Adventure II — an epic game for non-disc systems

We present an Adventure-creating program for almost any system.

by Ken Reed

ADVENTURE is a very popular and addictive computer game, run on many large systems throughout the world. However, despite its popularity among those with access to large machines, it does not seem to have appeared on many microcomputers. The reasons for that are easy to see; Adventure is complex, large, requiring disc back-up and difficult to modify to produce new games.

More general

Consideration of those problems led to the design and implementation of the simpler and more general Adventure program described in this article, suitable for most systems without the need for disc-backing store.

Adventure is a computerised version of the game Dungeons and Dragons, both of which are enormously popular. In Dungeons and Dragons a player is assigned the role of dungeon master, devising the dungeons which the other players explore in an attempt to acquire the treasures hidden there, often after having fought their way past monsters of various kinds.

Each player announces his action to the dungeon master who then tells him of the outcome, usually after spinning various dice to generate a random element.

Adventure, run on PDP-11 systems everywhere, is similar to Dungeons and Dragons except that the dungeons are devised by the original programmer and the computer then assumes the role of dungeon master.

Interactive technique

The computer describes the situation to the user who replies with the action he has decided to take. The computer, in turn, informs him of the result. This player/computer interaction is best illustrated by a sample run of the program — the user response is in capital letters.

```
   Somewhere nearby is Colossal Cave, where
others have found fortunes in treasure and
gold, though it is rumoured that some who
enter are never seen again. Magic is said to
work in the cave. I will be your eyes and hands.
Direct me with commands of one or two
words.
   You are standing at the end of a road before
a small brick building. Around you is a forest.
A small stream flows out of the building and
down a gully.
ENTER BUILDING
   You are inside a building. A well house for a
large spring.
   There are some keys on the ground here.
There is a shiny brass lamp nearby. There is
food here. There is a bottle of water here.
GET KEYS
OK
GET LAMP
OK
RUB LAMP
   Rubbing an electric lamp is not particularly
rewarding. Anyway, nothing exciting happens.
EXIT
   You're at the end of the road again.
GO SOUTH
   You are in a valley in the forest beside a
stream tumbling along a rocky bed.
GO SOUTH
   You are in a 20-ft. depression floored with
bare dirt. Set into the dirt is a strong steel grate
mounted in concrete. The grate is locked
UNLOCK GRATE
   The grate is unlocked.
```

Once having opened the grate, for which he must have the keys, the player then has access to Colossal Cave where there are problems to solve and treasures to gather. However, if he does not have the keys, there is no way that the grate can be opened. In fact, it may take him a while to find the entrance as it is all too easy to become lost in the forest.

As you can see from the example, playing Adventure is rather like reading a novel, with one important difference. Instead of following the story passively, the reader is involved actively, deciding what is the best action to take in a given situation, often having to think very carefully as the wrong decision may lead to death.

That afinity with a novel is Adventure's main disadvantage. Once all the problems have been solved, which may take several weeks, interest wanes and another Adventure is required.

The original version of Adventure, programmed by Will Crowther at Stanford Research Institute, is coded in Fortran, requires 64Kbytes of memory, disc back-up and is very difficult to modify to generate new games as many of its features are buried deep within the program code. That explains the current shortage of Adventures.

A better solution would be to have a general Adventure program driven by a separate database allowing new games to be generated without having to overcome the programming complexities every time. In fact, that approach was used by Scott Adams who has now produced a number of excellent adventures for some of the more popular systems such as the TRS-80 and Sorcerer.

The program described here carries this concept one step further. Instead of one person producing adventures for a limited range of systems the idea is to describe a program which can be implemented on almost any system and driven by an entirely separate and machine-independent database. That allows owners of the program to write adventures in a simple form and swap games with someone who may have an entirely different processor.

Two segments

As mentioned earlier, Adventure II is split into two parts. The first is the program and the second the driving database. Before describing the program, it is worthwhile to look at the general structure of the database which has four main sections:

1. The vocabulary of words recognised in the game.
2. The objects that may be manipulated.
3. The places that may be visited.
4. The actions performed by specific words.

All that is required to produce the database, and the program is an assembler and examples of various table entries are shown for a Z-80-type assembler.

The vocabulary is held as the first four letters of a word followed by an identifying code. That permits the program to reduce words to simple numbers which are much easier to manipulate. It also allows different words to have the same code and hence the same meaning.

Identifying code

For example, the words "DESCEND" and "DOWN", having roughly the same meaning, may be assigned the same identifying code. Hence the commands from the user "DOWN STEPS" and "DESCEND STEPS" may be handled by the same function. The table may be entered thus:

```
VOCAB: DEFM  'NORT' ; Word "NORTH"
       DEFB  1      ; Identifying code
                      "1"
       DEFM  'EAST'
       DEFB  2      ; EAST has code
                      "2"
```

DEFM is the instruction to define an ASCII string and DEFB is to define a byte. The table has the name "VOCAB" and terminated by a byte of 0FFH (255 or —1). The words for movement — north,south,etc. — must have codes in the range 1 to 12 as the program prints the message — I cannot go in that direction — if it cannot find anything to do with words in that range. Other unmatched words generate the simpler response: I can't.

Objects are anything which may be moved from one place to another and/or transformed from one thing to another. A lamp, for example, may be carried with the player and it may be transformed from a "LIT LAMP" to an "UNLIT LAMP" and, of course, back again.

Each object has an entry in each of two tables: the object location table which records the current position of the object and the object description table which contains the text used to describe the object.

The current location table is named "OBJLOC" and the descriptive text table "OBJTXT". OBJLO is terminated by a byte of 0FFH, OBJTXT needs no termination.

```
OBJLOC: DEFB    3,0         ; Object 0 at
                              location 3
        DEFB    5,0         ; Object 1 at
                              location 5
                            ; Similary for
                              other objects
OBJTXT: DEFW    M0          ; Address of text
                              for object 0
        DEFW    M1          ; Address of text
                              for object 1
M0:     DEFM    'A little   ; Description of
                axe'          object 0
        DEFB    80H         ; String termin-
                              ator
M1:     DEFM    'A bunch
                of keys'
        DEFB    80H
```

Note that the object position information is two bytes to allow it to be at a location — first byte is 0-225 — or in some special place, such as carried by the player — second byte is used. Also, the object description table OBJTXT contains the address of the actual description for each object.

Simple indexing

That allows simple indexing by object number into the table to locate the real text. If it were not done that way, the table would be much harder to use as each entry would be of an unknown length. The byte 80H is used to terminate the string.

The locations are the places that the player may visit. They may be rooms, caves or anything desired by the Adventure writer. Each location has an entry in two tables: The description of the location and the list of directions the player may go from there. The location descriptions are held in a table named LOCTXT and the possible movements in MOVEMT. Both of those tables consist of pointers to the actual data as described for the object descriptions above.

```
MOVEMT: DEFW    D0      ; Pointer to loc-
                          ation 0 moves
        DEFW    D1      ; Pointer to loc-
                          ation 1 moves
                        ; etc. for rest of
                          locations
```

The following example movement shows an entry that says that word 0 takes

(continued on next page)

(continued from previous page)

us to location 1 and word 3 will take us to location 5. Note that a —1 terminates the list.

```
D0:      DEFB  0,1,3,5,-1
LOCTXT:  DEFW  L0            ; Pointer to des-
                               criptions
         DEFW  L1            ;
L0:      DEFM  'I am in an empty room'
         DEFB  80H
L1:      DEFM  'I am by a stream'
         DEFB  80H
```

The action table is the section of the database interpreted or executed by the main program. It consists of words, conditions and actions performed. If there is an entry in the table for the words entered by the user and the conditions specified are met, the actions are performed.

Action table

For example, if the command "GET LAMP" has a corresponding entry in the action table and the condition that the lamp must be in sight are met, the database will instruct the program to mark the location of the lamp as carried. There is a similar table scanned before the player's turn to see if the computer wants anything to happen.

For example, he may be in the same room as a Vampire without a crucifix so the computer may make the Vampire attack. The user action table is named EVENT and the computer's table STATUS. Both have the same format:

```
EVENT:  DEFB  0,1           ; Words 0 and 1
        DEFW  C0            ; Pointer to
                              conditions
        DEFW  A0            ; Pointer to
                              actions

        DEFB   3,-1         ; Only word 3
                              required
        DEFW  C1            ; Pointers
        DEFW  A1            ;

C0:     DEFB  0,1,1,2,-1    ; Must be at
                              location 1 (0,1)
                            ; Ojbect 2 must
                              be here (1,2)

A0:     DEFB  5,3,-1        ; Print message 3
```

The lists of actions and conditions are terminated by a byte of 0FFH (-1). Note that the examples are only very small extracts from a real table. A full-size database may have up to 255 locations and any number of entries in the event and object tables.

Pseudo-code

The requirement that the program be as small as possible means that it must be entered as an assembler program. However, as we want the program described to be suitable for any system, it leads to a slight problem over how we represent it.

Assembly listings for every processor would occupy far more space than the magazine can provide and flowcharts would not really describe the action of the program at the level of detail we want. For those reasons, the program is represented in pseudo-code.

For those not familiar with the term, pseudo-code is a non-existent language or shorthand representation of a program often used by programmers for detailed design when the actual target language is not yet known. Pseudo-code provides far more detail than flowcharts and is, in fact, detailed instructions for the actual coding of the program.

Although the listing should be more or less self-explanatory, it is worth mentioning two conventions used. If a variable is preceded by a "@", it means that the variable is used as a pointer to the data. For example, if "HL" contains the value 100 and we say "A=@HL", "A" is loaded with the contents of memory location 100.

A similar convention is used to identify the address of a variable except the "#" character is used — also used by the IF statement for not equal to. For example, if we say "HL=#OBJLOC", it means that the variable HL is loaded with the address of OBJLOC and not the contents.

Variables used are defined as either BYTE, 8-bit, or WORD, 16-bit, and the contents are assumed to be set to zero unless a value is included between two 'P's. For example, to define two 8-bit variables in memory, one set to zero and the other to 3 we use:

```
BYTE VARA,VARB/3/
```

A memory block is reserved by:

```
BYTE VARC/<7>/
```

which means reserve seven bytes of memory starting at label "VARC".

Program arrays

That leads to the implementation of arrays used by the program. All references to arrays mean an offset to the base label of a memory area. For example, VARC(3) simply means the address found by adding three to the value of label VARC.

Thus VARC(0) is exactly equivalent to VARC. Remember that words occupy two bytes, so, if VARC was a word array, VARC(3) would actually be addressing VARC+6 and also VARC+7 for the top byte.

Knowing this, you should now be able to produce a version of the program for your particular system by working through the listing and generating the appropriate assembly code for your machine. If you have access to a medium-level language, such as PL/M for example, that is, of course, equally acceptable.

The pseudo-code program shown here has in fact been compiled by a specially-written compiler to ensure that it is sound.

Let us work through the program considering what makes it tick and explaining the meaning of the pseudo-code representation.

Referring to the listing, we can see that the first section is simply the definition of items not within this listing, that is the items marked "GLOBAL". Four subroutines not described here are called, but as these are relatively simple entities they should present no problem in coding.

The first subroutine required is called "$REPLY" and it is simply a routine to read a respose from the user and return a value of one if it was a "Y", and a value of zero if it was a "N". The routine should check that either a "Y" or a "N" was entered and prompt "PLEASE ANSWER YES OR NO" for any other reply.

The second routine is named $MESS and is the routine used to print messages on the console. It must take the ADDRESS of a message as a parameter and print all the bytes found there until a byte with the most significant bit set is encountered. The routine used also had the additional feature that it printed a return/line feed if it was called with an address of zero.

The next routine, $LINE, is the opposite of $MESS; it obtains a line from the user and passes back the address of the stored text.

Random numbers

Finally, $RAND is a routine which returns a random number in the range 0 to 100. Many people shudder at the thought of writing random-number generators but as we want only one number at a time and not a series, that is not as difficult as you may think.

It can be done by reading the refresh register if you have a Z-80 system, or if your keyboard is software-controlled, you may increment a counter in the keyboard — wait loop and use that value as the random number. If you want a more elegant solution, the random numbers used in the prototype program were generated using the algorithm:

[Generated number] = 11x[Last generated number] + 999 MOD 101 although this does require 16-bit multiply and divide.

The next group of globals refer to the addresses of the various tables in the database.

Variable definition

The last part of the section is the definition of the variables used by the program. Although some of them are defined as words, the only items which must be 16-bit are "Here" as it is compared to a 16-bit object location and the three "pointers" BC,DE and HL as they hold addresses used to point to the actual data required.

A further point is that some items are used for temporary storage only and may be replaced by the processor registers if you desire. The only variables that must be in memory are Here, the current location and User, the variables the database may access. If you run out of registers, remember they may be saved on the stack while a register is used for something else.

Proceeding to the code, we can see that the program begins at label "Start" which simply sets the first location to zero. The code beginning at "Desc" describes the current location by printing the description found adding the contents of 'HERE' to the base address LOCTXT and using the pointer there.

The current location will, of course, change as the game progresses. A small piece of code checks to see if the database has set user flag zero and if so, we are in dark locations and object zero must be present (a lamp) to obtain the location description. Otherwise the message "Everything is dark. I cannot see" is displayed.

Time limits

Two of the user flags are also decremented automatically if the database has set them non-zero. That allows time limits to be implemented for things like being in dark locations. A further flag is decremented a little later once per player's turn.

The code then goes on to scan through the object location table "Objloc" and if any objects position is the same as the current position "Here", the object description is printed.

Next, the program looks quickly as the status table which is effectively the computer's turn at the game. However, as the same mechanism which decodes the player's command is used, we will consider it later. That function, when completed, returns to the label "PROC".

The routine that obtains a line from the user is called ($LINE) and the address of the entered text obtained. The routine used returned the address on the top of the stack and the instruction "HL = @ SP" finds that address.

Line reduction

We then pick the first four letters of the first word and look it up in the vocabulary table. If the word is found, we do the same to the second word. If a particular word does not have an entry in the vocabulary table, we discard it and try the next.

That reduction of the line allows complex sentences like "TURN ON THE LAMP" to be reduced to simpler entities like "ON LAMP", provided the words TURN and THE are not in the vocabulary. Hence it is important to consider carefully which words are not in the vocabulary as well as which ones are.

If none of the entered words is found in the vocabulary, the message "I don't understand" is printed and we go and obtain another line from the player.

After we have converted the user's command into one or two single-byte codes, we take the first code and see if it is one of the words which cause movement at the location. If it is, the current position (HERE) is updated and we return to label "MOVED" to describe the new place. If it is not, we proceed to examine the main event table to see if there is an entry there.

If the first word code "W1" matches the first byte of an entry and "W2" matches the second byte, we proceed to extract the conditions and test them. If all the conditions are satisfied, we extract the list of actions and execute them.

If all the conditions do not match or the two-word codes do not match, we try the next entry in the table. That is repeated until an explicit command to leave the table is given or the table is exhausted.

The action or condition is decoded by using it as an index into a list of addresses for the function we want and simply moving the address to the program counter (PC). It can usually be done on most machines by pushing the address on to the stack and executing a return from subroutine instruction.

The comments in the program listing explain the operation in greater detail and indicate what actions and conditions are available.

Looking at some examples of a database should further clarify the operation of the program. To make the database more readable, the example extracts shown below were produced using a macro assembler and calling various macros to make the entries in the appropriate table.

Vocabulary details

The following is a small section of the vocabulary from the author's test database. Note how abbreviations are also entered for words and given the same code. Hence "E" is equivalent to "EAST".

```
VOCAB::
TABLE <SOUT>  ,1
TABLE <S>     ,1

TABLE <EAST>  ,2
TABLE <E>     ,2

TABLE <WEST>  ,3
TABLE <W>     ,3

TABLE <NE>    ,4
TABLE <NW>    ,5
TABLE <SE>    ,6
TABLE <SW>    ,7

TABLE <UP>    ,8
TABLE <U>     ,8

TABLE <DOWN>  ,9
TABLE < D>    ,9

TABLE <NORT>  ,12
TABLE <N>     ,12

TABLE <END>   ,13
TABLE <STOP>  ,13
TABLE <QUIT>  ,13
TABLE <ABOR>  .13
```

In the objects shown here, note how items which can change state are two objects although only one of the pair may exist at any given time.

```
OBJECT 0, <0,8>, <A lit lamp>
OBJECT 1, <S7,0>, <An old oil lamp>

OBJECT 2, <S5,0>, <A small cloth bag>

OBJECT 3, <S5A,0>,<A bottle of holy water>
OBJECT 4, <0,8>, <An empty bottle>
OBJECT 5, <0,8>, <A match>
OBJECT 6, <0,8>, <A spent match>
```

The first byte of the location information is used to mark the location of the object. If the second byte is non-zero, the object is at one of the special places. These are:

2 — Object is carried [512]
4 — Object is worn [1024]
8 — Object does not exist (yet) [2048]

The value in "[]" indicates the number obtained when the two bytes are considered as a single 16-bit word.

Movement words

The first two locations of the example illustrate how movement can be accomplished by any words and not just directions. For example, the word "HELP" moves the player to location Sl which simply contains instructions for him. The word "BEGIN" is used to start the game.

```
LOC S0, <HELP,S1, BEGI,S2>
TXT <Welcome to Adventure!>
TXT <If you know what to do type BEGIN
      otherwise type HELP>

LOC S1, <BEGI,S2>
TXT <I have managed to get myself lost in the
      forest on my>
TXT <quest for the seven golden keys of
      Waydor and I don't know>
TXT <what to do next. So it is up to you to
      help me.>
TXT <>
TXT <Give me your instructions and I will
      obey. For example, >
TXT <if you want me to go to the north, type
      "GO NORTH", if>
TXT <we should come across some keys and
      you want me to get>
TXT <them, type "GET THE KEYS".>
TXT <Some other words that you may find
      useful are: >
TXT <INVENTORY to find out what I'm
      carrying>
TXT <QUIT to give up.>
TXT <>
TXT <Type "BEGIN" when you are ready to
      to start.>

LOC S2, <S,S4, PATH,S4>
TXT <I am in a clearing in a very dense
      forest.>
TXT < There is a path leading off to the
      south.>
LOC S5, <N,S2,E,S5,W,S6>
TXT <I am at a "T" junction with exits to the
      north, west and east>

LOC S5, <W,S4,EXIT,S4,E,S5A,ALTA,
      S5A>
TXT <I am amongst the ruins of a church.
      At the far end there>
TXT <are the remains of an altar. The exit is
      to the west.>

LOC S5A, <EXIT,S5,W,S5>
TXT <I'm beside the altar.>

LOC S6, <E,S4, IN,S7, CRYP,S7>
TXT <I'm outside the entrance of a crypt.>

LOC S7, <EXIT,S6, DOOR,S6>
TXT <I'm in a vaulted chamber. Thick
      cobwebs hide the ceiling.
TXT <There is an empty coffin in the corner
      and a passage leading>
TXT <off into darkness to the north.>

LOC S8, <D,S9, STEP,S9>
```

(continued on page 73)

(continued from page 71)

```
TXT <I'm at the top of a steep flight of steps.
    I can see a>
TXT <dim light to the south.>
```

The event table is the real heart of the database as it contains the actions performed by each command from the user. This section also contains the various messages which may appear under database control.

```
EVT 2  <N   ,-1 >        <0,S7,-1> <9,0,8,
                                    S8,6>
EVT 3  <S   ,-1  >       <0,S8,-1> <10,0,
                                    8,S7,6>

EVT 4  <GET ,LAMP>       <1,0,-1>   <2,0,
                                    13>
EVT 5  <GET ,LAMP>       <1,1,-1>   <2,1,
                                    13>
EVT 6  <DROP,LAMP>       <1,0,-1>   <3,0,
                                    13>
EVT 7  <DROP,LAMP>       <1,1,-1>   <3,1,
                                    13>
EVT 8  <LIGH,LAMP>       <1,1, 1,5, <11,0,
                         -1>        11,5,
                                    5,11,18,
                                    ,-1>
EVT 9  <OFF ,LAMP>       <1,0,-1>   <11,0,
                                    13>
EVT 10 <LIGH,LAMP>       <1,1,-1>   <5,14,
                                    -1>

Status:
EVT A  <-1,-1>           <7,5,5,0,2,10,-1>
                        <5,7,15,2,8, 9,5, -1>
EVT B  <-1,-1>           <6,2,1,    <5,8,
                         -1>        12>
EVT C  <-1,-1>           <5,2,-1>   <5,5,
                                    -1>

EVT Z  <-1,-1>           -1         7

MSG 5,<I feel sick and dizzy!>

MSG 7,<Some one has lept out of the shadows
   and BITTEN MY NECK!!!!>
TXT <He vanished as suddenly as he
   appeared!>

MSG 8,<Everything is getting dark! I Think
   I'm dy ...>

MSG 11,<I have lit the lamp with the match
   which has now burned out>

MSG 14,<I don't have anything to light it
   with.>
```

It is worthwhile examining some of the entries in the table in detail to show just what can be accomplished in the database. For example, in the location S7 shown, there is a passage leading north, but there is no entry in the movement list for it.

That is because rooms past there are dark and we want to tell the program. So let us look at entry 2. The word codes which must match are "N" (north) and anything will do for the second. There is a single condition, namely that he must be at location S7. If that is so, actions are performed which are : 9,0 — Set flag zero; 8,S8 — Go to location S8; 6 — Describe the location and obtain another command from the player.

Updated positions

The lamp, being two objects, the lit lamp and the unlit lamp, has two entries for the GET and DROP commands. Each entry determines which of the objects is here and updates the position of the appropriate one. To light the lamp, the conditions are that the unlit lamp and the unused match must be present. If that is so, the unlit lamp is destroyed, the lit lamp created and the match is transformed to the spent match.

An informative message is also printed. The final command (18) aborts the scanning of the table as a little later in the table, there is an entry for LIGHT LAMP when no match is present — which gives message 14 — and we do not want to fall through to it if we have already lit the lamp.

The table is terminated by a word code of zero. Note that in the example the words GET, DROP etc., are shown but in a real table the word code is used.

The entries in the STATUS table show an example of how a "wandering monster" may be implemented. The conditions are: Flag 5 must be zero; he must be in "dark" locations and 10 percent probability will generate the actions. The actions are: print message 7; store 8 in flag 2 — counted-down by the program — and set flag 5 to prevent more vampires.

Message printed

EVT B checks if flag 2 has reached 1 and if so, says we are dead and EVT C prints message 5 if flag 2 is non-zero. Hence we have a 10 percent probability of being bitten by a vampire. We then receive the message 'I feel sick and dizzy' for seven turns before we die. In the authors' database, drinking some holy water clears flag 2 and hence prevents EVT B from executing so we survive the bite.

Now that we have described the operation completely perhaps some ideas on implementing it and swapping databases may be useful. In terms of hardware, all that is required is about 16K bytes of memory for a decent adventure.

The program should fit in less than 4K but you will find that the descriptive text, particularly for locations, will eat memory.

In terms of software, all you need are an editor and an assembler. However, if you have access to a disc-based system, all the better. Perhaps the best way to go about it is through your local computer club working on the program as a team and generating your own adventures.

As the database is pure data, any database will run on any machine — providing there is enough memory. However, the program still needs to know the position of the tables in memory.

Assembled listing

The simplest way to do this is to assemble the program to suit the database. Say we have Fred Smith's database which occupies memory from 0 to 2000 Hex and he has provided details of the start of each table. We assemble the program so that it starts above the database and we also define the table addresses by adding a small header to the program of the form:

```
        ORG 2000H ;Start of program
LOCTXT EQU 200H   ;Define table address
VOCAB  EQU 1000H ;ETC for rest of tables
OBJLOC EQU 50H
```

Another approach would be to make the tables of a fixed length and define specific addresses for them. That removes the need to re-assemble the program for each database but does not use memory very efficiently.

Unfortunately, Adventure is not the kind of game you can describe in such a way that the program can be blindly copied and played. However, I hope that the description given here will allow anyone to implement it on his system. If you're wondering if it is worth the effort, ask anyone who has played before.

```
!       ***********************************************************
!       *                                                         *
!       *                      ADVENTURE                          *
!       *                                                         *
!       * Programmed by - K Reed                                  *
!       * Date          - 12-May-80                               *
!       *                                                         *
!       ***********************************************************

        BEGIN DATA
!       External subroutines
!
!       $REPLY   - Gets a YES/NO response from the user
!       $MESS    - Ouput to console
!       $LINE    - Read a line from the console
!       $RAND    - Get a random number (Range 1-100)

        GLOBAL $REPLY,$MESS,$LINE,$RAND

!       Driving database labels
!       Message - User messages
!       Vocab   - Basic vocabulary
!       Loctxt  - Location descriptions
!       ObJloc  - ObJect locations
!       ObJtxt  - ObJect descriptions
!       Event   - Main event table
!       Movemt  - Location movements
!       Status  - Status check table

        GLOBAL Message,Vocab,Loctxt,ObJloc,ObJtxt
        GLOBAL Event,Movemt,Status

        WORD Here,HL,BC,DE,Rnum,I,J,User/<15>/

        BYTE Flag,W1,W2,Btemp,Ctemp,Doneit
        BYTE Word1/<4>/,Space/' '/,Cret/0/,Bneg1/-1/,Bzero/0/

        END DATA
        PROGRAM Adventure
start:  here=0                                  ! Start at location 0

moved:  CALL $mess(0)                           ! New line

!       User flag 0 when set indicates a dark location
!       He cant see unless object 0 is here
```

(continued on next page)

(continued from previous page)

```
!         Note that Objloc(0) is equivalent to simply Objloc
Desc:     BEGIN IF (User#0)
             IF (User(3)#0)User(3)=User(3)-1
             IF (Objloc=Here)GO TO Seen    ! Object here
             IF (Objloc=512)GO TO Seen     ! Carried
             TYPE 'Everything is dark. I cannot see'
             IF (User(4)#0)User(4)=User(4)-1
             GO TO Command
          ENDIF

Seen:     CALL $Mess(Loctxt(Here))          ! Describe Here

look:     Flag=0                            ! List objects here
          I=0
look1:    IF (objloc(I)=-1)GO TO command    ! End of objects
          IF (objloc(I)#here)GO TO next
          BEGIN IF (Flag=0)                 ! Object here
            CALL $mess(0)                   ! New line
            TYPE 'I can also see'
            Flag=1                          ! That message only once
          ENDIF
          CALL $mess(objtxt(I))             ! Describe object
          CALL $mess(0)
next:     I=I+1                             ! Next entry
          GO TO look1


Command:
          HL=#Status                        ! See if anything happens
          GO TO Active
Proc:                                       ! Returns here
          IF (User(2)#0)User(2)=User(2)-1   ! Count down active
          CALL $RAND(#Rnum)                 ! Keep random spinning
          CALL $Mess(0)
          CALL $Line                        ! Get a line
          HL=@SP                            ! Point to it
GETWD:    CALL Lookup(#W1)                  ! See if we know it

          BEGIN IF (W1=Bneg1)               ! Not found in table
             BEGIN IF (@HL=Cret)            ! No more words
Err1:           TYPE 'I just dont understand what you mean'
                BEGIN IF (Word1>90)
                   TYPE 'Perhaps if you used UPPER CASE ....'
                ENDIF
                GO TO Command               ! Try again
             ENDIF
Scan:        BEGIN IF (@HL=Space)           ! Next word
                HL=HL+1
                GO TO Getwd
             ENDIF
             IF (@HL=Cret)GO TO Err1        ! No more words
             HL=HL+1
             GO TO Scan
          ENDIF

!         If we fall out here we have a known word in W1. Now see
!         if we can find one for word number 2

          W2=Bneg1                          ! No word yet
Scan2:    IF (@HL=Space)GO TO Second        ! Found one
          IF (@HL=Cret)GO TO Allin          ! NO more
          HL=HL+1
          GO TO Scan2

Second:   HL=HL+1                           ! Point to word
          CALL Lookup(#W2)                  ! See if an object
          BEGIN IF (W2=Bneg1)               ! Not found
Scan3:       IF (@HL=Cret)GO TO Allin
             IF (@HL=Space)GO TO Second     ! Another word found
             HL=HL+1                        ! Keep looking
             GO TO Scan3
          ENDIF

!         See if this word causes a change of location

Allin:    HL=Movemt(Here)                   ! Point to movements
Moveit:   IF (@HL=Bneg1)GO TO Nomove        ! End of list
          BEGIN IF (@HL=W1)                 ! Entry found
             HL=HL+1                        ! Point to dest
             Btemp=@HL                      ! Go there.
             Here=Btemp                     ! Keep to bytes
             GO TO Moved
          ENDIF
          HL=HL+2                           ! Next entry
          GO TO Moveit

Nomove:


!         Look up the words in the main event table to see
!         what (if anything) happens.

          HL=#Event                         ! Point to table
          Doneit=0                          ! Clear flag
Active:   BEGIN IF (@HL=Bzero)              ! End of table
             IF (Doneit#0)GO TO Command     ! We did something
             BEGIN IF (W1<13)               ! Explicit movement
                TYPE 'I cannot go in that direction'
             ELSE
                TYPE 'I cant'
             ENDIF
             GO TO Command                  ! Get another command
          ENDIF

          IF (@HL=Bneg1)GO TO Entry         ! Any match
          IF (@HL=W1)GO TO Entry            ! Exact match
          HL=HL+6                           ! Next entry
          GO TO Active

Entry:    HL=HL+1                           ! Point to 2nd word
          IF (@HL=Bneg1)GO TO Match         ! Any match
          IF (@HL=W2)GO TO Match            ! Exact match
          HL=HL+5                           ! Next entry
          GO TO Active
```

```
Match:    HL=HL+1                           ! Condition pointer
          BC=@HL                            ! Get it
          HL=HL+2                           ! Point to actions

Check:    IF (@BC=Bneg1)GO TO Doit          ! End of conditions
          Btemp=@BC                         ! Get this condition
          BC=BC+1                           ! Next operand
          Ctemp=@BC                         ! Preload
          PC=TABLE1(Btemp)                  ! Computed GO TO

          BEGIN DATA
          WORD TABLE1/C0,C1,C2,C3,C4,C5,C6,C7,C8/
          END DATA



C0:       IF (Ctemp=Here)GO TO passed       ! Check current location
Cont:     HL=HL+2                           ! Next word pair
          GO TO Active                      ! Try next table entry

C1:       IF (Objloc(Ctemp)=Here)GO TO passed ! Object present
          IF (511<Objloc(Ctemp)<1025)GO TO passed
          GO TO Cont

C2:       CALL $Rand(#Rnum)                 ! Probable event
          IF (Ctemp>Rnum)GO TO passed
          GO TO Cont

C3:       IF (Objloc(Ctemp)=Here)GO TO Cont ! Object not here
          IF (511<Objloc(Ctemp)<1025)GO TO Cont
          GO TO Passed

C4:       IF (Objloc(Ctemp)#1024)GO TO passed ! Object not worn
          GO TO Cont

C5:       IF (User(Ctemp)=0)GO TO Cont      ! Flag not zero
Passed:   BC=BC+1                           ! Next condition
          GO TO Check

C6:       BC=BC+1                           ! Check flag value
          Btemp=@BC                         !
          IF (User(Ctemp)#Btemp)GO TO Cont
          GO TO Passed

C7:       IF (User(Ctemp)#0)GO TO Cont      ! Flag zero
          GO TO Passed

C8:       IF (Objloc(Ctemp)#512)GO TO Cont ! Object carried
          GO TO Passed


!         Condition met so perform the actions

Doit:     BC=@HL                            ! Point to actions
          HL=HL+2                           ! Point to next entry
          Doneit=1                          ! Say we have done something
Nxtact:   IF (@BC=Bneg1)GO TO Active        ! All done
          Btemp=@BC                         ! Get action
          BC=BC+1                           ! Point to next
          Ctemp=@BC                         ! Preload value
          PC=TABLE2(Btemp)                  ! Computed GO TO

!         In the following TABLE3 is simply a continuation of TABLE2
!         and not a sperate entity. It is done this way to keep the
!         compiler happy as it can't handle continuation lines

          BEGIN DATA
          WORD TABLE2/A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10/
          WORD TABLE3/A11,A12,A13,A14,A15,A16,A17,Done/
          END DATA


A0:       TYPE 'I Have with me'             ! Inventory
          Flag=0
          I=0
inven:    IF (Objloc(I)=-1)GO TO inven0     ! End of list
          BEGIN IF (511<Objloc(I)<1025)     ! Carried
             Flag=1
             CALL $mess(objtxt(I))
             BEGIN IF (Objloc(i)=1024)      ! Worn
                TYPE ' which I am wearing'
              ELSE
                CALL $mess(0)               ! New line
             ENDIF
          ENDIF
nextob:   I=I+1
          GO TO inven                       ! Next entry
inven0:   IF (Flag=0)TYPE 'Nothing at all'
          GO TO Done

A1:       BEGIN IF (Objloc(Ctemp)#1024)     ! Remove worn object
             TYPE 'I am not wearing it'
             GO TO Done
          ENDIF
          BEGIN IF (User(1)=4)              ! Hands full
             TYPE 'I cant. My hands are full'
             GO TO Done
          ENDIF
          Objloc(Ctemp)=512                 ! Say carried
          User(1)=User(1)+1                 ! Update tote
          GO TO Nxtop


A2:       BEGIN IF (User(1)=4)              ! Pick up object
             TYPE 'I cannot carry any more'
             GO TO Done
          ENDIF
          BEGIN IF (Objloc(Ctemp)=Here)
             Objloc(Ctemp)=512              ! Say carried
             User(1)=User(1)+1              ! Update total
             GO TO Nxtop
            ENDIF
          TYPE 'Im already carrying it'
          GO TO Done
```

```
A3:     BEGIN IF (Objloc(Ctemp)=Here)    ! Drop object
           TYPE 'I dont have it'
           GO TO Done
        ENDIF
        IF (Objloc(Ctemp)=512)User(1)=User(1)-1
        Objloc(Ctemp)=Here
        GO TO Nxtop


A4:     BEGIN IF (Objloc(Ctemp)=512)     ! Wear it
           Objloc(Ctemp)=1024            ! Say carried
           User(1)=User(1)-1
           GO TO Nxtop
        ENDIF
        BEGIN IF (Objloc(Ctemp)=1024)
           TYPE 'I am already wearing it'
        ELSE
           TYPE 'I dont have it'
        ENDIF
        GO TO Done


A5:     CALL $Mess(Message(Ctemp))       ! Type message
        GO TO Nxtop                      ! Get next action

A6:     GO TO Desc                       ! Describe location
A7:     GO TO Proc                       ! Procede

A8:     Here=Ctemp                       ! Immeadiate mqve
        GO TO Nxtop

A9:     User(Ctemp)=255                  ! Set flag
        GO TO Nxtop

A10:    User(Ctemp)=0                    ! Clear flag.
Nxtop:  BC=BC+1
        GO TO Nxtact

A11:    DE=Objloc(Ctemp)                 ! Swap objects
        Objloc(Ctemp)=Objloc(Ctemp+1)    ! Move 1st object
        Objloc(Ctemp+1)=DE               ! Move 2nd object
        GO TO Nxtop

A12:    STOP                             ! Stop the program


A13:    TYPE 'Okay'                      ! Say okay
        GO TO Done                       ! And procede

A14:    TYPE 'Are you sure you want to quit now'
        CALL $reply(#i)
        IF (i=0)GO TO Nxtact
        STOP 'Okay ... bye'
```

```
A15:    BC=BC+1                          ! Store value in flag
        Btemp=@BC                        ! Get it
        User(Ctemp)=Btemp
        GO TO Nxtop

A16:    Objloc(Ctemp)=Here               ! Create object
        GO TO Nxtop

A17:    Objloc(Ctemp)=2048               ! Destroy object
        GO TO Nxtop

Done:   GO TO Command


!       Lookup - Find word in table
!       Each entry consists of a four byte name
!       followed by an byte identification code. Eg 'FRED',2
!       This code is returned if found, otherwise -1.

        SUBROUTINE lookup(DE)

        LOOP J=0 TO 3                    ! Clear out word
           Word1(J)=Space
        ENDLOOP J

        LOOP J=0 TO 3                    ! Extract 1st 4 letters
           IF (@HL=Space)GO TO Gotwrd    ! End of word
           IF (@HL=Cret)GO TO Gotwrd     ! End of line
           Word1(J)=@HL                  ! Get character
           HL=HL+1
        ENDLOOP J


Gotwrd: BC=#Vocab                        ! Point to table
        @DE=Bneg1                        ! Assume no match

Find:   Flag=0                           ! fndit Flag
        LOOP I=0 TO 3                    ! 4 bytes
        IF (@BC=Bneg1)RETURN             ! End of table
        IF (Word1(I)#@BC)Flag=1          ! No match
        BC=BC+1
        ENDLOOP I

        BEGIN IF (Flag=0)                ! Matched
           Btemp=@BC                     ! Get ID
           @DE=Btemp                     ! Pass it to caller
           RETURN
        ENDIF

        BC=BC+1                          ! Skip over ID
        GO TO Find                       ! And try again

        END
```

ENTERPRISES

MAIL ORDERS, VISITS, TRADE ENQUIRIES WELCOME. CREDIT CARD ORDERS ACCEPTED BY TELEPHONE/TELEX
Payment must be in sterling, on a UK bank.

**Room PC
11, Cambridge House
Cambridge Road
Barking, Essex
IG11 8NT England
Tel: 01-591 6511
Telex: 892395 LPRISE**

BOOKS/MAGAZINES/SUBSCRIPTIONS

BY OSBORNE
Introduction to Microcomputers Series

Vol 0: Beginners Books	£6.95
Vol 1: BASIC Concepts	£8.30
Vol 2: Some Real Microprocessors (without binder)	£18.95
Vol 2: Some Real Microprocessors (with binder)	£24.70
Vol 2: Updating supplement set Nos. 1-6	£18.95
Vol 3: Some Real Support Devices (without binder)	£11.95
Vol 3: Some Real Support Devices (with binder)	£17.70
Vol 3: Updating supplement set Nos. 1-6	£18.95
1 Binder (Specify for Vol 2 or 3)	£5.75
1 Updating supplement (Specify for Vol 2o r 3)	£4.00
PET and the IEEE 488 (GPIB) bus	£9.95
6800 Programming for Logic Design	£6.30
8080 Programming for Logic Design	£6.30
Z80 Programming for Logic Design	£6.30
Z80 Assembly Language Programming	£8.15
6502 Assembly Language Programming	£ 8.25
8080A/8085 Assembly Language Programming	£7.95
6800 Assembly Language Programming	£7.95
Accounts Payable and Accounts Receivable	£13.15
Payroll with Cost Accounting	£13.50
General Ledger	£12.25
Some Common BASIC Programs	£7.95
Running Wild	£2.50

MAGAZINE BACK ISSUES

Micro 6502 Journal	£1.75
Personal Computing	£1.95
Interface Age	£2.95
Dr Dobbs Journal	£1.95
Computer Music Jurnal	£3.75
Recreational Computing	£1.95
BYTE	£2.95
Creative Computing	£1.95
Calculators and Computers	£1.95
Kilobaud Microcomputing	£2.95
Compute – for the 6502	£1.95
68 Micro	£1.95
80-Microcomputing	£2.25
On Computing	£1.95
Compute II – for the Single Board	£1.95
Magazine Storage Box (Holds 12)	£1.50

MAGAZINE SUBSCRIPTIONS (all processed within 3 weeks)

Micro 6502 Journal	(12 issues)	£13.50
68 Micro	(12 issues)	£17.50
Personal Computing	(12 issues)	£17.50
Interface Age	(12 issues)	£25.50
Dr Dobbs Journal	(10 issues)	£15.00
Recreational Computing	(6 issues)	£10.50
BYTE	(12 issues)	£25.00
Creative Computing	(12 issues)	£17.00
Kilobaud Microcomputing	(12 issues)	£22.00
Compute for the 6502	(6 issues)	£10.50
80 Microcomputing	(12 issues)	£21.00
Compute II – for the Single Board	(6 issues)	£10.50

FOR THE 6502
See Magazines and Subscriptions! See Osborne Books!

Best of Micro, Vol 1	£5.50
Best of Micro, Vol 2	£5.50
Programming the 6502 (Zacs)	£7.95
Programming the 6502 (Foster)	£6.75
6502 Applications	£7.95
6502 Software Gourmet Guide and Cookbook	£7.25
32 BASIC Programs for the PET	£10.10

FOR THE 8080
See Osborne Books!

8080 Programmers Pocket Guide	£1.95
8080 Hex Code Card	£1.95
8080 Octal Code Card	£1.95
8080 Software Gourmet Guide & Cookbook	£7.15
8080/8085 Software Design	£6.75
8080 Standard Monitor	£9.95
8080 Standard Assembler	£9.95
8080 Standard Editor	£9.95
8080 Special Package: Monitor, Editor, Assembler	£20.00
BASEX: A Simple Language and Compiler for the 8080	£5.50

FOR FUN

SARGON – A Chess Game	£9.50
BASIC Computer Games	£5.00
More BASIC Computer Games	£5.50
What to do After you Hit Return	£8.95
8080 Galaxy Game	£6.95
SUPER-WUMPUS – A game in 6800 Assembler Code & BASIC	£4.25
Computer Music Book	£6.75
Computer Rage (A board game)	£6.95
Artist and Computer	£3.95
Games, Tricks and Puzzles for a Hand Calculator	£2.49
Introduction to TRS-80 Graphics	£5.75
Take My Computer Please ... (Fiction)	£3.25
Introduction to Low Resolution Graphics for Pet, Apple, TRS8	£5.50
Starship Simulation	£4.50

FOR THE NOVICE
See Magazines and Subscriptions! See Osborne Books!

Your Home Computer	£5.95
Getting Down to Business With Your Microcomputer	£5.50
Introduction to Personal and Business Computing	£5.50
Getting Involved with Your Own Computer	£5.50
How to Profit from Your Personal Computer	£6.50
Microcomputer Potpourri	£1.95
Hobby Computers are Here	£3.95
New Hobby Computers	£3.95
Understanding Microcomputers and Small Computer Systems	£6.95
Understanding Microcomputers and Small Computer Systems and Audio Cassette	£8.75
How to Make Money with Your Microcomputer	£5.75
From the Counter to the Bottom Line	£10.00

• **Circle No. 168**