ADMS

Adam Dawes

COLLABORATORS

	TITLE : ADMS		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Adam Dawes	March 15, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ADMS

1.1 ADMS -- Amiga Dungeon Mastering System

```
                                                Welcome to


                   .....:.....         ...':..      .''        ........
          :     .''    ::    `::.    .'     ::.   .:         :'      ``:
         :::    :     .::      ::.  :'      :::  .::        ::
       .' ::    :     :::       ::  :      .':::.'::        `::....
      .'  `::   `:    :::       ::: :.     :  ::' :::         ````::::.
 '''::::...::.    .   :::       ::'  '    .'   :  :::.     .         ``:
 ..:'   ```:::.       :::       :: .     .'       ::::     :           ::
:''         `::.      ::'      :'   :::::'         ::::.   ::.        .:
              `::. `..::.....''      ``'            ``:::.. ``:::...::'



                    Amiga Dungeon Mastering System v1.1
```

Contents:

1.2 What is ADMS?

ADMS (Amiga Dungeon Mastering System) is a complete package which will allow you to create and play adventure games with absolute ease.

The program has been designed to be very simple to use, but yet to still offer incredibly flexible features.

ADMS contains an entire language which is used to create commands to be used in your adventure games. There are currently over 60 commands recognised by ADMS, each of which in itself performs only a relatively simple command. By building these commands together, you can easily construct the commands that the person playing your game will use.

ADMS comes in two parts; the compiler and the interpreter. More information on each of these can be gained from the main menu.

If you have any questions or find any bugs (of which there are probably many at this stage!) then please contact me and tell me!

1.3 The Compiler

The ADMS compiler takes 7 source-code files that must have been set up by you, and turns them into a block of data that the interpreter can understand.

The compiler must be run each and every time a change is made in any of your game source code.

Once the game has been compiled, the interpreter may be run and your game tested.

The compiler may only be run from the command line. The syntax for its usage is:

```
  ADMScompile <indexfile>
```

More information about the index file can be found in the

ADMS files
section.

1.4 ADMS Files

The ADMS compiler needs 7 files in order to compile an adventure game,
echo of which contains various information about the finished game.

The files are as follows:

The Index File

The Global Message File

The Object File

The Room File

The Language File

The Travel File

The Synonym File

Only when all of these files have been created can the game be compiled.

Also see:

Special Characters

Escape Codes

1.5 Escape Codes

ADMS uses various 'escape codes' to make printing of some pieces of information easier.

An escape code consists of an 'at' character ('@') followed by two characters that define what information is to be printed. These two characters can be any of the following:

tt	=	Title of the game (as defined in the Index file)
rn	=	Release number of the game (from Index file)
sn	=	Serial number (from Index file)
cs	=	Current Score
ms	=	Maximum Score
tn	=	Number of turns taken
vb	=	Current verb

n1	=	First noun from syntax list
n2	=	Second noun from syntax list
a1	=	Indefinite article of first noun
a2	=	Indefinite article of second noun
dn	=	Direction from syntax list
w1	=	The first word found from ’word’ or ’word=’ syntax
w2	=	The second word from ’word’ or ’word=’ syntax

Any of these escape sequences may be used at any time during the game, although they may not always make much sense (for example, if you’re executing a command that doesn’t involve any objects, printing the first and second noun won’t have very productive results).

For example, the following ADMS command:

```
   Print "Welcome to @tt, release number @rn, serial number @sn.^"
```

Might produce:

```
   Welcome to Kroz III, release number 1, serial number 940409.
```

It’s also possible in your verb command code to have statements such as:

```
   Print "You can’t do that to @a1 @n1!^"
```

Which might produce:

```
   You can’t do that to an apple!
```

Please note that ADMS only stores the first 8 characters of each verb, so if in the code for a verb called ’inventory’ you were to put the command:

```
   Print "I am about to do an @vb.^"
```

The output would be:

```
   I am about to do an inventor.
```

This will be changed in a future release of ADMS.

1.6 Special Characters

There are a few characters that have special meaning to the ADMS ↩ compiler
and interpreter.

First of all is the semicolon (’;’). Anything in your source code following a semicolon will be completely ignored by the compiler. This is used to add comments to the programs and data files so that you can understand exactly what everything means.

The backslash (’\’) character is used to split lines that are longer than the screen over several lines so that it is easier to edit them. Whenever a backslash is encountered at the end of a line, it is deleted, and the first character on the following line placed in the position it occupied.

For example, the following lines:

```
   Print "Hello, \
          how are you?"
```

..would be read by the compiler as:

```
   Print "Hello, how are you?"
```

The carat character ('^') is used to tell the compiler that you want to put a carriage return in to your text. In nearly all situations, carriage returns are not added to text in ADMS to increase the flexibility to the game writer, so it's important you remember to do so!

The following line:

```
   Print "Hello!^How are you?^"
```

..would produce the following output:

```
   Hello!
   How are you?
```

The tilde character ('~') is replaced by the ADMS compiler with double quotation marks (you can't use double quotes themselves because they are used to mark the beginning and end of text strings).

For example:

```
   Print "The sign says: ~Beware, all ye who enter here.~"
```

Would produce:

```
   The sign says: "Beware, all ye who enter here."
```

The at character ('@') is used to mark the beginning of escape codes .

1.7 The Index File

The Index file is the file that holds all the other files together. It's also the file that is passed as a parameter to the ADMScompile command then your game is to be compiled.

The index file contains the filenames of the 6 other files to be compiled in to your game, and also several pieces of information which define some of the game's characteristics.

The following lines of information must be included in the Index file:

```
       GameName = "<name of your game>"
```

This defines the name that is given to the game. The name will be printed at the top of the screen whilst it’s being played in the interpreter, and can also be accessed via the escape codes.

```
       ReleaseNumber = <a number>
```

This specifies the release number of your game. It’s a good idea to give each game you write a unique release number, then you can keep track of exactly which version of a game you are playing. The release number can also be accessed via the escape codes.

```
       SerialNumber = <6 characters>
```

The serial number is also just for your reference. Traditionally the date of release is put as a serial number in the form YYMMDD, but any 6 numbers or letters can be entered. Again, this data can be accessed via the escape codes.

```
       MaxScore = <a number>
```

This line defines the maximum score the player should be able to achieve during the game. There is no checking that the score is able to reach this value, and it’s also quite possible for the score to exceed this value, so you must be quite careful when you set it. The MaxScore value can be accessed within the language file, and also through the escape codes.

```
       ObjectCapacity = <a number>
```

The object capacity defines how many objects the player should be able to carry in the game. Again, this is not enforced, but should be maintained by the game programmer when the language file is written.

```
       WeightCapacity = <a number>
```

This defines the weight of objects that the player should be able to carry.

```
       ObjFile = <file path/name>
```

Gives the full path and filename of the

object file
.

```
       RoomFile = <file path/name>
```

Gives the full path and filename of the
room file
.

```
       TravelFile = <file path/name>
```

Gives the full path and filename of the
travel file
.

```
       LanguageFile = <file path/name>
```

Gives the full path and filename of the
language file
.

```
       SynonymFile = <file path/name>
```

Gives the full path and filename of the
synonym file
.

```
       GlobalMsgFile = <file path/name>
```

Gives the full path and filename of the
global message file
.

```
       OutputFile = <file path/name>
```

This tells the compiler in which file it should store the complete compiled game (as will be used by the interpreter).

Note that if any of these declerations, the compiler will stop compiling almost immediately, telling you which of the lines of information are missing. After it’s happy that all the necessary data is present in the index file, it will begin processing the other files.

1.8 The Global Messages File

The Global Messages file contains text strings that are frequently used by the ADMS command language -- for example, many of the commands need to print strings such as, "But you’re not carrying it!", or, "You can’t go in that direction!" so those strings can all be stored as global messages.

The first four messages in this file are used by the system, and so must be included, but it’s possible to include as many strings of your own as you like, and then use the ADMS command ’PrintMsg’ to print them. However, each message must be numbered in the range of 1 to 255.

The first four messages contains equivalent strings to the following:

1. Welcome to the game.
2. I didn’t recognise one of the words you typed.
3. I understood all the words, but I didn’t understand the syntax.
4. You’re not carrying an object. (used by the ’checkcarried’ command)

To include messages in the file, put the message number on its own at the start of a blank line. On the next line, put the message itself. Finally, leave a blank line to terminate the message. For example:

```
1
Welcome to my game!

2
Sorry, I didn’t understand that!

3
I almost understood what you said, maybe you could rephrase it?
```

Messages can be included in any order, but each message number can only be used once. If any of the messages from 1 to 4 are found to be missing, the game will not compile.

1.9 The Objects File

The objects file is where you define all the objects that will be used in your game. An object might be something like a lantern or a sword that the player can carry around, or it might be an oak table that is fixed in place. It might also be a piece of invisible scenery.

Objects can also have more complex properties such as the ability to be opened or locked, to provide light for locations that are otherwise dark, or maybe to be containers or supporters that can hold other objects.

The first thing to do is to tell the compiler that you’re about to start talking about an object. To do this, you put the string ’object=’ at the start of a line, followed by the object’s name. For example:

```
object = lantern
```

This name that you have given is the name that will be used by the compiler to reference the object, not the player. For example, if you had 2 doors in your game you could give them game names of ’door1’ and ’door2’, yet the player names for both objects could be simply ’door’.

The player name for the object is defined on the next line (it’s best to leave a space or tab before putting the rest of the details of an object to make the text more readable). Often the player name will be the same as the game name, and in our example that is the case.. The next line would be:

```
        lantern
```

After defining the player’s name for the object, we tell the game where the object starts its life. Put the location name for any location that you have defined, and the object will start there. If you specify instead of a location name, the name of another object this object will be put inside or on top of the object you specify (note that you should only place things inside objects which are set up to be containers or supporters (see below) or you may find odd things happening in your game.) If you wish the player to be carrying the object at the start of the game, put ’Player’ as the start location. This is the case for our example object, so we add the following line:

```
        player
```

On the next 3 lines we define 3 descriptions of the object.. The first is the shortest description, and is how the object should be described if it is in your inventory or inside another object. The second description is what will be given if the object is sitting on the ground, not contained or supported by anything. The third is the full description of an object that should be given when the object is examined. For our example, they may be as follows:

```
        small lantern
        There is a small lantern on the ground here.
        The lantern is constructed from glass and copper.
```

Note that some objects will be defined as scenery (see below) and for those objects only the longest description will ever be seen (it should impossible to pick up a scenery object or place it inside something, and you should never see a scenery object on the ground, but it is still possible to examine them).

Next we list the object’s attributes . These are all put on the same line and seperated by spaces. All objects MUST have at least one attribute, and a good attribute to use if you can’t think of any others is the ’article’ attribute. This defines the indefinite article (either ’a’ or ’an’) which will be used with the object.

For our example object, we’ll set the attributes as follows:

```
        weight=50 article=a
```

This sets the weight to be 50 units, they can be any units you like. When you define a container you can set the weight limit that can be put inside it.

Finally, define the properties for the object. An object needn’t have any properties, but usually it will have at least one. Properties include features such as being a container, being openable, providing light etc.

We want our object to be switchable (so that we can turn the light on or off), and at the start we want it to be turned on and providing light. We set up the object’s properties as follows:

```
switchable on light
```

That’s the end of the definition for that object. You can now leave a blank line and start to define another object. See the Example game’s object file for more details.

1.10 Object Attributes

The following are valid attributes for objects:

ObjCapacity=<x>	this object can contain/support <x> object being put in/on it (default = 0)
WeightCapacity=<x>	this object can contain/support objects up to a total weight of <x> units (default = 0)
Weight=<x>	this object weighs <x> units (default = 0)
Adj=<x>	adjective for word (ex. adj=large) (default = "")
Article=<a/an>	set article for this word (default = ’a’)

1.11 Object Properties

The following are valid properties for objects:

Light	Sets a room/object has light. If there is no light source in a room at any time, the ’look’ function (as well as others, probably) can be made unable to function.
Container	This object can contain other objects.

Supporter	This object can support other objects.
Opaque	For a container, this means you cannot see inside it when it’s closed.
Openable	This object can have an ’open’ or ’closed’ state.
Open	This object is currently open.
Lockable	This object can be locked.
Locked	This object is currently locked.
Clothing	It’s possible to wear this object.
Worn	The object is currently being worn.
Switchable	This object can be switched on and off.
On	This object is currently switched on.
Static	This object can not be picked up or moved around.
Invisible	This object starts off invisible.
Enterable	It’s possible to get inside this object.
Scenery	Not given by inventory listing.
Edible	This object can be eaten.
Taken	This flag should be unset until the object is picked up, used (for example) for scoring.
Nonexistant	This object doesn’t currently exist.

1.12 The Room File

The Room file allows you to set up each of the locations to be used in your game. Each location has a game-name that is used to reference it from within your source code, a short description (that can be printed when a room is entered after the first time to quickly convey exactly where a player is) and a long description that contains much more verbose detail about the room. Finally, each location can be given some properties that alter how the room functions.

To define a room, put the line ’room=’ at the start of a line, followed by the name of the room that will be used internally by the compiler. As an example, we’ll take a location standing outside a house. We can start defining the location as follows:

```
room = outsidehouse
```

Next we need to give the short name. It’s best to use a space or tab before each of the following lines of information to improve readability. Our short description could be as follows:

```
Outside of House
```

Now the long description on the next line:

```
You’re standing outside a small white house. The door and windows \
have been boarded and the garden looks very overgrown. There is a \
path which winds to the north and south through some trees.
```

(Note the use of the backslash (’\’) character, for more information see

Special Characters
)

That’s the full description of the location. You may want to define some location
properties
next, though a location doesn’t need to have any. If you want any properties to be given to this room, list them on the next line.

The properties for our location are as follows:

```
light startloc
```

That’s the end of the definition for this location. Leave a blank line, and start to define another. See the Example game’s Room file for an example.

1.13 Room Properties

The following are valid properties for rooms:

Light	This room has light.
StartLoc	This is the room the player starts the game in. Note that this property must be given to one location and one location only. If this is not the case, the compiler will produce an error.

Entered	This flag should be unset until the room is entered, used (for example) for scoring.

1.14 The Language File

The language file is probably the most complicated file used by ↩ the ADMS compiler, so take some time to understand exactly how it all works.

The language file is used to define all of the verbs that will be used by the player in your finished game, for example: get, look, put, examine, open etc. It's written in a language called the ADMS command language.

To define a game verb, you first need to tell the compiler the name of the verb you're going to write. In this text, we'll write a simple command, 'examine'. So to start off:

```
 verb = examine
```

Now the compiler knows which verb we're working with. Sometimes the verb you specify won't be the name of a command to be entered by the player in the game, see special verbs.

It's possible in ADMS to define several completely different ADMS command scripts for one command, and the interpreter chooses which one to execute depending on what words follow the verb. You tell the compiler which words should follow the verb using the syntax = command.

In our example, we want the player to type the verb ('examine') followed by a noun (whichever object they wish to examine), so our syntax is 'verb noun'. Enter this after a 'syntax=' command on the next line:

```
    syntax = verb noun
```

When the interpreter receives the 'examine' command, it will only execute the following code if the words typed by the player consist of the verb 'examine' followed by a noun. You can set more than one syntax for each verb, as you will see shortly.

Next we start to write the program that will be executed when the syntax matches what we have requested. Explanations of all these commands can be found in the section on the ADMS command language.

First of all, let's check we're carrying the object:

```
       a = GetParent noun1
```

The variable 'a' now holds the object or room which is the given object's parent (ie, the object or room that contains the given object). Next we check that that object is the player:

```
       If a <> player
           EPrint "But you're not carrying it!^"
       EndIf
```

If the object is not the player, a message is printed telling the player that they're not carrying the object, and execution stops.

Assuming the program gets past this stage, we can give the player the full description of the object:

```
       PrintObjFull a
       EndParse
```

..The full description of the object is displayed, and execution of the program stops.

Now we'll define another syntax for the 'examine' command, that of when the verb is entered on its own with no object following it.

```
    syntax = verb
```

We want to print some sort of error message when this happens, as follows:

```
       eprint "What do you want to examine?"
```

Altogether, the full definition for the command is as follows:

```
 verb = examine
    syntax = verb noun
       a = GetParent noun1
       If a <> player
           EPrint "But you're not carrying it!^"
       EndIf
       PrintObjFull a
       EndParse
    syntax = verb
       eprint "What do you want to examine?"
```

Now you could continue to add more 'syntax=' keywords to this verb if you wished, or you could start to define another verb underneath. See the Example game's language file for a set of commands that could be used as the basis for a complete adventure game.

1.15 Syntax= keywords

The following keywords can be used after the 'syntax=' command:

any
The syntax will match regardless of what has or has not been typed after this point.

verb
The syntax will match if any verb has been entered at this word position.

verb = <verb>
The syntax will match if a specific verb has been entered at this word position.

noun
The syntax will match if any noun has been entered at this word position.

noun = <noun>
The syntax will match if a specific noun has been entered at this word position.

direction
The syntax will match if a compass direction (north, northeast, east, southeast, south, southwest, west, northwest, up or down) has been entered at this word position.

word
The syntax will match if any unrecognised word has been entered. The word can be displayed on the screen using the {"escape codes" link EscapeCodes}.

word = <word>
The syntax will match if the specified word has been entered at this word position. The specified word should not be a valid verb or noun.

For examples on using the different syntax keywords, see the Example game's language file.

Note: You don't need to worry about the words 'the', 'a' or 'an' being entered in to the syntax line because these are all stripped from the user's input before being passed to the syntax processor.

1.16 ADMS commands

ADMS commands:

Move

Give

GiveRoom

NearTo

IsHere

Carried

```
ObjRoom

CanGo

VerboseOn

BriefOn

SuperbriefOn

Has

HasRoom

Children

Weight

WCapacity

WUsed

OCapacity

OUsed

Confirm

ResetStream

GetStreamObj

GetParent

AddScore

SubScore

SetTask

ClearTask

ClearAllTasks

GetTask

Push

Pop

ClearStack

SetTimer

ClearTimer
```

```
GetTimer

ExtendTimer

EndParse

Return

Quit
        Restart

Save

Load

Verbose

Brief

Superbrief

Print

EPrint

RPrint

PrintMsg

PrintValue

CheckCarried

PrintShortDesc

PrintLongDesc

PrintArticle

PrintObjShort

PrintObjLong

PrintObjFull

GetCR

Gosub

SubMove

Random

If

EndIf
```

```
 Loop

 EndLoop

 ExitLoop

 DebugObj
 Miscellaneous:


 Variables
```

1.17 ADMS command: Move

```
Command:        Move

Usage:          Move <object> <object/room>

Description:    Moves the given object to another object or room

Example(s):
                move apple forest       ; puts the apple in the forest

                move lantern player     ; gives the lantern to the player

                move noun1 noun2        ; puts first object inside second


Note:           Be very careful when putting objects inside other objects!
                Imagine you have a box and a table.. Put the box on the
                table, and then put the table in the box. Now whenever
                the ADMS interpreter scans the parent tree to find the
                location of the box or table, it'll end up in an infinite
                loop as it loops through the two objects again and again.
                The temporary solution to this is to make sure that when
                an object is put inside another, the parent of both objects
                is either (a) the player or (b) the location. I'll be
                implementing a command 'inside' in the next version of
                ADMS to solve this problem.

                Also, make sure an object is not moved to itself, this one
                is much easier to stop.
```

1.18 ADMS command: Give

```
                 Command:       Give

Usage:          Give <object> <property list>

Description:    Adds or removes object
                properties
                 from the specified
```

object. To add a property, simply list its name after the object. To remove the property, put its name with a minus sign ('-') before it.

Examples:

```
Give lantern on light   ; the lantern is now on and light

Give lantern -on -light ; when it's turned off again

Give noun1 open taken -edible   ; change several properties
```

See also:

Has

GiveRoom

1.19 ADMS command: GiveRoom

Command: GiveRoom

Usage: `GiveRoom <room> <property list>`

Description: Adds or removes room properties from the specified location. To add a property, simply list its name after the room name. To remove the property, put its name with a minus sign ('-') before it.

Examples:

```
Give cave1 light        ; something here is glowing..?

Give location -light    ; make player's current location
                        ; dark
```

See also:

HasRoom

Give

1.20 ADMS command: NearTo

Command: NearTo

Usage: `<var> = NearTo <object>`

Description: Returns 'TRUE' if the specified object is in the same room as the player or it is being carried by the player, or otherwise 'FALSE'.

Example:

```
a = NearTo noun1                   ; is the object here?
```

```
if a = false
    eprint "I can’t see the @n1!^" ; nope..
endif
printobjfull a                     ; otherwise describe it
endparse                           ; and stop.
```

See also:

IsHere

1.21 ADMS command: IsHere

Command: IsHere

Usage: `<var> = IsHere <object>`

Description: Returns ’TRUE’ if the specified object is in the same room as the player, but if it’s in another room or is being carried by the player, returns ’FALSE’.

Example:

```
a = IsHere noun1
If a = false
    EPrint "It’s not on the ground!^"
EndIf
```

See also:

NearTo

1.22 ADMS command: Carried

Command: Carried

Usage: `<var> = Carried <object>`

Description: Returns ’TRUE’ is the object is found anywhere in the player’s inventory tree. If you wish to find if an object is in the player’s first level of inventory, the

GetParent

command may be used instead.

Example:

```
a = Carried snake
If a = true
    EPrint "The budgie flys out of your reach.^"
EndIf
Move budgie player
```

See also:

GetParent

1.23 ADMS command: ObjRoom

```
Command:       ObjRoom

Usage:         <var> = ObjRoom <object> <room>

Description:   Returns 'TRUE' if the object is in the specified room,
               or 'FALSE' if it is anywhere else.

Example:
               a = ObjRoom crucifix altar
               if a = false
                   EPrint "Nothing happens..^"
               endif
               Print "There is a huge burst of multicoloured sparks!^"
               SetTask 1
               EndParse
```

1.24 ADMS command: CanGo

```
                Command:       CanGo

Usage:         <var> = CanGo <object> <direction>

Description:   Tests to see if the given object can go in the specified
               direction (note that the 'direction' variable may be
               used here instead of an explicit compass direction as long
               as 'direction' was included in the syntax= string).
               This is achieved by examining the code for the appropriate
               direction in the
               travel table
               . If there is no entry in
               the table, "You can't go that way!^" is printed, and
               'noroom' returned in the variable. Otherwise, the travel
               table code is executed. Assuming a room name is found in
               the travel table code, that room number will be returned.
               Otherwise, 'noroom' is returned.

Example:
               a = CanGo player direction
               If a = noroom
                   EndParse            ; can't go that way
               EndIf
               Move player a           ; move player in that direction
               PrintShortDesc          ; show new location information
               PrintLongDesc

See also:
                Move
                Note:          Here is an example piece of code from the travel ↩
                    table:

            room = forest1
               dir = north
```

```
    a = Carried apple
    If a = false
        EPrint "You need an apple to go north from here.^"
    EndIf
    Forest2
```

Now if the CanGo command is executed with the player as the object to test movement for ('CanGo player direction'), if they were not carrying the apple, the 'You need an apple..' text would be printed. If any other object is tested, the text will not be printed. This is so that you can move objects other than the player around without worrying about spurious messages appearing if the object cannot move in a certain direction.

Note: Because this command actually executes code from the travel table, it can sometimes be quite a time consuming command. Try to only use it when it's necessary, and not repeat it when you could just store the result of the first execution in another variable, etc.

Note: The CanGo command can not be used within the travel table code itself.

1.25 ADMS command: VerboseOn

Command: VerboseOn

Usage: `<var> = VerboseOn`

Description: Returns 'TRUE' if the current room-display mode is Verbose, or 'FALSE' if it anything else.

Example:

```
a = VerboseOn
If a = true
    EPrint "Verbose mode is on already!^"
EndIf
Verbose
EPrint "Verbose mode now active.^"
```

See also:

Verbose

BriefOn

SuperbriefOn

1.26 ADMS command: BriefOn

```
                 Command:       BriefOn

Usage:          <var> = BriefOn

Description:    Returns 'TRUE' if the current room-display mode is Brief,
                or 'FALSE' if it anything else.

Example:
                a = BriefOn
                If a = true
                    EPrint "Brief mode is on already!^"
                EndIf
                Brief
                EPrint "Brief mode now active.^"

See also:
                 Brief

                 VerboseOn

                 SuperbriefOn
```

1.27 ADMS command: SuperbriefOn

```
                 Command:       SuperbriefOn

Usage:          <var> = SuperbriefOn

Description:    Returns 'TRUE' if the current room-display mode is
                SuperBrief, or 'FALSE' if it anything else.

Example:
                a = SuperBriefOn
                If a = true
                    EPrint "Superbrief mode is on already!^"
                EndIf
                Superrief
                EPrint "Superbrief mode now active.^"

See also:
                 Superbrief

                 VerboseOn

                 BriefOn
```

1.28 ADMS command: Has

```
                 Command:       Has

Usage:          <var> = Has <object> <property list>
```

```
Description:   Tests to see if the given object has the specified
               properties. If it does, 'TRUE' is returned, otherwise
               'FALSE'. Note that you can check if properties are not
               set by preceeding the property name with a minus sign ('-').
Examples:
               a = Has noun2 supporter         ; can put things on here?
               If a = false
                   EPrint "You can't put things on the @n2.^"
               EndIf
               Move noun1 noun2
               EPrint "The @n1 is now on the @n2.^"

               a = Has noun1 edible
               If a = false
                   EPrint "You can't eat that..!^"
               EndIf

               a = Has noun1 openable open
               If a = false
                   EPrint "The @n1 is already open!^"
               EndIf

               a = Has noun1 container openable -open opaque
               If a = true
                   EPrint "I can't inside it!^"
               EndIf

See also:
                Give

                HasRoom
```

1.29 ADMS command: HasRoom

```
                Command:       HasRoom

Usage:         <var> = Has <object> <property list>

Description:   Tests to see if the given room has the specified
               properties. If it does, 'TRUE' is returned, otherwise
               'FALSE'. Note that you can check to see if a room's
               properties are not set by preceeding the property name with
               a minus sign ('-').

Example:
               a = HasRoom location light      ; can I see anything?
               If a = false
                   EPrint "It's dark here!.^"
               EndIf
               PrintShortDesc
               EndParse

               a = HasRoom location light -entered
               If a = true
```

```
                    Give location entered
                EndIf
```

See also:

GiveRoom

Has

1.30 ADMS command: Children

```
Command:        Children

Usage:          <var> = Children <object/location>

Description:    Returns the number of children in the given object or
                location. Only the first level of children are scanned.

Example:
                a = Children location
                If a = 1                ; only the player is here
                    Print "There are no objects here."
                EndIf
```

1.31 ADMS command: Weight

```
                 Command:       Weight

Usage:          <var> = Weight <object>

Description:    Returns the weight of the given object.

Example:
                a = Weight noun1
                If a > 100
                    EPrint "The object is much too heavy for you to lift.^"
                EndIf
```

See also:

WCapacity

WUsed

1.32 ADMS command: WCapacity

```
                 Command:       WCapacity

Usage:          <var> = WCapacity <object>

Description:    Returns the weight capacity of the given object (which
                should be a container or supporter object).
```

```
Example:
              a = WCapacity noun2
              b = WUsed noun2
              c = Weight noun1
              b = b + c
              If b > a
                  EPrint "There’s no space left in the @n2.^"
              EndIf

See also:
               Weight

               WUsed
```

1.33 ADMS command: WUsed

```
               Command:       WUsed

Usage:        <var> = WUsed <object>

Description:  Returns the weight which is currently used in a container
              or supporter object. Note that only its direct children
              are scanned.

Example:
              a = WCapacity noun2
              b = WUsed noun2
              c = Weight noun1
              b = b + c
              If b > a
                  EPrint "There’s no space left in the @n2.^"
              EndIf

See also:
               Weight

               WCapacity
```

1.34 ADMS command: OCapacity

```
               Command:       OCapacity

Usage:        <var> = OCapacity <object>

Description:  Returns the maximum number of objects than can be stored
              in the given (container or supporter) object.

Example:
              a = OCapacity player
              b = OUsed player
              If a >= b
```

```
                    EPrint "You can’t carry any more.^"
                EndIf

 See also:
                 OUsed
```

1.35 ADMS command: OUsed

```
                 Command:       OUsed

 Usage:         <var> = OUsed <object>

 Description:   Returns the number of objects than are currently stored in
                the given (container or supporter) object.

 Example:
                a = OCapacity player
                b = OUsed player
                If a >= b
                    EPrint "You can’t carry any more.^"
                EndIf

 See also:
                 OCapacity
```

1.36 ADMS command: Confirm

```
 Command:       Confirm

 Usage:         <var> = Confirm <text>

 Description:   Prints the specified text on the screen, and then waits
                for the player to press the ’y’ or ’n’ key. If the player
                selected ’y’, ’TRUE’ is returned in the variable, otherwise
                ’FALSE.

 Example:
                a = Confirm "Are you sure you want to quit? "
                If a = false
                    EPrint "No.^"
                EndIf
                Print "Yes.^^Your score is @cs out of @ms in @tn turns.^^"
                GetCR
                Quit

 Note:          The Confirm command may not be used within the travel
                table.
```

1.37 ADMS command: ResetStream

```
                Command:        ResetStream

Usage:          ResetStream <stream number> <object/room>

Descripton:     Resets an object stream to the first child object of the
                given object/location. The objects can then be read in
                sequence with the GetStreamObj command. These two commands
                are used together in order to scan object trees.

Example:
                ResetStream 0 location          ; reset strm 0 to curr.loc
                Loop
                    a = GetStreamObj 0          ; get obj from stream
                    If a = noobject
                        ExitLoop                ; reached the last one
                    EndIf
                    If a <> player              ; check it's not the player
                        PrintObjShort a         ; display object's name
                        Print "^"
                    EndIf
                EndLoop

See also:
                 GetStreamObj
                 Note:          The object stream number must be in the range of 0 ↩
                     - 255.
```

1.38 ADMS command: GetStreamObj

```
                Command:        GetStreamObj

Usage:          <var> = GetStreamObj <stream number>

Description:    Returns the next object in the list of the specified
                stream number, or 'noobject' if the end of the list is
                reached. This command must only be used after the object
                stream has been initialised with the ResetStream command.

Example:
                ResetStream 0 location          ; reset strm 0 to curr.loc
                Loop
                    a = GetStreamObj 0          ; get obj from stream
                    If a = noobject
                        ExitLoop                ; reached the last one
                    EndIf
                    If a <> player              ; check it's not the player
                        PrintObjShort a         ; display object's name
                        Print "^"
                    EndIf
                EndLoop

See also:
                 ResetStream
```

```
Note:          The object stream number must be in the range of 0 - 255.
```

1.39 ADMS command: GetParent

```
Command:       GetParent

Usage:         <var> = GetParent <object>

Description:   Returns the parent object/location of the given object.

Example:
               a = Parent microchip
               If a <> chip_socket
                   EPrint "Nothing happens.^"
               EndIf
               Print "The screen suddenly bursts into life.^"
```

1.40 ADMS command: AddScore

```
Command:       AddScore

Usage:         AddScore <amount>

Description:   Adds the specified amount to your current score.

Example:
               AddScore 20
               If currentscore = maxscore
                   EPrint "Congratulations, you have finished the game!^"
               EndIf

See also:
               SubScore
```

1.41 ADMS command: SubScore

```
Command:       SubScore

Usage:         SubScore <amount>

Description:   Subtracts the specified amount from your current score.

Example:
               SubScore 20

See also:
               AddScore
```

1.42 ADMS command: SetTask

Command: SetTask

Usage: SetTask <task number>

Description: Marks the specified task as having been completed.

Example:

```
a = GetTask 0
If a = false
    SetTask 0
    AddScore 15
    EPrint "You suddenly feel much more powerful!^"
EndIf
```

See also:

ClearTask

GetTask

Note: The task number must be in the range 0 - 63.

1.43 ADMS command: ClearTask

Command: ClearTask

Usage: ClearTask <task number>

Description: Marks the specified task as incomplete.

Example:

```
If noun1 = magic_orb
    a = GetTask 1
    If a = true
        ClearTask 1
        SubScore 25
        EPrint "You feel strangely sad after your action.^"
    EndIf
EndIf
```

See also:

ClearTask

GetTask

Note: The task number must be in the range 0 - 63.

1.44 ADMS command: ClearAllTasks

Command: ClearAllTasks

Usage: ClearAllTasks

```
Description:   Marks all the tasks as incomplete.

Example:
               If noun1 = sacred_crown
                   ClearAllTasks
                   EPrint "Suddenly you hear a crashing sound filling \
                           the air all around you! It seems all your \
                           hard work has been undone!^"
               EndIf

See also:
                ClearTask

                SetTask
                Note:          Be very careful with this command or you might  ←
                    find
               yourself clearing tasks which are at this point impossible
               to complete again.
```

1.45 ADMS command: GetTask

```
                Command:       GetTask

Usage:         <var> = GetTask <task number>

Description:   Tests if the specified task has been completed. If it has,
               'TRUE' is returned, otherwise 'FALSE'.

Example:
               Print "Tasks completed:^"
               a = 0                   ; current task to check
               b = 0                   ; task complete count
               Loop
                   c = GetTask a       ; test this task
                   If c = true
                       If a = 0
                           Print "  Magic orbs^"
                       EndIf
                       If a = 1
                           Print "  Crown jewels^"
                       EndIf
                       If a = 2
                           Print "  Lost treasure^"
                       EndIf
                       b = b + 1       ; increase completed ctr
                   EndIf
                   a = a + 1           ; move to next task
                   If a = 3
                       ExitLoop
                   EndIf
               EndLoop
               If b = 0                ; no tasks are complete!
                   EPrint "  None.^"
               EndIf
               EndParse
```

```
See also:
                SetTask

                ClearTask
                Note:          The task number must be in the range 0 - 63.
```

1.46 ADMS command: Push

```
                Command:       Push

Usage:         Push <object/room/number/verb/variable>

Description:   Pushes the given piece of information on to the top of
               the user stack. The Pop command can be used to retrieve
               it at a later time.

Example:
               Push a                  ; push contents of a onto stack
               Gosub .someroutine
               a = Pop                 ; get contents back from stack

See also:
                Pop

                ClearStack
                Note:          The user stack is maintained by the ADMS  ↩
                   interpreter so
               that if a verb finished execution with data still on the
               stack, it will be erased and the stack reset.
```

1.47 ADMS command: Pop

```
                Command:       Pop

Usage:         <var> = Pop

Description:   Retrieves the piece of information currently on the top of
               the user stack and stores it in the given variable.

Example:
               Push a                  ; push contents of a onto stack
               Gosub .someroutine
               a = Pop                 ; get contents back from stack

See also:
                Push

                ClearStack
                Note:          Be very careful not to use the Pop command if the  ↩
                   stack is
               currently empty -- you may experience odd results or
```

system crashes if you do!

1.48 ADMS command: ClearStack

Command: ClearStack

Usage: ClearStack

Description: This command clears all data that is currently on the stack, and returns it to a completely empty state.

See also:

Push

Pop

Note: Be very careful not to use the Pop command is the stack is empty!

1.49 ADMS command: SetTimer

Command: SetTimer

Usage: SetTimer <timer#> <verb> <delay>

Description: This command allows you to set a timed future event. After the number of turns specified in <delay> have elapsed, the given verb will be executed. This is useful for imposing time limits on games aswell as a whole host of other things. The timer number should be a unique number for each timed task, though it’s not necessary for it to be; if you use move than one timer with the same number simultaneously, however, you will be unable to cancel or get information on them.

Example:

```
If noun1 = match
    SetTimer 0 .endmatch 5    ; match burns out in 5 turns
    EPrint "You match bursts into flames."
EndIf
```

See also:

ClearTimer

GetTimer

ExtendTimer

Note: The timer number must be in the range 0 - 255.

1.50 ADMS command: ClearTimer

```
Command:       ClearTimer
```

```
Usage:          ClearTimer <timer#>

Description:    The ClearTimer command cancels a timed event that has
                previously been initialised with the SetTimer command.

Example:
                If noun1 = match
                    a = GetTask 0               ; is the match alight?
                    If a = -1                   ; no
                        EPrint "The match isn’t alight!^"
                    Endif
                    ClearTimer 0                ; stop it burning out
                    EPrint "The match is now extinguished.^"
                EndIf

See also:
```

```
SetTimer

GetTimer

ExtendTimer
Note:          The timer number must be in the range 0 - 255.
```

1.51 ADMS command: GetTimer

```
Command:       GetTimer
```

```
Usage:          <var> = GetTimer <timer#>

Description:    Returns the number of turns that still have to elapse
                before the specified timer triggers. If the timer is not
                active, the value -1 is returned.

Example:
                If noun1 = match
                    a = GetTask 0               ; is the match alight?
                    If a = -1                   ; no
                        EPrint "The match isn’t alight!^"
                    Endif
                    ClearTimer 0                ; stop it burning out
                    EPrint "The match is now extinguished.^"
                EndIf

See also:
```

```
SetTimer

ClearTimer

ExtendTimer
Note:          The timer number must be in the range 0 - 255.
```

1.52 ADMS command: ExtendTimer

```
                 Command:       ExtendTimer

 Usage:         ExtendTimer <timer#> <no. of turns>

 Description:   This command delays the triggering of the specified timer
                by the given number of turns. If the specified timer is
                not currently active, nothing happens.

 Example:
                If noun1 = petrol
                    If noun2 = car
                        extendtimer 0 200     ; car runs another 200 turns
                        eprint "You pour the petrol in to the car.^"
                    EndIf
                EndIf

 See also:
                 SetTimer

                 ClearTimer

                 GetTimer
                 Note:          The timer number must be in the range 0 - 255.
```

1.53 ADMS command: EndParse

```
                 Command:       EndParse

 Usage:         EndParse

 Description:   Stops the ADMS command processor completely so that the
                player can be prompted for his next move.

 Example:
                PrintShortDesc          ; print room description
                EndParse                ; and stop.

 See also:
                 EPrint
                 Note:          Every single syntax of every verb must be  ↩
                    terminated with
                an EndParse (or EPrint) command or you may experience
                strange results.

 Note:          The EndParse can not be used within the travel table, use
                the Return command instead.
```

1.54 ADMS command: Return

```
 Command:       Return

Usage:          Return

Description:    Stops the current ADMS command execution and returns to the
                command that called it. This can only happen if a verb
                is called with the Gosub command.

Example:
                PrintShortDesc          ; print room description
                Return                  ; and return to previous verb

See also:
                 Gosub

                 RPrint
                 Relevant topics:
                 Special Verbs

Note:           When you write a subroutine, you must end it with a Return
                (or RPrint) command. It is also acceptable (though not
                very good programming) to terminate it with EndParse.

Note:           When writing the travel table, if the direction chosen
                can not currently be accessed, use the Return command to
                pass processing back to the verb without allowing travel
                in that direction.
```

1.55 ADMS command: Quit

```
Command:        Quit

Usage:          Quit

Description:    Quits the game and returns to CLI/Workbench

Example:
                a = Confirm "Are you sure you want to quit? "
                If a = false
                    EPrint "No.^"
                EndIf
                Print "Yes.^^Your score is @cs out of @ms in @tn turns.^^"
                GetCR
                Quit

Note:           The Quit command does not ask for any confirmation, so it’s
                best to do it yourself, as shown in the above example.
```

1.56 ADMS command: Restart

```
Command:        Restart
```

Usage: Restart

Description: Restarts the game, as if it had only just been loaded.

Example:

```
a = Confirm "Are you sure you want to restart? "
If a = false
    EPrint "No.^"
EndIf
Restart
```

Note: The Restart command does not ask for any confirmation, so it’s best to do it yourself, as shown in the above example.

1.57 ADMS command: Save

Command: Save

Usage: Save

Description: Prompt the user for a filename, and then saves all changeable details to that file.

Example:

```
Save
Endparse
```

1.58 ADMS command: Load

Command: Load

Usage: Load

Description: Prompt the user for a filename, and then loads all changeable details from that file.

Example:

```
Load
Endparse
```

1.59 ADMS command: Verbose

Command: Verbose

Usage: Verbose

Description: Switches the room description mode into Verbose mode.

Example:

```
Verbose
EPrint "Verbose mode active."
```

See also:

Brief

Superbrief

VerboseOn

1.60 ADMS command: Brief

Command: Brief

Usage: Brief

Description: Switches the room description mode into Brief mode.

Example:

```
Brief
EPrint "Brief mode active."
```

See also:

Verbose

Superbrief

BriefOn

1.61 ADMS command: Superbrief

Command: Superbrief

Usage: Superbrief

Description: Switches the room description mode into Superbrief mode.

Example:

```
Superbrief
EPrint "Superbrief mode active."
```

See also:

Verbose

Brief

SuperbriefOn

1.62 ADMS command: Print

```
                 Command:       Print

Usage:          Print <text>

Description:    Prints given text to the screen.

Example:
                Print "Welcome to my game!"

See also:
                 EPrint

                 RPrint

                 PrintMsg

                 PrintValue
```

1.63 ADMS command: EPrint

```
                 Command:       EPrint

Usage:          EPrint <text>

Description:    Prints given text to the screen and then performs an
                EndParse command.

Example:
                EPrint "A voice say, ~Thankyou!~"

See also:
                 Print

                 EndParse
```

1.64 ADMS command: RPrint

```
                 Command:       RPrint

Usage:          RPrint <text>

Description:    Prints given text to the screen and then performs a
                Return command.

Example:
                RPrint "The coin lands with a ’splash!’"

See also:
                 Print

                 Return
```

1.65 ADMS command: PrintMsg

Command: PrintMsg

Usage: PrintMsg <message#>

Description: Prints a global message to the screen.

Example:

```
PrintMsg 10
```

Note: The message number must be in the range 0 - 255.

Note: Make sure the global message actually exists!

1.66 ADMS command: CheckCarried

Command: CheckCarried

Usage: CheckCarried <object>

Description: Performs exactly the same task as the Carried command, except that if the player is found not to be carrying the object, global message 4 is printed to the screen and an EndParse performed.

Example:

```
CheckCarried noun1
Move noun1 location
EPrint "You drop the @n1.^"
```

Note: This command can not be used in the travel table.

1.67 ADMS command: PrintShortDesc

Command: PrintShortDesc

Usage: PrintShortDesc

Description: Prints the short description of the player’s current location, as defined in the Room file.

See also:

PrintLongDesc

1.68 ADMS command: PrintLongDesc

Command: PrintLongDesc

Usage: PrintLongDesc

Description: Prints the long description of the player’s current location, as defined in the Room file.

See also:

PrintShortDesc

1.69 ADMS command: PrintArticle

Command: PrintArticle

Usage: PrintArticle <object>

Description: Prints the indefinite article (’a’ or ’an’) for the specified object, followed by a space.

Example:

```
PrintArticle noun1
PrintObjShort noun1
EndParse
```

See also:

PrintObjShort

PrintObjLong

PrintObjFull

1.70 ADMS command: PrintObjShort

Command: PrintObjShort

Usage: PrintObjShort <object>

Description: Prints the short description for the specified object, as defined in the Object file.

Example:

```
PrintObjShort noun1
```

See also:

PrintArticle

PrintObjLong

PrintObjFull

1.71 ADMS command: PrintObjLong

```
                 Command:       PrintObjLong

Usage:          PrintObjLong <object>

Description:    Prints the long description for the specified object, as
                defined in the Object file.

Example:
                PrintObjLong noun1

See also:
                 PrintArticle

                 PrintObjShort

                 PrintObjFull
```

1.72 ADMS command: PrintObjFull

```
                 Command:       PrintObjFull

Usage:          PrintObjFull <object>

Description:    Prints the full description for the specified object, as
                defined in the Object file.

Example:
                PrintObjFull noun1

See also:
                 PrintArticle

                 PrintObjShort

                 PrintObjLong
```

1.73 ADMS command: GetCR

```
Command:        GetCR

Usage:          GetCR

Description:    Displays the message "Press <RETURN> to continue" on the
                screen, and waits for the player to press the RETURN key.
```

1.74 ADMS command: Gosub

```
                Command:       Gosub

Usage:         Gosub <verb>

Description:   Stores the current position of command execution and passes
               command to the verb specified after the Gosub command.
               When a Return command is executed in that verb, command
               execution will pass back to the command immediately
               following the Gosub command.

Example:
               Print "You are currently carrying:^"
               Gosub .listinv
               EndParse

See also:
                Return
                Relevant topics:
                Special Verbs

Note:          You should only use the Gosub command with a Special Verb
               as it’s destination.

Note:          When a verb is executed via the Gosub command, the first
               syntax available for the verb is the one that will be
               executed, regardless of what the actual syntax is.
               Providing special verbs are being used as the targets for
               Gosub commands this should be no problem.
```

1.75 ADMS command: SubMove

```
Command:       SubMove

Usage:         SubMove

Description:   Subtracts one from the number of turns taken. This should
               be used with verbs which don’t really need to take one of
               the player’s turns, for example: score, save, etc.

Example:
               SubMove
               EPrint "Your score is @cs out of @ms, in @tn turns."

Note:          The SubMove command should be used as early as possible
               in the verb’s code so that the number of moves doesn’t
               temporarily increase by 1 for the commands preceeding it.
```

1.76 ADMS command: Random

```
Command:       Random
```

```
Usage:         <var> = Random <max value>

Description:   Returns a random number between 0 and <max value>
               inclusive.

Example:
               a = Random 50
               a = a + 200
               SetTimer 0 .lampout a   ; lamp out in 200 - 250 turns
```

1.77 ADMS command: PrintValue

```
                Command:       PrintValue

Usage:         PrintValue <variable>

Description:   Prints the numeric contents of the specified variable to
               the screen.

Example:
               a = Random 10
               s = s - a
               Print "You are hit! Your strength is now "
               PrintValue s
               EPrint ".^"

See also:
                Print
```

1.78 ADMS command: DebugObj

```
Command:       DebugObj

Usage:         DebugObj <object>

Description:   Prints the name, parent object/room, child object, next
               sibling and previous sibling objects of the specified
               object. Use this for debugging -- this command should
               not be accessible in your final games.

Example:
               DebugObj noun1
```

1.79 ADMS command: If

```
                Command:       If

Usage:         If <var> <relation> <var>

Description:   If the relation between the first and second variable is
```

true, the following commands are executed, but if the relation is not true, all further commands are ignored until a matching ’EndIf’ command is found.

Valid relation operators are:

```
  =     (are the variables equal?)
  <>    (are the variables not equal?)
  >     (is the first var greater than the second?)
  <     (is the first var less than the second?)
  >=    (is the first var greater than/equal to the second?)
  <=    (is the first var less than/equal to the second?)
```

You may find that some relations will not compile. The reason for this is that not all the relations make logical sense, for example there is little point asking if a table if greater than 2, or even if a table is greated than an apple.

Examples:

```
a = 2
b = 2
If a = b
    Print "a and b are equal^"
EndIf
If a <> b
    Print "a and b are not equal^"
EndIf
If a > b
    Print "a is greater than b^"
EndIf
If a < b
    Print "a is less than b^"
EndIf
EndParse

a = 2
b = 2
If a = 2
    Print "a equals 2"
    If b = 2
        Print "b also equals 2"
    EndIf
EndIf
EndParse
```

See also:

EndIf

Note: The indented spacing inside the If commands is not ↩
    enforced
in any way by ADMS but it makes reading your programs a lot easier.

Note: Every single If command must have a matching EndIf command. If this is not the case, an error will occur on compilation.

1.80 ADMS command: EndIf

```
                Command:        EndIf

Usage:         EndIf

Descriptions:  Marks the end of a conditional execution block set up by
               the If command.

Example:
               If noun1 = apple
                   Print "This will only happen if noun1 is an apple.^"
                   Print "So will this.^"
               EndIf
               EPrint "This will always happen.^"

See also:
                If
                Note:           Every single If command must have a matching EndIf ↩
                    command.
               If this is not the case, an error will occur on
               compilation. Also, it is illegal to have an EndIf that does
               not match to a previous If statement. Any occurances of
               this will also cause compilation errors.
```

1.81 ADMS command: Loop

```
                Command:        Loop

Usage:         Loop

Description:   Marks the start of a block of commands that can be executed
               multiple times. The block is terminated with the EndLoop
               command. When the EndLoop command is encountered, command
               execution will continue from the command immediately
               following the Loop command. You can break out of a loop
               with an EndParse or Return command, or by using the
               ExitLoop command, which will continue processing from the
               command immediately following the EndLoop command.

Example:
               a = 0
               Loop
                   Print "This will be printed 5 times.^"
                   a = a + 1
                   If a = 5
                       ExitLoop                ; break out of the loop
                   EndIf
               EndLoop                 ; keep looping until
               Print "Finished.^"

See also:
                EndLoop
```

```
                 ExitLoop
                 Note:          Every Loop command must have a matching EndLoop  ↩
                    command
                or compilation will fail.
```

1.82 ADMS command: EndLoop

```
                 Command:       EndLoop

 Usage:         EndLoop

 Description:   Marks the end of a block of code, the start of which was
                defined by a Loop command.

 See also:
                 Loop

                 ExitLoop
                 Note:          Every Loop command must have a matching EndLoop  ↩
                    command
                or compilation will fail. Similarly, you can not have an
                EndLoop command without a matching Lop command.
```

1.83 ADMS command: ExitLoop

```
                 Command:       ExitLoop

 Usage:         ExitLoop

 Description:   This command tells the command execution to stop until
                it finds the EndLoop command matching the Loop block
                the execution is currently in.

 See also:
                 Loop

                 EndLoop
                 Note:          You can’t have an ExitLoop command outside of a  ↩
                    Loop
                structure.
```

1.84 Special Verbs

```
 In addition to the verbs you create that can be entered by the player,
 ADMS also has ’special verbs’. These are verbs which cannot be entered
 by the player, but are used in the ADMS language file as targets for
 subroutines from other verbs.

 All special verbs start with a period (’.’), for example you may have
 special verbs called ’.doobjlist’ or ’.givescore’, as long as it starts
```

with a period. The simple reason for this is that all words entered by the player which start with a period are removed from the input line before it is processed, so there’s no chance of them accidentally entering a verb which has been allocated as a special verb.

In addition to the user defined special verbs are several preset special verbs. These are as follows:

```
   .direction          This verb is executed whenever the player enters
                       a direction command (north, northeast, east,
                       southeast, south, southwest, west, northwest, up
                       or down) as the first verb on their input line.
                       This allows you to handle all 10 directions with
                       the same piece of code. The language file entry
                       for movement could be something like:

               verb = .direction
                   syntax = direction
                       a = CanGo player direction
                       If a = noroom
                           EndParse
                       EndIf
                       Move player a
                         ; your routine for describing the location at
                         ; which the player has arrived goes here
                       EndParse

                       Note that this is the one special verb in which
                       the syntax is taken into account! It’s possible to
                       use the verb to handle input such as ’north apple’
                       etc.


   .startgame          The .startgame verb is executed immediately the
                       game begins, before the player has a chance to
                       enter any commands. You can use this verb to
                       print any copyright messages for the start of the
                       game on the screen, set up any timers that are
                       needed in the game, and also to describe the
                       player’s starting location. If this verb is not
                       present, when your game is loaded the player will
                       simply be presented with a prompt, with no
                       explanation of what is happening.

   .preturn            Not yet implemented -- I hope to change this in
                       the next version of ADMS.

   .postturn           Not yet implemented -- I hope to change this in
                       the next version of ADMS.
```

1.85 ADMS variables

ADMS has 26 user variables (each given a letter of the alphabet, ’a’ through ’z’). These can be assigned values by typing their name, an equals sign, and the value you wish then to take. For example:

```
   a = 0               ; a now contains the value 0

   a = apple           ; a contains the object ’apple’

   a = b               ; a contains whatever b contains
```

It’s also possible to do simple arithmetic during a variable assignment. Arithmetic is limited to a single operator per assignment, and the four basic functions are supported. For example:

```
   a = a + 1           ; increases the value of what is in variable a

   a = b - 10          ; a now equals 10 less than the value in b

   a = a * 2           ; doubles the value of a

   a = b / c           ; a now equals the value in b divided by that in c
```

If you wish to do more complicated arithmetic (for example, a = (b+c)*d), you’ll need to do it in several stepd:

```
   a = b + c
   a = a * d           ; a now equals (b+c)*d
```

Please note also that arithmetic should only be performed on variables that contain numeric values. The following code:

```
   a = apple
   a = a + 1
```

..will not produce any errors, but may have unexpected results when the code is executed!

Variable assignments can also be made through many of the ADMS commands. Each command will give individual information about exactly how it works and what result will be given to the variable after its execution.

In addition to these user variables, ADMS also has several game variables. These variables can not be changed by the game programmer, but can be used in comparation in If commands, and also in any ADMS command that takes parameters of the same type.

The game variables are as follows:

location — The location that the player is currently standing in (type = room)

player — The player himself (type = object). Note that the player is treated as an object just like any other object in the game, so any operation you can perform on objects can also be performed on the player.

```
        currentscore            The current score the player has achieved
                                (type = number).

        maxscore                The maximum score the player can possible
                                achieve (type = number).

        verb                    The current verb (type = verb).

        noun1                   The first object found in the verb's
                                syntax line (type = object).

        noun2                   The second object found in the verb's
                                syntax line (type = object).

        direction               The direction found in the verb's
                                syntax line (type = direction).
```

Finally, ADMS has some constants which can be used in If commands. There are:

```
        true                    Many commands return 'true' or 'false'
                                values. Compare variables with 'true'
                                to see if an assignment returned a 'true'
                                value.

        false                   As 'true', except the opposite.

        noobject                Some functions such as GetStreamObj will
                                attempt to return an object as their
                                result. However, sometimes they run out of
                                objects to return, and in these cases
                                'noobject' will be returned.

        noroom                  Some functions such as CanGo attempt to
                                return a room as their result. If however
                                they are unable to do so, 'noroom' will be
                                returned.
```

1.86 The Travel File

The Travel file is what tells ADMS how all your rooms are linked together.
It's quite a complex thing, and is programmed in ADMS command language -- the same language as used in the Language file, so if you haven't looked at that yet you should do so before continuing with this text.

The travel table does not use 'verb=' or 'syntax=' keywords, but the keywords it does use are very similar. First of all, it uses 'room=' to know which room you're talking about. Let's say you have created 3 locations, which have the names 'MudPath', 'OutsideHouse', and 'Kitchen'. We'll first look at the MudPath location.

```
   room = MudPath
```

Now ADMS knows which room we're talking about, you need to tell it which direction you want to define. Any directions which are not defined are assumed to be directions in which travel is not possible. To define a direction, use the 'dir=' keyword, followed by one of the 10 compass directions. We want to be able to leave the MudPath to the north, which will take us to the OutsideHouse location. That's achieved like this:

```
     dir = north
        OutsideHouse
```

Note that to tell ADMS where you want to go, you just put the location's name. In actual fact, what's happening is a little more complicated. You can actually write ADMS commands after the 'dir=' keyword and they will be executed (with a few exceptions which are detailed individually in the ADMS command explanations). To demonstrate this, we'll move to our next travel table entry.

From OutSide house, we wish to be able to move south back to the MudPath, but also east in to the house. However, we don't want the player to be able to move in to the house unless the object we've made called 'door' has the 'open' property. This is achieved as follows:

```
   room = OutsideHouse                  ; travel data for OutsideHouse
      dir = south
          MudPath                       ; south goes to MudPath
      dir = east
         a = Has door open              ; is the door open?
         If a = true
            Kitchen                     ; yes, go to the kitchen
         EndIf
         RPrint "You'll have to open the door first!^"
```

If the travel table command processor finds a Return command (or RPrint), it will return the 'noroom' constant (see
               ADMS variables
) to the command
that called it. If it finds a room name, that room will be returned.

Finally, we'll define the travel table entry for the kitchen, again making sure the door is open before allowing passage through it.

```
   room = Kitchen
      dir = west
         a = Has door open
         If a = true
            OutsideHouse
         EndIf
         RPrint "You'll have to open the door first!^"
```

It's possible to do more complicated things using ADMS commands in the travel table too. Let's say we have a lift, and we can take it to different floors by pressing buttons in the lift. The current floor number is contained within the variable ’f’ and can be in the range 0 to 2.

```
   room = InsideLift
      dir = north
         a = Has LiftDoors open
         If a = false
            RPrint "The doors are currently closed.^"
         EndIf
         If a = 0
            GroundFloor
         EndIf
         If a = 1
            FirstFloor
         EndIf
         SecondFloor
```

Note: The user variables are entirely global and are shared throughout the Travel file and the Language file. Care must be taken so that variables in the travel table don’t overwrite variables that you’re trying to use in the language command procedures. It’s a good idea to set aside a small group of (3 or 4) variables which you use only in the travel table, this way you can stop variable conflicts.

1.87 The Synonym File

In your game you may wish to refer to objects or verbs by more than one name. The Synonym file allows you to set up alternative names for verbs and objects.

Let’s say we have a book in the game. Now the player may type any of the following commands and expect to be able to pick up the book:

```
   get book
   get paperback
   get novel
```

To define a synonym, enter in the synonym file the original verb/object that the game currently supports, and then a list of alternative words that should also be accepted to mean the same thing, all seperated by spaces. To achieve the above example, we’d put the following line in the synonym file:

```
   book        paperback novel
```

You can also use synonyms for verbs. Whilst some players are happy typing ’get book’, other may prefer ’take book’. Synonyms for verbs are defined in exactly the same way:

```
   get         take
```

For more examples, see the Example game’s Synonym file.

Note: Remember that only the first eight characters of a verb are actually stored by ADMS, so don’t try things like:

```
   inventory   inventor
```

..because they will both be seen as exactly the same thing by ADMS. This should all be changed in a future version of the compiler/interpreter.

1.88 The ADMS Interpreter

The interpreter is the program that is used to replay your compiled games.

After successful compilation, you'll be left with a new file, the name of which was specified in your Index file. Run the interpreter from the command line as follows:

```
ADMSplay <compiler output file>
```

The output file contains all the information necessary to play the game.

If you wish to distribute your games, you can give people the compiled output file that you have created and a copy of the ADMSplay program. Please read the Copyright and Distribution section for more information.

Remember to mention ADMS when you distribute your games! :)

1.89 Copyright and Distribution

ADMScompile is not public domain. It may be distributed freely as long as no unreasonable charge is imposed on the buyer. However, ADMScompile may not be distributed commercially without express written permission from me, Adam Dawes (see Author Information). I hereby allow the program to be included on the AmiNet compact discs, and I will also allow it to be distributed by Fred Fish if he wishes to do so.

If you wish to send me anything in return for my many many long hours of hard work, then please do! Money, postcards, candy, letters of praise etc. will all be gratefully received! :) Also please drop me an EMail if you like/use the ADMScompile program.

Please be aware that ADMScompile may become ShareWare in the future.

ADMSplay is public domain, and freely distributable. You may give this program to anyone who wishes to play your games.

The output files you create using ADMScompile are entirely yours and you may distribute them entirely how you please, but spare a thought for little ol' me who slaved for many weeks to make this program possible, and give me a mention somewhere in your game. Please? :)

1.90 Legal Information

THERE IS NO WARRANTY FOR THE PROGRAMS, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAMS "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAMS IS WITH YOU. SHOULD THE PROGRAMS PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY REDISTRIBUTE THE PROGRAMS AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAMS (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAMS TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

1.91 Acknowledgements

I send my sincere thanks to the following people and programs, without whom ADMS would not be what it is:

Oliver Smith/Kingfisher software
and AMUL (Amiga Multi User games Language)

For many ideas including the verb syntax and escape codes.

Graham Nelson
and Inform

For explanations of the old Infocom games, and inspiring my ideas for object trees and object properties.

Nico François

From whom I stole the legal information text (I hope you don't mind, Nico! :)

..and last but most certainly not least:

Infocom

For producing what are still the most classic games around. Your memory lives on.

1.92 ADMS -- Past Present and Future

ADMS history

Version 1.0

Version 1.1
ADMS right now

Known Bugs
ADMS in the future

Planned improvements
I promise with my hand on my heart that I will try when I make future versions of the program to leave your old source files still working. Whatever changes to the current commands I do make should be resolveable with a quick run through your source with a search/replace command.

1.93 Program History -- ADMS v1.0

Initial version of the program.

Release date: Not released. Completed on 10th April 1994.

Compiler understands the following commands:

```
Move            Give            GiveRoom        NearTo
IsHere          Carried         ObjRoom         CanGo
VerboseOn       BriefOn         SuperbriefOn    Has
HasRoom         Children        Weight          WCapacity
WUsed           OCapacity       OUsed           Confirm
ResetStream     GetStreamObj    GetParent       AddScore
SubScore        SetTask         ClearTask       ClearAllTasks
GetTask         Push            Pop             ClearStack
SetTimer        ClearTimer      GetTimer        EndParse
Return          Quit            Restart         Save
Load            Verbose         Brief           Superbrief
Print           EPrint          RPrint          PrintMsg
CheckCarried    PrintShortDesc  PrintLongDesc   PrintArticle
```

```
PrintObjShort   PrintObjLong    PrintObjFull    GetCR
Gosub           SubMove         Random          If
Endif           Loop            EndLoop         ExitLoop
```

1.94 Program History -- ADMS v1.1

Release date: 14/07/94

```
o New ADMS command:    PrintValue
            Usage:    PrintValue <var>
      Description:    Prints the value contained in the variable to the
                      screen.

o New ADMS command:    DebugObj
      Usage:          DebugObj <object>
      Description:    Prints name, parent, child and next/prev siblings
                      of specified object. Use for debugging.

o New ADMS command:    ExtendTimer
      Usage:          ExtendTimer <timer#> <#of turns>
      Description:    Delays the trigger of the given timer by the
                      specified number of turns
```

o Has and HasRoom commands now accept a property list as parameters, and not just a single property. For example, you can check to see if an object is both openable and closed in just one command:

```
a = has noun1 openable -open
if a = true
    eprint "The @n1 is closed.^"
endif
```

o Fixed bug in ClearTask command (it actually performed a SetTask instead)

o Dramatic speed increase in text output.

o Automatic paging of text. If more than a screenful of text is printed between 2 of the user’s commands, the program will pause and wait for the user to hit the RETURN key.

1.95 ADMS bugs

The following bugs are currently known within ADMS:

o ADMSplay crashes if you type off the end of the line.

o ADMScompile shortens multiple spaces within quotes to a single space, so the command:

```
    print "   "
```

will only actually print a single space.

All these will be fixed as soon as I find a bit of spare time to do it.

If you find anything else that seems to be wrong, please
contact me
and
tell me what the problem is (in as much detail as possible!)

1.96 Planned Improvements

I have quite a few ideas in store for ADMS, as soon as I have time to
implement them. Some of these are as follows:

```
  o  Command history/editing in ADMSplay (using cursor keys).

  o  Partial compilation so that successfully compiled sections of code
     need not be recompiled if they haven’t been changed.

  o  Restructuring of conditional execution blocks so that they run
     much faster (the code is rather inefficient at the moment).

  o  Object priorities so that the game creator can program the order
     in which objects are displayed. This would allow, for example,
     objects with very high priorities to become a part of the room
     descriptions, allowing dynamically changing descriptions.

  o  Lots more ADMS commands.
```

If you have any suggestions or ideas for improvements, please don’t
hesitate to
contact me
and tell me about them!

1.97 Author Information

ADMScompile and ADMSplay were painstakingly written by Adam Dawes, a
student of computer science at Brighton University.

You can contact me at the following addresses:

```
       InterNet:       ad32@vms.bton.ac.uk

       FidoNet:        Adam Dawes@2:441/93.5

       SnailMail:      Adam Dawes
                       47 Friar Road
                       Brighton
                       BN1 6NH
                       England
```

```
Please don't expect a fast reply if you contact me by snail mail, but I
will do my best! Send any gifts or donations to the same address. :)
```

1.98 Index

ADMS Commands

`GetTimer`

`Give`

`GiveRoom`

`Gosub`

`Has`

`HasRoom`

`If`

`IsHere`

`Load`

`Loop`

`Move`

`NearTo`

`ObjRoom`

`OCapacity`

`OUsed`

`Pop`

`Print`

`PrintArticle`

`PrintLongDesc`

`PrintMsg`

`PrintObjFull`

`PrintObjLong`

`PrintObjShort`

`PrintShortDesc`

`PrintValue`

`Push`

`Quit`

`Random`

`ResetStream`