ALPS

Adventure Language Programming System

Alpine Software

PO BOX 25, Portadown, CRAIGAVON BT63 5UT

Adventure Language Programming System

Alpine Software

ALPS

Adventure Language Programming System

CONTENTS

APPENDICES

COPYRIGHT NOTICE

INTRODUCTION

The Adventure Language and Programming System or 'ALPS' consists of a 16K eprom which provides a program language specifically designed for the purpose of writing compact and fast machine code adventure games. The eprom also includes routines for setting up and editing the adventure data base and a dedicated text editor to enable easy entry of the room descriptions and other messages. A built-in text compressor ensures that as much text as possible can be fitted into the available memory.

A separate utility program is supplied to enable the final adventure game to be produced in a stand-alone form. This does not require the ALPS rom to be present in the host computer and programs may thus be given to friends or sold commercially. In either case no restrictions are placed on the use of the material, provided you state that ALPS was used to create the adventure.

FITTING

The eprom should be fitted in one of the paged rom sockets in the computer. The location of these depends on the model of computer and is given on a separate sheet. When the computer is switched on or if *HELP is entered, ALPS will report its presence with a message of the form:

ALPS - Version X.X

Please quote the version number in any correspondence. To begin using ALPS, enter the language by typing *ALPS and press <RETURN>. ALPS will announce itself and you will see an arrow and a flashing cursor. The arrow is the ALPS command prompt. The data and text editors are entered by pressing one of the function keys. A function key label is supplied and this should be slipped under the plastic strip above the red keys.

WRITING ADVENTURES WITH ALPS

The game consists of the following elements:

a. a database of information about:

1. the objects and their properties
2. the rooms and their connections
3. the verbs which the program recognises
4. the text of room and object descriptions and system messages

b. a program to control the actions that take place in the game.

PROGRAMMING THE GAME

Before you start to actually program your game or set up the data base it is best to plan the outline and jot down the plot ideas with pencil and paper. This section provides a brief overview of the programming process using ALPS. More detailed instructions for each stage are given later. At any stage you can save your work by entering the command SAVE "name" where "name" is a valid filename of your choice.

This will automatically save five files using the following directories (assuming a disk system):

```
Objects data in the O. directory          - filename "O.name"
Rooms data in the R. directory            - filename "R.name"
ALPS program in the current directory     - filename "name"
Text data in the T. directory             - filename "T.name"
Dictionary data in the D. directory       - filename "D.name"
```

For this reason the O, R, T and D directories should be regarded as reserved for ALPS use. That is, do not change the current directory to O, R, T or D. Tape users may like to note that the order of saving and loading is as given in the list above.

The "STARTER" files on the system disk provide a basic framework program (giving the movement verbs plus LOOK, INVENTORY, TAKE, DROP, SAVE and RESTORE) that will suit any adventure and includes messages 1-20. If you type LOAD "STARTER" before entering the rest of your data, the simple program will allow you to wander around your environment, pick up and drop the objects and generally test the layout before you add the various puzzles and other features to the program.

You should start by entering a list of the object names, using the OBJECT command. For example: OBJECT "LAMP","TORCH" enters an object which the player may refer to as a 'lamp' or a 'torch'. You will need to allocate a message number for the description of each object, so it's a good idea to jot these down on your sheet of paper alongside each object's name. ALPS will allocate a number to each object as its name is entered and the object data editor may be used to enter the data for each. The object data editor is entered by pressing <f4>.

Now you can set up the rooms data base which contains the details of your map and the message numbers for each room's description. The room data editor is entered by pressing <f5>.

The text of each message that will be required is entered using the text editor. The text may need to be compressed at some stage, if you start to run out of memory, and this compression is done with the aid of the COUNT and REPLACE commands.

You are now ready to write the actual game program using the ALPS language. The program consists of some initialising instructions followed by a loop which inputs the player's instructions and carries them out, subject to any constraints you care to program. Movement and behaviour of other 'characters' can be programmed in this loop also. The main loop is followed by the verb definitions which define the action to be taken when the player enters that verb, and a number of exit routines which can be defined so that certain conditions must be satisfied before a move is allowed through a particular exit in a given 'room'. The complete listing for a small sample adventure, "TINY", is given in Appendix D, along with explanatory comments for each instruction.

You may test your game at any stage by entering 'RUN'. The prompt will change to a colon (:) to distinguish it from the command level prompt. Because you will alter the database as you play the game, it is essential to be able to restore it afterwards. There is insufficient memory to do this automatically so the simplest way is to re-LOAD all the files again. This is rather slow for cassette users so

a better alternative is to save just the room and object data. Using a "SAVE" verb such as that defined in "STARTER", you can save the initial data by running the game, entering "SAVE" and giving "INIT" as the filename. You must do this before giving any other commands which could alter the database.

The game should then be carefully tested to ensure it performs as expected and that the adventure is solvable. Any corrections require alterations to either the data base or the ALPS program. To return to the command level simply press <ESCAPE>. If an error occurs you will also be returned to the command level where any necessary editing can be done.

To restore the database after a test, run the program again and use the "RESTORE" verb to reload the "INIT" file, pressing <ESCAPE> to return again to the command level.

As an alternative, particularly in smaller games, you could write a procedure to reset the database each time the game is run. This would need to replace all objects in their starting rooms and initialise those object and room states and flags that may have been altered.

When you have a fully de-bugged game you can produce a version of it which can run independently of the ALPS system by using the 'CREATE' utility on the system disk/tape.

THE OBJECT EDITOR

When you press <f4> the computer will enter the object editor, with the data for the current object displayed as shown below:

```
Object: LAMP (1)

Flag  Value
 7      1
 6      1
 5      0
 4      0
 3      0
 2      0
 1      0
 0      0
State:  0
Room :  0
MSG# :  0
```

If you want to enter the editor at a particular object, type the number of the required object before pressing <f4>.

At the top of the screen is the object name, if defined, and its number. In the centre of the screen are the current data values for this object. These consist of 8 FLAGS, which can each have a value of 1 or 0, and three other data items which may have values between 0 and 255.

The first two flags have reserved meanings as follows:

If Flag7=1 this indicates the object is a light source
If Flag6=1 this indicates the object can be taken

The other flags are free for you to use as you wish. For example, you may wish to use Flag5=1 to represent an object which breaks if it is dropped.

Below the flags on the editor screen are the object's current STATE (see 'STATES AND FLAGS' later), its ROOM number (where it will be found when the game is played) and the number of the message which describes it (MSG#).

The following keys are used to move around and alter the object list:

Cursor Left	- decrease current data value by 1
SHIFT+Left	- decrease current value by 10
Cursor Right	- increase data value by 1
SHIFT+Right	- increase data value by 10
Cursor Up	- move to preceding data item
Cursor Down	- move to following data item
CTRL+C	- clear current value to zero
CTRL+Left	- move to previous object
CTRL+Right	- move to next object
Return	- exit from editor
Escape	- exit from editor without updating changes

The allowed range of states and rooms is 0-255 and the range of message numbers is 0-65535. Object numbers outside the range 1-180 are not allowed.

THE ROOM EDITOR

This acts in a similar manner to the object editor but there are additional values required for each room. The flags, state and message values are handled in the same way. There is one reserved flag: again Flag7=1 indicates there is ambient light present in the room. The remaining 12 values are divided into two groups of six. The first group represents the destination room number for each of the directions N,S,E,W,U and D with a zero entry indicating 'no exit'.

Note that room 1 is the player - when he takes an object it is in fact moved to room 1. Thus an inventory command is easily implemented by a single DESCRM(1) (DESCribe RooM) instruction. Clearly the description of room 1 needs to be something along the lines of 'You are carrying ' so that DESCRM(1) produces:

'You are carrying a lamp, a gold coin....'

It is sometimes useful to include an extra room with no exits or entrances! This will prove useful as a convenient place to 'lose' objects, since the player will never be able to reach this room.

To enter your map into the rooms data base, press <f5> to select the room data editor. (If you wish to edit a particular room you can type the room number before hitting <f5>. The maximum number of rooms depends on the amount of memory available: 175 rooms on a model B with DFS and 255 on a Model B with cassette filing system or a Master series computer). You should see the data display for room 1, as follows.

```
Room: 1

Flag  Value
 7      1
 6      0
 5      0
 4      0
 3      0
 2      0
 1      0
 0      0
State:  0
MSG# :  14
DIRN     TO          EXIT ROUTINE
NORTH:   0                0
SOUTH:   0                0
EAST :   0                0
WEST :   0                0
UP   :   0                0
DOWN :   0                0
```

As explained above this is the 'player' room so the only important data are flag 7, which must be set, and the message number. Set the flag and message number, using the cursor keys, as described in the object editor section.

The second group of six values is headed 'EXIT ROUTINE'. If we wish to make exit in a particular direction conditional on certain tests or if something must happen when a player goes through a certain exit, we will define a numbered exit routine in our ALPS program and insert its number in the appropriate place in the rooms data base (you can have any number from 1-255, the only constraint is that each routine must have a different number). Note that the same exit routine can be used on several exits, perhaps in different rooms, provided it is suitably designed. You should note that although the player may be prevented from moving by an exit routine, the exit must still have a valid, non-zero destination value.

Note that you will cause an 'Undefined room' error, when a game is RUN, if you move to a room which has not been included in the database. When a room's data is viewed using the room editor this 'creates' all rooms up to and including that room. For example, when room 56 is examined with the editor, this will create rooms 1-56 as valid rooms.

THE TEXT EDITOR

Now we come to designing the messages, where you can allow your creative talents full rein and give your adventures the 'atmosphere' which is essential to a good game. Messages are the only means by which the computer can let the player know what is happening, both in terms of describing what he can see and in responding to his inputs. All such descriptions and responses are stored as numbered messages and may be printed by the 'MSG(m)' instruction where 'm' is the number of the required message.

Messages 1 to 10 must have the following meanings, though of course the exact wording is at your own discretion:

MSG	Meaning
1	nothing
2	You can see (Note 1)
3	Ok
4	I don't see that here!
5	You have that already!
6	You can't take that!
7	It is pitch dark
8	You don't have that!
9	You can't carry any more!
10	You can't go that way!

Note 1: This message should have a trailing space. After a room description is given, if there are objects present, message 2 will be printed followed by the list of objects. The object list is printed using the following format:

You can see a lamp, a revolver, etc... and a bunch of keys.

You will note that each object description should be in lower case letters and have no full stop. The commas, the word " and " and the full stop will be added by the interpreter.

Note 2: If you decide to use the exit reporting option - see entry for 'OPT' in the 'ALPS KEYWORDS' section - you will also need to reserve messages 11 and 12 (both must have trailing spaces):

11	The only exit is
12	There are exits

To enter the text editor at the current message press <f6>. You can go straight to a particular message by typing its number before hitting <f6>. The number of the message being edited is given at the top of the screen.

To try out the editor, hit <f6> and you should see a blank screen with 'MSG#: 1' at the top. To enter message 1, just type the word 'nothing'. If you make any mistakes you can use the cursor keys to position the cursor as required for editing. The delete key deletes the character to the left of the cursor. To insert characters just move the cursor to the required position and type as normal. There is no need to worry about words being split at the ends of lines as this will be taken care of automatically by the message display system which operates when the game is being RUN or played.

Use of cursor keys in the text editor

In the text editor the cursor keys have the following effects:

Right	- move cursor right one character
SHIFT+Right	- move cursor to right end of line
Left	- move cursor left one character
SHIFT+Left	- move cursor to left end of line
Down	- move cursor down one line
SHIFT+Down	- move cursor to end of message
Up	- move cursor up one line

SHIFT+Up - move cursor to start of message
CTRL+Right - move to next message (creates new one if necessary)
CTRL+Left - move to previous message
CTRL+Up - move to first message
CTRL+Down - move to last message
CTRL+S - Swap case of text
CTRL+E - Examine/Edit switches
DELETE - Delete character to left of cursor
ESCAPE - Exit from text editor.

CTRL+S changes upper case characters to lower case and vice versa. Holding down the keys will run the cursor quickly through the text, swapping cases continuously.

CTRL+E allows you to look at, and alter if necessary, the message switches. I will leave the explanation of that topic until the 'STATES AND FLAGS' section. For now, if there are no switches set for the current message, you will just see the text of the message replaced by

SWITCH VALUE

at the top of a blank screen.

If you wish to insert some switches press <COPY> for each switch to be inserted (up to a maximum of 10), otherwise press <RETURN> which will take you back to the text. Use the cursor keys, with shift for larger changes, as described in the OBJECT EDITOR section, to move through and alter the switch list. If you need to delete a switch just move the cursor under it and press <DELETE>.

If the message you have selected for editing has been tokenised - see the next section - it will be automatically expanded to enable editing to be carried out. It may happen that, after editing, the new version of the message is longer than the original and there may be insufficient memory left to insert it back into the text again. If this happens the computer will beep when you exit from the editor by pressing <ESCAPE> or if you move to a different message. Any changes will not have been carried out and the original message will be left intact.

Special message effects

There are two reserved characters which trigger special effects when the messages containing them are printed out by the interpreter. The first is the % character which must be immediately followed by one of the letters A-z. This has the effect of causing the message routine to print out the value of the corresponding variable. Thus to print out a score, stored in S%, you could use a message:

'You have scored %S points.'

which would produce 'You have scored 55 points.' when printed out, if S%=55. (NB there must be no space between the '%' and the 'S').

The second reserved character is '$' which may be followed by 'V' or 'O' and will cause the last-entered verb or object to be substituted. For example, if the player entered: 'WEST' or 'W', the message 'You can't go $V from here' would be printed as 'You can't go west from

here'. (Note the automatic conversion to lower-case and the expansion of the abbreviated verb to its full form).

Any other character (including a space) after the '$' will cause it and the following character to be replaced on printout by the last word that the player entered. For example you can implement the usual 'SAY' command using a message such as:

```
OK, "$ !"
```

If the player enters 'SAY HELLO' this message would produce

```
OK, "HELLO!"
```

when it is printed since 'HELLO' is the most recent word entered.

The insertion of a semicolon (;) in a message will cause the next message printed by the 'MSG' instruction to continue on the same line, immediately following the previous message. If the message does not contain a semicolon, 'MSG' will start printing the next message on a new line.

THE TEXT COMPRESSOR

Memory is always at a premium in adventure games, mainly because the text takes up so much of it. Normal text is stored as a sequence of ASCII codes, one for each character of the message. All printable characters have codes which are less than 128 (&80). This means that it is possible to replace selected groups of characters, words or whole phrases by single-byte tokens, whose values are between 128 and 255. The characters which are replaced have to be stored in a list, called the 'dictionary'. When printing out the text, if the computer detects a token code it looks up the corresponding dictionary entry (the original characters) and prints that instead of the token. For example, we might replace all the "you" groups (each taking up 3 bytes) with a single '128' byte, saving two bytes per "you" replaced. Of course one copy of the "you" has to be kept in the dictionary but, with careful selection of which groups to replace, you can make large memory savings. This is helped by the fairly specialised vocabulary used in adventure games. A small number of words are repeated fairly often within the text of a particular game.

The text compression is carried out with the aid of two ALPS commands: COUNT and REPLACE. Each is followed by a string, enclosed in quotes, and each responds with the number of occurrences of that string within the text. In the case of REPLACE, the error message 'String exists' is given if the replace string is already in the dictionary, otherwise you will be told how many bytes would be saved if the string were replaced and, if there would be a real saving, you will be asked

```
Replace? (Y/N)
```

If you want to go ahead with the replacement hit <Y> and the text will be very rapidly scanned and all occurrences of the string will be replaced by the next available token. If you decide not to replace this string just hit <N>. If you have used all the tokens REPLACE will report 'All tokens used'. If there is insufficient memory to store the string in the dictionary the 'No room' message will be given.

As mentioned earlier, each time you edit a message, even if you only look at it without making any changes, the message is expanded back to its full length. When new messages are entered they will not at that stage be compressed. Therefore, every so often, it is a good idea to recompress the text to prevent it exceeding the allocated memory space. This is done by pressing <f7> which initiates a scan of the whole text. This involves the computer checking each entry in the dictionary and replacing any occurrences throughout the entire text. During scanning the screen will show how it is progressing, giving each token value and dictionary entry as they are checked. If you have a printer and you would like a listing of the dictionary, press CTRL-B before pressing <f7>. Don't forget to disable the printer afterwards by pressing CTRL-C.

You will obviously want to identify which replacements will give the greatest savings. You will need to note down which strings you will want to replace. Don't overlook the fact that the most common character is the space! Of course you can't tokenise single spaces but the spaces at the ends of words are well worth getting rid of. This can be done by replacing the last letter, and the following space, of as many words as possible. The most common last letters are: e, s, t, n, d, r and y.

With careful selection of dictionary entries you should be able to achieve a compression rate (that is the ratio of the compressed length to the original length) in the region of 55 to 60%. For example, the compression rate in 'Mystery Mission' is about 58%, the compressed text being just over 13,000 characters compared with an original text length of over 22,000 characters. This does not include the effect of switched messages which can provide further memory savings.

WRITING THE 'ALPS' PROGRAM

When you have completed the setting up of your adventure's data base the time has come to write the program itself. This is done using the language 'ALPS' which has been specifically designed to provide those facilities required for adventure programming. In consequence it is a simple language to use for this purpose and it produces a compact intermediate code which runs very quickly under the control of the machine code interpreter program, 'ASI', as the adventure is being played.

You should find ALPS very easy to use. There are, however a number of important differences between ALPS and other languages you may be familiar with, such as BASIC.

One major difference between BASIC and ALPS is that there are no line numbers in the latter. Thus there are no GOTO's but a full IF...THEN...ELSE structure together with a REPEAT...UNTIL structure ensures that these are not required.

Several ALPS statements may be placed on the same line if they are separated by a space or a colon (:). These separators are not stored so when you LIST the line again you will see that it is listed with spaces between keywords where you may have entered colons. Each line is checked as soon as you press <RETURN> and any errors are reported at once. This will prevent most errors from reaching the finished program but some errors cannot be trapped at the compilation stage. For example 'Too many REPEAT's' and 'No REPEAT' will only occur when

the game is RUN - See the notes on 'ASI' errors in Appendix B.

In the case of 'Syntax error' the offending word is printed so you can see what you did wrong. Spaces are important - if you type 'EDITO' instead of 'EDIT O' you will get

'Syntax error: EDITO ?'

See Appendix A for a full list of error messages.

All your programs will consist of a fairly short main section to input the player's commands and carry out any necessary actions, followed by the definitions of verbs, exit routines and procedures. These are each explained below and you should refer for further details to Appendix D. This gives a full listing of the example program, "TINY", together with explanations of what each line does.

VERB DEFINITIONS

The "STARTER" file provides the basic verbs needed in any adventure. The definitions of 'TAKE' and 'DROP' are simple and can be handled just using the inbuilt TAKE(O) and DROP(O) statements. The zero object value causes these to act on the last object named by the player (stored in O%). Other more complex adventures may make taking an object conditional on certain tests and dropping objects in certain rooms may 'lose' them. These will obviously require extended definitions of the two verbs.

EXIT ROUTINES

In almost all adventures, whether the player is allowed to go through certain exits, or not, is made to depend on him having satisfied certain conditions. In ALPS the tests of these conditions are performed by a numbered 'exit routine' which is executed every time he tries to go through an exit controlled by that routine. After the exit routine is executed, the computer checks the value of G%. Only if this is zero will the requested move be permitted, otherwise nothing happens. G% should be set to zero each time around the program's main loop by the EQUATE(G%,0) instruction.

Aside from controlling movement through exits, exit routines can also make certain things happen when the player goes through the exit. An example would be to have a can of paint fall over him and his belongings.

PROCEDURE DEFINITIONS

Procedures in ALPS act as subroutines which can be used to carry out certain actions at various points in the game, without the need to type the same code many times. Please refer to the 'ALPS KEYWORDS' and 'USING PROCEDURES' sections for further details of their use.

CREATING A GAME FILE

All the files for two example games: 'TINY' and 'MISSION' are supplied on the system tape/disk and may be loaded into ALPS by typing LOAD "TINY" or LOAD "MISSION", respectively.

To turn these into a complete, stand-alone adventure game you need to run the game file creator program - "CREATE" - supplied on the system tape/disk. Disk users can access this from the menu which is obtained by booting the disk. Tape users should type CH."CREATE". The rest of these instructions assume a disk system but tape users should simply follow the on-screen prompts. You will require the system tape, the source tape with your ALPS files and a blank destination tape on which the game will be saved.

The menu displays the currently selected drive number for the data files. The default is drive 0 but you may change this to 1, 2 or 3 by pressing the appropriate number key. When the 'CREATE' option is selected you will be asked to supply a title for your adventure and your own name. You are then prompted for a filename to be used for your game and the filename under which you saved your ALPS files. To use the default values shown for these, just press <RETURN> when prompted.

If you are using a Master computer you will be given the option of creating a version to run on Master or tape-based machines only. This allows more memory to be made available. However if you want your games to be compatible with all computers you should not select this option.

You will then be asked to:

Insert destination disk in drive 0
and hit a key

The **destination** disk is the one on which you want the final game produced. Note that it must be in drive 0, even if you have a dual drive system. The system disk is no longer required and may be removed. When you have done this you will then be prompted to:

Insert source disk in drive D
and hit a key

where D is the drive selected for your data. The **source** disk is the one on which your ALPS data files are stored. The program will now load each file in turn, displaying the filenames of each. It will check that the files are of the correct type and stop if any are incorrect. If the file is too long to fit into the remaining memory the program will report 'File too long!' (not applicable to tapes). In either case you may need to return to ALPS for any editing that is required.

After loading the rooms data there will be a pause of a few seconds while the data is compressed. The screen will display:

PACKING - PLEASE WAIT

while this is happening, and this will change to 'PACKED' when the process is complete. If you are using tapes and you do not have motor

control please pause the tape until the packing is completed.

If all files are correctly loaded the program then saves a compacted file which includes all the data and program files, together with the machine code which is needed to run it. Make sure you insert the destination data disk (in drive 0) when requested. Two files are created on the destination disk: a short loader program and the game itself. The loader will have the filename you supplied, and the game itself will have the 'G.' directory added to its filename. The two files for a game called "QUEST" would thus be "QUEST" and "G.QUEST". The game would be run by typing: CH. "QUEST".

ALPS KEYWORDS

This section explains all the instructions that comprise the ALPS language. These are used to define what happens during the game, for example when the player types 'THROW BOMB'.

KEYWORD	DESCRIPTION
COMMANDS	
These do not appear in the program, and are mainly listing and editing commands.	
LIST [<linrng>|O|V|D|E|P]	List Program or Table. LIST O and LIST V list the object and verb names. LIST D <str> lists the definition of the specified verb. LIST E <num> and LIST P <num> list the specified exit routine and procedure. respectively. The D, E and P options list up to the next 'END' which is on a line of its own.
LISTO	Set list options: LISTO 1 - Display line numbers. LISTO 2 - display object names. LISTO 4 - display verb names. Options may be combined eg LISTO 3 displays with line numbers and object names.
LVAR	List integer variables
EDIT [<line>|O|V|D|E|P]	EDIT on its own or with a line number enters the program editor. EDIT O <str1>,<str2> replaces <str1> with <str2> in the object table. EDIT V does the same for the verb table. EDIT D <str> enters the editor at the first line of the specified verb. EDIT E <num> and EDIT P <num> enter the editor at the first line of the specified exit routine and procedure.
NEW	Delete all data and the program
SAVE <fsp>	Save program and data tables
LOAD <fsp>	Load program and data tables (deletes any existing data)
RUN	Execute the program in memory
STATUS	Display system status information. This consists of one line of information for each section of the database and program: O,V,T,D,R and P which represent the object and verb lists, the text and dictionary

	data, the room data and the program. Except for P, the display shows how many items have been defined, the maximum number of bytes available for that section and the free bytes remaining. The R size and free data gives the maximum number of rooms and the number remaining, not the number of bytes.
OBJECT <str>{,<str>}	Add object/s to list. Each object name must be in quotes. Several names may be supplied, separated by commas and each will be treated as a synonym of the first name in the list.
SYNONYM O¦V <str>,<str>	Create Object/Verb synonym. For example SYNONYM V "TAKE","GET" makes 'GET' behave the same as 'TAKE'.
COUNT <str>	Displays the number of occurences of the specified string within the text.
REPLACE <str>	Replaces the specified string throughout the text with the next token value. Displays possible saving and asks for confirmation before replacing.

OPERATORS

AND	AND result of two conditions. Both must be true before the action will be taken.
OR	OR result of two conditions. If either condition or both are true the action is taken.

PROGRAM STRUCTURES

IF <expr>	Part of IF...THEN...ELSE
THEN	Part of IF...THEN...ELSE
ELSE	Part of IF...THEN...ELSE
REPEAT	Marks start of loop structure
UNTIL <expr>	Marks end of loop. You can 'nest' up to 16 loops inside each other.
DEFINE <str>	Marks the start of a verb definition. Also enters the verb name in the verb table. You cannot include any other instructions on the same line as a DEFINE instruction.
EXIT(E)	Defines the start of exit program E
DEFPROC(P)	Defines the start of procedure P
END	Marks the end of verb/exit/procedure definitions. It may occur several times within each, as needed.

FUNCTIONS

Functions return either a true or a false value. They are used to control the actions to be taken by the program by writing:

IF <function expression> THEN <action list 1> ELSE <action list 2>

If <function expression> is true then <action list 1> will be performed, otherwise <action list 2> will be performed. Both THEN and ELSE are optional.

For example:

```
  IF CARRY(3) AND ROOM(91) MSG(18) ELSE EQUATE(F%,9) MSG(37)
```

Which means: if the player is carrying object 3 and he's in room 91 then message 18 is printed, otherwise F% is set equal to 9 and message 37 is printed.

VERB(V)	TRUE if verb V was typed
NOTVERB(V)	TRUE if verb V was not typed
OBJ(O)	TRUE if object O was typed
NOTOBJ(O)	TRUE if object O was not typed
RSTATE(R,S)	TRUE if room R's state is S
OSTATE(O,S)	TRUE if object O's state is S
NOTRSTATE(R,S)	TRUE if room R's state is not S
NOTOSTATE(O,S)	TRUE if object O's state is not S
CARRY(O)	TRUE if object O is carried
NOTCARRY(O)	TRUE if object O is not carried
ROOM(R)	TRUE if current room is R
NOTROOM(R)	TRUE if current room is not R
HERE(O)	TRUE if object O is present
NOTHERE(O)	TRUE if object O is not present
RFLAG(R,F,v)	TRUE if room R's Flag F has value v
OFLAG(O,F,v)	TRUE if object O's Flag F has value v
YESNO(M)	Do MSG(M), wait for an input from the keyboard and return TRUE if Y or y entered
TRUE	Always returns a true result
FALSE	Always returns a false result
EMPTY	TRUE if text buffer is empty
NOTEMPTY	TRUE if buffer is not empty

The next four functions are used to compare the value of one integer variable with either a number or a second variable.

EQ(<var>,<num>¦<var>)	TRUE if the values are EQual, eg 'IF EQ(A%,4)...' is true if A% is equal to 4
LT(<var>,<num>¦<var>)	TRUE if the variable is Less Than the number or second variable, eg IF 'LT(F%,B%)...' is true if F% is less than B%
GT(<var>,<num>¦<var>)	TRUE if the variable is Greater Than the number or second variable
NE(<var>,<num>¦<var>)	TRUE if the values are Not Equal

STATEMENTS

CALL(addr)	Loads A,X and Y registers from a%,x% and y%, then calls a user supplied machine code routine which starts at the address given (in decimal) by the argument. The routine must be assembled and *LOADed in a safe location and must end with an RTS instruction. The ALPS program will then continue with the instruction following the CALL instruction.
OPT(0¦1)	Exit report option: 1 = Do report; 0 = Don't. If exit reporting is to be used, messages 11 and 12 must be reserved (see TEXT EDITOR section) and the first six verbs must be north, south, east, west, up and

	down, in that order.
MSG(M)	Print message M
DESCRM(R)	Describe room R
DESCOB(O)	Describe object O
SETRS(R,S)	Set room R's State to S
INCRS(R)	Increment room R's State
DECRS(R)	Decrement room R's State
SETOS(O,S)	Set object O's State to S
INCOS(O)	Increment object O's State
DECOS(O)	Decrement object O's State
SETRF(R,F,v)	Set flag F of room R to value v
SETOF(O,F,v)	Set flag F of object O to value v
STOP	End game and return to the command level, if game is being RUN, or to BASIC if the stand-alone game is being played
GOTO(R)	Move player to room R
MOVE(d)	Move player in direction d. The values of d for each direction are: N=1, S=2, E=3, W=4, U=5 and D=6.
MOVEOB(O,R)	Move object O to room R
DEST(d)	Sets D% equal to the destination room that would be arrived at by taking direction d from the current room. If d=0 then the statement substitutes the value of d%.
TAKE(O)	Take object O. Messages 4, 5 and 6 are printed if the stated object isn't here, you already have it or if it is not takeable.
DROP(O)	Drop object O. Message 8 is printed if you're not carrying the object specified.
INPUT	Get player's input into buffer
GETV	Get a verb from the input buffer and convert into a verb number in V%. If no verb was recognised V% will contain zero.
GETO	Get an object from the input buffer and convert into an object number in O%. If no object was recognised O% will contain zero.
PERFORM	Carry out player's action as given by verb number in V%. You can force a verb to be PERFORMed by placing its number in V% and executing a PERFORM.
RESTORE	Reset pointer to start of input buffer
PROC(P)	Execute procedure number P and then continue with next instruction
INOUT(M,f)	Does a MSG(M) then either LOADs (f=0), or SAVEs (f=1), the game data. These are used for 'save' and 'restore' verbs. Note that data saved using the rom is not compatible with stand-alone data files.

The next five statements alter the value of a specified integer variable. See the 'CHANGING VARIABLE VALUES' section for more details.

INC(<var>)	Add 1 to variable
DEC(<var>)	Subtract 1 from variable
ADD(<var>,<num>¦<var>)	Let <var> = <var> + <num>¦<var>
SUB(<var>,<num>¦<var>)	Let <var> = <var> - <num>¦<var>
EQUATE(<var>,<num>¦<var>)	Let <var> = <num>¦<var>

Notes

(1) See APPENDIX C for details of minimum abbreviations.

(2) Meaning of symbols and allowed values:

¦	indicates alternatives - choose one.
[]	enclose optional items
()	indicates possible repetition of enclosed symbols zero or more times.
O,	object number, 0-180. If O=0 the function is evaluated for the current object. This is stored in O% and may be manipulated if required.
R,	room number, 0-175 (Model B) or 0-255 (Master or tape-based). If R=0 the function is evaluated for the current room. This is stored in R% and may be manipulated if required.
V,	verb number, 0-255
E,	exit routine, 1-255
P,	procedure number, 0-255
M,	message number, 0-65535. If M=0 then message M%+256*N% is printed.
S,	object or room state value, 0-255
F,	object or room flag number, 0-7
V,	object or room flag value, 0 or 1
<num>,	numeric value, 0-255
<var>,	integer variable, A%-z%. Each can hold a value up to 255.
<str>,	a string of characters in quotes
<fsp>,	a valid file name, in quotes
<linrng>,	a line number range, for example LIST 10,50 lists lines 10 to 50; LIST 800, lists from line 800 to the end of the program.
<line>,	a line number, for example EDIT 65 enters the editor at line 65.
<expr>,	any valid expression constructed from functions and operators.

INTEGER VARIABLES

Integer variables in ALPS consist of the variables A% to z% which may each hold values in the range 0 to 255. Although all variables are accessible to the programmer, a number of them have reserved uses as given in the following table. These reserved variables should be used with care if their internal use is not to be upset.

Variable	Usage
C%	Number of objects being carried
D%	Destination room number: set by 'DEST' keyword
G%	Exit routine flag - move only allowed if G%=0
M%	Message number variable (low byte)
N%	" " " (high byte)
O%	Number of object returned by 'GETO' keyword
R%	Current room number
V%	Number of verb returned by 'GETV' keyword
X%	Maximum number of objects which may be carried
Z%	Random number (0-255)
d%	direction value for 'DEST(d)' keyword if d=0

CHANGING VARIABLE VALUES

The values of integer variables may be incremented and decremented by the 'INC' and 'DEC' statements. For example if J%=5 then

INC(J%) gives J%=6
DEC(J%) gives J%=4

Note that, if J%=255, then INC(J%) will leave J% equal to zero. Similarly, if J%=0 then DEC(J%) will make J%=255.

The 'ADD' and 'SUB' statements allow a constant or the value of another variable to be added to or subtracted from the specified variable. For example if A%=8 and Y%=2 then

ADD(A%,3) gives A%=11
ADD(A%,Y%) gives A%=10
SUB(A%,6) gives A%=2
SUB(A%,Y%) gives A%=6

Again the maximum value that a variable can hold (255) affects the result obtained by the ADD and SUB statements.

EDITING 'ALPS' PROGRAMS

The editor prompt is a question mark (?) to distinguish when edit mode is active.

To edit the list of instructions making up the 'ALPS' program you will use the EDIT command. This may be followed by certain options but EDIT on its own will enter the editor at the first line of the program. Alternately, you may give a line number following the EDIT command and the editor will be entered at that line, if it exists.

The first line, or the specified line, is listed and the edit prompt is displayed immediately below the first character of the line. The line above the prompt is the 'current line'. If you press <RETURN> the first line is left as it is and the next line is listed and becomes the current line and so on until the end of the program is reached. The editor is then exited but you can also exit at any time by pressing <ESCAPE>.

You may delete the entire current line by pressing function key <f2>, in which case the next line is listed and may be deleted and so on.

To insert a new line immediately following the current line press <f0> and a second ? prompt will appear below the first one. Type in your new line and, if there are no errors, it will be entered and becomes the new current line. Any errors will be reported and the line rejected. Subsequent lines may be inserted in a similar way.

To simply make a correction to the current line use the cursor and <COPY> keys to copy any parts of the line that are correct and retype where necessary. Pressing <RETURN> will enter the corrected line, subject to normal syntax checking, or <ESCAPE> will exit the editor and leave the line as it was.

To create a new line preceding the first line of the program requires you to first insert a copy of the first line. To do this, enter EDIT, press f0 and use the cursor and <COPY> keys to copy the first line and press <RETURN> to enter it. Now exit the editor - press <ESCAPE> - and re-enter it again (type EDIT) now typing the new first line and pressing <RETURN> to enter it. To avoid this it is a good idea to make the first line of the program a blank line - just press <RETURN>.

For convenience in editing verb definitions, exit routines and procedures, the EDIT command may be followed by the D, E or P, options. These options enter the editor at the first line of the specified routine.

Never delete or attempt to alter the DEFINE statement at the start of a verb definition. If you want to change the verb name, use the 'EDIT V' command - see next section.

EDITING THE VERB & OBJECT TABLES

It may happen that you have mis-spelt a verb or object name or you wish to completely change the name of a verb or object. These corrections are made using EDIT with the V or O options. The V and O specify the verb and object tables, respectively. In either case the letter must be followed by two strings separated by a comma. The first string is that which you wish to replace and the second string is the new version.

For example: EDIT V "TAKE","GET" would replace the word "TAKE" in the verb table by the new word "GET". The verb number given to "GET" is the same as that of the old word "TAKE". If the old string is not in the requested table the 'Not found' error message will be issued followed by the offending string. If the new string is already in the table you will be given the error 'Object exists' or 'Verb exists' followed by the string concerned.

To add a synonym for a verb or object you use the command SYNONYM instead of EDIT but otherwise the process is the same as that just described. In this case however the old string remains in the table and the new string is added to it, with the same number as the old string. For example: SYNONYM V "TAKE","GET" will add "GET" to the verb table and ensure that 'TAKE LAMP' and 'GET LAMP' both produce the same response to the player's input.

You don't need to add synonyms such as 'N' for 'NORTH', 'D' for 'DOWN' and so on. The system allows the player to abbreviate verb and object names. Note, however, that it will take the first match that it finds in the list. Thus if you want 'D' to be 'DOWN' then 'DOWN' will have to be defined before, say, 'DROP'. As a rule, it is best to start your verb definitions with the movement verbs, since these are the most frequently used.

STATES AND FLAGS

a. States

Objects and rooms both possess a state which can be set to any value in the range 0 to 255. What the different state values represent is entirely at the programmer's disposal. For example, in 'TINY', I have

used the state of the bear, object 6, to represent whether it is alive or dead. I arbitrarily decided that a zero state value would represent a live bear and a value of one would indicate the bear was dead. These values operate in conjunction with the message switches mentioned in the text editor section. Each switch contains the number of another message. Again, consider the bear in 'TINY': its basic description is message 28: 'a huge hairy grizzly bear '. This message has two switches, switch 1 with a value of 0, and switch 2 with a value of 29. Message 0 is a 'null' message i.e. it prints nothing, and message 29 says 'lying dead on the floor.'

When the system is asked to describe the bear, it will print message 28. It then checks if that message has any switches. If it has none, the description ends immediately. However, if there are switches, as in this case, the message routine will check the state of the object being described, and then get the message number contained in the corresponding switch. If the state is zero, the first switch will be read and the system will switch to printing that message, which could itself have further switches. If the state is 1, switch 2 will be used; if the state is 2, switch 3 will be used and so on. For example, the two states, 0 and 1, of the bear produce the two messages:

State=0: 'a huge hairy grizzly bear '
State=1: 'a huge hairy grizzly bear lying dead on the floor.'

(Note the trailing space included in the first message). If the state is higher than the number of switches then the highest-numbered switch is used.

As another example of the use of object states, 'TINY' uses the lamp's state to indicate if it is off (state=0) or on (state=1). You can go further than this and I have actually included three switches in the lamp's basic message. Message 20 includes switches to messages 30, 31 and 32. Respectively, these say that the lamp is off, shining brightly and shining dimly. 'TINY' only uses the first two states but you could try adding a counter which will allow the batteries to run down after a time. You could perhaps include something along these lines, in the main program loop:

```
1. EQUATE(EX,200)

   REPEAT
   .
   .
2. IF EQ(EX,50) THEN MSG(58) SETOS(1,2)
3. IF EQ(EX,0) AND NOTOSTATE(1,0) THEN MSG(59) SETOS(1,0)
SETOFLAG(1,7,0)
4. IF NOTOSTATE(1,0) THEN DEC(EX)
   .
   INPUT
   .
   etc;
   .
```

1. Allow 200 turns with lamp on.
2. Warn that the lamp is getting dim and set its state to 'dim'.
3. If the count reaches zero and the lamp is not already off, say the lamp has gone out and turn it off.
4. If the lamp is not off decrement the lamp counter.

In the 'ON' routine, you would need to include a check of the value of E% and refuse to light the lamp if it is zero.

Room states can be used in similar ways: when you want to provide a description which changes, depending on certain conditions. As an example, you may want to have a secret passage appear when the player says a magic word. This can be accomplished, by changing the room's state in response to the correct word. Typically this could be done as follows:

Room message: 'You are standing in a room with '

switch 1: 'only one visible exit, to the north.'
switch 2: 'an exit north and a secret passage leading down.'

There will have to be 'north' and 'down' destinations in this room's data base entry. A suitable exit routine on the 'down' exit will prevent the player using the passage until it is opened. The exit routine will need to check the room's state and print message 10 ('You can't go that way!') if the player tries to go down before the passage appears. This will avoid giving the game away, since the movement routines don't give any specific message when movement is prevented by an exit routine.

Assuming the room's state is initially zero, he will get the description:

'You are standing in a room with only one visible exit, to the north.'

If he gives any direction, other than north, he will be told 'You can't go that way!' Note the slight clue, given by including the word 'visible' in the description.

When the player says the magic word, and if the other conditions are satisfied, the 'SAY' routine will set the room's state to one. This will now give the description:

'You are standing in a room with an exit north and a secret passage leading down.'

The exit routine should now allow him to go down, by setting G% to zero.

b. Flags

Unlike states, flags can only have one of two values: zero or one. Consequently they are most often used to indicate if a certain property is possessed or is not possessed by a room or object. In all ALPS programs flag number 7, for both rooms and objects, is used to indicate the possession of the property 'light source'. In 'TINY', for example, the gun's seventh flag is zero: it's not a light source. The lamp's flag 7 is one, when it is 'on', indicating that it does then act as a source of light. The only other reserved flag is object flag 6: takeable.

Other object properties that could be represented by flags are: breakable, valuable, eatable, killable, burnable, sharp, magical etc.

Room flags can also be used to represent a wide range of properties. Certain rooms may have the property 'high' which means, perhaps, that saying 'JUMP' or 'DOWN' in such a place will prove fatal. I have already mentioned that objects may disappear if dropped in certain rooms. Again, one of the room flags can be used to indicate which of the rooms this applies to. Other rooms may be 'underwater', needing an aqualung, or they could require a spacesuit because they are 'airless'. The range of possible uses for flags is wide.

SOME OTHER PROGRAMMING TIPS

a. Programming mazes

Mazes are very easy to design, simply consisting of a group of interconnected rooms, all having the same description number. The player can map this type of maze by dropping objects, but this could also be prevented by 'losing' the dropped objects - see b. in the previous section.

You should decide on the correct route through the maze and make all other exits lead back to one particular room. It is also permissible for an exit to lead back to the room the player has just come from. However, since the 'last room' (usually stored in L%) and 'current room' (R%) would be the same, the game wouldn't give a room description. This problem is solved using the same, simple, exit routine on each of the affected exits. This would simply set L% to zero to ensure L% is not equal to R%, and also set G% to zero to allow the move.

The following maze, which you may like to add to 'TINY', consists of four rooms, all with description 58: 'You're in a maze of little passages, all alike.' The player enters in room 9 and exits from room 12 (to 13). The shortest route through is W, S, E, E. The exit routine mentioned above needs to be set on the exits marked '*'.

Dest. Room	9	10	11	12
NORTH	9 *	9	9	9
SOUTH	9 *	11	11 *	9
EAST	9 *	10 *	12	13 (out)
WEST	10	10 *	9	12 *
UP	9 *	10 *	11 *	12 *
DOWN	9 *	10 *	11 *	12 *

b. Thinking ahead and saving memory

Some careful thought at the planning stage can save a lot of time and memory. Consider the inclusion of an 'EXAMINE' verb, which may tell the player more about certain objects. Obviously it is necessary to store messages relating to this extra information. In order to avoid a lot of memory-wasting IF statements, it will make sense to group these messages together. For example, if we have five objects, the 'examine' messages could be numbers 127-131. To get at the required message you could use this verb definition:

```
DEFINE "EXAMINE"
ADD(O%,126) EQUATE(M%,O%) EQUATE(N%,0)
```

```
MSG(0) ;print msg in M% (N%=0)
END
```

Note that message 126 would be 'You notice nothing special.' which would be given for an unrecognised object: O%=0. The second line adds 126 to the object number to get the relevant message number which it places in M%. N% is set to zero so MSG(0) prints message M%.

Taking the above example a step further, suppose the examine messages were:

Number	Message
126	You notice nothing special.
127	The note says: "Go W S E E in the maze"
128	You notice nothing special.
129	It's a very valuable antique!
130	It has a switch on the bottom.
131	You notice nothing special.

It would be a waste of memory to have to store the message 'You notice nothing special' three times. To avoid this, messages 128 and 131 can be null messages, each with a single switch to message 126. When asked for message 128 or 131, the message system will first of all print nothing, then switch to message 126, giving the desired effect.

This method of switching to a piece of text, which is needed several times, can be used to save memory in other circumstances also. Consider an example in which the player may drown, either because he has no aqualung, or because it has run out of air. The two necessary messages could be:

'You splash about for a while but you quickly drown.'

'Your air just ran out and you quickly drown.'

Since the last part of these messages is the same they could also be written as follows, saving some memory:

'You splash about for a while but' switch to message <XXX>

'Your air just ran out and' switch to message <XXX>

message <XXX>: ' you quickly drown.'

When asked for either message, the system will print the first part and then switch to print the second part.

USING PROCEDURES

Suppose you wanted, at some stage in a game, to have a dwarf rush out, grab some or all of the player's possessions and run away again. This would require a fairly complex sequence of instructions which have to be placed within the main loop. It would be more convenient to have the instructions within a procedure. This can then be called from the main loop using:

```
IF <condition> THEN PROC(1)
```

assuming that the dwarf procedure is number 1. If the specified

condition is satisfied, the program will execute the dwarf procedure and return to the instruction following the PROC statement.

In order to illustrate the correct structure for procedures, I will give the full routine here.

```
DEFPROC(1)

; dwarf steals one item or
; kills you if you're empty-handed

EQUATE(O%,1) EQUATE(Y%,0) ; Y% is a flag

REPEAT
; Does player Carry object O%?
; If so then put it in maze
IF CARRY(O) THEN MSG(61):DESCOB(O):MSG(62):MOVEOB(O,11):DEC(C%):I
NC(Y%)
INC(O%)
UNTIL EQ(O%,7) OR NE(Y%,0)
IF EQ(Y%,0) THEN MSG(63):EQUATE(F%,9)
END
```

The messages required are:

```
MSG(61) : "A dwarf appears and steals "
MSG(62) : ". Hi Ho! Hi Ho! Off to the maze I go!"
MSG(63) : "A dwarf appears and is furious that you have nothing
to steal. He kills you with his little axe!"
```

Note the interesting line following the REPEAT statement. If you are carrying the keys (object 1) then this would produce:

"A dwarf appears and steals a bunch of keys. Hi Ho! Hi Ho! Off to the maze I go!"

The procedure loops through all the objects until it finds one that the player is carrying, or until it reaches the last object.

SENTENCE ANALYSIS

The simple two word, verb-object type command system can be easily implemented in ALPS by the instructions:

INPUT GETV GETO PERFORM

The programmer should be aware of the effect of GETV and GETO on the buffer pointer, if he wishes to manipulate the buffer contents to achieve anything other than this simple verb-object command structure. The ALPS language provides all the tools needed to implement a fairly full analysis of simple English sentences such as:

'TAKE (THE) BOX (AND) OPEN IT'

(The bracketed words are optional).

The GETV and GETO statements are used to take groups of characters

(words) from the buffer and convert them into verb and object numbers. If they find a recognised verb/object, they will leave the buffer pointer (where the next word will be read) at the first character after the verb or object which they found. Thus, for the example quoted above:

```
TAKE THE BOX AND OPEN IT
     ^       ^       ^  ^
    (1)     (2)     (3)(4)
```

GETV would extract 'TAKE', assuming it is a defined verb, and leave the pointer where indicated (1). GETO would then find 'BOX' ('THE' is not a recognised word), leaving the pointer at position (2). Another GETV finds 'OPEN' ('AND' is not recognised) and leaves the pointer at (3). A second GETO would find the object 'IT', leaving the pointer at (4). The buffer is now 'empty'.

If there had been no further recognised verb after 'TAKE THE BOX AND', the second GETV would have left the pointer at position (2), allowing the second GETO to find the 'IT'.

To summarize the effect of GETV and GETO on the buffer pointer:

(i) No recognised word found: the buffer pointer is left where it is.

(ii) Verb or object found: the pointer is placed at the character immediately following the end of the verb/object.

The type of sentence given above can be handled using the following structure:

```
    EQUATE(F%,0) EQUATE(I%,0)

    REPEAT    ;Main loop

    ; Put any high priority conditions here

    MSG(270)  ;'What now?'
    INPUT

    REPEAT    ;Analysis loop
    GETV GETO
  * IF OBJ(1) THEN EQUATE(O%,I%) ELSE IF NOTOBJ(0) THEN EQUATE(I%,O%)
    .
    ; Put any special pre-conditions here
    .
    .
    PERFORM
    .
    ; Put any special post-conditions here
    .
    .
    UNTIL EMPTY OR VERB(0)   ;Buffer empty or nothing recognised?

    UNTIL NE(F%,0)           ;dead?
```

The line marked with the asterisk deals with the word 'IT'. 'IT' is defined as an object (object 1), and the marked line either replaces 'IT' with the last-named object's number, as held in I%, or, if a new recognised object has been entered, updates I% to the new object's

number.

Now consider the sentence:

'TAKE THE BOX AND THE GUN'

The meaning of the word 'AND' in this sentence differs from that in the previous sentence. In the previous case, it served purely as a cosmetic device: it could have been omitted. In this case, however, 'AND' must be interpreted as meaning 'repeat the action of the previous verb with the object that follows'.

To enable this to be handled correctly, it should be realised that, in the second case, there is no verb following the 'AND', whereas there is a following verb in the first sentence.

The logic of this could be accomplished by defining a verb 'AND' so that when 'AND' is encountered in a sentence, the definition of 'AND' will replace V% with the number of the previous verb, but only if there is no other verb following the 'AND'. If P% is used to store the previous verb number and assuming that 'AND' is verb number 7, then the definition may be written:

```
DEFINE "AND"
;Check for verb after "AND"
IF NOTEMPTY THEN GETV
;Get object and check for "IT"
GETO:IF OBJ(1) THEN EQUATE(O%,I%) ELSE IF NOTOBJ(0) THEN
EQUATE(I%,O%)
;Was there another verb? If so do it
IF NOTVERB(0) AND NOTVERB(7) THEN PERFORM END
;No other verb so replace AND with old verb
IF VERB(0) AND NE(P%,0) THEN EQUATE(V%,P%)
;If not "AND" then do it
IF NOTVERB(7) THEN PERFORM
END
```

In the main loop of the program, the following instructions can be used to keep P% updated:

```
.
.
INPUT
REPEAT
GETV
IF VERB(0) AND EQ(P%,0) THEN MSG(19)  : "Eh?"
IF NOTVERB(7) THEN EQUATE(P%,V%) GETO : Verb OK but not 'AND'?
.
.
UNTIL EMPTY OR VERB(0)

etc;
```

APPENDIX A: ALPS ERROR MESSAGES

(All error messages produced by ALPS are displayed in red text).

Syntax error:

This is produced if you type in something which the computer does not recognise as a valid ALPS keyword.

Bad parameter

This message is given if parameters are omitted or if a variable is used when a number was expected e.g. RSTATE(A%,3)

Numeric parameters should be in the range 0 to 255, except for the MSG, YESNO and INOUT instructions, which have a range of 0-65535. Values exceeding these ranges give the 'Bad parameter' message.

Bad variable

The computer was expecting a variable A% to Z% and found something else e.g. IF EQ(5,V%)...

Object exists

This error is given when you try to enter an object which is already in the object list. It occurs with the 'OBJECT', 'EDIT O' and 'SYNONYM O' commands.

Missing "

All strings in ALPS, including filenames (except those used with operating system (*) commands) must be enclosed in quotes. This message is given if either quote is missing.

Escape

This message is produced when the <ESCAPE> key is pressed. If used whilst RUNning a program control will be returned to the command level. If pressed in a finalised game it has no effect.

Verb exists

You have attempted to add a verb that already exists in the verb table. This can occur with the 'DEFINE', 'EDIT V' and 'SYNONYM V' commands.

String exists

Occurs with the REPLACE command if you attempt to add a string which is already in the dictionary.

All tokens used

Occurs with the REPLACE command if all 128 tokens have been used.

Missing ,

The comma has been omitted from the 'OBJECT', 'EDIT' or 'SYNONYM' commands e.g. OBJECT "DAGGER" "KNIFE".

Not found

The computer was unable to find the specified verb or object name or the number of an exit routine or procedure. This error can occur with the LIST D¦E¦P and EDIT D¦E¦P commands, if the verb definition, exit routine or procedure does not exist. It also occurs with the EDIT O¦V and SYNONYM O¦V commands if the old verb/object name is not in the relevant table.

List full

There is a maximum of 255 verbs and 180 objects. This error is produced if you try to define more than 255 verbs or 180 objects.

No room

This error is produced if there is insufficient memory to insert the current line in the program, if the memory allocated to the verb or object tables is full or if the memory allocated to the text or dictionary is full. If you define a lot of synonyms this will use up the table space before the maximum number of verbs/objects have been defined. The memory allocations and free space remaining can be displayed by the STATUS command.

No program

This error is given if you enter 'EDIT' with no program resident in the computer.

In addition to the errors above, which are specific to ALPS, any operating system and filing system errors may also occur. These are dealt with in a similar manner.

APPENDIX B: ASI ERROR MESSAGES

'ASI' is the machine code program which executes the intermediate code produced by the ALPS compiler. The errors discussed in this section may be produced when the game is being played.

No REPEAT (2)

The interpreter found an 'UNTIL' statement without a preceding 'REPEAT' statement.

Too many REPEATs (1)

Your program contains REPEAT...UNTIL loops which are 'nested' more than 16 deep.

Undefined room (3)

The interpreter found a statement with a room number greater than the number of rooms in the data base.

Undefined object (3)

The interpreter found a statement with an object number greater than the number of objects in the data base.

Bad flag number (3)

This error is caused if a 'flag' number exceeds 7.

Bad flag value (3)

This error is caused if a flag has a value other than 0 or 1.

Illegal statement (5)

The interpreter found the end of program marker or tried to execute a 'DEFINE', 'DEFPROC' or 'EXIT' statement. The verb definitions, exit routines and procedures should not occur in the main part of a program. They must be placed in a separate section which is not directly executed. There should be a 'STOP' statement between the main program and the definitions.

Bad dir'n (5)

Occurs if direction value in MOVE or DEST statements is outside the allowed ranges: 1-6 for MOVE and 0-6 for DEST.

Bad description (4)

This error is given when a DESCRM or DESCOB instruction refers to a room or object with a non-existent description. Room and object descriptions must refer to valid message numbers.

Bad message (4)

This error is given if you try to print a message which is not in the data base. For example MSG(78) would cause this error in 'TINY', as there are only 57 messages in the game. It will also occur when using the text editor, if you specify a message beyond the last one currently defined: for example, typing '61 <f6>' in 'TINY' would try to jump to a non-existent message and result in this error.

Can't open File (21)

The computer was unable to open the file specified by an INOUT statement.

EOF (12)

The computer found the end of file marker when loading a game file.

TREATMENT OF ERRORS

The errors listed above are reported and the program run is aborted, with the computer returned to the command level and the line containing the error listed, starting from the keyword that caused the error.

If an error occurs in a stand-alone game only the error number - shown in brackets above - is displayed. Control will then be returned to BASIC.

Try to note the circumstances which lead up to the error as this will help to track down the mistake. For example, does the error always occur after a certain verb has been used or does it only happen in a particular room?

Other errors may occur as a result of the player entering illegal filenames or pressing <ESCAPE>. These may be reported but they do not interrupt the program.

APPENDIX C: KEYWORD ABBREVIATIONS

All keywords occupy only one byte of memory so abbreviations are merely for your convenience when entering your programs. Abbreviations do not require a full stop.

KEYWORD	ABBREVIATION
LIST	L
LISTO	LISTO
LVAR	LV
EDIT	E
NEW	NEW
SAVE	SA
LOAD	LO
RUN	RU
STATUS	STA
OBJECT	OBJE
SYNONYM	SY
AND	A
OR	OR
IF	I
THEN	T
ELSE	EL
REPEAT	RE
UNTIL	U
DEFINE	DEF
EXIT	EX
DEFPROC	DEFP
END	EN
CALL	CAL
OPT	OP
VERB	V
NOTVERB	NOTV
OBJ	O
NOTOBJ	NOTO
RSTATE	R
OSTATE	OS
NOTRSTATE	NO
NOTOSTATE	NOTOS
CARRY	C
NOTCARRY	NOTC
ROOM	RO
NOTROOM	NOTRO
HERE	H
NOTHERE	NOTH
RFLAG	RF
OFLAG	OF

KEYWORD	ABBREVIATION
YESNO	Y
TRUE	TR
FALSE	F
EMPTY	EM
NOTEMPTY	NOTE
EQ	EQ
LT	LT
GT	GT
NE	N
MSG	MS
DESCRM	DESC
DESCOB	DESCO
SETRS	SETR
INCRS	INCR
DECRS	DECR
SETOS	SE
INCOS	INCO
DECOS	DECO
SETRF	SETRF
SETOF	SETOF
STOP	S
GOTO	G
MOVE	M
MOVEOB	MOVEO
DEST	DE
TAKE	TA
DROP	DR
INPUT	IN
GETV	GET
GETO	GETO
PERFORM	PE
RESTORE	RES
PROC	P
INOUT	INO
INC	INC
DEC	DEC
ADD	AD
SUB	SU
EQUATE	EQU

APPENDIX D: LISTING OF "TINY"

```
| "TINY" program for 'TINY' adventure
| Do not type comments unless using wordprocessor

GOTO(2)        ;Game starts in room 2
EQUATE(C%,0)  ;Not carrying anything
EQUATE(L%,2)  ;L% is Last room
EQUATE(F%,0)  ;Game over flag: F%=1: success, F%=9: death!
EQUATE(X%,3)  ;Maximum carry capacity
EQUATE(B%,3)  ;Number of bullets
EQUATE(A%,0)  ;Counts turns in cave

OPT(0)         ;No exit reporting
DESCRM(0)      ;Describe current room

| Main loop

REPEAT
EQUATE(G%,0) ;Clear exit routine flag
;If he's in the cave and the bear's not dead increment turns
IF ROOM(7) AND NOTOSTATE(6,1) THEN INC(A%)
;Get his input and convert to verb and object numbers
INPUT GETV GETO
;If he's had 4 turns then kill him
IF EQ(A%,4) THEN MSG(34) EQUATE(F%,9)
;If he has'nt had 4 turns then do his action
IF NE(A%,4) THEN PERFORM
IF VERB(0) THEN MSG(18)  ;Didn't recognise that!
IF NE(L%,R%) THEN DESCRM(0) EQUATE(L%,R%) ;Describe if its a new room
UNTIL NE(F%,0) ;Continue until dead or successful
IF EQ(F%,1) THEN MSG(51) ;If not dead then congratulate
STOP ;Back to BASIC

| Verb definitions

DEFINE "NORTH"
MOVE(1)
END
DEFINE "SOUTH"
MOVE(2)
END
DEFINE "EAST"
MOVE(3)
END
DEFINE "WEST"
MOVE(4)
END
DEFINE "UP"
MOVE(5)
END
DEFINE "DOWN"
MOVE(6)
END
DEFINE "LOOK"
DESCRM(0)
END
DEFINE "INVENTORY"
DESCRM(1)
```

```
END

DEFINE "SHOOT"
;Got the gun?
IF NOTCARRY(5) THEN MSG(40) END
;Any bullets?
IF EQ(B%,0) THEN MSG(52) END
;Is the bear here?
IF NOTHERE(6) THEN MSG(38) DEC(B%) END
;Is it dead already?
IF OSTATE(6,1) THEN MSG(57) END
;Kill it
MSG(39) SETOS(6,1) DEC(B%)
END

DEFINE "TAKE"
EQUATE(T%,C%)
TAKE(0)
IF NE(T%,C%) THEN MSG(3)        ; print OK if taken
END

DEFINE "DROP"
;dropping coin in room 7 and bear not dead?
IF OBJ(3) AND ROOM(7) AND NOTOSTATE(6,1) THEN MSG(55) END
;bear is dead but has player got coin?
IF OBJ(3) AND CARRY(3) AND ROOM(7) THEN EQUATE(F%,1) MOVEOB(3,8) END
;nothing special now
EQUATE(T%,C%)
DROP(0)
IF NE(T%,C%) THEN MSG(3)        ; print OK if dropped
END

;Turn on lamp
DEFINE "ON"
;Got lamp?
IF NOTCARRY(1) THEN MSG(8) END
;Is it on?
IF OSTATE(1,1) THEN MSG(41) END
;No - turn it on
SETOS(1,1) MSG(43) SETOFLAG(1,7,1)
END

;Turn off lamp
DEFINE "OFF"
;Got it?
IF NOTCARRY(1) THEN MSG(8) END
;Is it off?
IF OSTATE(1,0) THEN MSG(42) END
;No - turn it off
SETOS(1,0) MSG(44) SETOFLAG(1,7,0)
END

;Open door
DEFINE "OPEN"
;In the correct place?
IF NOTROOM(4) THEN MSG(45) END
;Got the keys?
IF NOTCARRY(2) THEN MSG(46) END
;Is it already open?
IF OSTATE(4,0) THEN MSG(47) END
```

```
;No - open it
SETOS(4,0) MSG(37)
END

;Print a random score
DEFINE "SCORE"
MSG(50)
END

DEFINE "SHOUT"
MSG(49)
END

; Exit routine for door
EXIT(1)
;If the door is not open (State=1) then stop movement and explain why
IF OSTATE(4,1) THEN MSG(48) EQUATE(G%,1)
END

; Objects and synonyms

OBJECT "LAMP","TORCH"
OBJECT "KEYS"
OBJECT "COIN"
OBJECT "DOOR","VAULT"
OBJECT "GUN","PISTOL","REVOLVER"
OBJECT "BEAR","GRIZZLY"

SYNONYM V "SHOUT","SAY"
SYNONYM V "TAKE","GET"
SYNONYM V "OPEN","UNLOCK"
SYNONYM V "SHOOT","FIRE"
SYNONYM V "ON","LIGHT"
```

MESSAGES USED IN 'TINY'

Switches are only needed for those messages which have switch numbers following them.

Message: 1: nothing

Message: 2: You can see

Message: 3: Ok

Message: 4: I don't see that here!

Message: 5: You have that already!

Message: 6: You can't take that!

Message: 7: It is pitch dark

Message: 8: You don't have that!

Message: 9: You can't carry any more!

Message: 10: You can't go that way!

Message: 11: You're in the hall.

Message: 12: You are in the disused vault of the 'Dwarf and Elf Bank, Inc." The shelves seem bare. The only way out is through the vault door, north.

Message: 13: You're in the vault.

Message: 14: You are on a landing at the top of the stairs. Exits lead west and south and there's a large hole near the floor in the north wall.

Message: 15: You're on the landing.

Message: 16: This is a small damp room with only one accessible exit, to the north.

Message: 17: You're in the damp room.

Message: 18: I don't understand that!

Message: 19: You're in the bedroom.

Message: 20: a small battery lamp
Switches: <30><31><32>

Message: 21: a lamp
Switches: <30><31><32>

Message: 22: a bunch of keys (could be useful!)

Message: 23: a rare gold coin!

Message: 24: a door
Switches: <25><33>

Message: 25: which is open.

Message: 26: a revolver
Switches: <27>

Message: 27: loaded with %B bullets.

Message: 28: a huge hairy grizzly bear
Switches: <0><29>

Message: 29: lying dead on the floor, poor thing.

Message: 30: which is off.

Message: 31: which is on and shining brightly.

Message: 32: which is on and shining dimly.

Message: 33: which is locked.

Message: 34: The bear sees you, grabs you affectionately and shakes you warmly by the throat! Oh dear, you're dead!

Message: 35: This room was once the master bedroom but the master isn't here anymore. A large fourposter bed stands in one corner, covered by cobwebs. Doorways lead off east and south.

Message: 36: You are carrying

Message: 37: The door swings open.

Message: 38: There's nothing here worth wasting a bullet on, but if you insist - BANG!

Message: 39: BANG!

You shoot the unfortunate bear which drops dead at your feet.

Message: 40: You have nothing to shoot with!

Message: 41: It's already on!

Message: 42: It's already off!

Message: 43: You turn on the lamp.

Message: 44: You turn off the lamp.

Message: 45: There's nothing here with a key.

Message: 46: You have no keys.

Message: 47: It's already open!

Message: 48: The locked door bars your way.

Message: 49: Ok, "$!"

Message: 50: You have scored %Z points.

Message: 51: As you drop the coin in the slot you hear a sudden noise of ancient machinery starting up. A mighty voice booms "Well done, mortal. Now go forth and program thine own adventure!"

Message: 52: There are no bullets left!

Message: 53: You're in the bear's cave.

Message: 54: You have dropped into a warm cosy cave. The only exit is too high to reach. In the far corner is a very old piece of equipment, with a small slot on the top panel.

Message: 55: The bear growls and swipes at you with his paw. He won't let you near the machine.

Message: 56: You are in the hall of a long-deserted and very strange house. It clearly once belonged to a professional adventurer, judging by the diplomas from the Institute of Advanced Adventurers which hang on the walls here. Rickety stairs lead up and a small hole exits to the west.

Message: 57: The bear is already dead!

APPENDIX E: MAKING A CASSETTE VERSION

Having developed an adventure on disk you may wish to give it to friends who do not have disk systems, or perhaps you want to sell it in cassette form. In either case, just follow the instructions below in order to produce a cassette version of a complete game that is already on disk. The instructions use "QUEST" as an example so you must replace "QUEST" with the filename of your own game.

(i) Insert the game disk and type:

```
*INFO #.QUEST
```

(ii) The computer should display something like:

```
G.QUEST    L 001100   001100   006AB9 070
$.QUEST    L 007000   007000   0004A8 010
```

These numbers are, respectively,

	Load Address	Start Address	File Length

(The last number in each case is the disk sector where the program starts and can be ignored).

Note the file lengths: L1 for QUEST and L2 for G.QUEST

(iii) Type:

```
*LOAD QUEST 1900
*TAPE
*SAVE QUEST 1900+L1
```

This will then save the loader program onto cassette. Use the value for L1 that you noted.

(iv) To save the main program type:

```
*DISC
*LOAD G.QUEST 1100
*TAPE
*SAVE G.QUEST 1100+L2
```

Use your own value for L2, noted at step (ii).

The main program will now be saved onto the cassette and can be run, after rewinding to the start of the loader program, by typing CH."QUEST".

PO Box 25, Portadown, Craigavon, BT63 5UB

Alpine
Software

PO Box 25, Portadown, CRAIGAVON, BT63 5UT ☎ 0762 42510

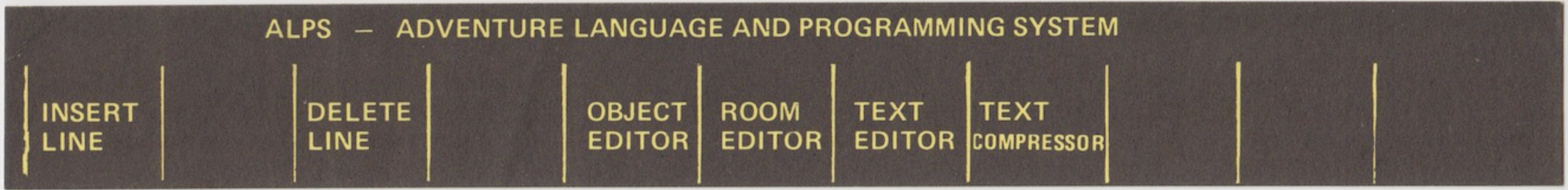
ALPS - MASTER COMPACT VERSION

The supplied 3.5 inch disk contains the ALPS utilities, sample adventures and an image of the ALPS rom for loading into sideways ram. Please ignore the references to the eprom in the User Guide.

When the disk is booted it will check if ALPS is present and if not it will load and initialise ALPS automatically before displaying the Utilities menu as described on page 11 of the Guide. To use the 'rom' just press <break> to leave the menu and then *ALPS <return>. Subsequent booting of the disk will go straight to the menu.

Appendix E: Making a cassette version

As the Compact has no tape hardware you will not be able to create tape games. However your games are compatible so you will just need a cooperative friend with BBC B or Master and a disk-drive adaptor lead to transfer your games from 3.5 inch to 5.25 inch disk and from that to tape as described in this appendix!


ALPS – ADVENTURE LANGUAGE AND PROGRAMMING SYSTEM
INSERT LINE
DELETE LINE
OBJECT EDITOR
ROOM EDITOR
TEXT EDITOR
TEXT COMPRESSOR