The Professional Adventure Writing System

Introduction

CPM 2.2
or
CPM+/3

The Professional Adventure Writer

An adventure writing system for CP/M computers.

Program: T.J.Gilberts, G.Yeandle and P.Wade
Graphics: A.Williams
Manuals: T.J.Gilberts and G.Yeandle

Acknowledgement

Thanks to Jill for putting up with the many hours I spent hammering away at the keyboard.

Contents

Introduction	Page 5
Getting Started	Page 6
Concepts	Page 9
Writing an adventure	Page 10
Start typing	Page 13
Playing the game	Page 17
Objects	Page 19
Process & Response	Page 26
The Bird	Page 41
The Dog	Page 46
Do it yourself	Page 52
End of the road	Page 53
Appendix A:	
EDIT: A simple Text Editor	Page 54
User Registration	Page 60

Introduction

Welcome to the world of adventure writing...

The Professional Adventure Writer (or P.A.W. as it is more commonly known) provides you with the facilities to produce high quality adventures (in machine code) of equal or better quality than many commercially available.

PAW will provide you with the basic framework for writing a game, but it is still up to you to provide an imaginative storyline and original puzzles.

The manuals supplied with PAW cover all aspects of its use. This manual provides a tutorial covering the construction of an adventure and we would recommend you work your way through it and its accompanying examples before attempting a game of your own. The other manual provides a detailed breakdown of the entire system and can be used as a reference guide while writing your own games.

Good luck...

A great deal of time and effort has been put into ensuring PAW deals with all conceivable situations in a logical and useful manner. This has resulted in a complex program of some 60K in size, and it is entirely possible that somewhere deep within the code a few well hidden bugs remain. Indeed a well known quote states that; "Testing only proves the presence of bugs, not their absence". If you should find a problem please tell us so that we can correct it if necessary.

Getting Started

The CP/M version of PAW consists of a Compiler and an Interpreter. Adventures are created by entering all information about the game into a text file known as a source file and then using the Compiler to produce a database. The Interpreter then uses the information in the database to provide the finished adventure.

A limited amount of knowledge of CP/M is assumed as follows:-

a) You should understand the format of CP/M file names.
   ie d:nnnnnnnn.ttt
b) You should know how to display a directory of your files.
   eg DIR
c) You should know how to delete a file eg ERA OLDFILE.TXT
d) You should know how to rename a file.
   eg REN NEWNAME.SCE=OLDNAME.SCE
e) You should know how to copy a whole disc to provide a backup.
f) You should know how to make a duplicate copy of a file eg to make a copy of START.SCE called TICKET.SCE you could use PIP TICKET.SCE=START.SCE
g) If you have more than one disc drive you should know how to refer to files on specific drives and how to copy files from one drive to another eg PIP B:=A:TICKET.SCE

Before we go any further ensure you have made a backup copy of the PAW disc and that you have put the original in a safe place.

It may be worth reading the READ.ME file on the release disc at this point to find out about any additions and or corrections since this manual was printed. This can be done using the CP/M command TYPE:

```
TYPE READ.ME
```

Installing the Interpreter

Before you can use the Interpreter it needs to be installed. The Interpreter needs to know, amongst other things, the number of columns on your screen, the number of lines on your screen and how to clear the screen. To install the Interpreter type:-
PAWINST PAWINT.COM

Your screen should now look something like this:-

```
Columns       is set to 90
Lines         is set to 31
Timeout       is set to 4680
Clear screen is set to '1B451B48' (hex)
Pause         is set to 17

Enter A to change Columns
```

```
      B to change Lines
      C to change Timeout
      D to change Clear screen
      E to change Pause
or    F to exit
Choose A-F?
```

Do not worry about the Timeout & Pause values for now as these are for 'fine tuning'. Check that the settings for Columns, Lines and Clear screen are right for your computer and if not correct them. Note that the Clear screen code has to be entered in hexadecimal and that the codes entered should clear the screen and place the cursor at the top left hand corner of the screen.

Some example values are:-

Computer	BBC with Z80 (2nd processor)	PCW8512 CP/M 3	CPC464 CP/M 2	CPC6128 CP/M +
Screen mode	MODE 3	24x80 off	MODE 2	MODE 2
Columns	80	90	80	80
Lines	25	31	25	24
Timeout	5461	4680	1705	1705
Clear screen	'0C'	'1B451B48'	'0C'	'1B451B48'
Pause	35	17	17	17

When you are satisfied that the values are correct for your computer enter 'F' to exit and you will be asked "Do you want to update the file on disc?". Answer Y if you have made any changes.

Choosing your Text Editor

To create your adventure you will need a Text Editor or Word Processor that can be used to produce 'Non-Document' files ie CP/M text files without control codes. ED80 by Hisoft, Protext or Wordstar in non-document mode are suitable examples. If you already have a suitable text editor then use it. If not you will need to use the one that we provide which is very basic. If you choose to use it then the instructions for installing it and using it are contained in appendix A of this manual, you will need to familiarise yourself with its use before proceeding.

Note: Be careful that certain wordprocessors use unusual symbols at the start of lines as control characters. eg The 'DOT' commands of Wordstar, and the '>' of Protext. The latter will cause problems with one of the system messages (33) which is the Marker used for the players input. Just put a space in front of the '>' or use a different character to prevent conflict.

A working disc

To get maximum use out of the PAW system, we advise that you make up a day to day working disc containing only those files which

are absolutely necessary, and any useful CP/M utilities. This is especially important in single drive CP/M 2 systems, where multi disc use is practically impossible. Our suggestions for various systems are as follows:

Single drive CP/M 2.2 (e.g. Amstrad 464 with DD or Amstrad 664).

Format a disc with the system option, so that you can use CTRL C to reset the disc system! Then copy STAT onto the disc. If you are using your own text editor copy that on, otherwise copy the installed EDIT.COM & EDIT.INS from the PAW disc. Finally copy the following files from the PAW disc:

PAWCOMP.COM
PAWINT.COM (after installing it correctly!)
START.SCE

When you then start a game merely rename START.SCE to be your game name. This will provide the maximum amount of space for your Source and Database files. STAT can be done away with if you don't want to know how much space is left on the disc!

Dual drive CP/M 2.2 or CP/M 3 (E.g. Acorn Z80, PCW 8512)

No problems here. Make a System disc up with any useful CP/M utilities and copy the major PAW files onto it. It is a good idea to leave about 50K free on this disc to accept your database files. Thus leaving the second drive totally free for Source. You will find the COMPILER faster if you compile the database to a different drive from that which holds the source.

Single drive CP/M 3 systems (E.g. CPC 6128)

With the virtual disc facilities of CP/M 3 you can store all your source and database files on a separate disc and then refer to it as drive B. NB. do not try to compile the database onto a different virtual disc to the source or you may end up with a sore arm from swapping discs!

RAM disc systems (E.g. PCW 8000 and 9000 series.)

Make up a day to day system disc as with the CP/M 2.2 single drive system, but with PIP as well. Then at the start of a session you merely copy all the files from this disc into RAM. (E.g. on a PCW you would PIP M:=A:*.*) then change the disc in drive A: to your Source disc. Make the RAM drive your current drive and away you go. Do not be tempted to use the RAM drive to hold your source, despite its speed it is volatile and the CEGB give no guarantees!

Concepts

It is probably a good idea at this stage to introduce some of the more important concepts which PAW embodies in its design.

The Source File

This is a collection of tables which contains all the information, in a human readable form, to define the adventure. File START.SCE is an example of a source file and is a useful starting point for an adventure.

The Compiler

PAWCOMP.COM is a program that checks the source file for errors and converts it into a PAW database.

The Database

This contains the output from the compiler and is a machine readable version of the Source File. It can thus be considered as a compiled adventure. START.PDB is an example of a database file.

The Interpreter

PAWINT.COM is the main part of PAW. It contains all the run time routines needed by a finished adventure. Essentially it fetches commands from the player and uses the information in the database to decode and respond to those commands.

Parser

Back to school for this bit:-

parse v.t. to classify a word or analyse a sentence in terms of grammar. **parsing** n. (Minster English dictionary)

PAW features a fairly powerful parser to convert what the player types when playing your adventure into a series of simplified 'Logical Sentences' (LS's) to which you will have defined the responses. The parser does this by extracting 'phrases' from the input string one at a time and allowing the rest of PAW to interpret their meaning. Phrases are separated by any punctuation mark and the conjugations 'AND' or 'THEN' - although you can change this if required. When it runs out of phrases in the current input string it will request another. A phrase consists of at least a Verb (a doing word!) and optionally two Nouns (words which name objects) possibly with associated Adjectives (which describe objects), an adverb (which modifies the verb), a preposition (shows the relation of one Noun with another word) and finally a string enclosed in quotes which is used for speech to other characters in the adventure.

Writing an adventure

Now the fun starts...

Planning

Planning your game is very important if you want to create a professional result. It is no use sitting at the machine and typing away in fits and starts as you wait for inspiration! You will merely entangle yourself in a maze of numbers and words, with no recourse but to start from scratch anyway.

To illustrate the recommended approach to writing an adventure we will consider the design and development of a simple game from initial idea to final testing.

Getting an Idea

This is always the hardest part of creating anything! An original storyline can provide a game with an interest which rescuing a princess will probably not evoke in a modern adventurer.

Subjects for adventures are all around in many day to day actions, in exotic places around the world and out of this world!

If you decide to base a game on a book or film you have enjoyed, and intend using it commercially, make sure you have obtained permission from the original author or copyright owners.

For our sample we will use a simple problem which beacts a passenger on his/her way home:-

While standing at the bus stop the passenger's ticket blows away in the breeze and is carried away by a small bird into an adjacent park. The computer will play the part of the passenger who you must direct to find the ticket before the bus arrives.

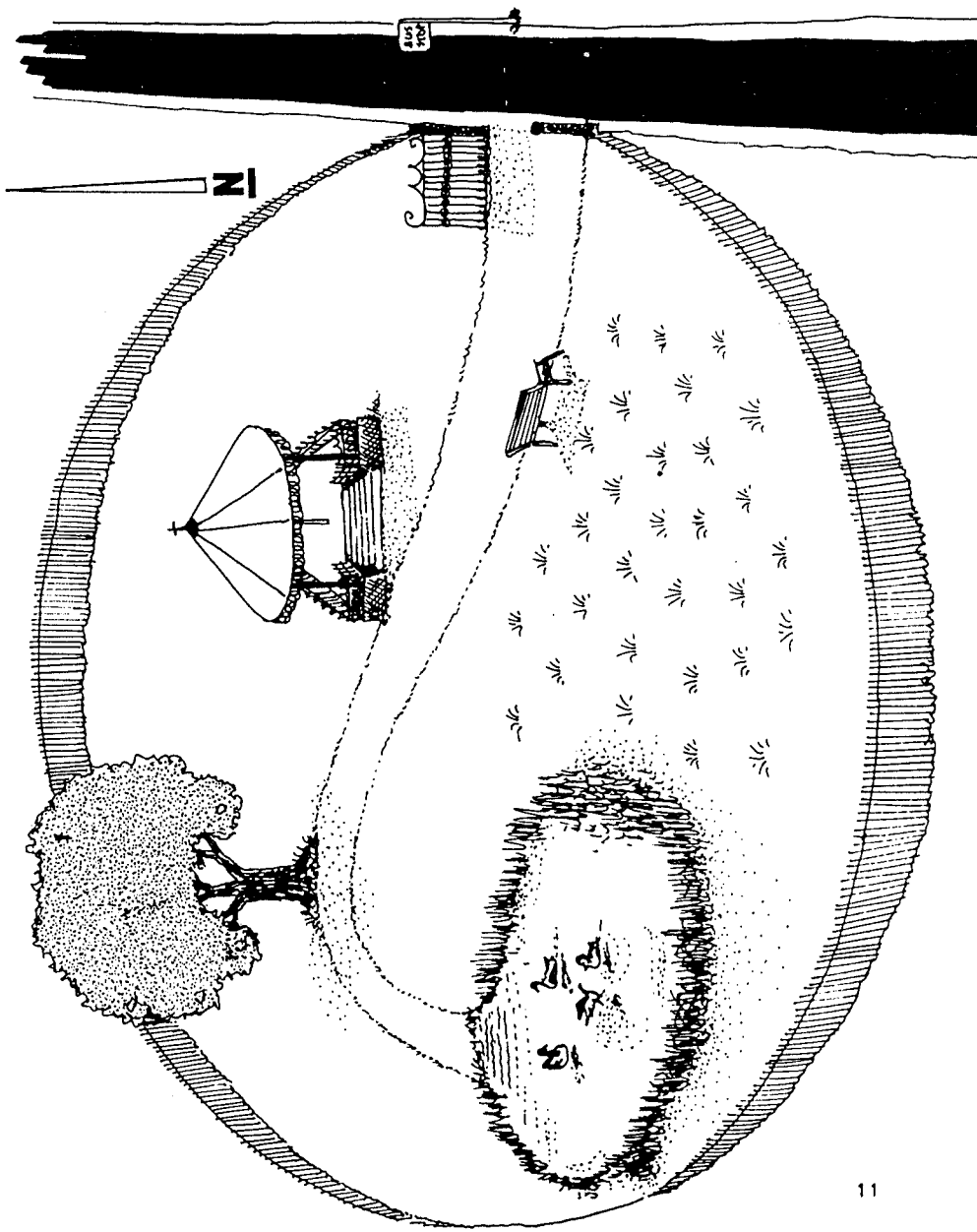
Game design

Now you have the idea, it is worth drawing a rough sketch of the area the game will take place within as we have done in diagram 1 (well actually our artist drew it...).

Note that any game design ought to be within a logically enclosed area or the player will wonder why they can't go in a direction when nothing appears to bar the way.

An adventure consists of a number of 'discrete' that is separate, 'locations' or places the player can visit. You must now decide which areas can become a location and number them individually.

Diagram 1



Writing an adventure

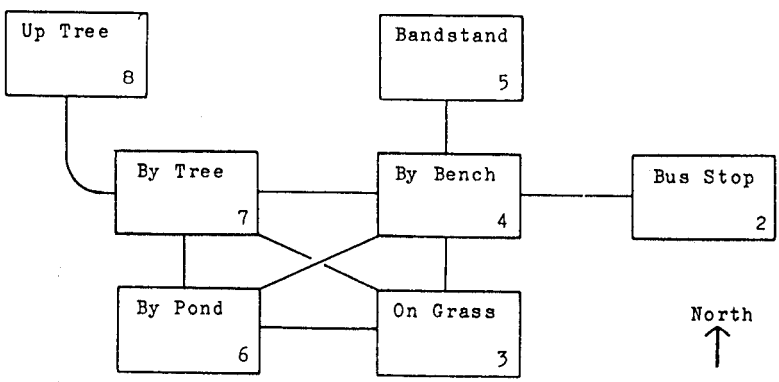
Try to make the scale consistent or logical (unless the game is illogical by intent!) as a single step to an airport earlier described as 10 miles away doesn't help the impression of realism. You can introduce a method of transport such as a taxi etc., if needed. Now location 0 is always reserved as a title screen for a game, and we want location 1 for a special use later, so we number the locations from 2 upwards.

For our example we have chosen 7 locations as follows:-

2 The Bus Stop.
3 On the Grass.
4 By the Bench.
5 The Bandstand.
6 The Ornamental Pond.
7 By the Tree.
8 Up the Tree!

To clarify the layout and to work out the possible movements, diagram 2 is a block diagram showing a stylized map of the game.

Diagram 2



Now we can start to write the textual description of each location. Try not to make them a dry and uninteresting monologue on the state of the nation. Short and snappy is just as effective in creating an atmosphere if a little imagination is brought to bear. Remember to stick to one form of address ('I' or 'You' usually) or the player will suffer a serious identity crisis. Note that if you use 'You' as a form of address you will need to change the system messages - see the technical guide for details.

Start Typing

We are going to call this adventure TUTORIAL so we need to create a source file called TUTORIAL.SCE. As a starting point make a copy of START.SCE called TUTORIAL.SCE. This can be done using the PIP program under CP/M. e.g. PIP TUTORIAL.SCE=START.SCE. Now edit TUTORIAL.SCE, using either your own text editor or the one we supply. In the case of EDIT you would type EDIT TUTORIAL.SCE.

The source file consists of a number of sections which describe the adventure. Let's take a quick look at the source file to see what's there. The first line is a comment, which tells us that this is a START source file and the date it was last altered. Any line which starts with a semi-colon is a comment. Comments are totally ignored by the compiler and thus take up no room in the database file when it is produced. Copious comments will help you remember what is going on in your game.

The CONTROL section begins with a line which starts with '/CTL' and gives some control information for the compiler.

The VOCABULARY section begins with a line which starts with '/VOC' and defines all the words which PAW will 'understand' eg QUIT.

The SYSTEM MESSAGES section begins with a line which starts with '/STX' and defines the messages that the system can issue eg "What next?"

The MESSAGE TEXTS section begins with a line which starts with '/MTX' and defines the messages actually needed by the adventure eg "It's a cheese sandwich!"

The OBJECT TEXTS section begins with a line which starts with '/OTX' and defines the descriptions of all the objects in the adventure eg "A rotten egg".

The LOCATION TEXTS section begins with a line which starts with '/LTX' and defines the descriptions to be used for every location in the adventure eg "You are in a prison cell. There is no way out."

The CONNECTIONS section begins with a line which starts with '/CON' and defines the connections between every location in the adventure.

The OBJECT DEFINITION section begins with a line which starts with '/OBJ' and defines every object in the adventure eg Object 0 starts the adventure being carried and has a weight of 1 unit, it is not a container and cannot be worn etc.

The PROCESS TABLE section begins with a line which starts with '/PRO 0' and provides the main game control. eg when the player

types in LOOK, action DESC is done to describe the current location.

Getting back to our tutorial, firstly change the comment in line 1 to read:-

```
;TUTORIAL source file                                    DD/MM/YY
```

Where DD/MM/YY should be replaced with the current Day, Month and Year. It is a good idea to keep track of file versions like this, so you can always tell if you are working on the latest version.

While we are at the top of the file, line 3 tells the Compiler to write the compiled database to drive A. You can change this if you prefer a different drive.

Now we are going to concentrate on the locations. Find the line which starts with /LTX and you will see that a description for location 0 already exists. The connections section, a few lines lower, also has an entry for location 0 but this is a null entry.

Change the location and connections sections so that they read as follows:-

```
/LTX ;Location Texts
/0   ;Intro
The Ticket


While standing on the bus stop my bus ticket has been blown
away, can you help me to find it?


/1
I'm inside the bag!
/2
I'm standing by a bus stop, on a road which runs North to
South.  To the West a park gate set in iron railings stands
open.
/3
The grass on which I stand is neatly trimmed. To the North is
a path and bench while to the West is an ornamental pond.
/4
I am on a gravel path running East to West, by a park bench,
to the South is a grassy area while to the North I can see a
bandstand.
/5
I am standing on the bandstand which appears to be made of
ornate cast iron painted white.  To the South is a path.
/6
The sun glitters on the surface of the ornamental pond, whose
waters ripple in the gentle breeze.  A path runs North towards
a large tree, while to the East is a grassy area.
```

```
/7
The path curves South and East here beside a large tree.
/8
I am sitting on a branch in a broad leaved tree, the park is
spread out before me, to the East I can see the bus stop
through the gate in the railings.
;    -       -       -       -       -       -       -       -
/CON ;Connectionn
/0                      ;Start of game
     N       2
/1                      ;In bag
/2                      ;Bus stop
     W       4
/3                      ;By bench
     N       4
     W       6
     NW      7
/4                      ;Path
     N       5
     E       2
     S       3
     SW      6
     W       7
/5                      ;Bandstand
     S       4
     SW      7
/6                      ;Pond
     N       7
     NE      4
     E       3
/7                      ;By tree
     U       8
     NE      5
     E       4
     SE      3
     S       6
/8                      ;Up the tree
     D       7
;    -       -       -       -       -       -       -       -
```

Points to note (lots of them):-

a) The text will be formatted by the Interpreter when the game is played to ensure that the text fits the screen width correctly without breaking words over lines.
b) Do not worry about location 1 for now.
c) The texts should not include any control codes. Any character with a value less than decimal 32 is considered a control code including TABS.
d) Ensure that there are no spaces on the end of any of the lines. The description of location 0 includes 4 null text lines. These lines must not include any spaces.

e) The compiler joins consecutive non-null text lines with a space.
f) The compiler converts null text lines into carriage returns.
g) For a 40 column screen the introduction would therefore be printed as:-

```
The Ticket

While standing on the bus stop my bus
ticket has been blown away, can you help
me to find it?
```

h) The connections entry for location 5 (for example) states that if the player is at location 5 he can type in 'S' (or SOUTH) to get to location 4 or 'SW' to get to location 7.
i) The words after the semi-colons are comments.
j) Each direction in the connections section eg 'SW' must be defined as a movement word in the vocabulary.
k) For every location text in the location section there must be a corresponding entry in the connections section.
l) We have included a way of getting from location 0 (the introduction) to location 2. There is a better way but this will do for now.

Check these entries thoroughly against diagrams 1 & 2 until you are sure they are correct then save TUTORIAL.SCE, and exit back to CP/M so that we can try the compiler. With EDIT our simple Text Editor CTRL & K will allow you to save the file and then return you to CP/M.

Now let's try out the Compiler. Assuming that PAWCOMP.COM and TUTORIAL.SCE are on the default drive you need to type:

```
PAWCOMP TUTORIAL
```

to compile the adventure as it stands at the moment. We have not included any deliberate errors so you should eventually get a message saying "Compilation ends OK" and a file called TUTORIAL.PDB. If you get any errors or warnings you will have to correct them and recompile the source file. Note that you can redirect the compilation listing to a disc file if you prefer by specifying PAWCOMP TUTORIAL P which will create TUTORIAL.PRN on the default drive or PAWCOMP TUTORIAL Pd which will create d:TUTORIAL.PRN. If you get really stuck with errors in the source file then the file TICKET.SCE may help you but note that this file contains the whole tutorial adventure.

Note. An extremely common mistake, which can cause apparently unfathomable problems, is inserting a blank line! Particularily a blank line between the end of one section of the source file and the header of the next. i.e. above the /VOC, /PRO etc., lines. Or a blank line anywhere in the /CTL section which PAW will get very upset about!

Playing the Game

Now we can have our first use of the Interpreter (and see if you installed it properly!). Assuming PAWINT.COM & TUTORIAL.PDB are on the default drive you need to type:

```
PAWINT TUTORIAL
```

The screen should be cleared and at the top it should say something like:-

```
The Ticket

While standing on the bus stop my bus ticket has been
blown away, can you help me to find it?

What now?
>
```

If not you will have to reinstall the Interpreter and try again. Assuming your Interpreter is installed correctly we can continue.

The input line is used to enter commands for PAW to interpret into things to do - according to the information you have entered when writing your game. So far we have only told it where to take us when certain directions are entered. So try starting the game properly by typing NORTH or N.

The screen will clear and the description for location 2 will appear - if it doesn't you probably have the entry in the connections section wrong. Don't worry, you can leave the Interpreter by typing QUIT (which is a command PAW knows to start with) and replying Y; you do want to quit and N; you don't want to try again; then you can correct the entry, recompile and try again.

We need to mention diagnostics now. If you press RETURN/ENTER (whichever you have on your keyboard) without typing anything in first, the Interpreter will be switched into diagnostics mode and you will see a line something like:-

```
Flag 38=2 ?
```

which shows you the value of flag 38. There are two things to be learnt here. Firstly, pressing RETURN/ENTER again (without typing anything in first) will switch the Interpreter out of diagnostics mode. Secondly, that flag 38 always contains the value of the current location, ie 2 in the example above.

You can now try moving between your locations, testing the possible moves (make a note of any which are wrong so that you can correct them).

You might also like to try some of the other commands which PAW knows, e.g. R or REDESCRIBE will display the location description again - which is useful if a lot of text has been output and the description lost. I or INVENTORY will list the 'objects' you are carrying, you will be carrying one object to start with but will be able to do nothing with it.

You are probably dying to see the parser in action by now (why not?) so if you work your way back to the bus stop and enter the following line you will get a flying visit round the game!

```
GO WEST THEN NORTH THEN SW THEN UP AND DOWN THEN SOUTH
AND EAST THEN NORTH THEN EAST. INVENTORY
```

By the way W.N.SW.U.D.S.E.N.E.I will have the same effect but doesn't look half as impressive...

Right back to the boring bit, QUIT from the game as shown previously so that we can deal with the next chapter in this saga.

Objects

Objects are anything which the player can manipulate within the game, for example, An apple which they could eat, a key which they could use to unlock a door, or a rucksack to contain the key and the apple!

In our simple game we will have the following objects (not all of which have a function in the final game).

```
Object 0        A lit torch.
Object 1        A bag.
Object 2        A sandwich.
Object 3        An apple.
Object 4        A ticket.
Object 5        A lead.
Object 6        An anorak.
Object 7        An unlit torch.
```

Note that the torch is in fact two separate objects which we could swap over if the player turned it on or off.

Start editing TUTORIAL.SCE again and find the line which starts with /OTX. You will see that a description for object 0 already exists. Change the object text section so that it reads as follows:-

```
/OTX        ;Object Texts
/0
A lit torch.
/1
A bag.
/2
A sandwich.
/3
An apple.
/4
A ticket.
/5
A lead.
/6
An anorak.
/7
An unlit torch.
; -      -      -      -      -      -      -      -
```

Points to note:-

a) Ensure that there are no spaces on the end of any of the lines.

Now find the line which starts with /OBJ ie the Object definition section and change it to read as follows:-

```
/OBJ ;Object Definitions
;obj  starts    weight  cont-    wear/   noun    adjective
;num  at                 ainer    remove
/0              1
/1    2         3       Y        _       TORCH   LIT
/2    CARRIED   1       _        _       BAG     _
/3    CARRIED   1       _        _       SANDWIC _
/4    8         1       _        _       APPLE   _
/5    3         1       _        _       TICKET  _
/6    WORN      3       _        Y       LEAD    _
/7    CARRIED   1       _        _       ANORAK  _
;     -         -       -        -       TORCH   UNLIT
                                          -       -
```

Points to note:-

a) All the _ characters are underlines.
b) You can use TAB's to separate the columns in the object definition section (if your text editor will allow you).
c) There must be the same number of entries in the object definition section as in the object text section.

What on earth does all that mean I hear you ask. Well the first column contains the object number (and they must be defined in the correct order). The second column states where the object is at the beginning of the game; at the start of the game object 2 is carried by the player, object 6 is worn by the player, object 5 is at location 3 and object 0 is nowhere, ie does not yet exist within the game. The third column defines the object's weight; objects 1 and 6 are three times as heavy as the other objects. The next column specifies which objects are containers, ie whether they can contain other objects. We have decided that only the bag is to be a container, you have to imagine that the anorak has no pockets (well ours hasn't anyway). The fifth column determines whether the object can be worn and removed - in our game only the anorak can be worn and removed. Note that any object which starts the game worn must also be marked with a Y in the fifth cclumn. The next column (6th) is used to specify the noun to be associated with each object. Note that only the first 5 characters of each word are significant so that SANDW, SANDWI, SANDWIC and SANDWICH would all mean the same thing. The final column is used to specify the adjective to be associated with an object if a noun alone is insufficient to distinguish an individual object.

This is an excellent time to show you some compiler errors so save TUTORIAL.SCE and try to compile it again. (PAWCOMP TUTORIAL if you've forgotten).

When the compiler finds an error, the line number of the source file and the contents of the line are printed (when appropriate)

then the error number and error reason eg;

```
344 /1  2       3       Y                BAG
*** ERROR 5 - BAG   is not in Vocabulary
```

We are expecting the compiler to find 8 errors so the compilation should end with the line:-

*** Compilation ends with 8 ERRORS

The 8 errors have been found because we have used words eg BAG, which have not been defined in the vocabulary section. Guess which section we are going to look at next!

Vocabulary

The vocabulary is a list of all the words which PAW is able to recognize in any input the player types in during the game. Thus any words which aren't in this section will have no effect at all! START.SCE contains about 70 common English words which will be required for most adventures.

Each entry for a word consists of up to five letters which will either be a complete word e.g. NORTH or the first five letters of a longer word e.g. ASCEN(D), a word value and a word type (e.g. Noun, Verb etc).

The use of only five letters to store a word reduces the amount of memory required to store the entire vocabulary, the amount of typing the player must do and makes PAW faster at looking up words when required. Five letters is also more than adequate to differentiate the majority of important words in the English language from each other.

Start editing TUTORIAL.SCE again and find the vocabulary section - /VOC. You should be able to see that each entry consists of a word followed by a word value and a word type eg

```
N       2       noun
```

PAW understands 7 types of words ie

```
noun
verb
adjective
adverb
preposition
pronoun
and  conjugation
```

If you look at the first part of the vocabulary section you should be able to find all the directions that we used in the connections section (eg N & SW). Note that any words with the same word value and the same word type are regarded as synonyms, ie they mean the same thing eg N & NORTH are synonyms. It does

not matter which order the words are defined in the vocabulary section. We have chosen to group all the adverbs together and all the adjectives together etc..

We need to increase the number of Nouns by inserting a word for each of our objects. Now the first free number appears to be 15, but Noun values less than 50 have special meanings thus;

Nouns less than 50 are Proper Nouns, for example people's names or places. More specifically for PAW they are Nouns which will not affect the subject of 'it'. Take the sentence:-

GET THE SWORD AND CLEAN IT

IT (a word known as a pronoun) refers to the SWORD obviously. PAW will (as long as SWORD is a Noun in the vocabulary with a value greater than 49) know this, and assuming you have dealt with the possibility of cleaning the sword, allow you to do so. But take the following sentence:-

GET THE SWORD AND KILL THE ORC WITH IT THEN DROP IT.

Normally PAW assumes 'it' to be the last used Noun but as long as ORC is a Noun in the vocabulary with a word value less than 50, PAW will remember 'it' as being the sword, and carry out the action correctly. This feature is noted in the comments along with mention of word values less than 20. These are Nouns, which if PAW cannot find a Verb in a 'phrase' containing one, will convert temporarily (i.e. it does not change the vocabulary) into a Verb. The major use of this is for things like NORTH which may be typed on their own implying GO NORTH, which in normal English is invalid but is common when playing adventures.

Finally verbs and nouns less than 14 are assumed to be movement words - any word which is a direction. This merely determines the message which will be printed if PAW cannot do anything with the phrase it has found (i.e. it determines if "I can't" or "I can't go in that direction" is displayed). Note that this tag of 'less than 14 is a movement' applies to both Verbs and conversion Nouns.

Since all our objects are 'its' we must give them word values greater than 49 as follows:-

TORCH	50	noun
BAG	51	noun
SANDW(ICH)	52	noun
APPLE	53	noun
BUS	54	noun
TICKE(T)	54	noun
LEAD	55	noun
ANORA(K)	56	noun

Note that there are two words with value 54. This makes BUS and TICKET synonymous so if the player types GET BUS TICKET or GET TICKET, PAW will know they mean the same thing.

We also need some words to describe the difference between our two torches to PAW. The words which describe a Noun are called Adjectives. We need two extra adjectives LIT and UNLIT as follows:-

LIT	10	adjective
UNLIT	11	adjective

All word values from 1 to 254 are available for each type of word and there is no limitation on the number of words with the same word value (synonyms), so the vocabulary can become quite large if you want.

Alter the vocabulary section to include our extra 8 nouns and 2 adjectives, save TUTORIAL.SCE and compile it again. Correct any errors and recompile until there are no errors or warnings.

Play it again...?

OKAY let's try the Interpreter again (PAWINT TUTORIAL) and type NORTH to get to the Bus Stop. Firstly a bit more about diagnostics (you can enter and leave diagnostics mode by giving a null input ie just by pressing RETURN). Do that now... (please!)

PAW contains 256 of what are known as 'flags'. Each flag can be used to contain a number from 0 to 255 and is used to indicate (or flag!) the state of some part of the game. e.g. You could decide that flag 11 when set to 1 meant that the park gate was closed, and when set to 0 meant it was open. We will see examples of the way flags can be set and used in the next section.

PAW has set aside several of the flags to indicate specific things (flags 0 to 10 and 29 to 59 actually). The value displayed is the contents of flag 38 which PAW knows is your current location eg

```
Flag  38=    2 ?
```

You can look at the values of other flags by typing their number before pressing RETURN again. Try 100 to look at flag 100, which will display:

```
Flag 100=    0 ?
```

A very powerful feature allows you to set the value of a flag by putting = in front of the number. Try =10 RETURN and the line should be redisplayed as:

```
Flag 100=   10 ?
```

Flag 100 does nothing in our game and its value is unimportant, but if you decide to practice on your own do not change the values of any other flags for the moment or you may get some funny effects if you happen on a flag which is important. Return to the input line when you have finished (press RETURN) so that we can see what else PAW can do.

We should now be able to manipulate the objects in the game. At the moment the bag will be at the bus stop with us. We will be carrying the sandwich, apple, unlit torch and wearing the anorak.

Use the diagnostics to look at the value in flag 1. This has the value 3, which is the number of objects carried but not worn. Return to the input line and type GET BAG. PAW will print the message "I now have the bag.", which is known as auto-reporting (PAW automatically reports any action it has carried out). This command has caused the current position of the bag to be changed from location 2 (the bus stop) to 'location' 254 (carried). Note that no change has occurred to the object definition section of the database, only to a copy of it which was made when the game started. If you look at the value of flag 1 again (notice how the flag you looked at last is displayed when you reselect diagnostics) you should find it has been increased to 4.

Now try REMOVE ANORAK and the report "I can't remove the anorak, my hands are full" will be printed. This is because PAW initially (by default in other words) allows the player to carry only four objects at any one time. This situation must prevent the player from taking off clothing etc (actually removing is changing an objects position from location 253 to location 254!) Try DROP BAG and then REMOVE ANORAK again - this time you should be able to do so. Look again at flag 1 and you should discover it is still four - this is because removing the anorak has increased the number of things you have in your 'hands'.

Try the following and see if you can work out why they do what they do:

GET BAG

REMOVE ANORAK

WEAR ANORAK

GET APPLE

GET TICKET

Notice that all except the last report actually mentioned the objects by name. This is because they were in plain sight and thus the player would know they existed. But to a player who did not know the game, the ticket has not yet been found and to mention it by name would imply that it existed or that there was

only one in the whole game thus giving a clue!

If you try to put anything in the bag you will discover that PAW drops that object instead. This is because we haven't yet told PAW **what** can be put in the bag only that it is a container. The next chapter deals with this subject.

Finally we will find a 'bug' in our game; type GET GATE which will result in "There isn't one of those here". It shouldn't say that because the description says there is a gate here!

The problem arises because although we told PAW about the apple the sandwich, the torch and so on, we didn't tell it about the gate. If you use GET (or DROP,WEAR and REMOVE) with any word which is not in the vocabulary then PAW assumes it is an object which is 'not here'. Of course once the word is in the vocabulary, PAW will know it isn't an object (if there is no entry for the word in the object definition section) and report "I can't do that.", which is correct.

So edit TUTORIAL.SCE and insert the following vocabulary entries:-

```
GATE         57   noun
RAILI(NGS)   58   noun
GRASS        59   noun
PATH         60   noun
BENCH        61   noun
POND         62   noun
BANDS(TAND)  63   noun
IRON         63   noun
TREE         64   noun
BRANC(H)     64   noun
LEAF         64   noun
```

Notice how all the ways the player can refer to the tree are catered for. We have no intention of allowing the manipulation of leaves or the branch, but if you did you would need to give them separate word values - this is an important design consideration.

You might like to use the Interpreter again to ensure that GET GATE does indeed produce the correct response - don't forget to re-compile TUTORIAL.SCE if you do, or the 'bug' will still be there! This arises of course because the database file reflects the state of TUTORIAL.SCE the last time it was compiled.

We have now dealt with; creating locations and connecting them together, creating and describing objects, assigning them a word from the vocabulary, a starting point in the game, a relative weight, flagging if they are wearable (and removable) and if they are a container. The next chapter goes on to create problems and characters to make the game world a more interesting place by allowing the player to do things!

Process & Response

We now come to the section of PAW which allows the problems and characters in the game to be created.

The Response table

The response table is a special form of what PAW terms a process table. In fact the response table is process table 0. When we say response table we mean process table 0. A process table can be thought of as a simple sequential (it does each command in turn) programming language, the commands which are carried out are called 'CondActs' because they can be divided mainly into two groups; Conditions and Actions.

Earlier we mentioned that the parser in PAW breaks sentences down into phrases, which are then organized into what is known as a LS (logical sentence). In the case of directions like NORTH (which are LS's on their own) it uses the connections section to discover where (if at all) it should move the player to. Before it does that however it carries out a check against the response table to see if that table contains an entry which can deal with the LS, i.e. give a response to part of/entire command the player originally typed.

Every possible phrase the player types and therefore every LS that your game will respond to, will have a corresponding entry in the response table, except for most movements which you set in the connections section.

The most important part of a LS is the Verb, this shows the purpose of the LS, next most important is the first Noun which shows the subject of the LS; e.g. GET APPLE, GET is the purpose and APPLE is the subject.

Look at TUTORIAL.SCE and find the start of the response table. It starts with the line:-

```
/PRO   0...
```

For the moment ignore the other entries and consider the first entry only:

```
I      _       INVEN
```

the two words "I" & "_" indicate the Verb and Noun respectively of the LS that this entry can deal with. Now I is a conversion noun (as we saw in the section on vocabulary) which means if it is the only word the player types in a phrase, it will become the Verb for the LS. The underline (_) indicates that the Noun is not important in this entry. What this means in simple terms is that if the player types I on its own PAW will match it up with the

first entry in the response table and carry out that entry as described next. In order to carry out the entry, PAW will execute each of the condacts (commands) in the list which follows. Now the first entry contains only one condact;

INVEN is an action (the act part of the word condact!). It is an action because it carries out the act of listing the objects the player is carrying and wearing on the screen, you do not need to worry how INVEN does this it just does.

When you typed I (or INVENTORY which is synonymous remember) during testing the game, it was this entry in response that caused something to happen because a logical sentence of "I _" was created by the parser, which PAW then found matched the first entry in the response table.

INVEN once it has listed any objects you are carrying, instructs PAW it has 'done' something, when PAW discovers this it asks the parser for another LS, which the parser provides by decoding the next phrase in the players input, PAW gets this LS and checks it against the entries in response and so on. This 'loop' is shown in diagram 3 in the form of a flowchart which you should follow from the box marked 'start'. The loop is slightly more complex than the diagram might lead you to believe and a complete one is given in the technical guide, but diagram 3 will do for now.

We advise you reread the above paragraphs and study the diagram until you are happy with the way PAW operates on LS's before proceeding.

Let's consider the second entry in response:

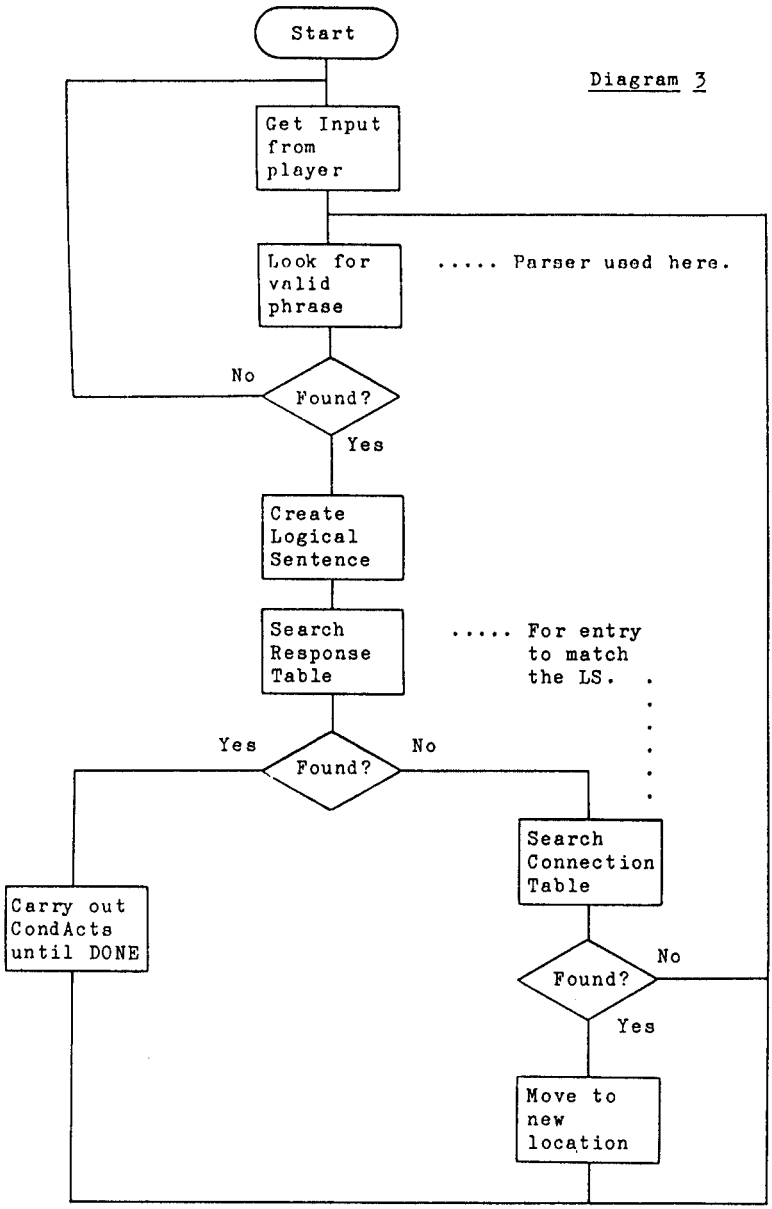
```
GET   I      INVEN
```

as you might have worked out this entry deals with the phrase TAKE INVENTORY (GET is a synonym of TAKE, I is a synonym of INVENTORY) this deals with another way the player might request a list of the objects he has with him.

We will skip the next few entries and move onto:

```
QUIT  _      QUIT
             TURNS
             END
```

Now, QUIT is a Verb in the vocabulary, so, as the minimum valid phrase is a Verb, if QUIT is typed on its own by the player then the parser will generate a LS of "QUIT _", on searching through the response table PAW will find the above entry and start to carry out the condacts which follow;


Start
Diagram 3
Get Input from player
Look for valid phrase
..... Parser used here.
No
Found?
Yes
Create Logical Sentence
Search Response Table
..... For entry to match the LS.
Yes
Found?
No
Search Connection Table
Carry out CondActs until DONE
No
Found?
Yes
Move to new location

QUIT is a condition, (the cond part of the word condact), do not confuse the Verb QUIT in the vocabulary with the condition QUIT, if you were to make STOP a synonym of QUIT and then delete the word QUIT from the vocabulary then the player would have to type STOP to end the game, but the condition QUIT would still be carried out i.e. the entry would then read:

```
STOP  _     QUIT
            TURNS
            END
```

A condition merely decides if PAW should carry out the next condact in the list. QUIT determines if the next condact should be carried out by asking the player "Are you sure?". If they reply "NO" then QUIT tells PAW it has 'done' something which causes PAW to go and get another LS (i.e. it stops the QUIT) this is slightly different to the normal way a condition works as you will see later. If the player types "YES" then QUIT does nothing and allows PAW to look at the next condact in sequence which is TURNS.

TURNS is an action, which prints "You have taken x turn(s)." on the screen where x is the number of phrases that PAW has carried out since the player started the game. Despite the fact it has done something it does not tell PAW to stop looking at condacts which proceeds to look at the next condact END.

END is a special action, which prints "Would you like another go?" on the screen. If the player types "YES" then END will cause the game to be restarted with all objects restored to their required position and so on. Otherwise END returns you to CP/M.

Note that you should always have an END action somewhere in the game (if you should happen to remove the QUIT entry that is) or you may not be able to return to CP/M very easily.

The other entries which are present in the response table deal with a number of other standard commands which the player of an adventure will usually need. The condacts used in the other entries are discussed below. You may be wondering why these entries are in the table and not part of PAW if they are needed in every game. Well apart from the fact it is easier to make them a table entry, **your** game might not need them and as they are a table entry they can be deleted.

DESC is an action, used by the "R _" entry in the table which causes PAW to abandon scanning the response table and reDESCribe the current location of the player.

SAVE and LOAD are two actions which allow the current state of the game to be saved and reloaded from disc, the current game position includes every piece of information needed to restore the game after a LOAD to exactly the same position it was before

the SAVE and includes the values of flags, position of objects plus sundry other information. Again do not confuse the Verbs SAVE and LOAD in vocabulary with the actions SAVE and LOAD. You could equally as well use STORE and RECAL(L) as your vocabulary Verbs but they would still use the SAVE and LOAD actions in the response table. Note that both SAVE and LOAD effectively do a DESC action when they have finished which means any condacts which follow will be ignored and that they also cause any further phrases in a players input to be ignored.

RAMSAVE and RAMLOAD are two actions similar to SAVE and LOAD, except that they use a 'buffer' (area of free memory) to store the game position. Only one position can be stored and as it is stored in memory it will be lost if the computer is turned off, this should be made clear to the player. The number after RAMLOAD is a parameter and tells the condact how many of the flags to restore from the previous RAMSAVE, this allows scores etc to be maintained even if the player 'cheats' by using RAMSAVE and RAMLOAD in a difficult part of the game. They are both followed by a DESC action as unlike SAVE and LOAD they just continue onto the next condact. Note that diagnostics are not available following a RAMSAVE.

If PAW runs out of condacts in a list without being told it has DONE something it will 'drop off' the end and realizing this will continue to search response for another matching LS. We also said that QUIT was a bit different to normal conditions, well for a start it is the only condition which asks the player for information and secondly it tells PAW something has been done if the player replies "NO" (they don't want to abandon the game) which causes PAW to get a new LS. A normal condition if it 'failed' would merely cause PAW to continue searching the response table for another entry matching the LS.

The other condacts which are used will be considered now in relation to the entries they are part of. To simplify our explanations we can consider the position of an object to be one of four places;

HERE: The current location of the player (the value stored in flag 38 if you remember).

CARRIED: 'location' 254, the imaginary location which is where all objects the player is carrying are stored.

WORN: 'location' 253, the imaginary location which is where all objects the player is wearing are stored.

NOTHERE: Anywhere else! This may also include 'location' 252 which is the imaginary location where any objects which do not yet 'exist' within the game are stored.

Take the following two entries in the response table:

```
GET   ALL   DOALL     HERE

GET   _     AUTOG
            DONE
```

these two entries allow the player to GET an object. GETting an object involves changing its location from HERE to CARRIED.

Ignoring the GET ALL for a moment let us look at the GET _ entry, as we said earlier underline means 'any word' so no matter what Noun the player types in, in the phrase containing the GET the GET _ entry will match (this is called triggering the entry). Take the phrase GET THE APPLE; THE will be ignored because it is not in PAW's vocabulary, so the LS will be "GET APPLE", this will 'trigger' the GET _ entry resulting in PAW looking at condact;

AUTOG is an action which AUTOmatically Gets the object specified by the Noun. This is where the object definition section comes into effect, AUTOG looks through the object definition section for an entry which matches the Noun in the LS, when it finds one (APPLE in the example) it then knows the number of the object it refers to (the apple is object 3), it then ensures that the current location of that object number is HERE and if so changes it to CARRIED and prints the message "I now have the _." where the underline is replaced with the description of the current object. i.e. the one AUTOG just looked up. If it does not succeed in finding an entry then there are five possibilities;

1/ The player has tried to get an object which they are already carrying or wearing in which case "I already have the _." is displayed.

2/ The player has tried to get an object which is NOTHERE in which case "There isn't one of those here." is displayed.

3/ The player has tried to get something which is not an object but does have a word in the vocabulary (e.g. GATE in the demo game) this results in "I can't do that.".

4/ The player has used a word which is not in the vocabulary which causes the parser to create a LS of "GET _" which triggers our GET _ entry anyway. AUTOG assumes this to be a Noun describing an object (which may or may not exist) and displays "There isn't one of those here.".

5/ The player is unable to carry any more objects or this object would cause the weight limit to be exceeded in which case a suitable message is displayed.

If AUTOG succeeds then PAW looks at the next condact DONE;

DONE merely tells PAW that this entry is finished and it should

go and get another LS.

Next we will look at the GET ALL entry, you may have guessed what this does (it attempts to GET all objects at the current location), so we shall explain the mechanism;

Should the player type the phrase GET ALL, the parser will create a logical sentence of "GET ALL", which will match the entry and cause PAW to look at the DOALL action;

DOALL is an action which is followed by a parameter which gives a location number to use. DOALL looks through the current location list for each object looking for entries that are at the same location as the parameter , when it finds one it looks in the object definition section to find the vocabulary word which describes that object number, this is placed in the current LS (thus replacing the Noun ALL), a flag is set to indicate that DOALL is active and the rest of response is scanned by PAW for an entry which matches the newly modified LS. This will be the GET _ entry discussed earlier, which will GET that object. Once this has been done PAW will discover that DOALL is active and go back to the GET ALL entry (actually it goes direct to the DOALL action) and allows DOALL to look for another object which generates a new LS and so on for all objects at the specified location. When DOALL runs out of objects it resets the flag to show it is not active and tells PAW to get a new LS.

This may seem a rather roundabout way to approach this task, but if you examine the very similar DROP,WEAR and REMOVE entries you will see that the same mechanism is used to create all four commands. AUTOD, AUTOW and AUTOR work in a very similar way to AUTOG while DOALL merely uses CARRIED as the parameter for DROP and WEAR (i.e. DOALL searches all the CARRIED objects when you try to DROP or WEAR ALL!) and WORN when you try to REMOVE ALL.

If the above seemed a bit heavy going don't worry about it for now as DOALL is one of the two most complex condacts in PAW and hopefully the penny will drop as we continue. At this point you might like to use the Interpreter to try out the 'all' commands which may make the mechanism clearer.

Messages

Before we continue with the response table we will insert some entries in the messages section which will be needed. So edit TUTORIAL.SCE again and find the messages section (MTX not STX).

Messages should be a breath of fresh air after that discussion of the response table, the purpose of messages is to contain all the text which will be displayed to describe what is happening in the game to the player, excluding the messages that PAW itself displays (like "I can't do that." etc).

We are going to deal with the player wanting to examine things in the game, e.g. EXAMINE APPLE. Now examining an object merely requires the writer to provide a message which gives more information about the specified object, so in the case of the apple we could say "The apple is crisp and green.".

Change the message text section so that it reads as follows:-

```
/MTX        ;Message Texts
/0
The apple is crisp and green.
/1
It's a cheese and pickle sandwich.
/2
The ticket has "City Bus Company" printed on it.
/3
The bench is firmly screwed to a concrete base.
;  -     -     -     -     -     -     -     -
```

We are going to deal with only four items in the demo game but in a large game you would usually provide detail for most things, even if they serve no purpose it provides a touch of realism which always makes the player feel involved.

Now back to the response table. Let's take the apple first; the phrase which the player will type will be EXAMINE THE APPLE (or EXAMI APPLE if they are lazy!) producing a LS of "EXAMI APPLE". So we need to insert an entry with these two words.

Now we must only allow the player to examine the apple if it is actually HERE, CARRIED or WORN (most normal people have a distinct disability at looking round corners or through walls!), this is collectively known as present and can be checked for using the condition PRESENT followed by the number of the object we are considering (the apple is object 3). If the object is indeed present then we can display our message (0) which describes the object, using the action MESSAGE which is followed by the number of the message you want to display. Finally we top it off with a DONE action to tell PAW that we have completed the task

So the entry we need is:-

```
EXAMINE     APPLE    PRESENT 3
                     MESSAGE 0
                     DONE
```

In the vocabulary LOOK is defined as a synonym of EXAMINE and in the response table we already have an entry of:-

```
LOOK          _      DESC
```

Process & Response

It makes no difference to the compiler or the interpreter whether our new entry uses the words LOOK APPLE or EXAMINE APPLE. However it makes reading and understanding the response table easier if we always choose the same word. Therefore change the response table to insert the following entry before the existing LOOK _ entry:-

```
LOOK        APPLE   PRESENT 3               ;Apple here?
                    MESSAGE 0               ;Describe it
                    DONE
```

The end of process table 0 (ie the response table) should now look like this:-

```
RAMLOAD     _       RAMLOAD 255             ;Reload all flags
                    DESC

LOOK        APPLE   PRESENT 3               ;Apple here?
                    MESSAGE 0               ;Describe it
                    DONE

LOOK        _       DESC
; -         -       -       -       -       -       -       -
```

Our LOOK APPLE entry is a classic example of a response table entry because if the condition PRESENT 3 fails then PAW will continue to look for an entry to match the LS, in this case it will find the next entry displayed which is LOOK _, this entry will trigger and describe the current location (the DESC action). Assuming that the APPLE is indeed present then PAW will continue with the condacts and display our description of the apple (MESSAGE 0) the DONE action tells PAW to go and get another LS because we have done something - this prevents the LOOK _ entry triggering as well.

So at the moment if the player tries to EXAMINE anything except the apple (or tries to EXAMINE APPLE when it isn't present) they will be rewarded with a fresh description of their current location. This is why the LOOK APPLE entry MUST be specified before the LOOK _ entry. Let's insert the entries to deal with the sandwich, ticket and bench - they are:-

```
LOOK  SANDW   PRESENT   2       ;The sandwich is here
              MESSAGE   1       ;Describe it
              DONE

LOOK  TICKET  PRESENT   4       ;The ticket is here
              MESSAGE   2
              DONE

LOOK  BENCH   AT        4       ;The bench isn't an object
              MESSAGE   3       ;so check location
              DONE
```

and they must be inserted before the LOOK _ entry.

AT is a condition which is followed by a location number which will succeed (i.e. allow PAW to continue onto the next condact) if the player is at the same location, this was used because the bench was not an object, but as it is part of the description for location 4 it will always be there!

Compile the adventure and test it to check that you can examine these four items correctly and that the location is described at any other time.

The Process Tables

We shall now turn our attention to the other Process tables.

It was stated earlier that the response table was process table 0. There can be up to 254 process tables as we shall see.

There are three process tables (0, 1 & 2)in the source file to start with, just like response PAW scans through them, but, unlike response, it scans them not after obtaining a LS, but;

Process 1 is scanned immediately after PAW has described a location. This allows information to be printed only once when the player first arrives at a location or when he requests a redescribe.

Process 2 just before requesting a new LS from the parser. This is used to provide PAWs 'turn' at the game.

So far while playing our demo game we have had to end the game by typing QUIT. Now the original storyline (if you can remember that far back!) was to help the passenger find the ticket before the bus arrived. Now we obviously could have an entry in response which if the player said GET TICKET (and it was present) could trigger the end of the game e.g.

```
GET   TICKET PRESENT   4        ;The ticket is here
             TURNS
             END                ;That's all folks!
```

but wouldn't it be much better to finish the game when the player gets back to the bus stop?

We shall do so, but, first we need a message to describe the arrival of the bus, so add the following message to the message text section:-

```
/4
The bus arrives. I hand the ticket to the driver who
smiles and says "Sorry I'm late, hope you haven't been
standing too long?".
```

Note that message 4 must be entered after message 3. The conditions for the end of the game are that the player is at the bus stop (location 2) and is carrying the ticket (object 4). The first condition of course will be AT 2, the other can be checked with CARRIED 4 (pretty unusual names these conditions have...) so the final entry needed in Process 2 is:-

```
_    _    AT        2        ;at bus stop
          CARRIED   4        ;with ticket
          MESSAGE   4        ;finished
          TURNS
          END
```

this entry will now be scanned just before PAW gets a new LS and as soon as both conditions are met the game will end independent of the commands the player uses to get to the bus stop with the ticket!

Add this entry to Process 2 now ie at the end of TUTORIAL.SCE.

If you look at process table 1 you will see that the following entries are present:-

```
_    _    NEWLINE
          ZERO      0      ;If it is light...
          ABSENT    0      ;and the light source is absent
          LISTOBJ          ;List the objects

_    _    PRESENT   0      ;If the light source is present
          LISTOBJ          ;List the objects
```

When each location is described the first action NEWLINE will always be executed;

NEWLINE prints a carriage return/line feed. It's main purpose here is to ensure that any text displayed will be on a new line because PAW does not start one at the end of displaying a location description, the technical guide shows how to use this to good effect to modify the location description to reflect changes in the location.

From now on the two entries must be considered as a pair, their ultimate purpose is to list the objects at the current location, first a bit of background information;

PAW uses flag zero to determine if there is light for the player to see by (this feature is not used at the moment in our demo game), if there is no light the flag will have a value other than

zero and PAW will say "It's too dark to see anything." instead of the description for the location. In this case the objects that are present must not be listed.

Object 0 is assumed by PAW to be an object which provides light which is why object 0 in our demo is a lit torch. If this is present while the game is 'dark' (flag 0 is non zero) then it will override the darkness and so the objects must be described.

The two entries provide an example of using PAW to create an OR situation ie List the objects if it is light OR if object zero is present;

ZERO is the first condition we have met which tests the state of a flag. ZERO 0 will succeed if flag zero contains 0 which means there is light.

ABSENT ensures that Object 0 is not present (opposite of PRESENT condition - all conditions have an opposite, e.g. AT has an opposite of NOTAT and so on.), the next _ _ entry lists the objects if object 0 (the source of light) is present so we do not want this entry to succeed as well (i.e. This deals with the situation of it being light and object 0 being present which would otherwise list the objects twice!).

LISTOBJ lists any objects that are present at the players current location, if none are present it does nothing - it would look a bit silly saying "I can also see nothing."!

Think about the above as it represents a fairly useful feature of PAW which you may well need to adapt for use in your own games.

Right, now we shall reveal the better way of getting from the introduction screen to the start of the game at the bus stop:

```
_    _    AT        0          ;Start of game
          ANYKEY
          GOTO      2
          DESC
```

Insert this into Process 1 in front of the 2 existing entries. This uses two new condacts ANYKEY and GOTO which are both actions;

ANYKEY prints "Press any key to continue." and waits for you to press a key, it then allows PAW to continue onto the next condact.

GOTO is followed by a location number and moves the player to that location, it effectively sets flag 38 (players current location) to the value given, it does nothing else so it is followed by a DESCribe to get PAW to display the new description.

This entry thus causes the title screen to be displayed (when PAW displays the first location description), a wait for a key and then the game itself is started at the correct location.

You might like to go to the connections section and remove the entry for NORTH in location 0 as this is not needed now.

Compile and test the adventure to see the above two entries in action. The following input while at location 2 (the bus stop) will 'solve' the game in one go:

GO WEST, WEST AND UP. GET THE TICKET. GO DOWN,EAST AND EAST.

You should then get the finishing message and an option to play again. If not check the entries in Process tables 1 and 2 thoroughly.

Let's have a break and go back to deal with the ability of the bag to contain objects. Thought we had forgotten about that, didn't you? Well we nearly did. This will require some entries in the response table and we are going to allow the player to LOOK IN BAG, so we need a new message "In the bag is:", add this (it should be message 5) to the messages section. We are going to provide the player with the option of saying PUT ALL IN BAG as well as PUT object IN BAG. We can use exactly the same system as GET/DROP ALL discussed earlier. PUT is a synonym of DROP (which takes care of DROP TICKET IN BAG and such similar phrases), so the LS we must check for will be DROP _ (i.e. player is trying to put or drop something), now if the player includes IN BAG as part of the phrase we want PAW to put the object in the BAG. This means we must override the DROP _ entry already present, and if the extra words are included in the LS put the specified object in the bag.

Therefore the entry we need is:-

```
DROP    _       PREP    IN      ;Put something in bag
                NOUN2   BAG
                PRESENT 1       ;Bag here?
                AUTOP   1
                DONE
```

so insert this between the existing DROP ALL entry and the existing DROP _ entry. This shows how we check for an extended LS (i.e. ensuring certain parts of the phrase were what we need).

PREP is a condition which is followed by a preposition from the vocabulary. Prepositions are words used before a Noun to show its relation to another word in the phrase, in this case the condition will succeed if the player has used IN as part of the phrase.

NOUN2 is a condition which is followed by a Noun from the

vocabulary. This will succeed if the player has used BAG in the phrase. Combined with the previous entry it effectively stops PAW looking at the condacts unless the LS was PUT _ IN BAG where the underline is any object.

AUTOP is followed by a location number. Now we set aside location 1 for a special purpose early on in the tutorial, this is it, it is used as the inside of the bag! So AUTOP just like AUTOD scans the object definition section for a Noun which matches the current first Noun in the LS, when it has found one it places it at the location given, reporting "I have put the _ in the bag."!

The DROP ALL entry which exists will also work to deal with PUT ALL IN BAG, because it does not ensure that IN BAG is part of the LS and will trigger on both occasions, and in both cases 254 (or CARRIED) is the location the objects will be coming from.

Now for a GET object OUT OF BAG type command we need an entry similar to the above to override the GET _ entry which already exists so insert the following entry between the existing GET ALL entry and the existing GET _ entry:-

```
GET     _       PREP    OUT     ;Get something out of BAG
                NOUN2   BAG
                PRESENT 1       ;Bag here?
                AUTOT   1
                DONE
```

AUTOT is followed by a location number which shows where the object to TAKEOUT will come from.

The implementation of an ALL version of the command needs an entry of its own, at the moment GET ALL causes a DOALL HERE (or 255), which is the current position of the player, to be carried out. In order to get all from the bag we need to generate all the objects that are inside it (location 1), so insert the following entry prior to the existing GET ALL entry:-

```
GET     ALL     PREP    OUT     ;Get all out of bag
                NOUN2   BAG
                DOALL   1
```

Before you try the adventure again insert the following entry before the existing LOOK _ entry. This allows the player to LOOK IN THE BAG:-

```
LOOK    BAG     PREP    IN              ;Look in bag
                MESSAGE 5
                LISTAT  1
                DONE
```

LISTAT is followed by a location number and lists any objects

present at that location, note that if no objects are present it will print "nothing." so the above would result in:

```
In the bag is:
nothing.
```

which is correct, unlike LISTOBJ which because of its main use does not print anything at all in that situation.

So compile and test the adventure again to ensure that you can indeed PUT ALL IN BAG and GET object OUT OF BAG etc.

The Bird

The tutorial game is a little bit simple to solve so we shall add some complexity in the form of puzzles by creating two characters to wander round our little world. These are termed 'Pseudo-Intelligences' (PSIs for short) because they cannot obviously think, but must appear to do so to the player. A PSI consists mainly of a collection of messages, flags and process table entries, but even a few simple entries can create a surprisingly realistic effect. Creating a complex PSI of say a human can take a fair bit of thought, but follows the same principles as we will take with our two PSIs; a bird and a dog.

The bird will complicate the scenario as follows; The bird will have the ticket at the start of the game (normally you would assign an unused location to contain the birds objects, but we will use location 252 - object does not exist in game - as we have only one PSI that can have an object). This means you must persuade the bird to drop the ticket, trying to GET it will result in an "I can't do that" message and the bird flying away. The bird also flies between the Bandstand and the Tree Branch at regular intervals. The way to get the bird to drop the ticket will be to drop the sandwich at the same location. So lets get that little lot working first.

Firstly change the object definition of object 4 to:-

```
/4   _      1      _      _      TICKET   _
```

This makes the ticket a does not exist object which we are using to indicate it is in the birds beak. Insert the Nouns DOG and BIRD in the vocabulary with word values 21 and 22 respectively. Then insert the following messages which will be needed.

```
/6
The bird drops the ticket to peck at the sandwich.
/7
The bird snatches the ticket.
/8
The bird ignores me.
/9
A small bird is here.
/10
The bird has a ticket in its beak.
/11
A small bird settles on the ground.
/12
A small bird lands on the branch.
/13
The bird sees the dog and flutters away quickly.
/14
The bird flies away.
```

The Bird

We will insert the messages to deal with the dog later after an ear bending on yet another feature of PAW.

In a large game which contains several PSIs and a lot of background action, Process tables 1 and 2 soon become so full of entries it is nigh on impossible to work out what they do. Enter stage left the other process tables to the rescue, these can be 'called' from Process 1,2 or Response and used as an extension of the table they are called from. Calling a process causes PAW to save where it is at the moment and shift the action to the indicated table. Note that when something is DONE in the called process, then PAW will still shift back to the original table, so some very powerful things can be achieved with thoughtful use of these sub-process'. Users who program in other languages will recognize this as a 'subroutine'.

While PAW is in a sub-process it is quite possible for it to be asked to call yet another sub-process - a sub-sub-process? and so on down to a sub-sub-sub-sub-sub-sub-sub-sub-sub-sub-process! That is 10 levels of subroutine calls may be carried out, this is called 'nesting' a call, attempts to go further will result in an Interpreter run time error.

We are not going to use anything like that here, only a sedate sub-process. This will contain all the entries to deal with the birds activities.

We will use process table 3 for the bird.

Flag 11 will be a 'working' flag to contain a value for use in a comparison.

Flag 12 will contain the current location number of the bird.

Flag 5 is a special flag which if it has a value other than zero PAW will reduce by one whenever it scans process 2 - this is called an auto decrement flag. In this case it is used to count the number of 'time frames' that have passed in the game, a time frame is a single time round the big loop shown in diagram 3, and at the moment this is done before every phrase the player types. The bird will change location every three phrases on behalf of the player which will create the appearance of action in the game independent of the players input.

Now insert the following entries, in order after process 2, each entry is preceded by an explanation of its purpose and any new condacts it uses:

First determine if the bird is going to fly away this time through the table, this is indicated by flag five being zero (as it counts down from 3), if the ticket is at the same location as the bird it will be destroyed (i.e. put at location 252 so the bird 'has' it) and if the player is at the same location as the

bird they will be told that the bird has snatched the ticket. Note that the bird will continue its cycle of movement even if the player does not see it, a tree certainly does fall even if there is no one to see it in PAW!

```
/PRO  3 ;Bird
_   _   COPYOF    4    11   ;Copy loc'n of obj4(ticket) to flag11
        SAME     11    12   ;ticket at same loc'n as the bird.
        ZERO      5         ;Bird going to fly?
        DESTROY   4         ;Bird 'GETS' the ticket
        SAME     12    38   ;Bird at same location as player?
        MESSAGE   7         ;Tell player about it.
```

Note there is no DONE action as we want PAW to do each entry in turn, the above entry shows how conditions and actions can be mixed together to create new conditions.

The /PRO 3 line marks the beginning of process table 3.

COPYOF is an action followed by an object number and a flag, it copies the current location of the specified object to the specified flag. We use it in this situation to see if the ticket is at the same location as the bird by following it with;

SAME is a condition which compares the contents of the two flags and succeeds if they are the same.

DESTROY is an action which places the specified object at location 252, the not-created location.

Now deal with the two possible movements of the bird. If the bird is at the bandstand and flag five has reached zero then move the bird, set flag 5 to 3 again and tell the player the bird is gone if they were at the same location. Vice Versa if the bird is on the branch.

```
_   _   EQ        12    8   ;Bird on branch?
        ZERO       5        ;Time to fly?
        LET       12    5   ;Move bird to bandstand
        LET        5    3   ;Three phrases 'till move
        AT         8        ;Player here as well?
        MESSAGE   14        ;Tell them bird has flown

_   _   EQ        12    5   ;Bird on bandstand
        ZERO       5        ;Time to fly?
        LET       12    8   ;Move to branch
        LET        5    3   ;Three phrases 'till move
        AT         5        ;Player here as well?
        MESSAGE   14        ;Tell them...
```

EQ is a condition which is followed by a flag number and a value and will succeed if the flag contains the value, in this case it is checking if the bird is at a specific location.

LET is an action which is followed by a flag and a value. It sets the flag to the value.

Now we have dealt with the birds departure, next we must deal with its arrival, and if it arrives in a location which contains the player tell them about it.

```
_    _      EQ        5    3   ;Bird just flown?
            SAME     12   38   ;Now at players location?
            AT        5       ;On bandstand?
            MESSAGE  11       ;Landed on ground

_    _      EQ        5    3   ;Bird just flown?
            SAME     12   38   ;Now at players location?
            AT        8       ;On branch?
            MESSAGE  12       ;Landed on branch
```

Now if the bird has the ticket in its beak we must tell the player.

```
_    _      EQ        5    3
            SAME     12   38
            ISAT      4  252   ;Ticket not-created?
            MESSAGE  10       ;Has a ticket in beak.
```

ISAT is a condition followed by an object and a location number and succeeds if the object is at the specified location.

Finally if the sandwich is at the same location as the bird the bird will drop the ticket to peck at the sandwich. This entry does not rely on flag 5 so it will be checked for every time PAW checks process 2, so even if the player drops the sandwich <u>after</u> the bird has arrived the correct sequence will still be carried out.

```
_    _      COPYOF    2   11   ;Sandwich
            SAME     11   12   ;at same location as bird?
            ISAT      4  252   ;Ticket in beak?
            COPYFO   12    4   ;Put ticket down
            SAME     12   38   ;Player here as well?
            MESSAGE   6       ;Tell them...
```

COPYFO is an action which copies the contents of the specified flag to the current location of the specified object. There are also COPYFF and COPYOO actions which you can probably guess the purpose of.

That completes the control routine for the bird, but we need an entry in Process 2 to call this table every time frame, so insert the following entry at the beginning of process table 2:-

```
_    _      PROCESS 3         ;Bird
```

which will cause PAW to execute our bird control table every pass round its main loop.

We must ensure the bird starts at the correct location and that the player knows the bird is there when the location is described (or they will see messages about a bird arriving and flying off, with the description containing no mention of it). So amend the entry we made earlier to process 1 to contain a LET 12 8, which will ensure the bird is on the branch at the start of the game. The modified entry should read thus:

```
_    _    AT        0        ;Start of game
          ANYKEY
          LET      12     8  ;Bird is on branch (locno. 8)
          GOTO      2
          DESC
```

Insert the following entry at the end of process table 1 to tell the player the bird is present and if it has the ticket.

```
_    _    SAME     12    38  ;Bird at same location?
          MESSAGE   9        ;Tell player
          ISAT      4   252  ;Ticket in beak?
          MESSAGE  10        ;Tell player
```

Finally in the Response section insert the entry:

```
GET  TICKE SAME     12    38  ;Bird at same location?
           ISAT      4   252  ;with ticket in beak?
           CLEAR     5        ;Force it to fly away
           NOTDONE            ;"I can't do that"
```

This should go before the GET ALL entries and prevent the "There isn't one of those here" message being produced if the bird is present with the ticket.

CLEAR is an action which is followed by a flag number and sets the flag to have the value 0. This will cause the bird to fly away (which it might have been going to anyway) simulating its fright at having a great hand descend on it to get its prized new possession.

NOTDONE is an action similar to the DONE action but it fools PAW into thinking that nothing was done and thus causes it to print the "I can't do that" message.

Now the moment of truth, upon testing the game you should be able to watch the bird fly in and out of the bandstand and the branch, play with the game for a while to see the fact that the bird does indeed continue its roving existence. Then try dropping the sandwich at the same location. Note that if you do not pick up the ticket before the bird flies away it will snatch the ticket back.

The Dog

The dog will be added to complicate the game a bit more. The dog will simply follow the player everywhere (being a very obedient dog) and frighten the bird off. Now a dog would not be able to climb the tree so we must prevent the player from tempting the bird with the sandwich on the branch. To do so we will arrange for any object dropped while on the branch to fall to the ground. The player will be able to get rid of the dog by putting the lead on it and then tying the lead to the bench. In addition the player will be able to 'speak' to the dog which will provide another way of getting rid of the dog by asking it to SIT or STAY.

Before we examine the entries in Process and Response needed to control the dog insert the following words into the vocabulary section:-

TIE	34	verb
UNTIE	35	verb
SIT	36	verb
STAY	36	verb
COME	37	verb
HERE	37	noun

and the following messages into the messages section:-

```
/15
The _ falls to the ground at the foot of the tree.
/16
The dog's bright eyes stare at me with mindless love.
/17
A dog is here.
/18
The dog follows me wagging his tail.
/19
A lead trails behind the dog.
/20
The dog is tied to the bench by a lead.
/21
Trustingly the dog lets me put the lead around its neck.
/22
I've tied the lead to the bench.
/23
Who should I say it to?
/24
The dog is sitting quietly.
/25
I've untied the dog from the bench.
```

Ensure you include the underline in message 15 as it serves a

special purpose we will discuss later.

There is no real need to make the control routine for the dog a separate process table as it is only one entry, but we shall do so in case you wish to expand the game later.

Flag 13 will contain the current location of the dog.

Flag 14 will contain: 0 - the dog is free to roam, 1 - the dog has the lead around its neck, 2 - the dog is tied to the bench, 255 - the dog is sitting quietly.

The Process table needed for the dog is:-

```
/PRO  4      ;Dog
_     _      NOTSAME  13   38  ;Dog not where player is?
             LT       14    2  ;Still able to move?
             NOTAT     8       ;Player isn't up the tree?
             COPYFF   38   13  ;Move dog to players locno.
             MESSAGE  18       ;Tell them its followed...
```

You should be able to work out what **NOTSAME**, **NOTAT** and **COPYFF** do but the technical guide will help you out if you have problems.

LT is a condition which succeeds if the flag specified contains a value Less Than the specified value.

Insert at the beginning of process table 2;

```
_     _      PROCESS  4         ;Dog
```

Note that this comes before the entry for the bird to ensure the dog will be moved to the players new location before the bird is checked.

Similarly to the bird, entries are required in process table 1 to inform the player of the dogs presence and these should be before the entry for the bird:-

```
_     _      SAME     13  38  ;Dog at same location?
             MESSAGE  17      ;Tell player
             EQ       14   1  ;With lead?
             MESSAGE  19      ;Yes so tell player

_     _      SAME     13  38
             EQ       14   2  ;Dog tied to bench?
             MESSAGE  20

_     _      SAME     13  38
             GT       14   2  ;255 is greater than 2 so
             MESSAGE  24      ;tell player dog is sitting
```

while you are in process 1 modify the first entry to contain a

LET 13 2 (before the GOTO) to make the dog start at the bus stop.

Now in order for the bird to be frightened away by the dog we need an extra entry in process table 3. Now the entry must go before the entry which decides to drop the ticket and after the entries which make the bird fly. This will ensure that the bird will fly away with the ticket if it has it and leave it if it does not. So we need to insert the following as the seventh entry in Process 3:-

```
_     _     SAME      12   13  ;Bird and dog at same location
            LET       12    8  ;Only ever on bandstand so
            LET        5    3  ;move to branch, three phrases
            AT         5       ;Player on bandstand?
            MESSAGE   13       ;tell them bird is gone..
```

The last change to the process tables is to insert a sub-process which we will be calling from Response to deal with speech to the dog. The mechanism works very simply. If the player includes a phrase in double quotes ("") in the input sentence, then the parser will save where it was and carry on with decoding the phrase. There is an action called PARSE which instructs PAW to use the parser to decode the string the player typed in, this then becomes the LS. It is only sensible to do this in a sub-process as PAW will try to match the new LS against the rest of the table. Insert the following to make Process table 5:-

```
/PRO  5     ;Speak to dog
_     _     PARSE                ;Convert string to LS
            MESSAGE   16         ;Not valid phrase so
            DONE                 ;dog does not understand!

SIT   _     ZERO      14         ;Dog not partially tied up?
            SET       14         ;Now sitting quietly
            MESSAGE   24         ;Tell player (always at same
            DONE                 ;place as dog) Then DONE

COME  _     EQ        14  255    ;Dog must be sitting
            CLEAR     14         ;Now normal
            MESSAGE   18         ;Dog follows
            DONE

_     HERE  EQ        14  255    ;Dog sitting?
            CLEAR     14         ;Now normal
            MESSAGE   18         ;Dog follows
            DONE

_     _     MESSAGE   16         ;Anything else.
```

We get around the limited vocabulary that the dog understands by making him wag his tail for most things!

PARSE will allow PAW to continue looking at condacts if it fails

to find a valid phrase, be careful here as the current LS may be a bit jumbled up (i.e. the parser managed to get some sense out of the phrase) so you should normally only print a message like "They didn't seem to understand" or some such similar and DONE to return to your calling action. If it does form a valid LS PAW will start to search the following entries for a match as with Response. PARSE should only be used in a sub-process called from Response it has no meaning in any other table.

Notice how the COME and HERE entries deal with a variety of phrases that the player might try to call the dog again having made it sit.

The last entry catches all the valid LS which may have been in the string and the dog has no specific response to.

Now to the Response table to allow us to insert the extra entries to control speech and the dropping of objects in the tree.

First off the mark is the entry which causes all objects dropped in the tree to fall to the ground, now this must go between the entry which deals with putting objects in the bag and the normal DROP _ entry.

```
DROP  _     AT        8          ;Player on branch?
            WHATO                ;I say old boy!
            LT       51   255    ;Valid object?
            EQ       54   254    ;Object carried?
            MESSAGE  15         ;Its now bottom of tree.
            PUTO      7         ;Put it there
            DONE
```

This is an example of creating an automatic action of your own, like AUTOG and so on.

WHATO is an action which looks up the first Noun in the current LS in the object definition section converting it into an object number. This number is then placed in flag 51. Flag 51 always contains the number of the last object referenced by PAW and whenever it is set the associated flags 54 to 57 are also set. Flag 54 contains the current location of the object.

PUTO is an action which changes the location of the currently referenced object to be the one specified.

Message 15 contained an underline. Whenever PAW meets an underline in text (be it message or location) it replaces it with the current object hence the message is changed to suit the object currently being dealt with.

Next a relatively simple entry to deal with PUT LEAD ON DOG and this should go after the DROP ALL entry:-

```
DROP   LEAD   PREP     ON          ;Ensure not just a DROP LEAD
              NOUN2    DOG
              CARRIED   5          ;Player has the lead
              SAME     13    38    ;is at same location as dog
              LET      14     1    ;Dog now has lead on
              DESTROY   5          ;So player hasn't
              MESSAGE  21          ;Tell them so.
              DONE
```

The entries which follow deal with a new concept again, the modification of the current LS. We want the game to understand both TIE DOG TO BENCH and TIE LEAD TO BENCH as the the same thing, now LEAD and DOG are separate word values, so the TIE DOG entry converts the Noun into LEAD (55) and allows PAW to carry out the TIE LEAD entry! A similar system is used for UNTIE. Insert the following entries at the end of Process table 0:-

```
TIE    DOG    LET       34    55  ;Convert DOG to LEAD

TIE    LEAD   PREP      TO
              NOUN2     BENCH
              AT         4        ;Where bench is.
              SAME      13    38  ;dog is here
              EQ        14     1  ;with lead on
              PLUS      14     1  ;now tied to bench
              MESSAGE   22        ;tell player about it
              DONE

TIE    _      NOTDONE             ;Ensure an I can't

UNTIE  DOG    LET       34    55  ;Convert DOG to LEAD

UNTIE  LEAD   AT         4        ;Where bench is
              EQ        14     2  ;dog tied to it
              CLEAR     14        ;Now free
              MESSAGE   25        ;Tell player
              CREATE     5        ;Recreate lead
              GET        5        ;Try and get it.
              DONE

UNTIE  _      NOTDONE             ;Ensure an I can't
```

The NOTDONE makes sure PAW reports "I can't do that" if you try and TIE or UNTIE anything other than the lead/dog.

PLUS is an action which is followed by a flag number and a value. The flag is increased by the value. If the result exceeds 255 then the flag is set to 255.

CREATE is an action which is followed by an object number. It causes that object to be at the position where the player is.

GET is an action which is followed by an object number. It

attempts to get the specified object.

We use these actions instead of just placing the object at 254 so that any weight and/or number of objects carried problems are reported.

Finally the entries to allow speech to the dog, we have also included the entry necessary to allow you to speak to the bird - it just ignores you! These can also go at the end of Process table 0:-

```
SAY   DOG   SAME      13   38  ;It's here
            PROCESS    5       ;Someone else to do the work
            DONE

SAY   BIRD  SAME      12   38  ;Bird here
            MESSAGE    8
            DONE

SAY   _     MESSAGE   23       ;Who?
            DONE
```

Notice that we do not ensure the preposition TO is specified - this allows the player to shorten their input if required. As a general guide don't check for an extended LS unless it is required to differentiate two similar phrases.

Now compile and test the adventure.

As a final test the following inputs should now work in the indicated situations, they show some of the power which the parser can provide your games with.

When on the path by the park bench with the lead and dog try;
PUT LEAD ON DOG AND TIE IT TO THE BENCH

then to untie it;
UNTIE DOG

When up the tree with the bag try;
PUT ALL IN BAG AND DROP IT. GO DOWN AND LOOK IN BAG

To make the dog sit down;
SAY TO DOG "SIT"

and get back up;
ASK DOG TO "COME HERE"

Do it yourself

Here are a few points that you might like to tidy up in the demonstration game as practice on using the system.

1/ EXAMINE should respond to all objects even if it is with a general reply such as "I see nothing special about the _.". Hint: so as not to lose the use of LOOK on its own you could use a condition LT 34 255 before triggering (i.e. ensure a Noun was actually specified).

2/ The bird should really fly away if you GET SANDWICH while the bird is present. i.e. it will be pecking at the sandwich and any normal bird would fly...

3/ UNTIE _ and TIE _ should have a message something along the lines of "Tie what to what?", NOTDONE was an easy copout!

4/ How might you deal with the player typing PUT object IN BAG when the bag is not present? at the moment the game will drop the object instead, why?

5/ Nothing was ever done with the torch, the following entries will allow it to be turned on and off (you will also need TURN as a verb in the vocabulary):

```
TURN  TORCH PREP     ON
            CARRIED  7
            SWAP     7    0
            OK

TURN  TORCH PREP     OFF
            CARRIED  0
            SWAP     0    7
            OK
```

Lookup the extra condacts in the technical guide and read the chapter on light and dark - perhaps a cellar could be created below the bandstand? The movement would have to be checked in the Response table with an entry such as: (assuming 9 is the new location).

```
DOWN  _     AT       5        ;Player on bandstand?
            SET      0        ;Flag 0=255=Dark!
            GOTO     9        ;New location
            DESC
```

Not forgetting an entry for UP which clears the flag!

6/ What happens if the player types CLIMB TREE or CLIMB UP TREE and what is the best way to check for this? Hint: there is only one thing you can climb in that location.

End of the road

We hope that the above tutorial has provided an insight into some of the many powerful facilities of the Professional Adventure Writer. Now it is time for you to expand your knowledge of the system by using it! The Technical Guide will provide an exact specification of everything that PAW contains and in conjunction with the essays in it on various subjects, will form essential - if a little heavy - reading when writing your own games.

The file TIMING.SCE (when compiled) can be used to 'fine-tune' your installation of the Interpreter. It helps you to set the values of PAUSE and TIMEOUT we encountered at the beginning of this manual.

Finally you will find a small game in source form on the disc called "TEWK". Looking through this should provide you with some more ideas on giving your game an individual look.

What should I do next?
HAVE FUN!
OK

Tim Gilberts - January 1988.

Appendix A: EDIT a simple Text Editor

Before you can use the Text Editor it needs to be installed. The Edit program is called EDIT.COM, the install program is called EDITINST.COM and the installation information is stored on a file called EDIT.INS. To install the Text Editor type:-

```
EDITINST
```

the EDIT.INS file will be read and your screen should now look something like this:-

```
---------------------------
Columns              is set to 90
Lines                is set to 31
Clear screen         is set to '1B451B48' (hex)
Print at             is set to '1B59' (hex)
Row before column    is set to Y
Col offset           is set to 32
Row offset           is set to 32

Enter A to change Columns
      B to change Lines
      C to change Clear screen
      D to change Print at
      E to change Row before column
      F to change col offset
      G to change row offset
or    H to exit
Choose A-H?
```

The Editor needs to know the number of columns on the screen, the number of lines on the screen, how to clear the screen and how to position the cursor to a particular row and column on the screen. Note that the Clear screen and Print at codes have to be entered in hexadecimal. The values needed for Columns, Lines and Clear screen are the same as needed for the Interpreter so you shouldn't have too much trouble with those but remember that the clear screen code should clear the screen and place the cursor in the top left hand corner of the screen. The other variables may need a bit more explanation. To position the cursor at a particular place on the screen the screen controller needs to be sent a particular sequence of codes to indicate that the following values represent a row and a column - this is the 'Print at' code which can also be known as the move cursor code or the cursor position lead-in sequence. The screen controller may require the row number before the column number (ie Row before column set to 'Y') or vice versa. When the screen controller is informed of the row & column it may require an offset to be added to each. The values required for the offsets are those to position the cursor at the top left of the screen (common values are 0, 1 and 32).

Some example values are:-

Computer	BBC with Z80 (2nd processor)	PCW8512 CP/M 3	CPC464 CP/M 2	CPC6128 CP/M +
Screen mode	MODE 3	24x80 off	MODE 2	MODE 2
Columns	80	90	80	80
Lines	25	31	25	24
Clear screen	'0C'	'1B451B48'	'0C'	'1B451B48'
Print at	'1F'	'1B59'	'1F'	'1B59'
Row before column	N	Y	N	Y
Col offset	0	32	1	32
Row offset	0	32	1	32

When you are satisfied that the values are correct enter 'H' to exit and you will be asked "Do you want to update the file on disc?". Answer Y if you have made any changes.

To test that the Editor has been installed correctly type:-

EDIT EDIT.KEY

which will activate the Editor and load in the file EDIT.KEY. Your screen should now look something like as follows with the cursor at the top left of the screen:-

EDIT.KEY

The Inputs allowed are	ASCII		ie value	32-126
	RETURN	CTRL M	ie value	13
	Cursor up	CTRL E	ie value	5
	Cursor down	CTRL X	ie value	24
	Cursor left	CTRL S	ie value	19
	Cursor right	CTRL D	ie value	4
	Page up	CTRL R	ie value	18
	Page down	CTRL C	ie value	3
	Top of file	CTRL T	ie value	20
	End of file	CTRL B	ie value	2
	Goto line	CTRL G	ie value	7
	Delete	DEL	ie value	127
	Delete line	CTRL Y	ie value	25
	Exit	CTRL K	ie value	11

Free:nnnnn Line:1 Col:1

If not you will have to reinstall the editor.

To exit from the Editor enter CTRL K ie hold down the CTRL key and press K. You will be asked "Save file (Y/N)?" and you will probably want to answer N.

Appendix A

If you have installed it correctly then you will see that the bottom line of the screen is a status line which tells you how much free space there is and where the cursor is. The file you are editing shows you the inputs that the Editor will accept eg CTRL X (ie hold down the CTRL key and press X) to move the cursor down. Note that on some machines it is possible to program the specific cursor keys to move the cursor. eg under CPM 3/+ the SETKEYS command can be used thus:

```
SETKEYS KEYS.WP
```

which sets up the cursor keys to produce Wordstar control codes - which is what EDIT uses. SETKEYS KEYS.CCP will restore the keyboard after you exit.

In Use

The text editor is invoked using the command 'EDIT fname' where fname is the name of a file which may or may not already exist. If the file does exist it will be read into memory. While the file is being read in any TAB characters found will be expanded (ie converted to spaces), any lines which are too long to be displayed on the screen will be truncated and if the file itself is too big that to will be truncated.

The length of a line is limited to one less than the screen width ie 79 for an 80 column screen. This is because each line contains an end of line character at its end. If you try to enter a line longer than this you will get a "Too long" error message.

The Inputs allowed are	ASCII		ie value	32-126
	RETURN	CTRL M	ie value	13
	Cursor up	CTRL E	ie value	5
	Cursor down	CTRL X	ie value	24
	Cursor left	CTRL S	ie value	19
	Cursor right	CTRL D	ie value	4
	Page up	CTRL R	ie value	18
	Page down	CTRL C	ie value	3
	Top of file	CTRL T	ie value	20
	End of file	CTRL B	ie value	2
	Goto line	CTRL G	ie value	7
	Delete	DEL	ie value	127
	Delete line	CTRL Y	ie value	25
	Exit	CTRL K	ie value	11

eg to move the cursor to the end of the file hold down the CTRL key and press B. The numbers shown above eg 5 for cursor up, are to help you if you have cursor control keys on your keyboard and you want to configure the keyboard to produce the correct codes for the Editor eg the cursor up key should be configured to produce a 5.

When you enter CTRL K to exit you will be asked if you want to save the file you have edited. If you answer 'Y' and you were editing a file called TICKET.SCE this is what will happen:-

1. Any existing file called TICKET.$$$ is deleted
2. The edited file is written out to a file called TICKET.$$$
3. If the file TICKET.SCE already exists on the disc then any existing file called TICKET.BAK is deleted and TICKET.SCE is renamed to TICKET.BAK
4. The file TICKET.$$$ is renamed to TICKET.SCE
5. If at any point the disc directory or the disc itself is found to be full then the Editor will attempt to make some more room and will retry. If there is still insufficient room it will give you the option of changing discs. NB If you are using CP/M 2.2 you should only change discs when prompted to do so!

THIS PAGE INTENTIONALLY LEFT BLANK!

Published by Gilsoft International Ltd.,
2 Park Crescent, Barry, South Glamogan CF6 8HD
Telephone Barry (0446) 732765