MUDDL

From Eastnet

Contents

Copyright and distribution notes

Notes on the original version

The original version of this document was a report into the MUDDL language and the Debugging system that I wrote for it back in 1988.

A lot of the debugging parts are irrelevant now so some sections have been removed entirely and some have been cut down to size - Hopefully this means it will make a little more sense. Some parts of the document may refer to other parts which have been removed (eg: The VMS specific items) but that shouldn't matter much and I have tried to cut them out. Since it is fair to consider that MUDDL is now a dead language I have made no additions to the contents and tried to avoid correcting too much although I did correct some spelling mistakes - Is correcting mistakes you made in a document you wrote 22 years ago a little anal? I also left a few bits in for their historical amusement value.

Ultimately this incarnation of the MUDDL Guide should be considered a long since closed project with just some tidying up done in 2010.

Michael 00:09, 18 May 2010 (ADT)

Summary

This document gives a summary of the database language MUDDL (Multi User Dungeon Definition Language) which is used for creating text based multi user adventure games. It introduces a debugger which is used to examine databases written in this language.

The report consists of an introduction to multi user adventure game systems and a history of MUDDL; an overview of the language's concepts and an informal definition the language itself. A description of the debugger, along with examples of its use on a real database are given at the end of the report.

The (now long since dead) appendices of the report give details of setting up the debugger to run on a VAX-VMS system, and a list of MUDDL's internal commands and functions.

Introduction

Computer based adventure games

The first well known computer based adventure game was called "Colossal Cave", and was written in Fortran in the late 1960's on a Dec-10. Colossal was an extension of the fantasy role playing type game which allowed a single player to explore an underground world without physically moving from his or her terminal. To play the game, the player would type in one or two word commands. The computer would then act on them and display the results. Using directional commands, he or she could "move about" in the caves and manipulate objects they found there using the GET or DROP commands.

Following Colossal there have been many single-user adventure games, though few with as much fame or success. All of them use the same kind of commands, but some of the modern ones allow much more sophisticated input and give more intelligent output (sometimes in pictorial form).

Computer based multi user adventure games

A multi user adventure game, at first glance, looks similar to a traditional adventure game. There is one main difference, however: the player is not alone. If the player cannot solve a problem or puzzle at some point, they can ask other players in the game for help. This is not the only different feature though, having other players on the game means that interaction between them is an essential part of play. A good multi user adventure (We will refer to this as an MUA from now on) will exploit the multi user aspect to its fullest potential. It is possible to have problems that need co-operation between players to solve (the classic problem of this type is in opening a portcullis which is too heavy for a single player to open). In a game of this nature, players can talk to the others, discuss problems, help one another or, if the character they are playing is feeling particularly antisocial, kill the others and steal all the treasures they have accumulated.

One thing that writers of MUAs have to take into account is whether the game is suited to having more than one user. At Essex University, a multi-user version of Colossal Cave was written and ran for a week or so. After a few minutes of testing the game with more than one user playing, it became apparent that although the game was a perfect copy of the classic adventure, it was not playable with more than one user. Colossal was never designed to be played in a multi user environment. There is only one of each object, and a player needs to have some or all of them in order to solve the game. If a player entered the game and the lamp (which is needed to see below ground) wasn't in the start room, there was no point in the player carrying on with the game.

In a single user game, the closest an author can hope to come to matching the level of interaction found in an MUA is by giving the computer a persona which can attempt to react to the human player's actions with some degree of artificial intelligence. With today's AI techniques, however, a programmer of a single user system cannot come close to the interaction levels found in even the simplest multi user adventures.

A brief history of MUD and MUDDL

The database language MUDDL was first developed in 1979 by Roy Trubshaw, then an undergraduate at Essex University. He was writing an adventure game called MUD (The Multi User Dungeon) but unlike other adventure games of the time, he wanted his system to have more than one player. As well as having many players, Trubshaw wanted MUD to have a modifiable database; this would allow changes to the game to be made without recompiling the whole MUD program. If the database was altered to add another room, then all that was required to make the change effect the game, was for the database to be interpreted by the MUD program. Trubshaw called the database language MUDDL (standing for "Multi User Dungeon Definition Language") and based it very losely on the database system used to write "Colossal Cave".

MUDDL started as a fairly simple game definition plan with a lot of parts to the game that should have been in the database still written into the main MUD program. In 1980, when Trubshaw left Essex University, MUD was taken over by Richard Bartle who continued to develop the system and added various features. MUDDL is still used today in developing the Essex games, but Bartle has gone on to develop a follow up to MUDDL that is much more powerful in many ways. In theory MUDDL could be used to write single user systems, but it would be a waste of its true power.

Aims of the project

This project comprises of a test and debugging system for a MUDDL database. At present the MUDDL databases are edited on the DEC-10 at Essex using TECO (the DEC-10 system editor) and compiled with DBASE (a program that converts MUDDL databases to a form understood by the MUD program). Testing of the program can take place only by playing the game and seeing what happens. In practice a game will be very big (the database used to test the project is over seven thousand lines long) and mistakes are easily missed. Also there is no way to examine, in context, any of the original MUDDL code so debugging becomes a major operation. This project has been developed to analyse a MUDDL database and develop a system which allows a database to be built, loaded and examined in detail, putting all of the major sections in context. The project has also been written with extendability firmly in mind. In the future, it should be fairly easy to extend the debugger into a full multi user MUA system.

An introduction to MUDDL

Introduction

This chapter will describe the language MUDDL and give a general overview of its uses, limitations, and some of its concepts. There is no reference to the MUDDL language in existence. In the past writers of databases have relied on the MUD source, and trial and error. This section is just an introduction to MUDDL; a more comprehensive, but still informal definition of the language is given in the next chapter.

Basic concepts of MUDDL

A MUDDL database is a file that defines a whole game world (often known as a "land"). Before we can look at the language, we must examine some of the main concepts behind it and define some of the terms that will be used in the rest of this report.

Rooms

The land is made up of a number of rooms, which are the basic building blocks of the game and contain all the other entities the land has in it. A room has a description and a number of attributes; for example, if it has a natural light source or if it is a small room. A room need not be a room as it is usually thought of; for example a description of a part of a forest is still a room.

Players

The players are human controlled entities that can move from room to room and manipulate things that they find. The term "persona" is used to describe the player's computer form.

Objects

An object is a thing that is present in a room that is not a player. Some of these can be taken and dropped by players but a lot are fixed. Everything in the game that is not a room or a player is an object. An object can also be a container, in which case the object itself can contain objects.

Each object has a "class" which groups together sets of similar objects, so an object named "stick" and an object named "tree" both have the class "wood". Operations can now be defined to act on anything of class "wood" and they would hold for both trees and sticks.

An object has a number of other attributes and can be in a number of states (or "properties"). For each different property an object will have a different description and the game can check which property the object has. For example, a stick could have three states each with their own description:

```
0 - A stick lies on the ground before you.
1 - A burning stick lies on the ground before you.
2 - A charred stick lies on the ground before you.
```

Mobiles

This is a special form of object that moves from room to room under the control of the computer. Normally the mobiles represent "monsters" that will possibly attack players if they meet them. This does not always have to be the case though: a more peaceful example of a mobile would be a tumbleweed which blows from room to room. Mobiles can also have instincts associated with them. This means that they can react to situations they encounter. A good example of an instinctive mobile is the cat in the MUD database that will kill any rat it encounters on its travels.

Demons

A demon cannot be seen in the land, but it is an important concept in MUDDL. It is a process which, once initalised, waits in the background until it is ready and then does something. The process of starting a demon running is known as "firing" the demon. Their use in MUDDL is threefold.

Firstly, a MUDDL instruction can only perform one task before it completes. However, there is an option for an instruction to "fire" a demon as the instruction terminates. If this demon has no delay incorporated and the demon itself triggers off more demons, then a sequence of instructions can be simulated.

The second use of a demon is to force another player to do something. Normally, all MUDDL instructions act on the person who executed them, but MUDDL has a mechanism for sending a demon to another player in order to make them execute a sequence of commands.

The final use of a demon is really an extension of the first, which is called an "animation". Unlike the first use of demons, an animation incorporates delays. The demon forces a number of actions on the player over a given time span. A good example of an animation sequence is found in the MIST database where the program simulates an old man in a pub talking to the unsuspecting player. It does not let them escape till the conversation has ended or the desparately harassed player has quit.

Player attributes

A player has associated with them a number of attributes. The main ones are sex, strength, dexterity, stamina, maximum stamina, score and class. These attributes define a number of things such as how many objects they can carry, how much chance they will have in combat and how much chance they have of casting spells successfully.

The Wizard

When a player has reached a certain score, they become wizards (or witches). This opens up a whole new set of commands for them; where they were once restricted to moving from room to room by following the correct paths, they can now move to any room they choose with a single command. Wizards do not have to worry about death or combat because the game ignores the fact they have died. Many of the commands defined in a MUDDL database are separated into wizard commands and non wizard commands. A non-wizard player is usually known as a mortal. For historical purposes the Wizard may sometimes be referred to as a debugger.

Commands and pronouns

A command typed into a MUDDL game can be a fairly complicated one, but it basically breaks down to one or more commands of the form:

```
verb   noun   noun
```

The verb is compulsory, the two nouns are optional.

Commands can be strung together with "." or "-" and the pronouns "me", "it", "him", "her", "them" and "there" (the latter only if the player is a wizard) can be used to represent the relevent players, objects or rooms. The pronoun table is updated every time something new is seen or heard.

An example of a command, together with the way it breaks down is as follows: Command: where melissa-goto there-sget flower-give it to her

This breaks down into the four commands:

- where melissa
- goto there
- get flower
- give it her

The first command returns the room that Melissa is in. As a result this updates the user's pronoun table, and the word "there" in the second command is replaced by the roomname last recorded. The GOTO instruction will now move the player typing the command to the same room as Melissa, and when they enter the room, their pronoun table is updated with anything seen there. The SGET instruction is now executed, and, the pronoun table's view of "it" is updated to "flower".

Finally, the GIVE command replaces its two pronouns from the table. However, if there was another female in the same room as Melissa, and she was seen after Melissa, her name will be substituted for "her" from the pronoun table and the flower will be given to the wrong person!

Game chaining

MUDDL has a mechanism for moving a player to a specified room in a totally separate game. In moving from one game to another, the player cannot take their objects with them, as this would require different MUDDL programs to share data and may produce ambiguities. On a fast machine the transition from one game to another takes place unnoticed by the player and it provides a very versatile system where a player can move, or be moved, between a selection of games with different databases without seeming to leave the game which the player started.

Overall structure of a MUDDL database

GET files and building

A database consists of a single large file, but this is built up from a smaller file and a number of standard library files (known as GET files after the DEC-10 implementation). The actual database for the land being written can include the GET files by using the "@" symbol followed by the name of the file. Bartle wrote a number of standard GET files that should be used with all databases to define a skeletal land. The debugger accesses these files so that a complete MUDDL database can be verified. Unlike the Essex implementation, the debugger allows GET files to include GET files themselves, making them more versatile but meaning that a separate building must be performed. During this building stage, all blank lines are removed, and all lines preceeded by a "%" sign are displayed on the screen. These lines are build time comments and as they are for the aid of the database writer, they are not included in the full database. Normally they are just used to indicate which stage the build process has reached.

Comments

Comments can be included in three ways. If there is a single semicolon at the start of a line, it is interpreted as a database comment, and is totally ignored by the database interpreter. A comment can be included at the end of a line by putting a tab followed by a semicolon, followed by the text of the comment. If there are two semicolons together at the beginning of a line it is still ignored, but it is displayed on the screen at load time (this is again for the writer's use).

Separators

The TAB character is the basic separator in a MUDDL database which separates individual parts of a line without ambiguity. If spaces are used instead of TAB characters, although they may look the same on the screen, the database loader will be unable to correctly interpret the database and will give errors about missing arguments. A newline character will always separate one line from another.

Headings

In a full MUDDL database, there are twelve section headings and a large number of item headings. A section heading signifies a distinct new section of the database and is preceded by an asterisk; these are discussed in detail in chapter 4. An item heading signifies that a new part of a section is being started.

A definition of MUDDL

Introduction

This chapter will examine each section of a MUDDL database in turn and define the syntax of the MUDDL language. In many ways, especially in presuming default values, this implementation is an extension of the Essex University version. The reason for this is that the debugger needs to be able to load an incomplete database whereas the Essex system can only deal with fully formed ones. To make writing a database to run on the Essex system possible, these differences are pointed out in the text.

Definition of syntax and terms used in this chapter

The rest of this chapter will look at each section of MUDDL in turn and give the syntax of the various parts.

Where a person is referred to in this chapter, the term "she" will be used.

The database requires a TAB to be used to seperate arguments but this text uses a number of spaces as opposed to a tab character to attempt to overcome the ambiguities in printing tabs.

This chapter uses the following terms and syntax symbols.

```
roomname   This is a text string, beginning with a letter and up to 132
           characters in length. It can contain numbers and most punctuation
           symbols except "$". The Essex system will only allow room names to
           be seven characters in length, and they may only contain letters and
           numbers. The debugger will give no warnings if the Essex convention
           is not adhered to. Roomnames are not case-sensitive so "start" is
           the same as "START". Although the program does not force the writer
           to do so, it is a very good idea to keep roomnames to just letters
           and numbers.
roomlist   This is a list of tab seperated roomnames. At any point, a set of
           roomnames can be enclosed in "[]" or "<>" parentheses. If the list is
           enclosed in square brackets, then a random roomname from the list
           will be inserted at LOAD time; if angle brackets are used, then
           a random roomname will be chosen from the list every time the list
           is consulted.
string     A string is a list of characters, of the same type as a roomname
           but it can begin with a number and contain any punctuation symbol.
number     A number can be positive or negative, in most cases the sign is
           important.
{  }       Items enclosed in braces are optional
[  ]       Items enclosed in square brackets mean that the item is an argument,
           which is explained in the definition.
 |         Items seperated by the "|" sign, specify that either the first item
           OR the second item may be inserted.
...        This signifies that a list will continue in the same way, and contain
           as many further items as are required.
```

Unless otherwise stated, if a definition has two entries of the same type then this signifies that the section can contain as many of these entries as are required. The end of the section will be marked by a new section header.

The *NAME section

```
*NAME    [game-name]
```

The *NAME section defines the name of the database, and of the game. The argument game-name is a string.

The *PERSONA section

```
*PERSONA    [persona-name]
```

The *PERSONA section tells the system which persona file the system will use. Persona-name is a string and is normally the same as the game-name. It will differ when, for example, two databases share the same persona file.

The *COMBAT section

```
*COMBAT    [count]
           [message]
           [message]
            ...
           [count]
           [message]
           [message]
            ...
           (etc etc...)
```

The *COMBAT section consists of 24 subsections. Each section is headed by a number (count) which says how many messages, of type string, are in the list that follows. The use of this section is to define the words used in the combat sequences.

In some MUA's, combat sequences are fairly boring, they consist of messages such as:

- Mattygrove hits you.
- You hit Mattygrove.
- You miss Mattygrove.

MUDDL has a powerful combat system that can handle such things as the player retaliating to a blow and hitting the opponent. These sections contain the terms used during such combat sequences. If a random choice from the list of messages in subsection 4 is signified by [4] then the messages in the lists fit into the combat system (with Mattygrove as an example opponent) as follows:

Mattygrove attacks:

- You [16] that Mattygrove is [1] at you [2]...
- Eg: You perceive that Mattygrove is staring at you ominously...

Mattygrove is hit:

- You [3] Mattygrove with a [4] [5]!
- Eg: You take aim at Mattygrove with a weighty blow!

Mattygrove misses:

- You [22] [6] a [7] [5] [9] Mattygrove.
- Eg: You comfortably duck a hasty thump from Mattygrove.

The player misses Mattygrove: Your [7] [5] is [10] Mattygrove. Eg: Your poor backhand is blocked by Mattygrove.

Mattygrove hits:

- The [12] of a [5] [9] Mattygrove sends you [21]; (or)
- You are [11] by the [12] of a [5] [9] Mattygrove!
- Eg: The savageness of a blow from Mattygrove sends you sideways; (or)
- You are jarred by the force of a forehand by Mattygrove!

The player is dead, Mattygrove wins the fight:

- You can [13] your [14] [15]...
- Eg: You can sense your last seconds slipping away...

The player is ok and attacks Mattygrove again:

- [17] you [18], and [19] into the [8].
- Eg: Yet you bear up, and advance into the slaughter.

Mattygrove has hit but the player hits back:

- Your [20] [5] sends Mattygrove [21]!
- Eg: Your follow-up thrust sends Mattygrove sprawling!

The player misses Mattygrove again:

- Your [7] [20] [5] at Mattygrove is [22] [23].
- Eg: Your puny follow-through clout at Mattygrove is comfortably ducked.

Mattygrove is dead:

- Your last [5] [24] Mattygrove!

- Eg: Your last slash murdered Mattygrove!

The *LEVELS section

```
*LEVELS
Male      [level  1]
          [level  2]
             ...
          [level 10]
Female    [level  1]
          [level  2]
             ...
          [level 10]
```

This is a list of strings that make up the level titles for both male and female players. When a player has a certain number of points, she has a title connected with them. For example, a player on 1,000 points may be known as "Mattygrove the hero". When a player has no points, she is on level 1, which is usually known as "novice". At 400 points, she will go up a level and every time her score doubles from then on she will go up another level. When a player reaches level-10 (normally this level name is known as "wizard" or "witch"), the score has no further effect on the level title, even if it decreases.

3.2.5. The *HOURS section.

```
*HOURS
Sun | 0     [open]     [close]
Sun | 0     [open]     [close]
  ...         ...          ...
Mon | 1     [open]     [close]
Mon | 1     [open]     [close]
        ...         ...
Fri | 5     [open]     [close]
Sat | 6     [open]     [close]
```

This section defines the opening times of the game. The values for open and close are integers between 0 and 24 which say the hours at which the game is open. In this version, but not the Essex version, a day can be specified by a three character name.

Any day can be defined more than once, so, if the game to be open on Wednesday from Midnight till 9am, 1pm till 2pm and then 5pm till Midnight, it should be specified as follows:

```
WED    0    9
WED    13   14
WED    17   24
```

The *ROOMS section

```
*ROOMS    [number]
[roomname]    [attributes]
              [short-description]
              [long-description]
                    ...
[roomname]    [attributes]
              [short-description]
              [long-description]
                    ...
```

This section defines the rooms that make up the land. The argument to the actual section header is the number of rooms to be defined in the section. The debugger ignores it (although it does give an error message if the number is exceeded).

Each new room is started with a roomname; these should all be unique. Following the room name, there is a list of attributes which can be selected from one or more of the following:

```
Light      The room contains a natural light source.
Water      There is water here.
Oil        There is oil here.
Death      If a mortal player comes in this room then she will die.
Sanctuary  If an object is dropped here, the player will have added to her
           score the amount of points the object was worth.
Hideaway   If a player is in here, she cannot be seen from outside by mortals.
Hide       If an object is in here, it cannot be seen by mortal players.
Small      Only one player or mobile can be in this room at a time.
Chain      The CHAIN attribute takes two arguments, [game-name] and [room]
           which follow on from the word CHAIN seperated by tabs. This attribute
           allows one game to run another game and put the player in the
           specified room.
Dmove      The DMOVE has a single argument [roomname] which again, follows the
           word DMOVE and a tab. It means that if any objects are dropped in
           here, either by accident (eg: by a player dying) or on purpose then
           the object is moved to the room specified in the roomname argument.
           The SANCTUARY attribute usually has a DMOVE attribute pointing to an
           inaccessable room as well. If this were not the case, players could
           drop objects, score points, pick them up and repeat the operation
           until they reached wizard level.
Nolook     This means that the room cannot be looked into from an adjacent room.
Silent     If a witch is in a silent room, then she receives no status messages.
```

The short room description can be one of two forms: a string, in which case this is taken literally as the short description of the room; or it can be a percent sign followed by a roomname. If this latter form is used, then the short description of the room is taken as being the same as the short description of the room specified. Using the "%roomname" form, complex areas, like forests, can be built up. Also, if the short room description that is referenced is changed, then that description will change for all the other rooms which reference it.

The long room description consists of zero or more tab indented lines which give the main description of the room.

A room has two descriptions for a few reasons. The short description is meant to give the player a basic idea of where they are in the land, the long one narrows it down, usually to a unique room. Normally, both descriptions of the room are given when a player enters it, however, if a player is playing in BRIEF mode then only the short description of a room will be seen. the LOOK command always displays both descriptions. Commands like "where", which allow players to find out where in the land something is, will only return the short room description.

The *MAPS section

```
*MAPS    number
[roomname]    [string]
              [string]
                ...
[roomname]    [string]
              [string]
                ...
```

This section is not implemented in the debugger, as so few databases actually use it. The strings following the roomname can be used to create a pictorial representation of the room and its exits.

The *VOCABULARY section

```
*VOCABULARY
CLASS     [classname]
            ...
SYN       [synonym]     [real-word]
            ...             ...
OBJECT    [object]    [classname]    [weight]      [value]
            ...           ...            ...           ...
CONJ      [string]
FPREP     [string]
BPREP     [string]
QUANT     [string]
PRON      [string]
MPRON     [string]
```

```
FPRON     [string]
PPRON     [string]
PLACE     [string]
SELF      [string]
INST      [string]
UNIVERSAL [string]
NOCLASS   [string]
PERSN     [string]
STHING    [string]
ATHING    [string]
MOTION    [motion]
          ...
NOISE     [string]
          ...
ACTION    [action definition]
          ...       ...
```

This section defines the vocabulary of the game, every word and action the game can interpret is in this section. Each of these headings can (and usually do) appear more than once in the section. It will help to look at the subsections in turn and to give an example of each.

Class

This is a list of strings that make up the class names of the objects in the database. For example:

```
class    wood
         weapon
         food
```

Syn

```
This defines synonyms which can be used in place of words that are already
```

defined in the database. The debugger will not check whether the words you are creating synonyms for already exist in the database. For example:

```
syn      gun      shotgun
         take     get
         carry    get
         wisky    whiskey
```

Object

```
All the objects in the game are defined here. There are four parts to each
```

definition: the object's name, its class which must have been defined under the class heading, a number representing its weight in grammes, and a number representing its value. For example:

```
object   stick     wood     200      0
         crown     gold     10000    300
         goat      animal   100000   20
```

The single argument subsections

These define words that MUDDL itself uses. The debugger will provide default values for all of them if they are missed, but the Essex system does not. The headings, default values, and meanings are as follows:

```
Conj       "and"         This is word used for stringing commands together, for
                         example: GO NORTH "and" GET AXE.
Fprep      "with"        This is the fight preposition, for example:
                         KILL FRED "with" AXE.
Bprep      "at"          This is used for example in LOOK "at" FRED.
Quant      "all"         This is the quantifier; it is used to manipulate a
                         quantity of objects in for example, GET "all"
Pron       "it"          This is the object pronoun; in a command, this word
                         will be replaced by the name of the last object seen.
Mpron      "him"         This is male pronoun; it is replaced by last male
                         seen or heard.
Fpron      "her"         This is the female pronoun; it is replaced by last
                         female referenced.
Ppron      "them"        This is the person pronoun; it is replaced by the last
                         male or female referenced.
Place      "there"       For wizards, this is replaced by the last room
                         referenced.
Self       "me"          This is always replaced by the player's persona name.
Inst       "whichever"   This is used in the demon definitions to pass an
                         argument on to the next demon called.
Universal  "shout"       This is the command that will be used to transmit a
                         message to everyone in the land.
Noclass    "none"        This is the word used in action definitions to signify
                         something that is not an object or a person.
Persn      "person"      This is used in action definitions to signify another
                         player.
Sthing     "something"   This is used in action definitions to signify an
                         object of any class.
Athing     "anything"    This is used in verb definitions to signify a person
                         or an object of any class.
```

Motion

This section contains a list of strings of valid directions that the game should understand. The directions can be either normal strings or, strings with a "$" sign at the beginning. A "$" means that this is a special direction, which can only be taken if a demon of that name is called. For example:

```
motion     north
           south
           east
           $special
           northwest
```

Noise

This section exists so the writer can declare words to be added to the database's dictionary but not to be used in the game. The normal use of this section is to stop players coming into the game as certain reserved words. For example:

```
noise      someone
           a
           the
```

Action

In this section, every verb that the game understands is defined. The format is not easy to define, so a lot of examples from the MIST database are included.

The way MUDDL executes commands is to search the verb definitions for any verbs that match the command typed in at the player's keyboard. If one is found, MUDDL checks if it is allowed to execute it. If it can run the command then the interpreter will search no further and the command will be carried out. If the command cannot be executed for some reason, it will go onto the next relevant action definition. If it fails on all definitions, or if a definition is not found, then it will give an error. It is a good practice for database writers to give all verbs a default definition with no conditions associated with it; this can be used to give a more useful error message.

The format of an action can be loosely described as:

```
verb {.command} arg1 arg2 function farg1 {farg2} m1 {m2} {m3} {dmn}
```

If the verb is preceeded by a dollar, it is a demon action as opposed to a play action, meaning that it cannot be typed in directly as a command.

The .command part is optional, if it is present, it calls an internal MUDDL function of that name.

Normally, arg1 and arg2 are the nouns that go with a verb. Arg1 can be an object class, or one of the words defined in the single argument section of vocabulary.

Arg2 will be described later.

M1, m2 and m3 are messages that will get sent to various groups of players.

It is always the case that m1 will be sent to the player who entered the command and m3 will go to the whole land but m2 could go to different places depending on the function.

The simplest type of action is one with no arguments that simply prints a message to the user. An example of this is the verb "MUD" which prints the message "No! This is MIST I tell you, MIST!"

The definition for this action is:

```
mud   none   none   null   null   1193   0
```

The "1193" is a reference to the message defined later on in the database in the *TEXT section.

In order to signify a message in the *TEXT section from now on, any message numbers that refer to it will be written after the action definition, preceded by a "#". This is not MUDDL but is a convenient way to avoid too much explanation. Using this method, the above definition would be written:

```
mud   none   none   null   null   1193   0
#1193  No! This is MIST I tell you, MIST!
```

Another simple type of action is one which displays a message, but that executes a MUDDL command as well. A good example is "sleep" which rests the player for a few seconds by calling the intenal MUDDL command ".sleep". This is written as:

```
sleep  .sleep   none    none   null    null    40    0
#40    ZZZzzz...
```

More complex actions can be written which allow an argument to a command. Take an example of picking up a piece of burning coal which is of class "hot".

```
get    hot    none    null     null     100     0
#100   You reach down to pick up the glowing coal, but as your hands get close
       you have second thoughts.
```

If it is decided that wizards should be able to pick up hot coals, then a function could be used to to check if the player is a wizard. This leads to an example of an action that uses something other than a null function.

The full definition of the "get" command when used with coal in this way now becomes:

```
get    hot    none    unlesswiz        null    100     0
get    .get   hot     none     null     null    101     0
#100   You reach down to pick up the glowing coal, but as your hands get close
       you have second thoughts.
#101   You take the red hot coal without so much as noticing that it is a little
```

```
    warm.
```

If arg1 is left as "none" (The string defined with NOCLASS) but arg2 is a class, then this means that the verb is executed if the player has an object of that class or if there is one in the same room. A good example is the verb "swim":

```
swim   none   river  null   null   200      0
swim   none   bath   null   null   201      0
swim   none   none   null   null   202      0
#200   You take one look at the fast flowing river and decide that swimming
       in there would not be a good idea.
#201   Its a little bath you fool, you can't swim in it!
#202   There is nothing here that you could possibly swim in.
```

Full three part commands can be built up by using both arg1 and arg2, the statement to handle "hit door with stick" could be:

```
hit    access  wood   null    null  450   0
#450   You bash the stick against the heavy wooden door, fun isn't it.
```

Functions can be inserted for a number of reasons. The one previously shown was used as a conditional, but more complex ones exist. An example is in touching a stone which increases the player's score by 250 and destroys the stone. (note, the expdestroy function has two arguments instead of the usual one):

```
touch  stone   none  expdestroy   null   250   800   0
#800   The stone vibrates and vanishes. As it does, you feel somehow different.
```

All of the commands so far have only used a single message argument, the following MIST command uses two messages (m1 and m3), a function called .flush and a MUDDL command.

```
charge  .flush  none   none   ifwiz  null   100    101    101
100     Certainly oh master, and I hope they pay too!
101     -+ Charging has just started for this account +-
```

The ".flush" command flushes the player's command buffer and the pronoun table. It is included to stop wizards typing CHARGE and then pressing return a number of times (which repeats the last command given to the game). Finally, the "dmn" argument allows a demon to be called. It is differentiated from the normal messages by being negative. If a command has a demon call in it then the demon is called after the command has been executed. Demons are defined in the *DEMONS section.

A full list of MUDDL internal commands and functions can be found at the end of this document.

The *DEMONS section

```
*DEMONS    number
[demon-number]  [demon-name]  [arg1]  [arg2]  [delay]  [attributes]
[demon-number]  [demon-name]  [arg1]  [arg2]  [delay]  [attributes]
```

This section defines all the demons in the game. The demon-number is the number by which the demon is called, and the demon-name is the name of a demon action as defined in the action section of vocabulary. It must always be preceded by a "$". The arguments arg1 and arg2 are nearly always "none" (or the word which is declared as "Noclass" in the Vocabulary section) with the exceptions being a couple of MUDDL system demons in the standard GET files.

The delay argument is the number of seconds that this demon should sleep before the demon action is called; if it is specified as "-1" then the action takes place immediately. If a random delay time is required then the lower limit and the upper limit should be seperated by a minus sign (eg: 50-300 would specify a random delay between 50 and 300 seconds).

Following the delay, there can be a list of zero or more tab seperated attribute keywords, these are:

```
global     The demon acts on everyone at once, not just the person who fired it.
enabled    The game automatically starts this demon running when it is started.
always     This demon never goes away. After it has gone off it remains
           enabled but dormant. One use of this is to check that someone has
           done something; if a demon with the "enabled" attribute is fired,
           the game can later check that this has happened using the ifenabled
           function.
```

The number after the *DEMONS token defines the maximum number of demons, this is important in the Essex system, but is ignored in the debugger.

The *OBJECTS section

```
*OBJECTS
[objname] {speed} {inst} {atp} [roomlist] stp  mxp  scp  {stam} [attrs]
[number]  [string]
          ...
[number]  [string]
  ...       ...
[objname] {speed} {inst} {atp} [roomlist] stp  mxp  scp  {stam} [attrs]
[number]  [string]
          ...
[number]  [string]
  ...       ...
```

This section defines fully every object and mobile in the game. The arguments to the definitions are as follows.

```
Objname    The name of the object to be defined, this must be an object
           previously defined in the Obj section of vocabulary. The obj
           section's definition of an object is very basic; it only defines
           class, weight and value. This section allows individual instances
           of an object be further sub-defined.

Speed      If speed is defined, then the object is presumed to be a mobile.
           The numeric argument is the time (in seconds) before the mobile
           will try to leave the room it is in. A value of "0" means that
           the mobile will never stay still, and a value of -1 means that
           it will never move.

Inst       This is a demon (and thus a negative number) that is executed every
           time the mobile enters a room. It defines the mobile's instincts.
           (see 2.2.4).

Atp        This is the mobile's attack probability; a number between 0 and 100
           that defines how much chance the mobile has of attacking any players
           it meets on its travels. An attack probability of 0 means that the
           mobile will never attack whereas one of 100 means the mobile will
           always attack.

Roomlist   This is a list of rooms the object is to be put into. If the name
           of another object (it must be a container) appears instead of a
           roomname, then the current object will be put inside the object
           appearing in the list.

Stp        This is the start property of the object. It is a numeric value
           defining which of its properties it will have at the start of the
           game. For an explanation of an object's property values, see 2.2.3.

Mxp        This is the maximum property that the object can have, the minimum
           is always 0. If this number is negative then the object will take
           on a random property value from zero up to the unsigned value of
           the number every time it is looked at.

Scp        This is the score property of the object. An object is only worth
           points to a player when it has a certain property, this number says
           what this property is.

Stam       In mobiles, this is its maximum stamina it has if it gets into a
           fight.

Attrs      This is a list of zero or more tab seperated attribute keywords that
```

definine the attributes of an object. These are as follows:

Bright The object provides a source of light when it is at property 0.

Noget This stops mortals picking this object up. The NOGET attribute would be given to objects like trees or doors.

Noit This object does not update the pronoun table when it is looked at, and thus cannot be refered to as "it" - This attribute also means that the object will not be affected by quantifiers, for example: "get all".

Contains This should be followed by a tab seperated number, it defines the object as being a container and thus able to carry other objects. The number is the weight in grammes that it can carry.

Disguised If the object is a container and has the DISGUISED attribute then mortal players will not be told that the object is a container.

Opened Normally, a player can only remove objects from a container if the container is in property 0, this attribute makes it so that objects can be removed from the container at any time.

Transparent The contents of a container can only be seen by mortals when the container is in property 0. Giving a container the TRANSPARENT attribute makes the contents of a container visible whatever property it has. Mortal players will be told that a non transparent container contains "something" when it has any other property.

Nosummon If a player is carrying an object with this attribute, then she cannot be summoned elsewhere by another player.

Fixed This defines an object as being an important part of the game and thus impossible to take, even for wizards. This is given to objects that form part of a room description, like a marker on a beach saying if the tide is in or out.

Following the initial definition, there is a list of property descriptions headed by the property number. The description of each individual property can take as many lines as are needed. If a "%" sign followed by an object name is used as a property description, then the description of the object named after the "%" sign is used in the property value being defined. If the description is simply a "?" then the object description is taken from a file called objname.DBA. The file is consulted each time a description is printed so the description can be changed as the game is running.

To make things clearer, there follow some examples.

```
beacon    beach4    1    1    2    noget    bright
0    You see before you a large beacon, burning brightly with magical flames.
1    A large pile of branches and twigs form an unlit beacon here.
```

A beacon is in the fourth beach location (beach4), it can have two property values (0 and 1) because Mxp is defined as 1. The beacon starts off in property 1 (Stp is defined as 1). The "bright" attribute is defined, so if it is in property 0, then it gives off light. The score property of the object is defined as 2, which is above the maximum number of properties that the object is allowed to have, thus ensuring that the object can never be worth any points. Finally, the "noget" attribute means that the object cannot be taken away by mortal players.

```
tide  cause1  cause2  cause3  pool1  pool2  pool3  1  1  2  noget  fixed
0    The tide is out.
1    The tide is in.
```

This is a tide marker, placed on every beach location It has two states, in and out - it cannot be taken by mortals or wizards.

```
door  court1  armour  2  2  3  noget
0    There is an open door here.
1    There is a closed door here.
2    There is a locked door here.
```

```
door  pantry   kitch    1    2    3     noget
0     %door
1     %door
2     %door
```

This declares two typical doors. The first, the connecting door from the courtyard to the armoury, starts off locked; whereas the second, the connecting door from the kitchen to the pantry, is merely closed. This section of code declares two objects, but the players will actually see four doors, one in each of the rooms mentioned. We define a connecting door as being in two rooms rather than one because when we set the door in one room to a new property value, the other door in the other location will also be set to that property.

```
teleport  start    <good1     bad1>     0    0    1      noget     bright
0     A teleport beam pulsates before you.
```

This declares a glowing teleport connecting room start to either room good1 or room bad1. In the object section (but not in other sections), it does not matter which form of bracketing you use in room lists. This is because all of the items are placed in the land when the game is loaded and so this kind of roomlist is only ever referenced once.

```
grave     grave1   grave2   grave3   grave4    0    -3    4     noget   bright
0     Inscribed on a modestly asture tombstone here is the name "Richard"
1     There is a large monument here, bearing the inscription "Roy, the
      grand-daddy of multi user games."
2     Before you, shining with a mystical light stands a massive golden
      monument encrusted with diamonds. The legend reads "Lorry, modest
      to the end."
3     ?
```

This defines a glowing gravestone that takes on a random property from 0 to 3 each time it is looked at. If it is looked at when has the property 3 (defined as "?", then the file "grave.dba" is consulted for the description.

The *TRAVEL section

```
*TRAVEL      [motion-word]
[roomname]   [condition]   {roomto}  [directions]
                ...          ...          ...
[roomname]   [condition]   {roomto}  [directions]
                ...          ...          ...
```

This section defines the connections between the rooms in the land. It defines which room a player will arrive in if they travel in a certain direction and what, if anything, there is to stop the player moving. In the Essex version there must be a travel definition defined for each of the rooms defined in the *ROOMS section.

The motion-word argument is the player's command used to move in a direction. In an English language database, this will nearly always be the word "go". As this word does not actually have to be typed to introduce a direction, its use is largely obsolete and so the debugger ignores it.

The roomname is the name of the room that the current travel definition refers to.

The condition defines whether movement is allowed or not. If the condition has a tilda in front of it then the logic of the condition is reversed. The following is a list of conditions allowed:

```
n | none          No condition, it is a simple movement command.
e | empty         The condition is true if the room is empty and if the player
                  is carrying nothing.
[object|class]    The condition is true if the object or an object of the right
                  class is in property 0 and is present in the room or carried
                  by the player.
[number]          If the number is positive, the message of that number is
                  printed, otherwise a demon is fired.
```

The roomto argument can be a list of roomnames, and is the room the player will be moved to if the condition is sucessful. If a random destination is required, then a roomlist enclosed in either square or angle brackets can be used. If the argument supplied is a "0" then the message "You cannot go that way" will be displayed.

The list of directions is one or more of the directions defined in the motion section of vocabulary, and can be synonyms. If random directions are required, then the whole list or a sublist can be surrounded in square or angle brackets depending when the random direction is to be worked out.

When the travel tables are processed by the game, each line is taken in order.

If a condition fails then the next entry is looked at, otherwise the relevant action is taken and processing stops. If all conditions fail, the message "You cannot go that way" will be printed.

There follows two examples of travel entries from the MIST database. They show how powerful this system can be when used properly.

```
mazea8  trog     0   s   se  sw
     ali      0   s   se  sw
     undead   0   s   se  sw
     talisman  mazea9  se  s   sw
     n        mazeb9  se  s   sw
     -200     n   e   w   nw   ne
     1109     u   d   in  jump
     1145     o   pit
#1109   You blunder into the darkness...
#1145   Naaaaah! That would be too easy!
```

In the room "mazea8" players cannot go south, south-east or south-west if there are objects of the type trog (the class for most of MIST's monsters), the type ali (moles) or the type undead (zombies) present.

If an object of type talisman is being carried, or there is one in the room, and there are none of the above mobiles in the room then going south, south-east or south-west will take the player to the room "mazea9" otherwise, it will move them to room "mazeb9" with the next line.

If the player goes north, east, west, north-west or north-east, then demon 200 will be started (This checks what objects the player is carrying and moves her to various other rooms). If the player goes up, down, in or jumps, the message 1109 is printed but no actual movement takes place. If the player goes out, or gives the direction command "pit", then message 1145 is printed.

```
mazea9  undead   0   o
     n        mazea7   [o   e]
     ~talisman mazea4   o    e
     door     <crypt   barn   pantry>    o   e
     1145     o   e   w   n   s   e   ne  nw  se  sw  u   d   in   pit
```

In room "mazea9", the player cannot go out in the direction "out" if there is a zombie in the room.

The loader will insert a random one of "o" or "e" in the next condition. It will be assumed that the loader chose "e". The line now becomes:

```
     n        mazea7   e
```

If the player goes east, they will be placed in room "mazea7".

The next condition checks that the player is NOT carrying a talisman, if this is so then they are moved to mazea4.

Now, if there is a door in property 0 (opened), going out or east will take the player to either the crypt, the barn or the pantry. As east has been trapped previously (moving the player to mazea7) only the out command will be valid here. The decision about which room the player will be placed in is taken at the time the command is executed; if square brackets had been used instead of angle ones, the decision would have been made when the game was loaded.

Finally, any other direction that can be typed will produce message 1145.

The *TEXT section

```
*TEXT      number
[number]   [string]
           ...
[number]   [string]
           ...
```

This section defines every text message used in the game. The value following the *TEXT section header is the maximum number of messages you expect to define. It is needed in the Essex version but ignored by the debugger.

The message body can consist of any number of tab indented strings; a new message is started when a new number is found as the header. If the text of the message is simply a "?" then the program will look in the file "number.DBA" for the message text; this file is read every time the message is referenced, so if it is updated while a game is running, the change will take place immediately.

For example: The two text messages 40 and 41 could be defined as follows.

```
40    ZZZzzz...
41    One does not cook cats, one eats them raw - I thought EVERYONE
      knew that!
```

The MUDDL debugger

Introduction

Note to the 2010 text revision: If you are reading this document as a reference to the MUDDL language then most of this chapter may be of little interest. I considered deleting it but there are some useful examples in here and it does show some more examples of how the language worked in practice. This is especially true of the demons, which are always something of a black art.

This chapter will give a basic overview of the system and its use. It will discuss running the program, building and loading databases, debugger commands and give examples of the debugger in use. It does not discuss installing the system or its environment, details of these are not available in this version of the document since they are now entirely obsolete.

Starting the debugger

The format of the command is:

MUDDL { [database] } { -HELP } { -NOWINDOW } { -NOCOMMENT }

The arguments are as follows:

```
database      The name of the database to load.
-HELP         Print a brief help text for the system.
-NOWINDOW     Do not use windows. This argument must be used if a DEC terminal
              is not being used.
-NOCOMMENT    Do not display comments during the loading stage.
```

If a database name is specified on the command line, the program will start up and automatically try to load that database.

If the -NOWINDOW option has not been specified then the screen will split up into four windows. The first window, labelled "STATUS", is for any game or operating system status messages. The second, labelled "OUTPUT", is for any general game output. "DEBUG" is where any error messages are displayed, or where

extra information is printed. The bottom window "INPUT" will contain the prompt "--*>" when it is waiting for input. It will allow the user to type up to 76 characters; if this limit is exceeded then it will automatically terminate the line and send it for processing.

Building the database

The debugger will only deal with a single large database file. In order to create this, the game text file and the GET files must be merged together. This process is called binding. The database file is called database.TXT and should be in the directory MUD_DATA:[SOURCE], the GET files should all be in the directory MUD_DATA:[GET]. When the build is finished, its output will go to MUD_DATA:[DATABASE].

To build the database, the command "/BUILD database" (where database is the name of the source file) is entered from within the debugger. The screen will clear and a number of diagnostic messages will be printed. If the build is sucessful, it will give a success message to the user, otherwise it will print an error message. In both cases, it will return to the debugger after a key is pressed.

Loading the database

Before the database can be examined or debugged, it must be loaded into the debugger. This can be done in two ways: if the MUDDL program is called with the database name as an argument, then it is automatically loaded; otherwise the command "/LOAD database" should be entered.

The load process takes quite a long time, so it gives messages on the output window saying which section it is currently loading; in the *VOCABULARY and *LEVELS sections, it will also display the subsection names. Unless the -NOCOMMENT flag has been specified, database comments that begin with ";;" will also be displayed. Any errors found in the loading stage are database syntax errors. These will be displayed in the debug window with enough information for the source of the error to be traced. Not many errors will be serious enough to abort the loading process, the ones which are being major format errors such as unknown section names.

If a lot of error messages are being produced on the debug window and are scrolling too fast to be seen, then the command "/LOG" should be typed and the load restarted. This will cause all debug output to be sent to the file MUDDL.DBG, which can be printed later for a hard-copy of the errors. Once the load stage is complete the status window will display the current database name and the number of rooms in the land.

Debugger commands

The debugger understands two types of commands, internal commands and game commands. The internal commands are the actual debugger commands, which are always preceeded by a "/". Game commands are the verbs contained in the database itself, and will not work unless a database has been loaded.

The following is a list of internal commands and their functions:

```
/HELP          Displays a list of internal commands and brief descriptions of
               their functions.
/BUILD [name]  Builds a database called "name". If the argument is missed then
               it is prompted for.
/LOAD [name]   Loads a database called "name", again, if the database name is
               missed then it will be prompted for.
/LOG           Logs all the output on the DEBUG window to the file MUDDL.DBG.
/DETAIL        Displays detailed analysis messages for every command in the
               debug window.
/QUIT          Terminates the debugger.
/SPAWN {com}   Spawns a VMS command from the debugger, if no command is
               specified, then it will spawn to VMS.
/AUTO          Enables/disables the facility to put the user into the PHONE
               or MAIL system if they are being phoned or mailed whilst in
               the debugger.
```

The following commands, although internal commands, will produce results only if a database is loaded:

```
/LOOK {room}   Looks at the current room or if an argument is given, at
               that room. This will give the name, attributes, short and full
               description of the room looked at. The argument is normally
               the name of a room, but if it is numeric and starts with a "$"
```

```
            sign, then it will look at the room by its internal room number.
/EXITS {room}   Displays the travel table for the current room, or if an
            argument is given, at that room. The room can be numeric as
            long as it starts with a "$" sign.
/GOTO [room]    Sets the current room to the specified room. The room can again
            be numeric if it starts with a "$"
/MSG [n]        Displays the text message "n" unless the number is negative in
            which case it displays the definition of the demon call.
/BRIEF          If /BRIEF is typed, then only the short description of a room is
            displayed when /GOTO is used.
/LOC            Lists all the rooms in the database along with their short
            descriptions.
/LEVELS         Displays the level tables.
/HOURS          Prints a table of the opening hours of the game.
/COMBAT {arg}   If this command is used alone, it will give help on the
            combat system. The arguments are as follows:
            *           Prints the whole of the combat table.
            n           Prints section "n" (n is a number from 1 to 24) of
                        the combat table.
            .           Simulates a combat sequence.
            keyword     Prints a relevant fight message for the keyword
                        specified, keywords are: START, IHIT, THEYMISS,
                        IMISS, THEYHIT, IMDEAD, IMOK, IHITBACK, IMISSAGAIN,
                        VICTORY. (See 3.2.3)
/VOCAB {arg}    If this command is used alone, it will give help on the
            vocabulary system, the arguments are as follows.
            *           Displays every word the system knows.
            .           Displays every verb the system knows.
            -           Displays every synonym the system knows.
            [l]         [l] is a single letter from "A" to "Z" and includes
                        "$". It displays a list of every word the
                        system knows beginning with that letter.
            .[l]        As above, but it is limited to just verbs.
            -[l]        As with [l], but it is limited to just synonyms.
/EXAM [obj]     Allows an object in the current room to be examined.
```

If a database is loaded and a command is typed that is not preceeded by a "/" or a ";" then it is interpreted as a game command. The existence of the verb is checked in the synonym tables and any substitutions that may be needed are performed. The verb tables are then searched for the verb and if anything relevant is found, the action definition is displayed on the output window. An exception to this is the UNIVERSAL command (normally "shout") as defined in the vocabulary section: this verb is executed straight away. Commands beginning with a ";" are simply echoed to the debug window. This allows a debugging log to be commented.

Moving around

The /LOOK, /EXITS and the /GOTO commands can be used to test the rooms and the travel tables. After a database has been loaded, the debugger will always start off with the current room defined as the room with the attribute "startrm". If one of these is not found, the room called "START" is defined as the current room. From this room, /GOTO can be used to travel to another room, /LOOK to examine the room in detail and /EXITS to examine the travel tables.

The /MSG command can be used in conjunction with /EXITS to examine messages or demons present in the travel tables. The use of these commands is best illustrated by means of examples.

After the database has been loaded loaded the first room is examined.

```
--*>/look
Room: CHAT [2].
(Startrm)
(Light)
A room with a view.
You are in a nice, warm cozy room full of seats and comfy chairs.
There is a single exit to your north, but the view through the
portal is marred by a wall of mist.
--*>/exits
    n    <start   sfor2   fyard1>    n    o
```

Moving north or out from this room would lead to either start, sfor2 or fyard1.

The choice is made randomly after the command is entered.

The room sfor2, which forms part of the southern forest, is now examined.

```
--*>/goto sfor2
Room: SFOR2 [268].
(Light)
Southern forest.
You are pushing your way through the trees of the southern forest. The
only ways seem to be north, back to the road, or east.
--*>/exits
    n     road3   n    o
    n     sfor1   e
    1700  s  w    ne   nw   se   sw  in
--*>/msg 1700
The forest is too dense in that direction.
```

There are three exits from this room, two of them (north and out) lead to road3 whilst the other (east) leads to sfor1. The other directions produce message 1700.

The two rooms leading from this one are now examined, the first, by actually going to it, the second by using an argument to /LOOK and /EXITS.

```
--*>/goto sfor1
Room: SFOR1 [267].
(Light)
Forest's edge.
You are standing in the fringes of the southern forest. To your east
your way is blocked by the castle moat, but the trees to the west are
thin enough to pass through.
--*>/exits
    n     sfor2   w    o
    1700  n  s    e    ne  nw   se   sw   in
--*>/look road3
Room: ROAD3 [206].
(Light)
Forest road.
You are walking along the road through the forest. To both the north and
south, the trees crowd closely to the roadside, making it very difficult
to see into the depths of the forest.
--*>/exits road3
    n     nfor2    n
    n     sfor2    s
    n     road2    e     o
    n     road4    w
    1700  ne   nw  se    sw   in
```

Looking at verbs

If a verb is typed directly into the debugger, every relevant action definition is displayed on the output window. Individual action definitions can now be examined. The following example illustrates using the debugger to examine the verb "kiss". The /MSG command is used to examine the messages produced by the actions.

```
--*>kiss
KISS is a known verb.
kiss froggy none destroycreate prince 1778 0
kiss princey none expdestroy null 1779 0
kiss pplayer none null null 1886 0
kiss wetfish none null null 1891 0
kiss person none emotion null 50 328
```

There are five actions defined for the verb kiss. The first, "kiss froggy" destroys the object of class "froggy" and creates a prince. It then displays message 1778.

```
--*>/msg 1778
The frog looks in amazement as you bend down to kiss it, it tries to hop
away but, you catch it and do the evil deed. With a flash, the frog that
was always quite happy being a frog transforms into well... Something.
```

The next action, "kiss princey", destroys the prince and gives the player who executed the command the amount of points the prince was worth. It then prints message 1779.

```
--*>/msg 1779
The Prince's eyes light up as you approach him and he grabs you in his
arms and gives you a big sloppy kiss; at that, he flies off into the
distance shouting "Wheee, Chase me!"
```

The third action "kiss pplayer" (pplayer is the class for all of MIST's "human" mobiles) prints message 1886.

```
--*>/msg 1886
Yeuch! Why on earth would you want to do that!
```

Kissing a live fish is handled with the fourth action "kiss fishy", this prints message 1891.

```
--*>/msg 1891
Hmmm.. seems fishy to me guv.
```

Finally, kissing other players is taken into account, this calls the function "emotion" with an argument of 50 and prints message 328.

```
--*>/msg 328
You give them a great big sloppy kiss.
```

Looking at demons

If a verb fires a demon, then the demon can be traced using the /MSG command. There follows an example of tracing a game demon that alters the land's weather cycle to winter. It is a complicated demon that would take a great deal of time to examine with just a game listing. First, the verb to change the season is examined, this is the "winter" command.

```
--*>winter
WINTER is a known verb.
winter none none ifwiz null 958 0 -33
```

This checks that the user is a wizard, and if so, prints message 958 on the screen and calls demon 33.

```
--*>/msg 958
OK, it's winter...
--*>/msg -33
$winter  none  none  -1  global
```

Demon 33 calls a verb "$winter" with no time delay, it is a global demon so it affects everyone. The "$winter" verb is now examined.

```
--*>$winter
$WINTER is a known verb.
$winter none none ifdisenable null 28 0 -34
$winter none none ifdisenable null 29 0 -34
$winter none none ifdisenable null 30 0 -34
$winter none none ifdisenable null 31 0 -34
$winter none none ifdisenable null 32 0 -34
```

The "$winter" verb checks to see any of five other demons are enabled. If they are then they are disenabled (turned off) and demon 34 is fired. The five demons that it checks for are 28, 29, 30, 31 and 32 - These can be checked:

```
--*>/msg -28
```

```
$brewing none none 15-30 global
--*>/msg -29
$pouring none none 300-600 global
--*>/msg -30
$calm none none 600-1500 enabled global
--*>/msg -31
$brewing1 none none -1 global
--*>/msg -32
$pouring1 none none -1 global
```

These demons are all connected with the weather cycle, so the end result of these actions is to turn off whichever one of the five demons is currently controlling it, and then to fire demon 34. This should now be examined.

```
--*>/msg -34
$snow none none -1 global
```

Demon 34 calls the action "$snow", this is checked by entering the verb into the debugger.

```
--*>$snow
$SNOW is a known verb.
$snow none none set rain 2 0 -35
```

"$snow" sets the object rain to property 2 (using the SET function) and then fires off demon 35.

```
--*>/msg -35
$snow1 none none -1 global
--*>$snow1
$SNOW1 is a known verb.
$snow1 none none sendeffect rain 959 0 -36
```

Demon 35 calls "$snow1" with no time delay, "$snow1" then sends message 959 to every location containing the object "rain" using the sendeffect function.

It also fires demon 36 when the command terminates.

```
--*>/msg 959
It has started to snow.
--*>/msg -36
$snow2 none none 300-1200 global
--*>$snow2
$SNOW2 is a known verb.
$snow2 none none set rain 0 0 -37
```

Demon 36 sleeps for a random time between 300 and 1200 seconds, it then calls the action "$snow2", this sets the object rain to property 0 and fires demon 37.

```
--*>/msg -37
$snow3 none none -1 global
--*>$snow3
$SNOW3 is a known verb.
$snow3 none none sendeffect rain 960 0 -38
```

Demon 37 calls the action "$snow3" with no delay, "$snow3" sends message 960 to each room and then fires demon 38.

```
--*>/msg 960
The skies clear, and it stops snowing.
--*>/msg -38
$snow4 none none 300-1200 global
--*>$snow4
$SNOW4 is a known verb.
$snow4 none none sendeffect rain 955 0 -39
```

Demon 38 again sleeps for a random time between 300 and 1200 seconds. When it wakes, it calls the action "$snow4". This sends message 955 to every room containing the object "rain" and then fires demon 39.

```
--*>/msg 955
Clouds are beginning to gather in the sky.
--*>/msg -39
$snow  none  none  15-30  global
```

Demon 39 sleeps for a random time between 15 and 30 seconds, it then calls the action "$snow" thus creating an infinite loop simulating a wintery weather cycle.

Conclusions

Throughout development and testing of the system, the MIST database was used for test data. MIST is a real game that is used every night at Essex University by players from all over the country; it is currently maintained by myself, and two other students from Essex. It is a long and laborious job to try and cross reference all the database parts to find errors, but using the debugger this task has been made much easier. Many faults in the MIST database have been discovered and traced, some of which have been there since it was first written in 1985.

The informal definition of MUDDL has also proved to be very useful: previously writers of MUDDL databases have only had the original MUD database to give a basic overview of MUDDL syntax. Many of MIST's objects were incorrectly defined and some of the demons looked like GOTO's in badly written BASIC programs. The debugger was written using modular techniques, as a result it is fairly simple to extend to a full multi user game system. The structure definition module already contains all of the data structure definitions needed for the players, even though the debugger does not require them. Writing the debugging system has proved both intersting and practical. As a result of all the testing, both the MIST database, and my own understanding of MUDDL have improved.

Appendix: MUDDL functions and internal commands

This appendix contains a list of MUDDL functions and internal commands. These are only listed for completeness, to provide a reference rather than a tutorial for the MUDDL language.

MUDDL functions

The only reference to the actions that these functions perform is found in the source to the MUD program.

backrot	create	dead	dec	decdestroy
decifzero	decinc	delaymove	destroy	destroycreate
destroydec	destroydestroy	destroyinc	destroyset	destroytogglesex
destroytrans	disenable	emotion	enable	exp
expdestroy	expinc	expmove	expset	fix
flipat	float	floatdestroy	flush	forrot
holdfirst	holdlast	hurt	ifasleep	ifberserk
ifblind	ifdead	ifdeaf	ifdestroyed	ifdisenable
ifdumb	ifenabled	iffighting	ifgot	ifhave

ifhere	ifheretrans	ifill	ifin	ifinc
ifinsis	ifinvis	iflevel	iflight	ifobjcontains
ifobjcount	ifobjis	ifobjplayer	ifparalysed	ifplaying
ifprop	ifpropdec	ifpropdestroy	ifpropinc	ifr
ifrlevel	ifrprop	ifrstas	ifself	ifsex
ifsmall	ifsnooping	ifweighs	ifwiz	ifzero
inc	incdec	incdestroy	incmove	incsend
injure	loseexp	losestamina	move	noifr
null	resetdest	retal	send	sendeffect
sendemon	sendlevel	sendmess	set	setdestroy
setfloat	setsex	ssendemon	stamina	staminadestroy
suspend	swap	testsex	testsmall	toggle
togglesex	trans	transhere	transwhere	unlessberserk
unlessdead	unlessdestroyed	unlessdisenable	unlessenabled	unlessfighting
unlessgot	unlesshave	unlesshere	unlessill	unlessin
unlessinc	unlessinsis	unlesslevel	unlessobjcontains	unlessobjis
unlessobjplayer	unlessplaying	unlessprop	unlesspropdestroy	unlessrlevel
unlessrstas	unlesssmall	unlesssnooping	unlessweighs	unlesswiz
writein	zonk			

MUDDL internal commands

The only references to the actions that these commands perform is found in the source to the MUD program (specifically in the file MUD5.BCL). The MUD and MIST help files will also give an indication to what the commands do.

.assist	.attach	.autowho	.back	.begone
.berserk	.blind	.brief	.bug	.bye
.change	.converse	.crash	.ctrap	.cure

.deafen	.debug	.demo	.detach	.diagnose
.direct	.drop	.dumb	.eat	.empty
.enchant	.exits	.exorcise	.flee	.flush
.fod	.follow	.freeze	.get	.go
.haste	.hours	.humble	.ignore	.insert
.inven	.invis	.keep	.kill	.laugh
.log	.look	.lose	.make	.map
.mobile	.newhours	.p	.paralyse	.password
.peace	.police	.pronouns	.proof	.provoke
.purge	.quickwho	.quit	.refuse	.remove
.reset	.resurrect	.rooms	.save	.say
.score	.set	.sget	.sgo	.shelve
.sleep	.snoop	.spectacular	.stamina	.summon
.tell	.time	.unfreeze	.unkeep	.unshelve
.unsnoop	.unveil	.value	.verbose	.vis
.wake	.war	.weigh	.where	.who
