

Message 0 can even amend its own text! If for example, message 0 already contained the text "hello", then th command

```
  Set Message Zero To [MSG(0)+MSG(0)+MSG(0)]
```

would result in message 0 containing the text

```
hellohellohello
```

## USING MEMORY EFFICIENTLY

When you first load up the **SAS** source editor, you immediately have 727040 bytes of memory available for your adventure source (794624 bytes if you have an external 1Mb memory interface connected). If you develop your adventures from the START starter file, then this figure is further reduced by 21106 bytes. The amount of free memory available for your adventure source might seem limitless at first, but you should bear in mind the following :

Every command line entered in a source bank will reduce the amount of free memory by exactly 64 bytes, regardless of how long the command line appears to be on the screen. This applies even to blank lines, **REMARKs** and line labels (whether they are referenced by your routines or not). Therefore, if free space starts to get a bit tight, you can dramatically increase your usable memory by deleting all blank lines, **REMARKs** (except the very first **REMARK** in each source bank which cannot be deleted and is used by the interpreter for reference purposes) and unused line labels.

Every message defined uses up 256 bytes of memory regardless of how many characters the message actually contains.

Every location defined uses up 306 bytes of memory regardless of how many characters the location description text contains and how many exits have been defined in its exits table.

Every line in the vocabulary uses up 16 bytes of memory. You can free memory by deleting any **REMARKs** present in the vocabulary listing (except for the first one which cannot be deleted, and is used by the interpreter for reference purposes).

Every definition of verb, direction, movable object and unmovable object names uses up 16 bytes of memory. All definitions of verb and unmovable object names can be safely deleted (thus freeing 16 bytes of memory for each one that was defined) **PROVIDED** that your adventure does NOT make use of the **V@(n)** and **U@(n)** string functions in the **SET MESSAGE ZERO** command. Direction and movable object names CANNOT be deleted in this way, because certain source commands (such as **DESCRIBE**) need to access the information stored there.

When defining messages, locations and names, care should be taken that the item just defined is CONSECUTIVE to the last defined item (ie. - you have not have left any blank definitions between your last defined item and the one just entered), otherwise memory space will be assigned to the intervening blank definitions - despite the fact that they will probably remain unused by your adventure! When such an item is deleted (by filling its definition with blank spaces), its memory space will be reclaimed including any blank definitions immediately below it in sequence, PROVIDED that there are no existing item definitions above it in sequence.

Very often certain routines can be re-written to use up less memory. As an example, consider the routine

```
If Verb = [8] DO
  {
  If Movable Object = [2] DO
    {
    Print Message [18]
    }
  }
```

which could be re-written as

```
If Verb = [8] AND
If Movable Object = [2] THEN
  Print Message [18]
```

which saves 256 bytes of memory space - enough for an extra message definition or 16 extra words in the vocabulary.

Here are some more examples :

```
  If Flag [10] = [5] DO
    {
    Print Message [13]
    Jump To End
    }
```

could be re-written as :

```
  If Flag [10] = [5] GOTO F10.5
```

with the following lines elsewhere in the same source bank

```
F10.5:
  Print Message [13]
  Jump To End
```

(64 bytes saved)

Careful use of IF.. commands and numeric functions can also bring rewards :

```
  If Movable Object = [2] THEN
    Drop Movable Object [2]
  If Movable Object = [3] THEN
    Drop Movable Object [3]
  If Movable Object = [4] THEN
    Drop Movable Object [4]
```

could be replaced by

```
  If Movable Object > [1] AND
  If Movable Object < [5] THEN
    Drop Movable Object [MNO]
```

(in this case, 192 bytes saved)

# CUSTOMISING SAS

This chapter gives details on how you can change some of the features in **SAS** to suit your own personal preferences.

## Using BASIC Subroutines Within SAS

As well as allowing you to write routines using its own programming language, **SAS** also lets you use normal SAM BASIC as well!

You can execute any BASIC command by using the **EXECUTE BASIC COMMAND** command.

Example

```
Execute BASIC Command : PRINT AT 0,0;"Hello from BASIC"
```

The parameter for **EXECUTE BASIC COMMAND** may contain multiple BASIC commands separated as normal by colons

Example

```
Execute BASIC Command : FOR q=1 TO 10:PRINT q:NEXT q
```

It is up to you to make sure that what you have defined as a parameter is valid BASIC, as no BASIC syntax checking is done by the compiler except to detect a leading BASIC line number which is not allowed.

It should be possible to execute most MASTERDOS and MASTERBASIC commands as well, provided that the relevant DOS/Extended BASIC code is present in memory (possibly loaded by altering the compiled adventure's "Auto" BASIC loader). You should bear in mind however, that any MASTERDOS or MASTERBASIC routines used in your own published games will infringe copyright if you include a copy of MASTERDOS or MASTERBASIC BASIC extension code on your disk as well.

Control will be passed back to **SAS** if a BASIC error occurs (even the STOP command counts as an error in this case) or a BASIC RETURN command is encountered without a previous BASIC GOSUB command. In fact **SAS** uses a BASIC RETURN command itself to return back to the interpreter after your BASIC command(s) have been executed.

It is possible to execute larger BASIC routines located elsewhere in the BASIC "Auto" loader, by using **EXECUTE BASIC COMMAND** to perform a BASIC GOTO, GOSUB or call a PROCEDURE. Your BASIC lines should not use line numbers in the range 1 to 15, as this area in the BASIC program is reserved for use by **SAS** itself.

There is about 8k available in the BASIC program area for more BASIC program lines or BASIC variables. The **SAS** interpretrer code starts at memory address 32768, so it is a good idea to type in CLEAR 32767 as a direct command before typing in extra BASIC program lines. This will prevent you from accidentally over-writing the **SAS** code area. It is usually best to insert your extra BASIC program lines after you have compiled your **SAS** source file for the last time (and are reasonably sure that your adventure is error-free), as each time you compile an adventure source and save the compiled adventure to disk, a completely new BASIC "Auto" file is created.

The following example illustrates how a large BASIC subroutine could be called from within **SAS**.

The routine is called by the command

```
Execute BASIC Command : basproc
```

("basproc" will be the name of a BASIC PROCEDURE located at BASIC line number 100)

The actual BASIC subroutine located at line 100 of the "Auto" loader would look something like

```
100 DEFPROC basproc
110 REM various BASIC commands here
120 REM more basic commands
130 END PROC
```

## Calling Machine Code Subroutines

Machine code subroutines can be called very easily by using a **EXECUTE BASIC COMMAND** command to provide a normal SAM BASIC CALL or USR command.

Example

```
Execute BASIC Command : CALL 16384
```

Your machine code may use any registers, but if interrupts are used, they will not work for most of the time when **SAS** is re-entered, as the interpreter very often pages out SAM's ROM and runs with interrupts disabled.

Care must be taken as to where to place your actual machine code routines. Ideally, they could be placed at the first free 16k RAM page below the DOS (normally RAM page 12 starting at address 212992 on a 256k machine, or RAM page 28 starting at address 475136 on a 512k SAM). You must of course, ensure that there are at least 16384 remaining bytes (16k) free after your adventure has been compiled and had any graphics added by the graphics extension program.

Alternatively, your machine code could be placed at the start of RAM page 0 (address 16384) in the system heap area. There is about 2.7k available for your code here, though "officially", before loading your code into this area, you should reserve some space by calling the ROM routine "JHEAPROOM" at address &0106. This will prevent your machine code from being over-written by the BASIC stack (used by the GOSUB and DO BASIC commands and PROCEDURES). If your machine code is quite small (say less than 1k), you will probably get away without having to call JHEAPROOM.

JHEAPROOM should be called with BC containing the number of bytes to be reserved. If the call is successful, then the carry flag is set on exit, and DE points to the old HEAP END (the start of the space just reserved) and HL points points to the new HEAP END (one byte past the end of your reserved space). If there was not enough room in the system heap to reserve the number of bytes in BC, then the carry flag is reset and HL holds the number of bytes that your request exceeded the available space by.

## Using Colour

Normally, the **SAS** interpreter prints all text on the screen using the rather boring colours of white PEN on black PAPER. PALETTE position 0 is used to hold the PAPER colour (normally colour value 0 - "pitch black") and PALETTE position 1 to hold the PEN colour (normally colour value 120 - "turnip"). however, it is possible to instantly change the default on-screen colours by using the **EXECUTE BASIC COMMAND** command to directly alter PALETTE positions 0 and 1.

Example

```
Execute BASIC Command : PALETTE 0,16:PALETTE 1,96
```

This example would instantly change the on-screen colours, and would result in **SAS** printing all subsequent text in a rather nice shade of yellow on a blue background.

Probably the best place to put such a command would be as the first command line in source bank 1.

It is also possible to define the colours that messages will be printed out in, by using the **EXECUTE BASIC COMMAND** command to provide a normal SAM BASIC PEN or PAPER command.

Example

```
  Execute BASIC Command : PEN 5:PAPER 10
  Print Message [80]
```

This example would result in message number 80 (and any subsequent messages) being printed in the PEN colour held in PALETTE position 5 and the PAPER colour held in PALETTE position 10.

The PEN and PAPER colours defined in this way will remain in use until either they are changed again by using another **EXECUTE BASIC COMMAND** command, or until a **JUMP TO INPUT** or **CONTINUE WITH INPUT** command is executed, which will revert back to PAPER 0 and PEN 1.

## Using Foreign Characters And User Defined Graphics

As well as allowing you to use normal letters in messages and location descriptions, **SAS** also allows you to use foreign characters (useful if you are writing software for use abroad) and/or user defined graphics (known as UDGs) as well.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 128 | Ç | SYM + 0 | 143 | Å | CTRL + 0 | 158 | ₧ | SYM + B |
| 129 | ü | SYM + 1 | 144 | É | CTRL + E | 159 | ƒ | CTRL + F |
| 130 | é | SYM + 2 | 145 | æ | CTRL + S | 160 | á | CTRL + Z |
| 131 | â | SYM + 3 | 146 | Æ | CTRL + A | 161 | í | CTRL + K |
| 132 | ä | SYM + 4 | 147 | ô | CTRL + O | 162 | ó | CTRL + = |
| 133 | à | SYM + 5 | 148 | ö | SYM + O | 163 | ú | CTRL + H |
| 134 | å | SYM + 6 | 149 | ò | CTRL + P | 164 | ñ | SYM + N |
| 135 | ç | SYM + 7 | 150 | û | CTRL + U | 165 | Ñ | SYM + I |
| 136 | ê | CTRL + 7 | 151 | ù | SYM + V | 166 | ª | CTRL + ; |
| 137 | ë | CTRL + 6 | 152 | ÿ | CTRL + Y | 167 | º | CTRL + : |
| 138 | è | CTRL + 5 | 153 | Ö | CTRL + L | 168 | ¿ | SYM + C |
| 139 | ï | CTRL + 4 | 154 | Ü | CTRL + J | | | |
| 140 | î | CTRL + 3 | 155 | ¢ | CTRL + C | | | |
| 141 | ì | CTRL + 2 | 156 | £ | SYM + D | | | |
| 142 | Ä | CTRL + 1 | 157 | ¥ | SYM + Y | | | |

Fig 15. - The foreign character set.

Figure 15. shows the entire foreign character set that is available from the **SAS** editor, along with their character codes and the key combinations which are required to produce the foreign charcter when entering text in a data entry field.

Altogether, there are 41 different foreign characters available with character codes ranging from 128 to 168 (the normal printable character set ranges from a space - charcter 32 to the copyright symbol - character 127).

If you wish to use foreign characters in your adventures, then your compiled adventure disk must also contain the file "fontEDITOR" which is found on the "Utilities" disk. It is this code file that contains the actual definitions for each foreign character.

You will also have to modify your compiled aventure's BASIC "Auto" loader program so that it loads up the foreign character font before running the adventure. The best way to do this, is to type

MERGE "filename"

Where "filename" represents the "Auto" file's true filename. This will load in the compiled adventure's "Auto" loader and suppress it from auto running.

Now insert the following BASIC line

```
5 LOAD "fontEDITOR" CODE UDG CHR$ 128
```

And re-save the modified program back onto the disk by typing

```
SAVE "filename" LINE 1
```

If you do not wish to use foreign characters in your adventure, you may use some or all of the space which they occupy to hold UDGs instead. To design your UDGs, you can use the font editor in the FLASH art program, or one of the many UDG designing programs which are widely available (one of which is to be found on SAMCO's SAMDOS disk) If your adventure is to use either 85 or 42 text columns, then there will be some limitations in designing your characters - See the next section "Using Different Text Fonts" for more details.

Each foreign character/UDG definition uses up 8 bytes and are stored consecutively at address &5490 upwards.

Care should be taken to ensure that the length of the code being loaded is not greater then the number of UDGs/foreign characters multiplied by eight. This prevents accidental over-writing of other areas of computer memory such as the PALETTE table. Fonts saved by the FLASH art program will be no less than 1024 bytes long (no matter how many charcters have been defined) and may well need to be shortened accordingly before loading into your compiled adventure.

If for example, you wanted to load only two UDGs as charcters 160 and 161, you could use BASIC commands along the lines of

```
LOAD "UDGcode" CODE UDG CHR$ 160
```

to load the code into the correct place in memory. In this case, the actual code loaded should be 16 bytes long (8 bytes for each UDG).

If you were loading an entire block of 41 UDGs, the destination address would be UDG CHR$ 128 and the code length would be 328 bytes (8x41 = 328).

While entering UDGs into text in the source editor however, you should remember that they will still be represented on the screen as foreign characters, so make a note of the character numbers of the UDGs you are using.

One very important note to remember is that if you are using either foreign characters or UDGs in your text, you should NOT use text compression while compiling your adventure. This is because the compiler uses characters numbered 128 upwards as text expansion tokens, and if you try to use text compression, you will end up with a strange mess of unintended text where your foreign characters/UDGs should have been!

## Using Different Text Fonts

It is possible to totally re-define the design of the 96 normal text characters used in an adventure. This can add greatly to the general atmosphere of the game. You could for example, use a Western style text font if your adventure was set in the Wild West, or a space age font if your adventure was set on another planet.

Again, you can use the font designer in FLASH or a normal UDG designer program to design your text font. As with UDGs, each character definition takes up 8 bytes, and if a full character set is designed (from the space to the copyright symbol), the font code would be 768 bytes long (8x96 = 768) and would be loaded by using a BASIC line in the compiled adventure's "Auto" loader such as

```
5 LOAD "font" CODE UDG CHR$ 32
```

As when designing UDGs, each character is defined as 64 pixels stored in 8x8 grid such as the one shown in Figure 16. which illustrates a typical pattern used to represent the letter "A".

However, if your adventure is using either 85 or 42 text column mode, then any pixels defined in the shaded area shown in Figure 16. will not be shown on the screen at alll (this is price we have to pay for being able to fit so many charcters onto the screen at once). Therefore, care is needed when defining characters for use in these text modes (the standard SAM character set is suitable for use in any text column mode). In addition, care should be taken so that there is at least one blank column of pixels to one side of each defined character. This "margin" prevents printed characters becoming cluttered together making the text unreadable.

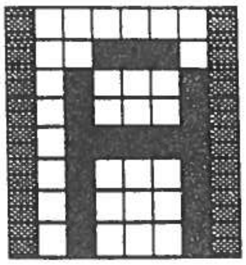


Fig. 16.

## Useful Addresses To POKE

This section lists various fixed address within the interpreter machine code which can be altered by a BASIC POKE command in the compiled adventure's "Auto" file before the adventure is run, in order to give useful results.

| ADDRESS | NAME | NO. OF BYTES | COMMENTS |
|---|---|---|---|
| &9000 | STRIP.CHR1 | (1) | Character used for window dividing strip at even character positions. Initially a dash "-" character. |
| &9001 | STRIP.CHR2 | (1) | Second character used for window dividing strip at odd character positions. Initially a dash "-" character. If 85 text column mode is being used, then STRIP.CHR1 and STRIP.CHR2 should contain the same value. |
| &9002 | CURSOR.CHR | (1) | Character used as the cursor. Initially an underline character "_". |
| &9003 | PROMPT.CHR | (1) | Character used as an input prompt. Initially a greater-than character ">". |
| &9004 | MORE.TXT | (8) | Text displayed when the upper window fills up with text. Initially the text " More ↑ ". |
| &900C | PRMPT.INV | (1) | If non-zero, the input prompt is printed in inverse. Initial value is 1. |
| &900D | BEEP.FLG | (1) | If non-zero, a BEEP is made every time a player types a character in his input. Initial value is 1. |
| &900E | INP.CAPS.FLG | (1) | If non-zero, all characters typed-in appear in upper-case. Initial value is 0. |
| &900F | LMESS.INV | (1) | If non-zero, text printed by the **PRINT LOWER MESSAGE** command is in inverse. Initial value is 1. |
| &9010 | LMESS.SPC | (1) | If non-zero, text printed by the **PRINT LOWER MESSAGE** command is padded out by a single space each side. Initial value is 1. |
| &9011 | LMESS.CAPS | (1) | If non-zero, all text printed by a **PRINT LOWER MESSAGE** command is printed in upper-case. Initial value is 1. |

| ADDRESS | NAME | NO. OF BYTES | COMMENTS |
|---|---|---|---|
| &9012 | A.MSG.LEN | (1) | Length of text used for the "a " movable object prefix. Initial value is 2. |
| &9013 | A.MSG | (10) | Text used for the "a " movable object prefix. Initially the text "a ". |
| &901D | AN.MSG.LEN | (1) | Length of text used for the "an " movable object prefix. Initial value is 3. |
| &901E | AN.MSG | (10) | Text used for the "an " movable object prefix. Initially the text "an ". |
| &9028 | SOME.MSG.LEN | (1) | Length of text used for the "some " movable object prefix. Initial value is 5. |
| &9029 | SOME.MSG | (10) | Text used for the "some " movable object prefix. Initially the text "some ". |
| &9033 | THE.MSG.LEN | (1) | Length of text used for the "the " movable object prefix. Initial value is 4. |
| &9034 | THE.MSG | (10) | Text used for the "the " movable object prefix. Initially the text "the ". |
| | | | (By altering the above, the user can effectively re-define his own movable object prefixes) |
| &903E | AND.TXT.LEN | (1) | Length of text used for "and " when listing movable objects. |
| &903F | AND.TXT | (10) | Text used for "and " when listing movable objects. |
| &9049 | BANK.NO | (1) | Source bank number of the current SAS command. |
| &904A | LINE.NO | (2) | Command line number of the current SAS command. |
| | | | (the following area is used to hold the game's current status, and is saved by the SAVE source command) |
| &F530 | FLAGS | (255) | Area used to store values of flags. |
| &F62F | SFLAGS | (30) | Area used to store values of system flags. |
| &F64D | MOBJ.LOCS | (255) | Current locations of movable objects in the game. |
| &F74C | UOBJ.LOCS | (255) | Current locations of unmovable objects in the game. |
| &F84B | CARRIED | (255) | Area used to indicate whether a movable object is currently carried by the player. 1 = object is carried, 0 = object is not carried. |
| &F94A | SCORE.MAP | (100) | Current Score Map. |
| | | | (The following area is used to store the current game status by the RAMSAVE command, and is essentially a copy of the above area) |
| &EF54 | RAMSAVE.BUFF | (1150) | RAMSAVE buffer area. |

# GLOSSARY OF SOURCE COMMANDS

This appendix lists all source bank commands in alphabetical order, along with any parameters required, and at least one example of the command in normal usage.

The parameters are represented as follows :

n - numeric expression

Numeric expressions may consist of numbers restricted to the range 0 to 255 inclusive (sometimes when used in conjunction with string functions, the range is extended to 0 to 1024 inclusive), arithmetic (+ and -) and numeric functions which are detailed below. Expressions are strictly evaluated from left to right, and brackets except as part of a function name are not permitted. Numeric expressions may be as complex as entry space will allow.

Numeric Functions :

| | |
|---|---|
| **VNO** | Returns the number of the last verb mentioned in the last player's input. If no verb was mentioned, then **VNO** returns a value of zero. |
| **DNO** | Returns the number of the last direction mentioned in the last player's input. If no direction was mentioned, then **DNO** returns a value of zero. |
| **MNO** | Returns the number of the last movable object mentioned in the last player's input. If no movable object was mentioned, then **MNO** returns a value of zero. |
| **UNO** | Returns the number of the last unmovable object mentioned in the last player's input. If no unmovable object was mentioned, then **UNO** returns a value of zero. |
| **PNO** | Returns the number of the last preposition mentioned in the last player's input. If no preposition was mentioned, then **PNO** returns a value of zero. |
| **LNO** | Returns the location number of the current location. |
| **CNT** | Returns the current counter value used by the **FOR/NEXT** command. If accessed outside a **FOR/NEXT** loop, the result can be unpredictable. |
| **FLG(n)** | Returns the contents of flag number **n**. **n** may be either a number in the range 1 to 255 o another valid numeric expression. The interpreter will display an error message if the valu of zero is used. |
| **SFLG(n)** | Returns the contents of system flag number **n**. **n** may be either a number in the range 1 t 30 or another valid numeric expression. The interpreter will display an error message if th value used is zero or greater than 30. |
| **RND(n)** | Returns a random number in the range 0 to **n**. **n** may be either a number in the range 1 t 255 or another valid numeric expression. The interpreter will display an error message if the value of zero is used. |

Some examples of valid numeric expressions :

```
23+VNO

FLG(8)+1+DNO

FLG(8+FLG(11))-SFLG(20)

10+FLG(18)-RND(9)

FLG(RND(FLG(88+2)+1))-UNO
```

**s$** - string expression

String expressions may consist of string functions which are detailed below and arithmetic (+ only). Again, expressions are strictly evaluated from left to right, and brackets except as part of a function name are not permitted. String expressions may be as complex as entry space will allow.

String functions :

| | |
|---|---|
| **MSG(n)** | Returns the complete message number **n**. **n** may either be a number in the range 0 to 102 or a valid numeric expression.The interpreter will display an error message if message number **n** has not been defined. |
| **SPC(n)** | Returns a string consisting of **n** spaces. **n** may be a number in the range 1 to 1024 or a valid numeric expression. The interpreter will display an error message if the value of zer is used. |
| **STR$(n)** | Converts the numeric expression **n** into a string (similar to SAM BASIC's STR$ function). r may be a number in the range 0 to 1024 or a valid numeric expression resulting in a valu in the range 0 to 65535. A leading space is added to the string representation of **n**. |
| **V$(n)** | Returns the name of verb number **n**. **n** may be a number in the range 1 to 255 or a valid numeric expression. The interpreter will display an error message if the name of verb **n** has not been defined, or **n** is outside the range 1 to 255. |
| **D$(n)** | Returns the name of direction number **n**. **n** may be a number in the range 1 to 99 or a valid numeric expression. The interpreter will display an error message if the name of direction **n** has not been defined, or **n** is outside the range 1 to 99. |
| **M$(n)** | Returns the name of movable object number **n**. **n** may be a number in the range 1 to 255 or a valid numeric expression. The interptreter will display an error message if the name of movable object **n** has not been defined, or **n** is outside the range 1 to 255. |
| **U$(n)** | Returns the name of unmovable object number **n**. **n** may be a number in the range 1 to 255 or a valid numeric expression. The interpreter will display an error message if the name of unmovable object **n** has not been defined, or **n** is outside the range 1 to 255. |

Some examples of valid string expressions :

```
MSG(11)

MSG(13)+MSG(14)

M$(11-FLG(30))+SPC(1)+MSG(1000)

MSG(88+FLG(99-SFLG(1)+2)-1)+SPC(1)+STR$(FLG(8))+MSG(3)
```

l# - line label

Line labels may consist of up to eight of any printable characters. Line labels may not be duplicated within the same source bank.

c - comparison

Used in **IF..** commands, comparisons may be any of the following :

| | |
|---|---|
| = | Is equal to |
| <> | Is not equal to |
| < | Is less than |
| > | Is greater than |
| >= or => | Is greater than or equal to |
| <= or =< | Is less than or equal to. |

**a** - action to take

Used in all **IF..** commands. **a** is the action to take if the **IF..** command proves true, and is one of the following :

| | |
|---|---|
| **G,l# (Goto)** | Jump to the line label **l#**. |
| **S,l# (Gosub)** | Call the subroutine located at line label **l#**. |
| **T (Then)** | Execute the following source line, else ignore it. |
| **D (Do)** | Execute the following source lines enclosed within a structure, else ignore them. |
| **O (Or)** | Chain with a following **IF..** command. |
| **A (And)** | Chain with a following **IF..** command. |

Any number of **IF..** commands may be chained together with either **And** or **Or**, creating in effect a much longer single **IF..** command. However, both **And** and **Or** cannot be present in the same chain. Instead, a **Then** could be used to "connect" two chains together. For example, instead of -

```
If Movable Object = [8] OR
If Movable Object = [9] AND
If At Location [10] AND
If Carried [11] GOTO JUMP
```

Use -

```
If Movable Object = [8] OR
If Movable Object = [9] THEN
  If At Location [10] AND
  If Carried [11] GOTO JUMP
```

**st#** - character string

A string of any printable characters. Used in the **REMARK** and **EXECUTE BASIC COMMAND** source commands.

## Act Upon Direction

This command checks the exits table for the current location for the exit corresponding to the last direction which was mentioned in the last player's input (the value DNO).

If a valid connection was found, then a "move" is made to the appropriate new location. LNO will equal the new location number, the exits table for that location will be transfered and a jump is made to the start of source bank 4.

If the connection was not valid, or no direction was specified in the last player's input (in which case DNO will equal zero), then no action is performed.

Example

```
  Act Upon Direction
```

## Add Direction [n1] Leading To [n2]

This command adds a connection to the exits table for the current location. The direction **n1** will lead to the location **n2**.

If a connection using the direction **n1** already exists in the exits table, then the new addition will take precedence while it is in force.

However, it is important to remember that this new addition to the exits table is temporary and will only last until a move is made to another location. If you wish this change to become permanent, then it is probably best to make this command conditional to the value of a flag (indicating that the new exit is permanent from now on) and place the command in source bank 4, so that the "new" exit will be added each time the location is re-entered.

Example

```
  If At Location [35] AND
  If Flag [10] > [0] THEN
    Add Direction [1] Leading To [50]
```

## Blank Line

Strictly speaking, this is not a command at all! This option from the CONTROL menu, simply allows the insertion of a blank line into the source listing. This can make the listing a lot easier to read and follow

Inserting a blank line will have no effect at all on the compiled adventure code apart from increasing it overall length by three bytes.

Example

```
  Remark : *** This subroutine prints message 1 ***

SUBR.1:
  Print Message [1]
  Return

  Remark : *** This subroutine prints message 2 ***

SUBR.2:
  Print Message [2]
  Return
```

## Continue With Input

When a sentence has been entered by the player, the interpreter usually scans and parses the input until the first occurance of a comma, full stop or the words **THEN**, **AND** or the ampersand character (&). This command allows the interpreter to continue evaluating the player's input from the point where the last comma, full stop, **THEN** or **AND** was found. If there are no more full stops, commas **AND**s or **THEN**s remaining in the input, then the command behaves as a **JUMP TO INPUT** command would and the interpreter awaits a new input from the player.

This very powerful feature enables the player to enter complex inputs such as

```
TAKE THE ROPE AND EXAMINE IT THEN DROP IT
```

or to enter multiple commands using full stops or commas such as

```
TAKE ROPE. EXAMINE ROPE, GO N. E, W. CLIMB LADDER.
```

In both cases, each part of the player's input will be acted upon in turn.

When using full stops or commas, it is important however that there is at least one space before the following word, otherwise the interpreter will assume an input such as

```
EAST,NORTH
```

to be a single word!

In the case of full stops or commas, the functions **VNO**, **DNO**, **MNO**, **UNO**, and **PNO** will all be set to return a value of zero before the remaining text is scanned.

**AND**, **THEN** and ampersand (&) however, maintain the previous values of **VNO**, **DNO**, **MNO**, **UNO** and **PNO** before any remaining text is dealt with.

This would allow an input such as

```
TAKE THE APPLE AND EAT IT
```

to be understood, but with

```
TAKE THE APPLE. EAT IT
```

the interpreter would have problems in deciding exactly what the player was supposed to eat as **MNO** would now return a value of zero.

The words **AND**, **THEN** and ampersand (&) should not be defined by the adventure author in the Vocabulary section of the editor, as the compiler automatically adds these words upon compilation.

Probably the best place to use **CONTINUE WITH INPUT** is as the last command in source bank 3.

Example

```
  Continue With Input
```

## Decrement Flag [n]

This command decreases the value held in the flag **n** by one. If flag **n** already holds zero, then the value of zero will be maintained.

Example

```
  Decrement Flag [10]
```

## Describe

This command simply describes the current location, listing all exits and any movable objects present.

If system flag 6 contains a non-zero value, then **DESCRIBE** will omit printing the exit details, and similarly, the listing of any movable objects present can be suppressed by giving system flag 7 a non-zero value. The exact way that any movable objects present are displayed will depend on the value of system flag 9 (see the appendix on system flags for more details on this).

Although the most obvious use of this command would be in source bank 4, so that any new locations are described as they are entered, it would be useful to use this command in other banks, for example linking it to a "Look" verb in bank 2.

Example

```
  Describe
```

## Describe Location [n]

This command works exactly the same as **DESCRIBE**, with system flags 6, 7 and 9 having the same effect, but instead of describing the current location, this command describes location **n** as if you were already there!

Possible uses of this command could be if you wanted to "Look" through a door etc to "see" what was on the other side!

Of course,

```
Describe [LNO]
```

would have exactly the same effect as the shorter (and more memory efficient)

```
Describe
```

Example

```
  Describe [80]
```

## Drop Movable Object [n]

This command enables the player to "drop" the movable object **n**. Once "dropped", the movable object is placed in the current location. System flag 1 is also decreased by one.

No action is performed if movable object **n** is not currently being carried by the player.

Example

```
Drop Movable Object [MNO]
```

## Execute BASIC Command : st$

This very powerful command enables the user to effectively create and call his own BASIC subroutines from within the compiled adventure environment! Customised machine code routines may also be executed by using this command to provide a normal BASIC "CALL" or "USR" statement. Of course, any machine code called would have to be already present in the memory, but you could even use this command to "LOAD" the machine code into memory in the first place! (although it might be better to modify the "Auto" BASIC loader instead).

The expression **st$** may contain multiple BASIC statements separated as normal by colons, but **st$** must not contain a leading BASIC line number.

If MASTERBASIC, MASTERDOS or any other extended BASIC commands are used, then the user must of course ensure that the appropriate DOS or extension code is present in memory first. (again, by probably altering the compiled adventure's BASIC "Auto" file)

It is up to the user to ensure that the **st$** expression is valid BASIC, as no BASIC syntax checking is done by the compiler except to detect a leading BASIC line number. If a BASIC error occurs, then control will pass directly back to the source command following the **EXECUTE BASIC COMMAND** command where the problem occured, resuming control after first displaying an interpreter error message - Even **STOP** counts as an error in this case.

Once your BASIC command(s) have been performed, control is passed back to the next source command in the current bank.

Example

```
Execute BASIC Command : CLS : PRINT AT 0,0;"Hello"
```

## Exit Structure

This command will jump directly to the end of the current structure.

If placed outside a structure, this command will generate an error message within the source compiler.

Example

```
If Flag [20] = [1] DO
  {
  If Flag [21] > [0] THEN
    Exit Structure
  Print Message [99]
  }
```

## For / Next : Start [n1] : End [n2] : Step [n3]

This command allows a section of source to be looped a number of times in a similar way to SAM BASIC's "FOR" command.

**n1** should be the start value of the loop, **n2** the end value and **n3** the step value. All three of these must be specified.

The **FOR / NEXT** command should be immediately followed by a structure in which the actual looping will take place.

while the loop is executing, the numeric function **CNT** can be used to access the current loop counter value.

**FOR / NEXT** loops may not be nested. If nested loops ARE required (unlikely), then a similar effect may be achieved by using a flag to hold the loop counter and a **GOTO** command to jump back to the looping point. An example of such a nested loop is shown below.

Example

```
 For / Next : Start [1] : End [10] : Step [1]
   {
   Print Message [CNT]
   }
```

Nested loop example

```
 Set Flag [1] To [10]
OUTER.LP:
 For / Next : Start [1] : End [20] : Step [1]
   {
   }
 Decrement Flag [1]
 If Flag [1] > [0] GOTO OUTER.LP
```

## Gosub l$

This command allows you to call subroutines in a similar way to normal SAM BASIC. l$ must be a valid line label located within the current source bank. One bank may NOT call a subroutine located within another source bank.

**GOSUBs** may form a part of all of the **IF..** commands to form in effect a more secconomical single command.

The actual subroutines are always begun with a line label and terminated by a **RETURN** command. Subroutines may be nested up to a maximum of 255 levels. Once a **RETURN** command is encountered, control is passed back to the command following the one which originally called the subroutine.

The user must ensure that the normal program logic bypasses any subroutines located within a bank, as a **RETURN** command encountered without a previous **GOSUB** will cause a run-time error message in the interpreter.

Example (1)

```
 Gosub SUBR.1
```

Example (2)

```
 If Flag [20] > [0] GOSUB SUBR.1
```

## Goto l$

This command allows you to jump directly to the line label l$ located in the current source bank. You may NOT use **GOTO** to jump from one source bank to another. Instead, use the **JUMP TO END**, **JUMP TO INPUT**, **ACT UPON DIRECTION** and **MOVE TO** commands.

It is not permitted to **GOTO** a line label located within a structure, although using **GOTO** from within a structure to jump to a line label located outside structures is fine.

As with **GOSUB**, **GOTO** may form a part of all of the **IF..** commands to form in effect a more ecconomical single command.

Example (1)

```
Goto LABEL.1
```

Example (2)

```
If Flag [10] = [1] GOTO LABEL.A
```

## If At Location [n] a

This command is used to check if the player is at location **n**. If so, then action **a** is performed, otherwise no action is taken.

Example

```
If At Location [20] GOTO L.LABEL
```

## If Carried [n] a

This command is used to check if the player is currently carrying movable object **n**. If so, then action **a** is performed, otherwise no action is taken.

Example

```
If Carried [18] THEN
   Exit Structure
```

## If Chance [n] a

This command is used to perform actions conditional to the result of a random number in the range between zero and the numeric expression **n**. If the random number generated is zero, then action **a** is performed, otherwise no action is taken. **n** must be in the range 1 to 255. The interpreter will display an error message if the value of zero is used.

So in effect, this command will result in a one in **n**+1 chance of action **a** being performed. In the example below, the message will have a one in four chance of being printed.

Example

```
If Chance [3] THEN
   Print Message [25]
```

## [illegible]

This command is used to compare the number of the direction mentioned in the last player's input (the value returned by the numeric function DNO), against the numeric expression **n** using the comparison **c**. If the comparison holds true, then action **a** is performed, otherwise no action is taken.

Example

```
 If Direction > [0] THEN
   Act Upon Direction
```

## If Direction Not Valid a

This command looks in the current location's exits table for the exit corresponding to the direction mentioned in the player's last input (the value returned by the numeric function DNO).

If a valid connection was NOT found (or no direction was mentioned - in which case DNO will return a value of zero), then action **a** is performed, otherwies no action is taken.

Example

```
 If Direction Not Valid GOTO DIR.BAD
 Act Upon Direction
DIR.BAD:
```

## If Flag [n1] c [n2] a

This command is used to compare the contents of flag **n1** against the numeric expression **n2** using the comparison **c**. If the comparison holds true, then action **a** is performed, otherwise no action is taken.

Example

```
 If Flag [80] = [FLG(81)] GOTO JUMP
```

## If Location c [n] a

This command is used to compare the number of the current location (the value returned by the numeric function LNO) against the numeric expression **n** using the comparison **c**. If the comparison holds true, then action **a** is performed, otherwise no action is taken.

Of course, the instruction

```
 If Location = [4] GOTO JUMP
```

would have exactly the same effect as the shorter (and more memory efficient)

```
 If At Location [4] GOTO JUMP
```

Example

```
 If Location > [50] THEN
   Increase Score [LNO]
```

## If Movable Object c [n] a

This command is used to compare the number of the movable object mentioned in the last player's input (the value returned by the numeric function **MNO**), if any, against the numeric expression **n** using the comparison **c**. If the comparison holds true then action **a** is performed, otherwise no action is taken.

Example

```
If Movable Object = [2] GOTO MO.2
```

## If Movable Object Not Present [n] a

This command is used to check whether the movable object **n** is NOT present in the current location. If the movable object is indeed not present, then action **a** is performed, otherwise no action is taken.

This command could be useful if the user wanted to detect whether the player was trying to pick up a movable object that was not actually present!

Example

```
If Movable Object Not Present [MNO] DO
   {
   Print Lower Message [100]
   Jump To Input
   }
```

## If Movable Object Present [n] a

This command is used to check whether the movable object **n** is in the current location. If so, then action **a** is performed, otherwise no action is taken.

Again, this command could be used in checking that a movable object was actually present before trying to pick it up!

Example

```
If Movable Object Present [MNO] THEN
   Take Movable Object [MNO]
```

## If Not Carried [n] a

This command is used to check whether the player is currently NOT carrying movable object **n**. If the player is indeed not carrying the movable object, then action **a** is performed, otherwise no action is taken.

This command could be useful in checking whether the player is trying to use an object which he is not actually carrying!

Example

```
If Not Carried [MNO] DO
   {
   Print Lower Message [90]
   Jump To Input
   }
```

## If Preposition c [n] a

This command is used to compare the number of the preposition mentioned in the last player's input (the value returned by the numeric function **PNO**), if any, against the numeric expression **n** using the comparison **c**. If the comparison holds true, then action **a** is performed, otherwise no action is taken.

When used along with the **IF VERB** command, this command can be used to alter the verb number when a certain preposition is present. Consider for example, the phrases

TAKE

and

TAKE OFF

which obviously have different meanings.

Example

```
 Remark : *** Convert one verb into another when a preposition is present ***

 If Verb = [8] AND
 If Preposition = [2] DO
   {
   Set Verb To [20]
   Set Preposition To [0]
   }
```

## If System Flag [n1] c [n2] a

This command is used to compare the contents of system flag **n1** against the numeric expression **n2** using the comparison **c**. If the comparison holds true, then action **a** is performed, otherwise no action is taken.

**n1** must be a valid numeric expression in the range 1 to 30, otherwise the interpreter will display an error message.

Example

```
 If System Flag [1] > [9] GOTO LABEL
```

## If Unmovable Object c [n] a

This command is used to compare the number of the unmovable object mentioned in the player's last input (the value returned by the numeric function **UNO**), if any, against the numeric expression **n** using the comparison **c**. If the comparison holds true, then action **a** is performed, otherwise no action is taken.

Example

```
 If Unmovable Object = [18] GOTO LAB.U18
```

## If Unmovable Object Not Present [n] a

This command is used to check whether the unmovable object **n** is NOT present in the current location. If the unmovable object is indeed not present, then action **a** is performed, otherwise no action is taken.

This command could be useful if the user wished to print a response to the player trying to examine etc, an unmovable object that was not present in the current location.

Example

```
 If Unmovable Object Not Present [UNO] DO
   {
   Print Lower Message [110]
   Jump To Input
   }
```

## If Unmovable Object Present [n] a

this command is used to check whether unmovable object **n** is in the current location. If so, then action **a** is performed, otherwise no action is taken.

Again, this command could be used to check whether an unmovable object was actually present before trying to examine it etc.

Example

```
 If Unmovable Object Present [UNO] AND
 If Verb = [18] DO
   {
   Print Message [30]
   Goto V18.U
   }
```

## If Verb c [n] a

this command is used to compare the number of the verb in the last player's input (the value returned by the numeric function **VNO**), if any, against the numeric expression **n** using the comparison **c**. If the comparison holds true, then action **a** is performed, otherwise no action is taken.

Example

```
 If Verb = [10] GOTO VERB.10
```

## Increase Score [n]

This command is used to increase the player's score at various points throughout the game. **n** must be a numeric expression in the range 1 to 100. The interpreter will display an error message if values outside this range are used.

**n** represents a one byte position in a 100 byte long "increase score map" which is included in the saved game data when the **SAVE** and **RAMSAVE** commands are used.

When an **INCREASE SCORE** command is executed, it first checks if the contents of the byte at position **n** in the score map is non-zero. If so, then no further action is taken. If however, the byte does contain a value of zero, then the contents of the byte are increased to a one and the contents of system flag 2 (which is used to hold the player's current score) are also increased by one, thus increasing the player's score by "one percent" (At the start of each new game, each byte in the score map is initialised to zero, as is system flag 2, so representing a score of nil percent).

If the user decides that the score should be increased at a certain point in the game, by for example, opening a certain door, then by using a score map position, we can ensure that the player cannot achieve a score of one hundred percent by simply opening and closing the door one hundred times! Instead, the score will only be increased the FIRST time that the door is opened, as subsequently, the score map position will now hold a non-zero value.

The user can if he wishes, simply manipulate the current score by directly altering the contents of system flag 2 by using the **SET SYSTEM FLAG** command, but care must be taken to ensure that the contents of the flag remain in the range 0 to 100 (otherwise scores of 101% etc may be displayed !!!).

The player's current score can be displayed by using the **SCORE** command. The **SCORE** command simply looks at the contents of system flag 2 for the current score percentage.

Example

```
Increase Score [10]
```

## Increment Flag [n]

This command increases the value held in the flag **n** by one. If the flag already holds a value of 255, then the value of 255 will be maintained.

Example

```
Increment Flag [5]
```

## Inventory

This command is used to display an inventory of all movable objects currently being carried by the player. The precise way that the list of carried objects is displayed on the screen, depends on the contents of system flag 9. If system flag 9 contains a value of zero, then INVENTORY will display each movable object carried on a new line, for example

```
Some matches
An apple
A large stick
```

If however, system flag 9 contains a non-zero value, then the same objects would be displayed as

```
Some matches, an apple and a large stick.
```

If System flag 8 contains a non-zero value, then any movable object prefixes such as "An", "Some" etc. will not be printed.

System flag 10 should hold the number of a message which contains text along the lines of "You have with you" or "I am currently carrying:" etc. which will be printed on screen before the actual list of carried objects. If system flag 10 contains zero, then no such message will be printed.

System flag 11 should hold the number of a message which contains text along the lines of "Nothing at the moment" which will be printed if the player is not actually carrying any objects at all at the moment. Again, if system flag 11 contains a value of zero, then no such message will be printed.

You should remember that since system flags can only contain values in the range 0 to 255, the messages that system flags 10 and 11 refer to must be messages with numbers less than 256!

Ideally, system flags 9, 10 and 11 should all be set with appropriate values in source bank 1, which is executed before the game actually begins, but you can constantly re-define the values held in these system flags from within the rest of the adventure source throughout the game if you wish.

Example

```
If Verb = [5] DO
   {
   Inventory
   Jump To Input
   }
```

## Jump To End

This command simply forces a direct jump to the first command line in source bank 3 where logic routines common to all locations and all circumstances within the game (low priority conditions) are executed.

Example

```
Jump To End
```

## Jump To Input

This command forces a direct jump to the input routine which awaits the player's next instruction input. However, unlike the **CONTINUE WITH INPUT** command, any remaining multiple commands in the player's last input which have not yet been dealt with are ignored.

Once the player has typed his next instruction and pressed "RETURN", each word in the input is checked against the list of words in the vocabulary, and the source code is entered at the first command line in source bank 2 as normal, where the user would place his main input evaluation routines.

It is often convenient to use this command after a **PRINT LOWER MESSAGE** command reporting an "error" such as "You can't do that" or "You cannot go that way" etc.

Example

```
If Direction Not Valid DO
  {
  Print Lower Message [80]
  Jump To Input
  }
```

## Line Label l$

This is not really a command at all. It should be thought of as a "marker" marking the start of a specific section in the source listing. l$ should be a string of up to eight characters (most printable characters can be used) Line label names cannot be duplicated in the same source bank.

Both the **GOTO** and **GOSUB** commands require a valid line label to jump to. Extra line labels can be used as useful reference points within the listing, as the editor will allow you to jump directly to them. Line labels have absolutely no effect on the final compiled adventure except to increase its overall length by three bytes for every line label used.

Example

```
LABEL.A:
```

## List Exits

This command is used to list the exits available from the current location (including any exits which have been added by the **ADD DIRECTION** command). If there are no exits available from the current location, then no action is taken.

The names of the directions printed will be accessed from the "Direction Definitions" section of the editor. The interpreter will display an error message if the name of the appropriate direction has not been defined.

System flag 14 should hold the number of a message which contains text along the lines of "Exits from here are:" which will be printed on screen before the actual list of available directions. If system flag 14 contains zero, then no such message will be printed.

You should remember that since system flags can only contain values in the range 0 to 255, the message that system flag 14 refers to must be a message with a number less than 256!

Ideally, system flag 14 should be set with an appropriate message number in source bank 1, which is executed before the game actually begins.

Example

```
List Exits
```

## List Movable Objects Present

This command is used to list any movable objects which are present in the current location. If no movable objects are actually present, then no action is taken.

The precise way that the list of present movable objects is displayed on screen depends on the contents of system flag 9 (see the appendix on system flags for more details on this).

The names of the movable objects printed will be accessed from the "Movable Object Definitions" section of the editor. The editor will display an error message if the name of the appropriate movable object has not been defined.

System flag 15 should hold the number of a message which contains text along the lines of "Movable objects present are:" or "You can see here:" which will be printed before the actual list of present movable objects. If system flag 15 contains zero, then no such message will be printed.

You should remember that since system flags can only contain values in the range 0 to 255, the message that system flag 15 refers to, must be a message with a number less than 256!

Ideally, system flag 15 should be set with an appropriate message number in source bank 1 which is executed before the game actually begins.

Example

```
List Movable Objects Present
```

## Load

This command enables a saved game position to be loaded from cassette or disk (according to SAM BASIC's DEVICE status), enabling a player to resume play at the point in the game where the saved game position was previously saved.

It is usually a good idea to follow this command with a **DESCRIBE** (to describe the new location) and **JUMP TO INPUT** commands.

Example

```
If Verb = [12] DO
  {
  Load
  Describe
  Jump To Input
  }
```

## Move to [n]

This command moves the player directly to location **n**. **LNO** will now return the new location number, and a jump is made to the first command line in source bank 4.

Example

```
Move To [12]
```

## NOT Flag [n]

This command alters the contents of flag **n** as follows:

If the contents of flag **n** has a value of greater than zero, then it will be made to hold a value of zero.

If however, the contents of flag **n** are already zero, then the flag will be made to hold a value of one.

This command would be useful for toggling flags which might for example, indicate whether a coat is worn or not, or whether a door is open or not.

By using this command, it is possible to reverse the status of the flag without needing to inspect its contents first. Using this command twice in succession will restore the contents of the flag to its original status (either a zero or a one).

Example

```
NOT Flag [35]
```

## Pause

This command simply pauses the game until the player presses any key.

This would be useful if for example, the player was to be prompted to insert a data disk before loading/saving a saved game position.

Example

```
If Verb = [12] DO
   {
   Print Lower Message [127]
   Pause
   Save
   Jump To Input
   }
```

## Print Lower Message [n]

This command prints the message number **n** in the lower screen window normally used for the player's input.

**PRINT LOWER MESSAGE** would normally print the message in upper case and in INVERSE (ie PAPER on PEN ), with one extra space added to the start and end of the message. However these features can be altered according to the user's taste. See the chapter "Customising SAS" for more details on this

It is intended that this command should be used for displaying "error" messages such as "You can't go that way" etc.

Example

```
If Direction Not Valid DO
   {
   Print Lower Message [200]
   Jump To Input
   }
```

## Print Messsage [n]

This command prints the message number **n** in the upper screen window. After the last line of text in the message has been printed, the upper window is scrolled an extra time so as to leave a blank line before the next message to be printed, thus keeping the screen display nice and neat and avoiding clutter.

This command can therefore be thought of as printing a "paragraph" of text onto the screen. If "paragraphs" longer than 256 characters are required (the maximum length of a single message), then using a **PRINT MESSAGE SUPPRESSED** command before a **PRINT MESSAGE** command will have the desired effect by chaining two messages together.

Example

```
Print Message [80]
```

## Print Message Suppressed [n]

this command prints the message number **n** in the upper screen window in exactly the same way as the **PRINT MESSAGE** command. However, the screen is NOT scrolled an extra time at the last line to be printed.

This command could be used to print large "paragraphs" of text when used with the **PRINT MESSAGE** command which is used to print the end of the "paragraph".

This command could also be used to have an extra message line printed depending upon the value of a flag, but still enabling the current "paragraph" of text to remain nice and neat. The example below does this.

Example

```
Print Message Suppressed [75]
If Flag [99] > [0] THEN
   Print Message Suppressed [76]
Scroll Screen
```

## Put Movable Object [n1] At [n2]

This command enables the movable object **n1** to be placed at the location **n2**. If **n2** equals zero, then the movable object is in effect destroyed.

This command will have no effect if the movable object **n1** is currently being carried by the player.

Example

```
Put Movable Object [30] At [65]
```

## Put Unmovable Object [n1] At [n2]

This command enables the unmovable object **n1** to be placed at the location **n2**. If **n2** equals zero, then the unmovable object is in effect destroyed.

Example

```
Put Unmovable Object [5] At [LNO]
```

## Quit

This command is used to re-start a game from scratch. It does the following tasks before jumping to the first line of source bank 1 :

a) All flags and system flags have their contents set to zero EXCEPT for system flag 3 (used to indicate whether **RAMSAVE** has been used) whose value is retained.

b) All movable object and unmovable object locations are set to zero.

c) All movable objects are marked as "not currently being carried" by the player.

d) The contents of each position in the score map are cleared.

e) The screen is cleared

It will therefore become apparent that the commands in source bank 1 must be used to set the intitial locations of any objects (movable or unmovable), the initial location, and set values of any flags which are required hold non-zero values at the start of the game.

## Ramload

This command enables a saved game file to be loaded from RAM (computer memory) enabling the player to resume play at the point where the player's game position was previously **RAMSAVE**d.

**RAMLOAD** will only be executed if system flag 3 contains the value of one (indicating that **RAMSAVE** has been previously used. The user can make use of this feature to display an appropriate error message if nescessary. The example below does this.

It is usually a good idea to follow this command with a **DESCRIBE** command (to describe the "new" location) and a **JUMP TO INPUT** command.

Example

```
If Verb = [19] DO
  {

  Remark *** Print "error" message if no RAMSAVED file ***

  If System Flag [3] <> [1] DO
    {
    Print Lower Message [23]
    Jump To Input
    }
  Ramload
  Describe
  Jump To Input
  }
```

## Ramsave

This command enables the player to store in RAM (computer memory) details of the current game position, so allowing the player to resume playing from the same point later on if things should go disasterously wrong!

Because the game position data is saved in RAM rather than disk or cassette, this data will be lost as soon as the computer is turned off!

As **RAMSAVE** is used, the contents of system flag 3 is set to the value of one, indicating that there is now a **RAMSAVEd** file in memory.

Example

```
 If Verb = [18] DO
    {
    Ramsave
    Jump To Input
    }
```

## Remark st$

This command is simply used to comment a section of the source listing, so aiding the user to follow his own program logic.

**REMARK** has no effect whatsoever upon the compiled adventure except for increasing its overall length by three bytes for every **REMARK** used.

**st$** may be a string of any printable characters.

Example

```
 Remark *** Any text you like HERE !!!!!!! ***
```

## Return

This command is used to return from subroutines in a similar way to normal SAM BASIC.

Actual subroutines are always begun with a line label and terminated by a **RETURN** command. Any number of source commands may reside between the start and end of a subroutine. Subroutines may be nested up to a maximum of 255 levels.

Once a **RETURN** command is encountered, control is passed back to the command following the one which originally called the subroutine.

The user must ensure that the program logic bypasses any subroutines located within a bank, as a **RETURN** command encountered without a previous **GOSUB** will cause the interpreter to display an error message.

Example

```
SUBR.1:
  Print Message [FLG(1)]
  Return
```

## Save

This command enables the user to save a game position to cassette or disk (according to SAM BASIC's DEVICE status), enabling the player to resume play later at the point in the game where the position was saved, by using the **LOAD** command.

Example

```
If Verb = [14] DO
   {
   Save
   Jump To Input
   }
```

## Score

This command is used to display the player's current score (the value which is held in system flag 2).

System flag 12 should hold the number of a message which contains text along the lines of "Your current score is" or "You have completed" which will be printed directly before the score (a number in the range 0 to 100). If system flag 12 contains a value of zero, then no such message will be printed.

System flag 13 should hold the number of a message which contains text along the lines of "/100" or "% of this adventure" which will be printed directly after the player's score. Again, if system flag 13 contains a value of zero, then no such message will be printed.

You should remember that since system flags can only contain values in the range 0 to 255, the messages that system flags 12 and 13 refer to, must be messages with numbers less than 256!

Ideally, system flags 12 and 13 should be set with appropriate values in source bank 1, which is executed before the game actually begins, but you can constantly re-define the values held in these system flags from within the rest of the adventure source throughout the game if you wish.

Example

```
If Verb = [17] DO
   {
   Score
   Jump To Input
   }
```

## Scroll Screen

This command is used to scroll the upper text window up by one text line.

**SCROLL SCREEN** can be used along with the **PRINT MESSAGE SUPPRESSED** command in order to leave a blank line before the next message to be printed.

Example

```
Print Message Suppressed [8]
If Flag [45] > [0] THEN
  Print Message Suppressed
Scroll Screen
```

## Set Direction To [n]

This command is used to "fool" the interpreter into thinking that the player had typed the name of direction number **n** in his last input.

This command also directly alters the value that will be returned by the numeric function **DNO**.

Example

```
Set Direction To [5].
```

## Set Flag [n1] To [n2]

This command is used to set the contents of flag **n1** to hold the value of the numerical expression **n2**.

Example

```
Set Flag [75] To [FLG(20)+10]
```

## Set Location To [n]

This command is used to directly alter the the current location to location number **n**. Unlike the **ACT UPON DIRECTION** and **MOVE TO** commands, a jump is NOT made to the first source line of source bank 4.

It is ESSENTIAL that this command is used in source bank 1 to define the initial location number for the location that the player starts out from at the beginning of the game.

This command also directly alters the value that will be returned by the numeric function **LNO**.

Example

```
Set Location To [1]
```

## Set Message Zero To [s$]

This command is used to build up the string expression **s$** into message number zero. Message zero can then be printed in the normal way by using the **PRINT MESSAGE**, **PRINT LOWER MESSAGE** or **PRINT MESSAGE SUPPRESSED** commands.

The total length of message zero may not exceed 1024 characters (the interpreter will display an error message if an attempt is made to exceed this length).

Message zero may be constantly re-defined throughout the adventure source within any bank.

Example

```
Set Message Zero To [MSG(61)+SPC(1)+V$(VNO+1)+MSG(62)]
Print Message [0]
```

## Set Movable Object To [n]

This command is used to "fool" the interpreter into thinking that the player had typed the name of movable object number **n** in his last input.

This command also directly alters the value that will be returned by the numeric function **MNO**.

Example

```
Set Movable Object To [5]
```

## Set Preposition To [n]

This command is used to "fool" the interpreter into thinking that the player had typed the text of preposition number **n** in his last input.

This command also directly alters the value that will be returned by the numeric function **PNO**.

Example

```
Set Preposition To [1]
```

## Set Scroll Counter To [n]

This command is used to directly alter the scroll counter. When using this command, the screen can be scrolled (by using the **SCROLL SCREEN**, **PRINT MESSAGE** or **PRINT MESSAGE SUPPRESSED** commands) **n**-1 times before the "More ↑" message is displayed, and the interpreter waits for a key-press before continuing to scroll the upper text window.

This would be useful if for example, the user wished to display a graphic picture of an object which was EXAMINEd, yet wished to let the player read any text which remained upon the screen first. The example below could be used to immediately print a "More ↑" message before displaying such a graphic.

Example

```
Set Scroll Counter To [1]
Scroll Screen
Show Graphic [1]
```

## Set System Flag [n1] To [n2]

This command is used to set the contents of system flag **n1** to hold the value of the numerical expression **n2**.

The interpreter will display an error message if **n2** has a value of zero or is greater than 30.

Example

```
set System Flag [9] To [1]
```

## Set Unmovable Object To [n]

This command is used to "fool" the interpreter into thinking that the player had typed the name of unmovable object **n** in his last input.

This command also directly alters the value that will be returned by the numeric function **UNO**.

Example

```
Set Unmovable Object To [8]
```

## Set Verb To [n]

This command is used to "fool" the interpreter into thinking that the player had typed the name of verb **n** in his last input.

This command also directly alters the value that will be returned by the numeric function **VNO**.

As an example of how this command could be used, consider the input " Get into the taxi", where "Get" is defined as a verb in the vocabulary, "into" a preposition and "taxi" is an unmovable object. In most adventures, "get" would be defined as a synonym of the verb "take". So initially, as far as the interpreter is concerned, the player wants to pick up the taxi! By detecting the verb number for "get" and the synonym number for "into", we can transform the verb number into the one for the verb "enter" which is obviously what the player really meant. The example below shows how this could be done.

Example

```
If Verb = [4] AND
If Preposition = [2] DO
   {
   Set Verb To [10]
   Set Preposition To [0]
   }
```

## Show Graphic [n]

This command is used to display location graphics. **n** is the picture number to display, and refers to the **n**th picture to be added to the compiled adventure by using the graphics extension program on the compiler disk. If no graphics have yet been added to the compiled adventure, then no action is taken.

**SHOW GRAPHIC** will automatically scroll the upper text window, clearing any text before showing the picture.

Although this command would mostly be used in source bank 4 to display location graphics as the location is entered, it is perfectly possible to use this command in other source banks. For example, in source bank 2, the user could use this command to display a picture of a map in response to it being examined or read.

Example

```
If At Location [54] THEN
   Show Graphic [11]
```

## } - Structure End

This is used to mark the end of a structure. Structures may be nested up to a maximum of 255 levels.

Example

```
For / Next : Start [120] : End [130] : Step [1]
  {
  Print Message [CNT]
  }
```

## { - Structure Start

This is used to mark the start of a structure which can contain any number of commands. Structures may be nested up to a maximum of 255 levels.

The **FOR / NEXT** command and all **IF..** commands with a **DO** action, require to be followed immediately by a structure.

Example

```
If Not Carried [10] DO
  {
  Print Message [210]
  }
```

## Swop Movable Objects [n1] And [n2]

This command reverses the current locations of movable objects **n1** and **n2**. This applies whether the objects are being carried or not.

A possible use of this command could be to "swop" objects such as a "lit match" and an "unlit" one, when in reality, both movable objects have been separately defined. You could even have one movable object magically transformed into a different one in the game!

Example

```
If Verb = [3] AND
If Movable Object = [23] THEN
  Swop Movable Objects [23] And [24]
```

## Take Movable Object [n]

This command enables the player to pick up movable object number **n**. This command does not check whether movable object **n** is actually in the current location first, so the user will probably have to detect this himself.

Once "taken", the movable object is removed from the current location (if it was there at all!) and the contents of system flag 1 (used to hold the number of objects currently carried by the player) is increased by one.

No action is taken if Movable object **n** is already being carried by the player.

Example

```
If Movable Object Present [MNO] THEN
  Take Movable Object [MNO]
```

# SYSTEM FLAGS

This appendix gives details on the system flags used by the interpreter. Where names are given to the flags, the names are purely for reference and do not refer to any numeric function names which can be accessed by the editor.

The contents of system flags can be directly altered by using the **SET SYSTEM FLAG** command, and their contents read by using the **SFLG(n)** numerical function. **n** represents the flag number to be read and must be a valid numerical expression in the range 1 to 30.

## System Flag 1 name : "NO.CARRIED"

This flag is used to hold the number of movable objects which are currently being carried by the player. The contents of this flag are increased by one when a **TAKE MOVABLE OBJECT** is executed, and decreased by one when a **DROP MOVABLE OBJECT** command is executed.

By checking the contents of this flag before allowing the player to pick up a movable object, the user can impose a limit on the number of movable objects that the player is allowed to carry at any one time. The example below imposes a limit of four objects carried.

Example

```
 If System Flag [1] > [3] DO
   {
   Print Lower Message [19]
   Jump To Input
   }
 Take Movable Object [MNO]
```

In the example shown, message 19 would contain text along the lines of "Sorry, you cannot carry any more"

## System Flag 2 name : "SCORE"

This flag is used to hold the player's current score (normally a number in the range 0 to 100). The contents of this flag are read by the **SCORE** command and increased by the **INCREASE SCORE** command.

## System Flag 3 name : "RAMSAVE.FLG"

This flag contains the value of one if a game position has been stored in RAM (computer memory) by using the **RAMSAVE** command, otherwise it will contain a value of zero.

The user can inspect this flag in order to display an "error" message if the player tries to use **RAMLOAD** without using **RAMSAVE** first.

This is the only system flag whose contents are not reset to zero by the **QUIT** command.

Example

```
 If System Flag [3] = [0] DO
   {
   Print Lower Message [23]
   Jump To Input
   }
 Ramload
```

## System Flag 4 name : "LOCATION"

This flag is used to hold the location number of the current location. The contents of this flag are read by the numeric function **LNO**.

## System Flag 5 name : "OLD.LOCATION"

This flag is used to hold the location number of the location you have just "left" when using the **MOVE TO** or **ACT UPON DIRECTION** commands. Some users might find this useful.

## System Flag 6 name : "DESC.E.FLG"

If the contents of this flag are set to a non-zero value, then the **DESCRIBE** and **DESCRIBE LOCATION** commands will NOT list any available exits (some users might prefer to list available exits in the actual location descriptions instead).

Altering the contents of this flag does not affect the **LIST EXITS** command.

## System Flag 7 name : "DESC.O.FLG"

If the contents of this flag are set to a non-zero value, then the **DESCRIBE** and **DESCRIBE LOCATION** commands will NOT list any movable objects which are present in the relevant location.

Altering the contents of this flag does not affect the **LIST MOVABLE OBJECTS PRESENT** command.

## System Flag 8 name : "PREFIX.FLG"

If the contents of this flag are set to a non-zero value, then any movable objects listed by the **DESCRIBE**, **DESCRIBE LOCATION**, **INVENTORY** and **LIST MOVABLE OBJECTS PRESENT** commands will NOT have their prefixes ("the", "some" etc.) printed (if they have one).

Note that this flag does NOT suppress the movable object prefix when the string function **M@(n)** is used. System flag 17 can be used for this purpose if desired.

## System Flag 9 name : "INV.TYPE"

This system flag controls the way that movable objects are displayed on the screen by the **DESCRIBE**, **DESCRIBE LOCATION**, **INVENTORY** and **LIST MOVABLE OBJECTS PRESENT** commands.

If the flag contains a value of zero, then each movable object will be printed on a new line. For example

```
An orange
Some stones
The computer
```

If however, system flag 9 contains a non-zero value, then the same objects would be displayed as

```
An orange, some stones and the computer.
```

## System Flag 10 name : "INV.MSG"

This flag is used by the **INVENTORY** command. The user should set this flag to contain the number of a message which should contain text along the lines of "You are currently carrying :" which is printed before the actual list of movable objects currently carried by the player. If the flag holds the value of zero, then no such message will be printed.

The message that this flag refers to must have a message number in the range 1 to 255.

## System Flag 11 name : "INV.NOMSG"

This flag is used by the **INVENTORY** command. The user should set this flag to contain the number of a message which should contain text along the lines of "Nothing at all!" which is printed in the case of the player typing INVENTORY when he is not actually carrying anything. If the flag holds a value of zero, then no such message will be printed (although this will probably look sloppy).

The message that this flag refers to must have a message number in the range 1 to 255.

## System Flag 12 name : "SCORE.MSG"

This flag is used by the **SCORE** command. The user should set this flag to contain the number of a message which should contain text along the lines of "Your current score is" which is printed directly before the player's actual score (a number in the range 0 to 100). If the flag contains the value of zero, then no such message will be printed.

The message that this flag refers to must have a message number in the range 1 to 255.

## System Flag 13 name : "SCORE.MSG2"

This flag is used by the **SCORE** command. The user should set this flag to contain the number of a message which should contain text along the lines of " out of 100" or "% so far" which is printed directly after the player's actual score (a number in the range 0 to 100). If this flag contains a value of zero, then no such message will be printed.

The message that this flag refers to must have a message number in the range 1 to 255.

## System Flag 14 name : "EXITS.MSG"

This flag is used by the **DESCRIBE, DESCRIBE LOCATION** and **LIST EXITS** commands. The user should set this flag to contain the number of a message which should contain text along the lines of "Exits from here are :" which is printed before the list of available exits. If this flag contains a value of zero, then no such message will be printed.

The message that this flag refers to must have a message number in the range 1 to 255.

## System Flag 15 name : "OBJ.PRES.MSG"

This flag is used by the **DESCRIBE, DESCRIBE LOCATION** and **LIST MOVABLE OBJECTS PRESENT commands.** The user should set this flag to contain the number of a message which should contain text along the lines of "Objects present here are :" which is printed before the list of movable objects present. If this flag contains a value of zero, then no such message will be printed.

The message that this flag refers to must have a message number in the range 1 to 255.

## System Flag 16 name : "PIX.FLG"

If this flag contains a non-zero value, then the **SHOW GRAPHIC** command will not display any pictures on the screen. Setting the contents of this flag to zero, enables **SHOW GRAPHIC** again. The user can make use of this flag to enable the player to switch between a text-only adventure, and one with graphics.

Example

```
PIX.ON:
  Set System Flag [16] To [0]
  Jump To Input

PIX.OFF:
  Set System Flag [16] To [1]
  Jump To Input
```

## System Flag 17 name : "M⊕.FLG"

This flag can be used to suppress the prefix of the name of the movable object returned by the string function **M⊕(n)**.

If the flag contains a value of zero, then any movable object prefix (such as "the ", "an ","some " etc.) will be included in the string returned by **M⊕(n)**.

If however, the flag contains a non-zero value, then the movable object prefix will be omitted from the name of the movable object returned by the string function **M⊕(n)**.

## System Flag 18 name : "FIRST.VNO"

This flag contains the number of the FIRST verb mentioned in the player's last input (or section of input if the player has used multiple commands). This is not necessarily the same value as will be returned by the numeric function **VNO** which returns the number of the LAST verb mentioned in the player's input (or section of input if multiple commands were used).

The user might find this flag useful when trying to evaluate exceptionally complex inputs entered by the player. As an example of how this works, consider the input

```
TAKE DROP
```

which obviously contains two verbs, "take" and "drop". The numeric function **VNO** will return the number of the LAST movable object mentioned in the input (in this case "drop"), but system flag 18 will contain the number of the verb "take".

If no verb at all was mentioned in the last player's input, then this flag will contain a value of zero.

## System Flag 19 name : "FIRST.DNO"

As system flag 18, but dealing with the first direction mentioned by the player.

## System Flag 20 name : "FIRST.MNO"

As system flag 18, but dealing with the first movable object mentioned by the player.

## System Flag 21 name : "FIRST.UNO"

As system flag 18, but dealing with the first unmovable object mentioned by the player.

## System Flag 22 name : "FIRST.PNO"

As system Flag 18, but dealing with the first preposition mentioned by the player.

## System Flag 23 name : "LAST.ERROR"

This flag will contain a value of one if an interpreter error occured (for whatever reason) while the interpreter was trying to execute the previous source line. If the previous source line was executed with no problems, then this flag will contain a value of zero.

By inspecting the contents of this flag, the user can take appropriate action if for example, a disk error occured while the player was trying to load or save a game position to disk.

Example

```
  Remark *** Take appropriate action if a disk error occurs while using SAVE ***

RETRY:
  Pause
  Save
  If System Flag [23] = [1] DO
    {
    Print Lower Message [123]
    Goto RETRY
    }
  Jump To Input
```

## System Flag 24 name : "ERROR.RPT.FLAG"

This flag can be used to suppress the printing of interpreter errors on screen. The errors still "occur" and the contents of system flag 23 is still set to one, but the player is unaware that anything has gone wrong, as no interpreter error message has appeared.

It is therefore absolutely essential that the user only sets this system flag once he is certain that his adventure has been vigourously play-tested and is free from errors.

By checking the contents of system flag 23 (see above), the user can still detect and take appropriate action on errors over which he has no control such as disk loading/saving errors.

## System Flag 25 name : "TOP.FREE"

This flag can be used to reserve an area at the top of the upper text window which will not be scrolled off of the screen by the **SCROLL SCREEN**, **PRINT MESSAGE** and **PRINT MESSAGE SUPPRESSED** commands. Some users might want to keep the name of the current location permanently here, or keep location graphics permanently displayed while text continues to scroll underneath as normal.

However, it is important to remember that the **SHOW GRAPHIC** command will continue to over-write the area reserved at the top of the screen as normal.

The maximum area that can be reserved is 10 text rows (each 8 pixels deep). The user should set this flag to contain the number of text rows he wishes to reserve. If the flag contains a number outside the range 1 to 10, then the interpreter will default to the normal upper window size of 17 text rows. It is a good idea to immediately alter the scroll counter after setting this flag.

Example

```
Remark : *** Reserve 3 rows at top of upper window ***

Set System Flag [25] To [3]
Set Scroll Counter To [17-SFLG(25)]

Remark *** Now put something at the top of the screen ***

Execute BASIC Command : PRINT AT 0,0;" Program Title "
```

## System Flags 26 - 30

These system flags are currently unused by the interpreter, but are reserved for use by any future versions of SAS.

# COMPILER ERROR MESSAGES

This appendix lists all possible error messages which may be displayed by the source compiler when compiling an adventure source created by the source editor.

While compiling a source, the compiler automatically displays on-screen the title of the section of the source currently being processed. If an error message should be displayed, it will be something in this last section that has caused the trouble.

In all instances, once an error is reported, the compilation will cease immediately, and the "compiled" adventure will be unplayable. The adventure source files being compiled will be unaffected. Wherever possible, the compiler will detail the line or item number in the current source section which caused the fault, and in the case of vocabulary or source bank errors, even display the offending line on-screen as well.

The user is recommended to make a note of the item in his source which caused the error, and return to the editor immediately to remedy the problem before attempting another compilation.

| | |
|---|---|
| Bad Compilation | In theory, this is one error message that you should NEVER see! It almost certainly indicates either a corrupted source file, or a programming "bug" in the compiler program itself! If this is the case, you should contact me so that I can investigate the problem. |
| Disk Error | A disk error occured when trying to load a source section from your "source" disk. Compilation was immediately aborted. |
| Exit Structure Misplaced | You have tried to use an **EXIT STRUCTURE** command outside a structure. The use of this command is only permitted somewhere between the **STRUCTURE START** and **STRUCTURE END** commands. |
| Expression Too Complex | You have tried to define a numeric or string expression which involved more than 255 calculations or operations - You are most unlikely to generate this error! |
| Invalid Action | An IF.. command was detected containing an invalid action. Only the following actions are permitted : THEN, DO, GOTO, GOSUB, AND and OR. |
| Invalid Command In Source Line | A source line did not contain a valid command code. This error message almost certainly indicates a corrupted source file on your "source" disk. |
| Invalid Comparison | An IF.. command was detected containing an invalid comparison. Only the following comparisons are permitted : < (is less than), > (is greater than), = (is equal to), <> (is not equal to), <= (is less than or equal to) or >= (is greater than or equal to). |
| Invalid Entry In Exits Table | You have defined exit data in a location incorrectly. Exits must be numbers in the range 1 to 99, and locations led to must be numbers in the range 1 to 255. |
| Invalid Expression | You have defined a numeric or string expression incorrectly. Possible causes of this error could be mis-spellings of function names or invalid arithmetic. |
| Invalid If.. Chain | You have defined an illegal chain of IF.. commands. An IF.. command with either an AND or OR action may only be chained with another IF.. command with the same action type. |

| | |
|---|---|
| Invalid Movable Object Prefix | You have defined the prefix for a movable object's name incorrectly. Only the following Prefixes are permitted : A, AN, THE, SOME or a space (indicating no prefix). |
| Invalid Vocabulary Line | A vocabulary line did not contain a valid command code. This error message almost certainly indicates a corrupted source file on your "source" disk. |
| Line Label Already Exists | You have duplicated the name of a line label in the current source bank. Only one line label per name is permitted. |
| Line Label Not Defined | You have tried to jump to a line label (by using the **GOTO**, **GOSUB** or **IF..** commands) which does not exist in the current source bank. |
| Line Label Within Structure | You have tried to position a line label within a structure, between the **STRUCTURE START** and **STRUCTURE END** commands. Line labels are only permitted outside structures. |
| Location Greater Than 255 In Exits Table | You have tried to define a location exit leading to a location with a number greater than 255. Only locations with numbers in the range 1 to 255 are permitted. |
| Message Greater Than 1024 | You have tried to print a message (using the **PRINT MESSAGE**, **PRINT MESSAGE SUPPRESSED** or **PRINT LOWER MESSAGE** commands), or tried to access a message (using the **MSG(n)** string function) with a number greater than 1024. Only messages with numbers in the range 0 to 1024 are permitted. |
| Missing Command Following THEN | An IF.. command containing a THEN action was not followed by another command line. A valid command line must be placed immediately after an IF.. command with a THEN action. |
| Missing Direction In Exits Table | You have not defined a direction number before a corresponding location number in a location exits table. |
| Missing Expression | You have omitted a string or numeric expression in a source command which required such an expression as one of its parameters. |
| Missing If.. Command | You have omitted an IF.. command. IF.. commands with either AND or OR actions require to be chained with another IF.. command following immediately on the next line. |

|  |  |
|---|---|
| Missing Location In Exits Table | You have not defined a location number after a corresponding direction number in a location exits table. |
| Missing Structure Start | You have omitted a **STRUCTURE START** command. Both the **FOR / NEXT** and **IF..** commands with a DO action, require to be immediately followed by a **STRUCTURE START** command on the next line. |
| More Than 255 Entries In Word Type | You have defined more than 255 "new" words in a "class" in the vocabulary. |
| More Than 255 Synonyms | You have defined more than 255 synonyms for a single word in the vocabulary. |
| No Free Memory | There is no free buffer space in RAM to continue compiling your adventure source. You are unlikely to encounter this error message (and even then, only when compiling exceptionally large source files). Try slightly reducing the size of one of your source sections (the largest is probably best) before attempting compilation again. |
| No Leading BASIC Line Number Allowed | You have tried to include a leading BASIC line number while using an **EXECUTE BASIC COMMAND** command. Leading BASIC line numbers are not permitted. |
| No Preceding Structure Start | You have used a **STRUCTURE END** command without previously using a **STRUCTURE START** command. |
| Number Greater Than 255 | You have tried to include a number greater than 255 in a numeric expression. Only numbers in the range 0 to 255 are allowed. |
| Number Too Big | You have tried to include a number consisting of more than 4 digits in a numeric or string expression. Only numbers with up to 4 digits are allowed. |
| Numeric Function Not Allowed | You have tried to use a numeric function in an expression. A string function or expression should have been used instead. |
| String Function Not Allowed | You have tried to use a string function in an expression. A numeric function or expression should have been used instead. |

| | |
|---|---|
| Structure Start In Wrong Place | You have mis-placed a STRUCTURE START command. STRUCTURE START should be used directly after a FOR/NEXT command, or an IF.. command with a DO action. |
| Structures Are Unbalanced | There are an unequal number of STRUCTURE START and STRUCTURE END commands in the source bank. |
| Synonym Misplaced | You have wrongly placed a synonym word in the vocabulary. Synoyms may only be placed after a "new" word, or another synonym. |
| Too Many Nested Structures | You have nested too many structures. Structures may only be nested up to a maximum of 255 levels. |
| Zero Not Allowed In Exits Table | You have used zero as either a direction number or location number in a location exits table. Direction numbers must be numbers in the range 1 to 99, and location numbers, numbers in the range 1 to 255. |

# INTERPRETER ERROR MESSAGES

This appendix details all possible error messages which may be displayed by the interpreter while a compiled adventure is being played.

When an error is generated, the screen is cleared and an error message displayed along with the details of the source line which caused the trouble. By pressing a key, the player can continue playing the adventure. The interpreter will now jump to the next source line AFTER the one which generated the error (this may or may not be what you intended when defining your source bank logic path).

In some cases, the error may have unforseen implications on the rest of the adventure, and it might be better to rectify it before continuing with the play-testing of the game. On rare occasions, continuing play after an error has been generated may even result in the computer "crashing" or appearing to "hang-up".

It is possible to detect in the source whether an error has been generated, or even to suppress the printing of error messages altogether (see the appendix on system flags for more details on this), but this is not advisable until the adventure has been as thoroughly play-tested as possible.

| | |
|---|---|
| Calculation Result Is Greater Than 255 | A numerical expression resulted in a value greater than 255. This was not permitted for the command being executed at the time. |
| Calculation Result Is Zero | A numerical expression resulted in a value of zero. This was not permitted for the command being executed at the time. |
| Calculator Stack Empty | An attempt was made to unstack a non-existant value from **SAS**'s calculator stack. This may have been caused by a syntax error in a numerical or string expression which was not detected by the compiler. |
| Calculator Stack Full | There was not enough room available on **SAS**'s calculator stack to process the current numerical or string expression. Try making the expression less complex. You are most unlikely to generate this error. |
| Direction Calculated To Be Greater Than 99 | An attempt was made to use or define a direction with a number which a numerical expression calculated to be greater than 99. This is not allowed. |
| Direction Calculated To Be Zero | An attempt was made to use or define a direction with a number which a numerical expression calculated to be zero. This is not allowed. |
| Direction Name Not Defined | An attempt was made to use the name of a direction whose name had not been defined. The error could have been generated by using the D@(n) string function, or by using the **DESCRIBE**, **DESCRIBE LOCATION** or **LIST EXITS** commands. |
| Disk Error | A disk error occured while trying to read or write data to the disk drive. |
| Error In BASIC Command | An error occured while trying to use an **EXECUTE BASIC COMMAND** command. This could indicate that there was a BASIC syntax error in the BASIC command you were trying to execute. |

| | |
|---|---|
| Exits Table Full | There was not enough room to insert a new exit for the current location by using the **ADD DIRECTION** command. The current location may hold a maximum of 20 different exits at one time. |
| Flag Calculated To Be Greater Than 255 | An attempt was made to use a flag with a number which a numerical expression calculated to be greater than 255. This is not allowed. |
| Flag Calculated To Be Zero | An attempt was made to use a flag or system flag with a number which a numerical expression calculated to be zero. This is not allowed. |
| Gosub Stack Full | There is no room available in **SAS**'s Gosub stack to perform a **GOSUB** command. **GOSUB**s may only be nested up to a maximum of 255 levels. |
| Graphic Does Not Exist | An attempt was made to display a location graphic which was not present in memory. (If no graphics had yet been added to the compiled adventure by the graphics extension program, then the **SHOW GRAPHIC** command would have been simply ignored). |
| Invalid Calculator Op-Code | Something has gone seriously wrong while trying to process a numeric or string expression. In theory, this is one error message which you should NEVER see! It almost certainly indicates either a corrupted compiled source file, or a programming "bug" in the interpreter program itself! If this is the case, you should contact me so that I can investigate the problem. |
| Location Description Not Defined | An attempt was made to describe a location (using either the **DESCRIBE** or **DESCRIBE LOCATION** commands) which did not have any location description text defined for it. |
| Location Does Not Exist | An attempt was made to use, define or jump to a location which did not have a description or exits table defined for it. |
| Location Calculated To Be Greater Than 255 | An attempt was made to use or define a location with a number which a numerical expression calculated to be greater than 255. This is not allowed. |
| Location Calculated To Be Zero | An attempt was made to use or define a location with a number which a numerical expression calculated to be zero. This is not allowed. |
| Message Calculated To Be Greater Than 1024 | An attempt was made to print or use a message with a number which a numerical expression calculated to be greater than 1024. This is not allowed. |

| | |
|---|---|
| Message Not Defined | An attempt was made to print or use a message which did not exist. |
| Message Zero Full | There is not enough room for the definition of message zero as specified in the **SET MESSAGE ZERO** command. The length of message zero may not exceed 1024 characters. |
| Movable Object Name Not Defined | An attempt was made to use the name of a movable object whose name had not been defined. The error could have been generated by using the **M@(n)** string function, or by using the **INVENTORY, LIST MOVABLE OBJECTS PRESENT, DESCRIBE** or **DESCRIBE LOCATION** commands. |
| Number Too Big | An addition in a numeric expression resulted in a number greater than 65535. |
| Object Calculated To Be Greater Than 255 | An attempt was made to use or define a movable or unmovable object with a number which a numerical expression calculated to be greater than 255. This is not allowed. |
| Object Calculated To Be Zero | An attempt was made to use or define a movable or unmovable object with a number which a numerical expression calculated to be zero. This is not allowed. |
| Return Without Previous Gosub | A **RETURN** command was encountered, without a **GOSUB** command previously being used to call a subroutine. |
| Score Map Position Calculated To Be Greater Than 100 | The Numeric expression in an **INCREASE SCORE** command calculated the score map position to be greater than 100. This is not allowed. |
| Score Map Position Calculated To Be Zero | The numeric expression in an **INCREASE SCORE** command calculated the score map position to be zero. This is not allowed. |
| Spaces Calculated To Be More Than 1024 | The numeric expression within a **SPC(n)** string function was calculated to be greater than 1024. This is not allowed. |
| Spaces Calculated To Be Zero | The numeric expression within a **SPC(n)** string function was calculated to be zero. This is not allowed. |

| | |
|---|---|
| Subtraction Result is Negative | A subtraction in a numeric expression resulted in a number less than zero. |
| System Flag Calculated To Be Greater Than 30. | An attempt was made to use a system flag with a number which a numeric expression calculated to be greater than 30. This is not allowed. |
| Unmovable Object Name Not Defined | An attempt was made to use the name of an unmovable object whose name had not been defined. The error would have been generated by using the **U@(n)** string function. |
| Verb Calculated To Be Greater Than 255 | An attempt was made to use or define a verb with a number which a numerical expression calculated to be greater than 255. This is not allowed. |
| Verb Calculated To Be Zero | An Attempt was made to use or define a verb with a number which a numerical expression calculated to be zero. |
| Verb Name Not Defined | An attempt was made to use the name of a verb whose name had not been defined. This error would have been generated by using the **V@(n)** string function. |

# "START" STARTER FILE SOURCE LISTING

This appendix gives a complete listing of the commands used in the "START" starter file to provide a basic adventure framework, from which complete adventures may be developed. Users might find this listing handy as a reference source.

## BANK 1

```
Remark : *** START.BK1 ***

Remark : *** Set up INVENTORY messages ***

Set System Flag [10] To [1]
Set System Flag [11] To [2]

Remark : *** Set up SCORE messages ***

Set System Flag [12] To [3]
Set System Flag [13] To [4]

Remark : *** Set up EXITS message for DESCRIBE ***

Set System Flag [14] To [5]

Remark : *** Set up OBJECTS PRESENT message for DESCRIBE ***

Set System Flag [15] To [6]

Remark : *** Now define the initial location and describe it ***

Set Location To [1]
Describe

Jump To Input
```

## BANK 2

```
Remark : *** START.BK2 ***

Remark : *** Convert "look at" into "examine" ***

If Verb = [8] AND
If Preposition = [5] DO
  {
  Set Verb To [11]
  Set Preposition To [0]
  }

Remark : *** If a direction was mentioned, then move ! ***

If Direction > [0] GOTO DIRS

Remark : *** Now jump to the routines dealing with each verb ***

If Verb = [1] GOTO SAVE
If Verb = [2] GOTO LOAD

If Verb = [3] GOTO RAMSAVE

If Verb = [4] GOTO RAMLOAD

If Verb = [5] GOTO QUIT

If Verb = [6] GOTO SCORE

If Verb = [7] GOTO INV

If Verb = [8] GOTO LOOK

Remark : *** For the remaining verbs, either a movable or unmovable
Remark : object MUST be specified ( EXCEPT for EXAMINE ) ***

If Verb <> [11] AND
If Movable Object = [0] AND
If Unmovable Object = [0] GOTO SORRY..

If Verb = [9] GOTO TAKE

Remark : *** For the following verbs, movable objects MUST be carried
Remark : and unmovable objects MUST be present ***

If Movable Object > [0] AND
If Not Carried [MNO] DO
  {
  Print Lower Message [13]
  Jump To Input
  }

If Unmovable Object > [0] AND
If Unmovable Object Not Present [UNO] GOTO NOT.HERE

If Verb = [10] GOTO DROP

If Verb = [11] GOTO EXAMINE

Goto SORRY..

Remark : *** The program jumps here if a direction was mentioned ***

DIRS:

 Remark : *** Print aan error message if there is no exit for the ***
 Remark : *** mentioned direction ***

 If Direction Not Valid DO
   {
   Print Lower Message [11]
   Jump To Input
   }

 Remark : *** The direction was valid, so move to the new location ***

 Act Upon Direction

 Remark : *** SAVE command routine ***
```

```
SAVE:
  Print Lower Message [7]
  Pause
  Save
  Jump To Input

  Remark : *** LOAD command routine ***

LOAD:
  Print Lower Message [7]
  Pause
  Load
  Describe
  Jump To Input

  Remark : *** RAMSAVE command routine ***

RAMSAVE:
  Ramsave
  Jump To Input

  Remark : *** RAMLOAD command routine ***

RAMLOAD:
  If System Flag [3] = [0] DO
    {
    Print Lower Message [8]
    Jump To Input
    }
  Ramload
  Describe
  Jump To Input

  Remark : *** QUIT command routine ***

QUIT:
  Score
  Print Message [9]
  Pause
  Quit

  Remark : *** SCORE command routine ***

SCORE:
  Score
  Jump To Input

  Remark : *** INVENTORY command routine ***

INV:
  Inventory
  Jump To Input

  Remark : *** LOOK command routine ***

LOOK:
  Describe
  Jump To Input
  Remark : *** TAKE command routine ***

TAKE:

  Remark : *** The movable object MUST be present ***

  If Movable Object = [0] GOTO SORRY..

  If Movable Object Not Present [MNO] GOTO NOT.HERE

  Remark : *** Now pick up the object ***

  Take Movable Object [MNO]
  Gosub OK
  Jump To End

  Remark : *** DROP command routine ***

DROP:
  Drop Movable Object [MNO]
  Gosub OK
  Jump To End

  Remark : *** EXAMINE command routine ***

EXAMINE:

  Remark : *** Insert any responses to objects examined here, followed
  Remark : by a JUMP TO END command ***

  Remark : *** Message 15 will be printed as a default ***

  Print Message [15]
  Jump To End

  Remark : *** The program will jump here if an input is not understood
  Remark : for some reason ***

SORRY..:
  Print Lower Message [10]
  Jump To Input

  Remark : *** The program will jump here when an object is not ***
  Remark : *** present ***

NOT.HERE:
  Print Lower Message [12]
  Jump To Input

  Remark : *** Subroutine to print an "OK" message ***

OK:
  Print Message [14]
  Return

  Remark : *** Routine to print "You cannot do that" message ***

CAN'T:
  Print Lower Message [16]
  Jump To Input
```

## BANK 3

```
Remark : *** START.BK3 ***

Remark : *** Insert any low priority conditions here ***

Remark : *** Now deal with any remaining multiple commands ***

Continue With Input
```

## BANK 4

```
Remark : *** START.BK4 ***

Remark : *** Describe the new location as we enter it ***

Describe

Remark : *** Insert any local conditions here ***

Remark : *** Now jump to bank 3 ***

Jump To End
```

# SAS SPECIFICATIONS

## System requirements

**SAS** requires 512k internal RAM, ROM version 2.1 or later and at least one disk drive fitted. If present, **SAS** will recognise and use an extra disk drive, the SAM mouse and a 1Mb external memory interface (requires MASTERDOS).

Compiled adventures may be run on either a 512k SAM, or a 256k machine if the adventure is small enough.

Up to 727040 bytes are available for adventure "source" (794624 bytes if a 1Mb external memory interface is used)

## Source code banks

Four source banks are available. Each bank has a specific purpose. Bank 1 is used for game initialisation, Bank 2 for input evaluation, Bank 3 for low-priority conditions, and Bank 4 for local conditions. Each source bank may hold up to 4096 command lines. Each command line uses 64 bytes of source space (irrespective of line length).

Each source bank may hold up to 4095 line labels.

## Vocabulary

Words consist of five different "classes"; Directions, Verbs, Movable Objects, Unmovable Objects and Prepositions. Each class of word may have up to 255 different entries, each with up to 255 different synonyms.

The vocabulary may have up to 2048 entry lines, with space for up to 2042 words. Each vocabulary entry line will use up 16 bytes of source space. Each word may be up to 15 characters long.

The user can define the number of characters in each word of the player's input that will be compared to each vocabulary entry.

## Messages

Up to 1024 messages are available. Each message may be up to 256 characters in length. Each message defined will use up 256 bytes of source space.

In addition, there is a special message (message 0) which can be constantly re-defined throughout the adventure. Message 0 may be up to 1024 characters long.

## Locations

Up to 255 locations may be used. Each location may have a description up to 256 characters long, and have up to 10 exits defined for it. Each location may actually hold up to 20 exits at one time, the extra exits being added by the **ADD DIRECTION** command.

Each defined location will use up 306 bytes of source space.

## Movable Objects

Up to 255 movable objecte may be defined. Each movable object may have a name up to 15 characters long. In addition, the movable object name may also have a prefix. There are four prefixes available - "a ", "the ", "some " and "an ". Each movable object defined uses up 16 bytes of source space.

## Unmovable Objects

Up to 255 unmovable object names may be defined. Each unmovable object may have a name up to 16 characters long. Each unmovable object defined uses up 16 bytes of source space.

## Verbs

Up to 255 verb names may be defined. Each verb may have a name up to 16 characters long. Each verb defined uses up 16 bytes of source space.

## Directions

Up to 99 directions may be defined. Each direction may have a name up to 16 characters long. Each direction defined uses up 16 bytes of source space.

## Flags

255 flags are available for use by the user. Each flag may contain a value in the range 0 to 255. In addition, there are 30 system flags used by the interpreter.

## Graphics

Location graphics are added to the compiled adventure using the graphics extension program. Graphics are converted from FLASH screens or normal SCREEN$ files in either MODE 3 or MODE 4 and are automatically compressed.

The number of graphics which can be added will vary according to the amount of free memory available and how well the graphics compress, but the interpreter will support a maximum of 255.

# INDEX

ALSO AVAILABLE FROM AXXENT SOFTWARE

# SHERIFF GUNN

**SHERIFF GUNN** is the first SAM adventure yet to make FULL use of the SAM Coupe's graphics, using 16 colour MODE 4 graphics, and 64 column MODE 3 text to provide the perfect display for your exploration of the rootin'-tootin' world of Sheriff Gunn.

The game is a two part adventure which sees YOU taking the part of the sheriff in his attempts to catch the infamous outlaw Poisonous Pete (he's hiding somewhere in tham thar' hills). On your way through over 80 locations, you will meet many other inhabitants of the town going about their business, but will they help or hinder your progress ? There are lots of tricks and puzzles to solve before you finally get your chance to bring in Pete, DEAD or ALIVE.

**SHERIFF GUNN** was not written using **SAS**, but nethertheless features an advanced parser capable of accepting multiple inputs using AND as well as RUN TO, FIND, GET ALL, DROP ALL, RAMSAVE and RAMLOAD, PALETTE, FONT SELECTION, SAY TO and much much more!

**SHERIFF GUNN** is available for only 9.99 (please make POs/cheques payable to C. Jordan) from COLIN JORDAN, 89 TREWENT PARK, FRESHWATER EAST, PEMBROKE, DYFED, SA71 5LP.

**SAS** registered users (see form on next page) may claim a special discount of 1.00 on the above price.

---

If you're interested in playing or writing adventures on the SAM, then why not get in touch with the SAM COUPE ADVENTURE CLUB. The club prouduces a bi-monthly club disk magazine which includes all the latest news in the SAM adventure world, including hints and tips on how to get the most out of **SAS**.

You can obtain a copy of the latest issue free of charge in return for a blank formatted disk and an S.S.A.E.

For your copy and more details contact :

Phil Glover
S.C.A.C.
43 Ferndale Road
Hall Green
Birmingham
West Midlands
B28 9AU