# DSA Reference

March 27, 2009

# Contents

# Chapter 1

# Introduction

*Dungeon Specific Actuators* (*DSA*s) are a programming system for Paul Steven's "*Chaos Stricks Back*" clone CSBWin. They provide dungeon designers a set of tools to create ... and well as modifiy .... .

*DSA*s are finite state-machines which execute *FORTH* (stack machine) like instruction sequences, which operate solely on 32-bit integer data.

The term *state* refers to the fact that at any given moment a given *machine* is a particular state which often, but not necessarily, reflects a logical physical state: open/closed, standing/walking/running, etc.

The machine responds to *events* based on its state at the time the event occurred and the type of event. For *DSAs* events are signaled or triggered by messages (2.3). Since the machine's behavior is dependent on two things (state and event) these are often thought of as being a two-dimensional grid (2D state transition table) where the rows and columns represent states and messages respectively.

## 1.1 Target Audience

The goal of this document is to be accessible to a wide target audience and a has couple of main goals.

On one hand it is desired to present exact details on how *DSAs* operate so those with workable programming skills can use as it as complete operational reference. To achieve this requires that parts of document will be very difficult to understand if one is not familiar with formal programming documentation.

On the other hand it is desired ...XX

For those readers which have little to no programming experience, it is recommend to skim through the *Introduction* and *Basic Concepts* chapters and then to proceed to *Explanations.* Do not bother attempting to understand everything you read. In fact do not be daunted if everything seems like gibberish. The goal of the read-through is to expose you to the basic terminology and components of the system.

## 1.2 Overview

- A dungeon may contain up to 256 *DSA* definitions.
- A given level may use at most 32 of these definitions.
- Each instance of a *DSA* has two persistent storage slots, termed *Parameters* (2.4.1).
- A dungeon may provided shared persistent storage slots termed *Global Variables* (2.4.4).
- Instructions additionally have access to a *stack* (2.4.2), an *array* (2.4.3) and *Temporary Variables* (2.4.5) during execution.

# Chapter 2

# Basic concepts

## 2.1 Instances

*DSAs* are game elements logically similar to items (specifically they are objects). The designer creates the defination and the places an *instance* of that defination at a given position within a cell.

Like other objects the engine allows multiple *DSA* instances in a given position of a cell, which form a logical *pile* in the same manner any other game item would create.

It is allowed for a given position to contain more than one instance and in that case the set of instances from a logical pile (one on top of the other).

## 2.2 States

States in *DSAs* are represented by integers and each definition allows a choice from three mechanisms of storage:

1. Local – Each instance implicitly stores its state, where state is limited to a range of 0-31.
2. Global – All instances share the same state, and that state may be any 32-bit value.
3. Parameter B – Each instance stores its state within parameter B, which allows states on the range of 0-1023.

The choice of state storage method is per *DSA* definition.

## 2.3 Messages

Events are signaled to *DSAs* by message-passing.

*DSAs* communicate via message-passing (Actor model)

It is important to note that message are sent to a cell position and not to a specific *DSA* instance. The engine allows multiple instances in a given cell position and in that case each will receive the message in turn.

Is the order specified?

### 2.3.1 Message Type

XXX

## 2.4 Memory

*DSAs* have access a number of logical elements which are arrays. In all cases these arrays are zero-based indexed, or specifically for an *n* element array, the set of valid indices are on zero to *n-1*, or formally: $[0,\ n)$.

### 2.4.1 Parameters

Each instance of a *DSA* has two local persistent storage elements termed parameters, which are called *A* and *B* respectively.

As mentioned in the description of an instance 2.1, multiple instances may occupy a given cell position. A given instance in this pile may directly access the parameters of the instances where are *below* it. For the instance to access the "A" and "B" parameters of the instance below the current executing the designer specifies "C" and "D" respectively and "E" and "F" for two instances below, etc. This mechanism cannot be used to access the parameters of instances above.

All values stored to a parameter are automatically truncated to 18-bits.

What is the promotion rule?

### 2.4.2 Stack

The *stack* (LIFO) consists of 100 32-bit integers, which is empty on execution entry. If during execution the stack either *overflows* or *underflows*, the engine displays an error message and exits.

### 2.4.3 Static Array

The *static array* is shared (global) memory consisting of 100 32-bit integers.

| |
|---|
| Assuming that a value isn't overwritten, will a given element retain the same value throughout execution (ignoring game loading)?<br>What about on start-up or just after a load? Is the array zeroed? |

| | | |
|---|---|---|
| &@ | read value from stack | (4.8.1) |
| &! | set value from stack | (4.8.2) |

### 2.4.4 Global Variables

Global variables are shared persistent storage elements. Specifically they retain their values across *DSA* calls and are loaded/saved with the game.

These are enabled by the designer through the "*Edit/Global Info*" panel, press the *"Edit Database*" button and pull-down to "*Global Variables*" in *"Database Type*". They are enabled in banks (or groups) of 16. As of this writing the maximum allows is four banks (up to 64 global variables).

Instructions which access these will be denoted as array accesses to *global.*

| | | |
|---|---|---|
| "GV" *n* "@" | read value from stack | (4.10.1) |
| "GV" *n* "!" | set value from stack | (4.10.2) |

### 2.4.5 Temporary Variables

*Temporary Variables* are a set of local variables that only survive a single execution of a *DSA*. Additionally references are *checked*. Specifically this means that it is invalid to read a from given index before it has been written to. If an attempt to read prior to

write occurs, the engine will display a programming error message but not terminate the game.

Instructions which access these temporary variables will be denoted *tempVar*.

| | | |
|---|---|---|
| "V" *n* "@" | read value from stack | (4.11.1) |
| "V" *n* "!" | set value from stack | (4.11.2) |
| &MONL&D | | (4.17.5.2) |
| &PARAM@ | | (4.21.2) |
| &PARAM! | | (4.21.3) |
| | | |

## 2.5 Cells

Each level of a dungeon is composed of a 2D grid of elements termed *cells*, which are smallest logical element. So, when a player move forward one position, he/she is moving from one cell to another. Cells can be broken into two types: *open* and *closed*, which correspond to floors and walls respectively.

Each cell has four positions within it, which are described in *Cell Locations* 2.6.1.

## 2.6 Enumerations and Bit Encoding

Since *DSA* instructions always operate on integer values, there must be mechanisms to map integers to various different required meanings.

### 2.6.1 Cell Locations

Instructions which ... will denote these as: *cellLocation*.

Cell locations are encoded using 2 bits for the position (2.6.4) within the cell, 6 bits for the level and 5 bits each for the x and y coordinates.

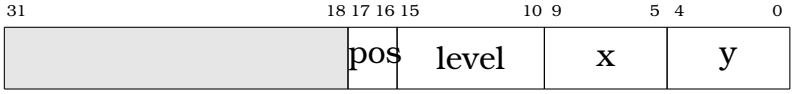



Table 2.1: Cell Location Format

Algebraically this can be expressed as:

$$(((64pos) + level)\,32 + x)\,32 + y \tag{2.1}$$

which expands to the following:

$$65536\,pos + 1024\,level + 32x + y$$

### 2.6.2 Locations

Some instructions allow...

| | | |
|---|---|---|
| cursor | $-1$ | -1 |
| character | | $-(100\,charID + packIndex)$ |
| monster | | $-(1000\,monsterID)$ |

Table 2.2: Extended Locations

1. cursor
2. character

location = -1 (negative 1). The object is placed in the cursor. -1000 < location <= -100. The object is placed on a character. location <= -1000. The object is placed on a monster. location >= 0. The object is placed in one of the cells of the dungeon.

——In the cursor—— If the cursor already contains an object then no object is created or placed. The 'posMsk' parameter is ignored.

——On a character—— The location parameter is of the form: - ( 100 * characterOrdinal + packIndex ). If an object is already present at that spot in the backpack then no object is created or placed. The 'posMsk' parameter is ignored. See Pack Indices for the numeric location numbers.

——On a Monster—— The location parameter is of the form: - (1000 * monsterID). The "posMsk' parameter is ignored.

——In a cell of the dungeon—— The location parameter determines the cell in which the object is placed. The 'posMsk' parameter determines the position within the cell.

'posMsk' is a bit-encoded value with bit zero standing for direction zero (north), etc. If 'posMsk' is non-zero then the object is placed randomly in one of the positions for which the corresponding bit is non-zero. If, for example, 'posMsk' is set to 5 then the object will be placed randomly in either position zero (north) or position two (south). If 'posMsk' is zero (no bits set) then the object is placed at position zero. The location parameter is encoded as three bitfields. Bits 10-15 are the 6-bit level number, bits 5-9 are the x (or column) coordinate, and bits 0-4 are the y (or row) coordinate.

### 2.6.3 Manhattan Distance

Distance calculations are based on Manhattan distances.

The standard definition, which will be termed *manhattanDistance*, is:

$$|x_0 - x_1| + |y_0 - y_1| \tag{2.2}$$

A similar definition, which will be termed *neighborDistance*:

$$|x_0 - x_1| + |y_0 - y_1| - 1 \tag{2.3}$$

Where $(x_0, y_0)$ and $(x_1, y_1)$ are the two sets of "x" and "y" cell coordinates, similar to *cellLocations* 2.6.1.

Since moving objects are restricted to moving in the cardinal directions (N,S,E,W), the Manhattan distance is the number of *steps* that would be required to move between the two cells. The *Neighbor distance* is the number to steps to get to be one-step-away, which will typically be used for comparing things like party-to-monster distances.

### 2.6.4 Facing and Positions

| ID | Facing/Closed | Open | mask |
|---|---|---|---|
| 0 | NORTH | NW | 1 |
| 1 | EAST | NE | 2 |
| 2 | SOUTH | SE | 4 |
| 3 | WEST | SW | 8 |

Table 2.3: Facings and Closed/Opens Cells

#### 2.6.4.1 Position Masks

When instructions read or write data related to a specific position within the cell, this will be denoted: XXX.

....mask columns for open (**??**) and closed (2.3) cells respectively.

### 2.6.5 Character Related

#### 2.6.5.1 Character Numbers

Each hero has an associated ordinal: 0,1,2,3. These values indicate their position left-to-right in the top display. Instructions which expect these values will be denoted as:

*charID* ::={0-3}

Some instructions additionally allow specifing the active leader (by the value 4), they will denote:

*partyID* ::={0-4}

Other instructions allow asking about *any* party member (by the value 5), they will denote:

*charAnyID* ::={0-5}

#### 2.6.5.2 Character Fingerprint

The *fingerprint* of a character is the initially set to the bottom 16-bits of the location in the dungeon of the text that defined the character when he was reincarnated or resurrected.

This value can be used to uniquely identify each character and is read and written as part of the character's variables (4.19.5). As such, it's value may be modified as the designer wishes. Keeping character fingerprints unique is advised for all but expert programmers.

Example usages include:

- determining if a given character is in the party.
- allows to find the character's name even when changed through reincarnation.

The following instructions are related to the fingerprint:

| Instruction | | Reference |
|---|---|---|
| &CHAR@ | read character variables | (4.19.5.1) |
| &CHAR! | write character variables | (4.19.5.2) |
| &CHARNAME | retrieve character's name | (4.19.7) |
| &SWAPCHARACTER | add, remove and swap from party | (4.19.2) |
| &WHEREISCHAR | characters current location | (4.19.6) |

#### 2.6.5.3 Skills

Each character has 20 skills. Four of these are *primary* skills, which are displayed to the user, and the remaining are *secondary* (or *hidden*) skills.

| ID | parent | name |
|---|---|---|
| 0 | - | fighter |
| 1 | - | ninja |
| 2 | - | priest |
| 3 | - | wizard |
| 4 | 0 | swing |
| 5 | 0 | thrust |
| 6 | 0 | club |
| 7 | 0 | parry |
| 8 | 1 | steal |
| 9 | 1 | fight |

| ID | parent | name |
|---|---|---|
| 10 | 1 | throw |
| 11 | 1 | shoot |
| 12 | 2 | identify |
| 13 | 2 | heal |
| 14 | 2 | influence |
| 15 | 2 | defend |
| 16 | 3 | fire |
| 17 | 3 | air |
| 18 | 3 | earth |
| 19 | 3 | water |

Table 2.4: Character Skills

| Instruction | | Reference |
|---|---|---|
| &EXPERIENCE+ | add experience points to a skill | (4.19.9.2) |
| &MASTERY | query a skill's level | (4.19.9.3) |

Dungeon Master Encyclopaedia

#### 2.6.5.4 Spells

Spells are cast by speaking 2-4 runes.

| Power | |
|---|---|
| 1000 | Lo |
| 2000 | Um |
| 3000 | On |
| 4000 | Ee |
| 5000 | Pal |
| 6000 | Mon |

+

| Elemental | |
|---|---|
| 100 | Ya |
| 200 | Vi |
| 300 | Oh |
| 400 | Ful |
| 500 | Des |
| 600 | Zo |

+

| Form | |
|---|---|
| 10 | Ven |
| 20 | Ew |
| 30 | Kath |
| 40 | Ir |
| 50 | Bro |
| 60 | Gor |

+

| Class/Align | |
|---|---|
| 1 | Ku |
| 2 | Ros |
| 3 | Dain |
| 4 | Neta |
| 5 | Ra |
| 6 | Sar |

Table 2.5: Spell Runes

Further XXX: Dungeon Master Encyclopaedia

#### 2.6.5.5 Inventory Slots

Each hero has 30 slots in which XXX

| | |
|---|---|
| 0 | Left hand |
| 1 | Right hand |
| 2 | Head |
| 3 | Body |
| 4 | Legs |
| 5 | Feet |
| 6 | Pouch, bottom |
| 7 | Quiver, top right |

| | |
|---|---|
| 8 | Quiver, bottom left |
| 9 | Quiver, bottom right |
| 10 | Neck |
| 11 | Pouch, top |
| 12 | Quiver, symbol position |
| 13 | Backpack, symbol position |
| 14-21 | Backpack, top row – left to right |
| 22-29 | Backpack, bottom row – left to right |

Table 2.6: Inventory Ordinals

### 2.6.6 Object

THESE HUGE TABLES SHOULD BE MOVE TO AN APPENDIX

| ID | Type | Mask | |
|---|---|---|---|
| 0 | Door | 1 | 0x0001 |
| 1 | Teleporter | 2 | 0x0002 |
| 2 | Text | 4 | 0x0004 |
| 3 | Actuator | 8 | 0x0008 |
| 4 | Monster | 16 | 0x0010 |
| 5 | Weapon | 32 | 0x0020 |
| 6 | Clothing | 64 | 0x0040 |

| ID | Type | Mask | |
|---|---|---|---|
| 7 | Scroll | 128 | 0x0080 |
| 8 | Potion | 256 | 0x0100 |
| 9 | Chest | 512 | 0x0200 |
| 10 | Miscellaneous | 1024 | 0x0400 |
| 11 | Expool[1] | 2048 | 0x0800 |
| 14 | Missile | 16384 | 0x4000 |
| 15 | Cloud | 32768 | 0x8000 |

Table 2.7: Object Types and Masks

There are 16 possible 'classes' or types of objects. Several DSA command require a 'mask' of the object types of interest. For example, you may want only to examine weapons and scrolls. In that case, you would use a mask of ( 0x0020 + 0x0080 = 0x00a0 ) ( or 32 + 128 = 160 ).

Here are values necessary for manipulating monsters, items and the party inventory slots.

Items numbers (with type number added) are used in &type calls (eg 50002 for torch). For potions, you will get an &type value that includes the power (eg 200 powered DES potion = 80000 + 512 + 200 = 80712). In order to identify a potion type, you need to strip the power rating from it. This is made harder by the +80000 for potiontype. Easiest to us '&type L80000 &% L256 &/' and then compare the basic potion type number directly (so for the DES potion, number recovered = 2). This wil not be necessary for water/empty flasks (0 will be power number associated - eg 85120 for empty flask)

Basic cloud numbers are used in &createcloud calls (as documented). Add the 'cloudtype' number for &type calls (eg 150050 for a fluxcage) Inventory numbers are used for both &chposs directly (eg L0 L1 &CHPOSS to get the right hand of first character) and also in &move/&add/&del command using -(char num * inventory number * 100) for the position.

#### 2.6.6.1 Monster Types

Basic monster numbers are used in &monster calls and all filters (eg 24 for dragon). Add the 'monstertype' value for &type calls (eg 40024 for a dragon).

MONSTERTYPE +40000

| | |
|---|---|
| 0 | Scorpion |
| 1 | Slime Devil |
| 2 | Giggler |
| 3 | Flying Eye |
| 4 | Hellhound/Rat |
| 5 | Ruster |
| 6 | Screamer |
| 7 | Rockpile |
| 8 | Ghost/Rive |
| 9 | Golem |

| | |
|---|---|
| 10 | Mummy |
| 11 | Black Flame |
| 12 | Skeleton |
| 13 | Couatl |
| 14 | Vexirk |
| 15 | Worm |
| 16 | Blue ogre/Antman |
| 17 | Wasp/Muncher |
| 18 | Knight |
| 19 | Zytaz |

| | |
|---|---|
| 20 | Water elemental |
| 21 | Oitu |
| 22 | Demon |
| 23 | Lord Chaos |
| 24 | Dragon |
| 25 | Lord Order |
| 26 | Grey Lord |

Table 2.8: Monster Types

#### 2.6.6.2 Item Types

**Weapons** WEAPONTYPE +50000

| | |
|---|---|
| 0 | Eye of Time |
| 1 | Storm Ring |
| 2 | Torch |
| 3 | Flamitt |
| 4 | Staff of Claws |
| 5 | Storm |
| 6 | Ra Blade |
| 7 | Firestaff A |
| 8 | Dagger |
| 9 | Falchion |
| 10 | Sword |

| | |
|---|---|
| 11 | Rapier |
| 12 | Biter |
| 13 | Samurai |
| 14 | Side splitter |
| 15 | Diamond Edge |
| 16 | Vorpal Blade |
| 17 | Dragon Fang |
| 18 | Axe |
| 19 | Executioner |
| 20 | Mace |
| 21 | Mace of Order |

| | |
|---|---|
| 22 | Morningstar |
| 23 | Club |
| 24 | Stone Club |
| 25 | Claw Bow |
| 26 | Crossbow |
| 27 | Arrow |
| 28 | Slayer |
| 29 | Sling |
| 30 | Rock |
| 31 | Poison Dart |
| 32 | Throwing Star |

| | |
|---|---|
| 33 | Stick |
| 34 | Staff |
| 35 | Wand |
| 36 | Teo Wand |
| 37 | Yew Staff |
| 38 | Staff of Irra |
| 39 | Cross of Neta |
| 40 | Serpent staff |
| 41 | Dragon Spit |
| 42 | Sceptre of Lyf |
| 43 | Horn of fear |
| 44 | Speedbow |
| 45 | Firestaff B |

Table 2.9: Weapon Types

**Clothing** CLOTHINGTYPE +60000

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | Cape | 15 | ElvenBoots | 30 | WoodenShield | 45 | PoleynOfRa |
| 1 | CloakOfNight | 16 | LeatherJerkin | 31 | SmallShield | 46 | GreaveOfRa |
| 2 | TatteredPants | 17 | LeatherPants | 32 | MailAketon | 47 | ShieldOfRa |
| 3 | Sandals | 18 | SuedeBoots | 33 | LegMail | 48 | DragonHelm |
| 4 | LeatherBoots | 19 | BluePants | 34 | MithralAketon | 49 | DragonPlate |
| 5 | TatteredShirt | 20 | Tunic | 35 | MithralMail | 50 | DragonPoleyn |
| 6 | Robe | 21 | Ghi | 36 | CasqueNCoif | 51 | DragonGreave |
| 7 | FineRobeA | 22 | GhiTrousers | 37 | Hosen | 52 | DragonShield |
| 8 | FineRobeB | 23 | Calista | 38 | Armet | 53 | Dexhelm |
| 9 | Kirtle | 24 | CrownOfNerra | 39 | TorsoPlate | 54 | Flamebain |
| 10 | SilkShirt | 25 | BezerkerHelm | 40 | LegPlate | 55 | PowerTowers |
| 11 | Tabard | 26 | Helmet | 41 | FootPlate | 56 | BootsOfSpeed |
| 12 | Gunna | 27 | Basinet | 42 | SarShield | 57 | Halter |
| 13 | ElvenDoublet | 28 | NetaShield | 43 | HelmOfRa | | |
| 14 | ElvenHuke | 29 | CrystalShield | 44 | PlateOfRa | | |

Table 2.10: Clothing Types

**Scroll Types** SCROLLTYPE +70000

**Potion Types** POTIONTYPE +80000 (Strength: +0 to +255) potion_MonPotionA = 0,->0 potion_UmPotion = 1,->256 potion_DesPotion = 2,->512 potion_VenPotion = 3,->768 potion_SarPotion = 4,->1024 potion_ZoPotion = 5,->1280 potion_RosPotion = 6,->1536 potion_KuPotion = 7,->1792 potion_DanePotion = 8,->2048 potion_NetaPotion = 9,->2304 potion_AntiVenin = 10,->2560 potion_MonPotionB = 11,->2816 potion_YaPotion = 12,->3072 potion_EEPotion = 13,->3328 potion_ViPotion = 14,->3584 potion_WaterFlask = 15,->3840 potion_KathBomb = 16,->4096 potion_PewBomb = 17,->5352 potion_RaBomb = 18,->4608 potion_FulBomb = 19,->4864 potion_EmptyFlask = 20,->5120

**Chest Types** CHESTTYPE +90000

**Miscellaneous Types** MISCTYPE + 100 000

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | Compass | 15 | Onyx key | 30 | Corn | 45 | Rope |
| 1 | Waterskin | 16 | Skeleton key | 31 | Bread | 46 | Rabbits foot |
| 2 | Jewel Symal | 17 | Gold key | 32 | Cheese | 47 | Corbum |
| 3 | Illumulet | 18 | Winged key | 33 | Screamer slice | 48 | Choker |
| 4 | Ashes | 19 | Topaz key | 34 | Worm round | 49 | Lockpicks |
| 5 | Hero's Bones[2] | 20 | Sapphire key | 35 | Shank | 50 | Magnifier |
| 6 | Sar coin | 21 | Emerald key | 36 | Dragon steak | 51 | Zo Kath Ra |
| 7 | Silver coin | 22 | Ruby key | 37 | Gem of Ages | 52 | Bones |
| 8 | Gor coin | 23 | Ra key | 38 | Ekkhard Cross | | |
| 9 | Iron key | 24 | Master key | 39 | Moonstone | | |
| 10 | Key of B | 25 | Boulder | 40 | The Hellion | | |
| 11 | Solid key | 26 | Blue gem | 41 | Pendant Feral | | |
| 12 | Square key | 27 | Orange gem | 42 | Magical Box A | | |
| 13 | Tourquoise key | 28 | Green gem | 43 | Magical Box B | | |
| 14 | Cross key | 29 | Apple | 44 | Mirror of Dawn | | |

Table 2.11: Miscellaneous Types

#### 2.6.6.3 Foo Type

CLOUDTYPE +150000 Fireball = 0 Dispel = 3 Poison = 7 Death = 40 Fluxcage = 50,

========== OBJECT TYPES ==========
enum OBJECTTYPE { obj_first = 0, obj_Compass_N = 0, //0 0 obj_FirstModifiableObject = obj_Compass_N, obj_Compass_E, //1 1 obj_Compass_S, //2 2 obj_Compass_W, //3 3 obj_Torch_a, //4 4 obj_Torch_b, //5 5 obj_Torch_c, //6 6 obj_Torch_d, //7 7 obj_Waterskin, //8 8 obj_Water, //9 9 obj_JewelSymal_a, //0a 10 obj_JewelSymal_b, //0b 11 obj_Illumulet_a, //0c 12 obj_Illumulet_b, //0d obj_Flamitt_a, //0e obj_Flamitt_b, //0f obj_EyeOfTime_a, //10 obj_EyeOfTime_b, //11 obj_StormRing_a, //12 obj_StormRing_b, //13 obj_StaffOfClaws_a, //14 20 obj_StaffOfClaws_b, //15 21 obj_StaffOfClaws_c, //16 22 obj_Storm_a, //17 obj_Storm_b, //18 obj_RABlade_a, //19 obj_RABlade_b, //1a obj_TheFirestaff_a, //1b obj_TheFirestaff_b, //1c obj_TheFirestaff_c, //1d obj_OpenScroll //1e obj_Scroll, //1f obj_LastModifiableObject = obj_Scroll, obj_Dagger, //20 32 obj_Falchion, //21 33 obj_Sword, //22 34 obj_Rapier, //23 35 obj_Biter, //24 36 obj_SamuraiSword, //25 37 obj_SideSplitter, //26 38 obj_DiamondEdge, //27 39 obj_VorpalBlade, //28 40 obj_DragonFang, //29 41 obj_Axe, //2a obj_Executioner, //2b obj_Mace, //2c obj_MaceOfOrder, //2d 45 obj_Morningstar, //2e obj_Club, //2f obj_StoneClub, //30 obj_ClawBow, //31 obj_Crossbow, //32 obj_Arrow, //33 obj_Slayer, //34 obj_Sling, //35 obj_Rock, //36 obj_PoisonDart, //37 obj_ThrowingStar, //38 obj_Stick, //39

obj_Staff, //3a 58 obj_Wand, //3b 59 obj_TeoWand, //3c 60 obj_YewStaff, //3d 61
obj_StaffOfIrra, //3e 62 obj_CrossOfNeta, //3f 63 obj_SerpentStaff, //40 64 obj_DragonSpit,
//41 65 obj_SceptreOfLyf, //42 66 obj_TatteredShirt, //43 obj_FineRobe_a, //44
obj_Kirtle, //45 obj_SilkShirt, //46 obj_ElvenDoublet, //47 obj_LeatherJerkin, //48
obj_Tunic, //49 obj_Ghi, //4a obj_MailAketon, //4b obj_MithralAketon, //4c obj_TorsoPlate,
//4d obj_PlateOfRa, //4e obj_DragonPlate, //4f obj_Cape, //50 80 obj_CloakOfNight,
//51 81 obj_TatteredPants, //52 82 obj_Robe, //53 83 obj_FineRobe_b, //54 84
obj_Tabard, //55 85 obj_Gunna, //56 obj_ElvenHuke, //57 obj_LeatherPants, //58
obj_BluePants, //59 obj_GhiTrousers, //5a obj_LegMail, //5b obj_MithralMail, //5c
obj_LegPlate, //5d obj_PoleynOfRa, //5e obj_DragonPoleyn, //5f obj_BezerkerHelm,
//60 obj_Helmet, //61 obj_Basinet, //62 obj_CasqueNCoif, //63 obj_Armet, //64
100 obj_HelmOfRa, //65 101 obj_DragonHelm, //66 102 obj_Calista, //67 103 obj_CrownOfNer
//68 104 obj_NetaShield, //69 obj_CrystalShield, //6a obj_SmallShield, //6b obj_WoodenShiel
//6c obj_SarShield, //6d obj_ShieldOfRa, //6e obj_DragonShield, //6f obj_Sandals,
//70 obj_SuedeBoots, //71 obj_LeatherBoots, //72 obj_Hosen, //73 obj_FootPlate,
//74 obj_GreaveOfRa, //75 obj_DragonGreave, //76 obj_ElvenBoots, //77 obj_GemOfAges,
//78 obj_EkkhardCross, //79 obj_Moonstone, //7a 122 obj_TheHellion, //7b 123
obj_PendantFeral, //7c 124 obj_SarCoin, //7d 125 obj_SilverCoin, //7e 126 obj_GorCoin,
//7f 127 obj_Boulder, //80 128 obj_BlueGem, //81 129 obj_OrangeGem, //82 130
obj_GreenGem, //83 131 obj_MagicalBox_a, //84 132 obj_MagicalBox_b, //85 133
obj_MirrorOfDawn, //86 134 obj_HornOfFear, //87 135 obj_Rope, //88 136 obj_RabbitsFoot,
//89 137 obj_Corbum, //8a 138 obj_Choker, //8b 139 obj_Dexhelm, //8c 140 obj_Flamebain,
//8d 141 obj_Powertowers, //8e 142 obj_Speedbow, //8f obj_Chest, //90 obj_OpenChest,
//91 obj_Ashes, //92 obj_Bones_a, //93 obj_MonPotion_a, //94 obj_FirstFullFlask
= obj_MonPotion_a, obj_UmPotion, //95 obj_DesPotion, //96 obj_VenPotion, //97
obj_SarPotion, //98 obj_ZoPotion, //99 obj_RosPotion, //9a obj_KuPotion, //9b obj_DanePotion
//9c obj_NetaPotion, //9d obj_AntiVenin, //9e obj_MonPotion_b, //9f obj_YaPotion,
//a0 160 obj_EePotion, //a1 obj_ViPotion, //a2 obj_WaterFlask, //a3 obj_LastFullFlask
= obj_WaterFlask, obj_KathBomb, //a4 obj_PewBomb, //a5 obj_RaBomb, //a6 obj_FulBomb,
//a7 obj_Apple, //a8 168 obj_Corn, //a9 169 obj_Bread, //aa 170 obj_Cheese, //ab
171 obj_ScreamerSlice, //ac 172 obj_WormRound, //ad 173 obj_Shank, //ae 174
obj_DragonSteak, //af 175 obj_IronKey, //b0 176 obj_FirstKey = obj_IronKey, obj_KeyOfB,
//b1 177 obj_SolidKey, //b2 178 obj_SquareKey, //b3 179 obj_TourquoiseKey, //b4
obj_CrossKey, //b5 obj_OnyxKey, //b6 obj_SkeletonKey, //b7 obj_GoldKey, //b8
obj_WingedKey, //b9 obj_TopazKey, //ba obj_SapphireKey, //bb obj_EmeraldKey,
//bc obj_RubyKey, //bd obj_RaKey, //be obj_MasterKey, //bf obj_LastKey = obj_MasterKey,
obj_LockPicks, //c0 192 obj_Magnifier, //c1 obj_BootsOfSpeed, //c2 obj_EmptyFlask,
//c3 obj_Halter, //c4 obj_ZokathraSpell, //c5 obj_Bones_b, //c6 obj_Special_a, //c7
obj_Special_b, //c8 obj_Special_c, //c9 obj_Special_d, //ca 202 obj_Special_e, //cb

203 obj_Special_f, //cc 204 obj_Special_g, //cd 205 obj_Special_h, //ce 206 obj_Special_i, //cf 207 obj_Special_j, //d0 208 obj_Special_k, //d1 obj_Special_l, //d2 obj_Special_m, //d3 obj_Special_n, //d4 obj_last = obj_Special_n, obj_NotAnObject = 0xff }

### 2.6.7 Sounds

Two types of sounds are supported....blah, blah

| ID | |
|---|---|
| 0 | no sound |
| 1 | dropped dagger |
| 2 | switch |
| 3 | door |
| 4 | dragon snarl |
| 5 | wall tapping |
| 6 | fireball |
| 7 | party scream |
| 8 | mummy scream |
| 9 | gulp |
| 10 | oof |
| 11 | ugh |

| ID | |
|---|---|
| 12 | snipped sound |
| 13 | ugh |
| 14 | zap |
| 15 | screamer |
| 16 | scorpion |
| 17 | swoosh |
| 18 | teleport |
| 19 | wall ugh |
| 20 | worm roar |
| 21 | explosion |
| 22 | giggler |

Table 2.12: Built-in Sounds

# Chapter 3

# Filters

*CSBWin* allows modifying of parts of built-in game behavior via a filtering mechanism. This is accomplished by setting up the matching "Special Location" within the "Global/Edit" menu. At this special location, the designer sets up a *DSA* to perform any required work.

Unless otherwise noted, when a filter is triggered the corresponding *DSA* is sent a S0 (Set 0) message. A rough outline of processing is as follows:

1. Get the set of parameters with the &PARAM@ (4.21.2) instruction.
2. Perform the desired modification of the parameters.
3. Return the set of parameters with the &PARAM! (4.21.3) instruction.

The remainder of this chapter describes each of the possible filters, as well as another special location *Indirect Actions* (3.16) which may be required by filters to perform game state changes which are disallowed directly within filter execution.

## 3.1 Spell Filters

Allows intercepting and modifying all spells that are cast.

Zyx Notes:
I think those are the current in build msg:
L0: normal action, normal message
L1: no action, no message
L2: this spell is too difficult. study it
L3: the spell fizzles
L4: the spell fizzles and dies
L5: this spell is too difficult.
L6: %caster% needs more practice with this spell.
L7: this spell requires a component.

This filter has 14 parameters:

| | | | |
|---|---|---|---|
| 0 | Action | (3.1) | r/w |
| 1 | Incantation | (3.1) | r |
| 2 | *charID* | (2.6.5.1) | r |
| 3 | Disable time | | ?? |
| 4 | Missile Type | | |
| 5 | Party Location in Dungeon | | r |
| 6 | Party facing direction | | r |
| 7 | Skill Required | | |
| 8 | Spell Byte 5 | | |
| 9 | Spell Class | | |
| 10 | Unused 1 | | |
| 11 | Unused 2 | | |
| 12 | Unused 3 | | |
| 13 | Unused 4 | | |

Table 3.1: Spell Filter Parameters

**Action** This is the action to be taken when the DSA exits. It is initially set to zero. The possible actions are:

0 – Proceed to cast the spell as usual using the Spell Parameters. If you do not modify the spell's parameters then this will cause the spell to be cast just as though there were no filtering in effect. If you have modified the parameters then the spell will be cast using those modified parameters.

1 – Cancel the spell. Print no message. The spell just quietly fizzles.

2 through 100 – Cancel the spell and print a message. See the list of possible built-in messages later in this discussion. TODO – add table of values.

**Incantation** The incantation is specified by a 4-digit decimal number. The four digits are the four runes that are selected in the casting. If fewer than four are used then the unused ones are zero. A ZO-Spell might be 1600 or 2600 or 3600 or 4600 or 5600 or 6600 depending on the 'power'. A fireball could be 1440 or 2440 or 3440 or 4440 or 5440 or 6440 depending on the 'power'. That should be enough examples to get the idea across. A spell of 1700 would be nonsense because there is no seventh rune.

TODO – add a table of the runes and values

## 3.2 Skill Adjust Filter

There is a function in the runtime engine named *AdjustSkills*. You can see a diagram of it (RECREATE OR ADD LINK). The *Adjust Skills Filter* is activated before the first code in that function. The Filter receives the following parameters :

| | | |
|---|---|---|
| 0 | *charID* | |
| 1 | Skill Number | (2.6.5.3) |
| 2 | XP gain (positive) | |
| 3 | *Reason* | (see below) |
| 4-8 | for designer usage | |

Table 3.2: Skill Adjust Filter Parameters

The values of parameters 4-8 are assigned by the designer with &SETADJUSTSKILLSPARAM (4.19.9.1) instruction.

Where *Reason* has one of the following values:

| | |
|---|---|
| 0 | Unknown |
| 1 | Physical Attack |
| 2 | WarCry, etc |
| 3 | Attack |
| 4 | Cast Spell 1 |
| 5 | Cast Spell 2 |
| 6 | Monster damages character |
| 7 | Skill Increaser 1 |
| 8 | Skill Increaser 2 |
| 9 | Throw by character |

Table 3.3: Skill Adjust Reasons

The filter can change any of the parameters and the changed parameters will be used by the function *AdjustSkills*. Only the first three (character index, skill number, and experience to be applied) will have any affect on the assignment of skills. The remaining parameters are simply for the filter to use to orient itself and make decisions.

## 3.3 Party Attack Filter

This filter will be called whenever the party invokes any of the attack options such as SHOOT, BASH, FLIP, etc.

The *DSA* associated with this filter will (usually) be activated three times per *action*:

| message | when triggered | |
|---|---|---|
| S0 | Pre-Attack | |
| C0 | Attack | |
| T0 | Post-Attack | |

Table 3.4: Party Attack Filter Messages

An overview of the sequence is as follows:

- All parameters are set to zero.
- Computes the values of parameters 0 through 10.

- PreAttack - A S0 message is sent to the DSA. The parameters that will be used to determine the results and effectiveness of the attack are recorded in the parameters. The DSA can modify the attack type, abort the attack, change some parameters, etc.
- If the filter sets *attackType* to a negative value in the previous step, all of the remain steps are skipped.
- Attack - A C0 message is sent to the DSA just prior to the attack. Again, the DSA can examine and modify various parameters. Many of the attack types have extra parameters that are defined only for that one attack type.
- PostAttack - A T0 message is sent to the DSA - After the attack is completed, the DSA will be called again and given an opportunity to modify the character's disable time, decrease in stamina, and the experience gained by the attack. Also, at this time we will *activate* the monster that was attacked. Normally the monster would have to wait for a *Movement Timer* to expire. But if the parameter *activateMonster* is non-zero then the monster is give a chance to move immediately. Lord Chaos might attempt to escape from a Fusion attack, for example.
- Adjust character stats and active monster if *activateMonster* is has been set to a non-zero value.

All of the variables used by the attack code are available to the DSA in the parameter area. Additionally, ten variables in the parameter area that are not used by the attack code will be set to zero before the first call to the DSA, will never be changed by the attack code, and are available to the DSA to *remember* things between calls. It is guaranteed that each of the three calls will occur once even in cases like FLIP or CLIMBDOWN where no monster is involved and no damage is done, unless aborted as mentioned above.

| | |
|---|---|
| 0 | |
| 1 | War Cry |
| 2 | Physical |
| 3 | Spell |
| 4 | Hit Door |
| 5 | Shoot |
| 6 | Flip |
| 7 | Shield |
| 8 | Flux Cage |
| 9 | Fusion |

| | |
|---|---|
| 10 | Heal |
| 11 | Window |
| 12 | Climb Down |
| 13 | Freeze Life |
| 14 | Light |
| 15 | Throw |
| 16 | Default |

Table 3.5: Attack Data Type

Official Parameter list ******** PRELIMINARY *********

struct SHIELD // dataType = ADT_Shield { // atk_SPELLSHIELD // atk_FIRESHIELD i32 mustHaveMana; i32 strength; };

struct FLIP // dataType = ADT_Flip { // atk_FLIP i32 heads; };

struct SHOOT //dataType = ADT_Shoot { // atk_SHOOT i32 success; i32 range; i32 damage; i32 decayRate; };

struct THROW { i32 side; // ReadOnly - 0=left, 1=right i32 abort; // default=0 - non-zero to abort action i32 facing; // ReadOnly - N,E,S,W i32 range; // WriteOnly i32 damage; // WriteOnly i32 decayRate// WriteOnly };

struct HITDOOR //dataType = ADT_HitDoor { // atk_BASH: // atk_HACK: // atk_BERZERK: // atk_KICK: // atk_SWING: // atk_CHOP: i32 strength; };

struct WARCRYETC //dataType = ADT_WarCry // atk_CONFUSE: // atk_WARCRY: // atk_CALM: // atk_BRANDISH: // atk_BLOWHORN: { i32 mastery; i32 skillIncrement; i32 effectiveMastery; i32 requiredMastery; };

struct PHYSICALATTACK //dataType = ADT_Physical { // atk_BASH: // atk_HACK: // atk_BERZERK: // atk_KICK: // atk_SWING: // atk_CHOP: // atk_DISRUPT: // atk_JAB: // atk_PARRY: // atk_STAB2: // atk_STAB1: // atk_STUN: // atk_THRUST: // atk_MELEE: // atk_SLASH: // atk_CLEAVE: // atk_PUNCH: i32 monsterDamage; i32 staminaAdjust; i32 skillAdjust; i32 attackedMonsterOrdinal; };

struct SPELLATTACK //dataType = ADT_Spell { // atk_LIGHTNING: // atk_DISPELL: // atk_FIREBALL: // atk_SPIT: // atk_INVOKE i32 spellRange; i32 spellType; i32 decrementCharges; // if non-zero (default = 1) };

struct HEAL { // atk_HEAL i32 HPIncrement; };

struct FREEZELIFE { i32 oldTime; i32 deltaTime; };

struct LIGHT { i32 deltaLight; i32 decayRate; i32 time; };

union ATTDEP { WARCRYETC warcryetc; PHYSICALATTACK physicalAttack; SPELLATTACK spellAttack; HITDOOR hitDoor; SHOOT shoot; FLIP flip; SHIELD shield; HEAL heal; FREEZELIFE freezeLife; LIGHT light; THROW throw; };

struct ATTACKPARAMETERS { i32 charIdx; i32 attackType; i32 attackX; i32 attackY; i32 monsterUnderAttack; //0 if none i32 monsterType; //or mon_undefined (=99) if none i32 skillNumber; i32 staminaCost; i32 experienceGained; i32 disableTime; i32 neededMana; i32 unused; //damageToMonster; i32 decrementCharges; i32 activateMonster; i32 userInfo[10]; i32 dataType; // = ADT_***** ATTDEP attdep; };

The Parameters that are computed prior to the PreAttack call to the Filter Character Index of atacking character Attack Type ( See Attack Types ) AttackX AttackY ID of Monster Under Attack Type of Monster Under Attack ( See Monster Types ) Skill Number Stamina Cost Experience Gained (Many attacks modify this) Disable Time (Many attacks modify this). Needed Mana

Parameters that are computed prior to Attack call to the Filter Damage to monster Decrement Charges flag.

Parameters for use by DSA 14 - 0 15 - 0 16 - 0 17 - 0 18 - 0 19 - 0 20 - 0 21 - 0 22 - 0 23 - 0

## 3.4 Feeding Filter

This filter is called when a hero attempts to consume an item.

The sequence is as follows:

1. The parameters are cleared.
2. The standard feeding process is executed, except no changes are made to either the character or the item. Instead, the changes that would have normally occured are placed in the parameters.
3. The filter is executed.
4. The parameters are applied to the character and to the object.

| 0 | perform | b | 1 |
|---|---|---|---|
| 1 | charID | ur | |
| 2 | object number | ur | |
| 3 | object type | ur | |
| 4 | *objectID* | ur | |
| 5 | Food amount | s | 2 |
| 6 | Water amount | s | 2 |
| 7 | potion type | ur | 3 |
| 8 | potion strength | ur | |
| 9 | Strength adj | s | |
| 10 | Dexterity adj | s | |
| 11 | Wisdom adj | s | |
| 12 | Vitality adj | s | |
| 13 | Antiven | b | |

| 14 | | s | |
|---|---|---|---|
| 15 | | u | |
| 16 | | u | |
| 17 | | s | |
| 18 | | s | |
| 19 | | u | |
| 20 | | b | |
| 21 | | b | |
| 22 | | b | |
| 23 | | s | |
| 24 | | ur | |
| 25 | | s | |
| 26 | | s | |

Table 3.6: A

Parameters: "s" = signed; "u" = unsigned; "b" = boolean; "r" = readonly

1. Set to *false* to do abort the action and do absolutely nothing.
2. Amounts to increase the hero's food and water.
3. Value is -1 if not a potion. ADD LINK

0 - b - perform feeding; Set false to abort the feeding and do absolutely nothing.

1 - ur - *charID*;

2 - ur - object database number; 10=misc; 8=potion;

3 - ur - object type; See Object Type Numbers.

4 - ur - object ID.

5 - s - foodValue; Food value to be added.

6 - s - waterValue; Water value to be added.

7 - ur - potion type; See Potion Type Numbers. (-1 if not potion)

8 - ur - potion strength

9 - s - strength adjustment;

10 - s - dexterity adjustment

11 - s - wisdom adjustment

12 - s - vitality adjustment

13 - b - antiVenin

14 - s - stamina adjustment

15 - u - shield strength adjustment

16 - u - shield duration

17 - s - mana adjustment

18 - s - hitpoint adjustment

19 - u - heal count

20 - b - empty flask. Remove Potion from Flask or water from Waterskin.

21 - b - empty hand. Remove the object from hand and delete from dungeon.

22 - b - chew

23 - s - swallow sound. Usually 8. Set to -1 to make no sound.

24 - ur - Party location

25 - s - antiMagic Adjust

26 - s - antiFireAdjust

Parameter #19 is a bit strange. A healing potion causes 'ouches' to disappear. The code clears random 'ouches' locations. It will try this several times until at least one 'ouch' is removed or until this count is reached. So if you set this count to zero then no 'ouches' will be removed. Setting it to one give a small chance that 'ouches' will be removed. Setting it to 10 will make the probability almost certain.

Parameter #20 controls whether a waterskin has water removed from it and whether a potion is converted to an empty flask. This is initially *true* for Potions or non-emtpy Waterskins. If you set it *false* (zero) then the waterkin or potion can be used again.

Parameter #21 controls whether the object will disappear. This in normally *true* for food items and *false* for potions and waterskin.

## 3.5 Character Death Filter

Single parameter, which contains the *charID* (2.6.5.1)of the dying character.

## 3.6 Viewing Filter

This filter is called when the player clicks on the hero's eye in the inventory screen.

There are three read-only (*PhraseMask* may be implictly modified as noted below):

I ASSUME THE READ-ONLY STATMENT IS CORRECT.

| | | |
|---|---|---|
| 0 | *PhraseMask* | |
| 1 | *objectID* | |
| 2 | *charID* | 2.6.5.1 |

Table 3.7: Viewing Filter Parameters

The bits in *PhraseMask* specifies the set of text to be displayed (see below).

The parameter *objectID* specifies the object being examined:

- If the value is negative 1 (0xFFFFFFFF), then no object is being viewed. Instead, the character's current stats are about to be displayed. In this case *PhraseMask* is set to zero and the filter is only being notified of the event. It can do nothing to modify what is displayed.
- If the object is a scroll then the *PhraseMask* is set to zero and the filter can change the *objectID* to be a different scroll, so that perhaps different members of the party see different words on the scroll.

The parameter *objectID* specifies the character performing the action.

There are a total of eight phrases that can be displayed:

The first six are placed in parentheses and are initially set to "CONSUMABLE", "POISONED", "BROKEN", "CURSED", "", and "".

The last two are placed on separate lines following the parenthesised list. The first of these two is generally initialized to the "WEIGHT x.x KG", and the second of these two is initialized to the empty string. Each of the eight phrases is printed only if the *PhraseMask* has the corresponding bit set. Bit zero for the first phrase, etc. This mask can be manipulated with the &DESCRIBE operation. The PhraseMask is initialized to print those phrases that apply to the object.

For example, an Apple is 'CONSUMABLE' so bit zero would be set.

Summary:

| | Initial Value | |
|---|---|---|
| 0 | CONSUMABLE | 1 |
| 1 | POINSONED | 1 |
| 2 | BROKEN | 1 |
| 3 | CURSED | 1 |
| 4 | | 1,2 |
| 5 | | 1,2 |
| 6 | | 2,3 |
| 7 | | 2,3 |

Table 3.8: Object Description Text

1. Parentheses are placed around these six when displayed.
2. xx
3. On a separate line

Manipulating the *PhraseMask* and the associated text is acompished with the instruction &DESCRIBE (4.20.3.6).

## 3.7 Cursor Filter

This filter is activated whenever the contents of the cursor is changed. The cursor represents the hand of the lead character and is used to manipulate items.

There are six parameters, the first indicates the type of the event and the remainder are type specific. The following indicates a common usage:

| | | |
|---|---|---|
| 0 | type | |
| 1 | level | |
| 2 | x | |
| 3 | y | |
| 4 | position | |
| 5 | ? | |

Table 3.9: Cursor Filter Parameters

The *objectID* of the object being put into the cursor or taken from the cursor A code telling what sort of transaction is taking place (see enum CURSORFILTER_TYPE below) P1 P2 P3 P4

| Type | | | |
|---|---|---|---|
| 0 | unknown | N | P1-P4 = 0 |
| 1 | read game | N | P1-P4 = 0 |
| 2 | pick from floor | Y | |
| 3 | place on character | Y | |
| 4 | pick from character | Y | |
| 5 | throw | Y | |
| 6 | entering prison | N | P1-P4 = 0 |
| 7 | drop object | Y | |
| 8 | eat | N? | |
| 9 | resume saved game | N? | |
| 10 | DSA DEL | N | |
| 11 | DSA ADD | N | |
| 12 | take from sconce | Y | (3.7) |
| 13 | place in sconce | Y | (3.7) |
| 14 | swap remove | | |
| 15 | swap replace | | |
| 16 | gift from god | | |
| 17 | take key | | |
| 18 | DSA MoveFrom | | |
| 19 | DSA MoveTO | | |
| 20 | CANCEL | | used in filter to cancel |

Table 3.10: Cursor Filter Types

**Pick From Floor**

**“Place on” and “Pick from” Character**

**“Take from” and “Place in” Character**

## 3.8 Attack Option Name Filter

This is called whenever the name of an action is about to be displayed.

The filter is called one to three times, once for each of the possible actions of a given item.

The parameters are:

| | | | |
|---|---|---|---|
| 0 | *charID* | 2.6.5.1 | |
| 1 | *ActionIndex* | (0,1,2) for three possible | |
| 2 | *ActionName* | initialized to -1 | |
| 3 | Number of valid attack options. | | |
| 4 | *AttackType* | | |

Table 3.11: Attack Option Name Parameters

- *charID* specifies the index of the XXX
- *ActionIndex* specifies the (0,1,2)
- *ActionName* may be modified by the filter. If it has a value on the range of 0 to 65535, then xx
- XXX
- *AttackType*

The filter may replace *ActionName* with another integer. After the DSA executes, this integer is examined. If it is an integer between 0 and 65535 then it is used to fetch the associated *Global Text Variable* (add LINK), which is displayed as the action's name. Otherwise the default action name is displayed.

Additionally it's important to note that *Attack Type* (parameter 4) can be changed to values other than those in-built. The value of this parameter is transmitted to the *Party Attack Filter* (3.3), thus allowing the designer to add custom actions (with custom names and effects) through the use of these two filters.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | – | 11 | Freezelife | 22 | Confuse | 33 | Spellshield |
| 1 | Block | 12 | Hit | 23 | Lightning | 34 | Fireshield |
| 2 | Chop | 13 | Swing | 24 | Disrupt | 35 | Fluxcage |
| 3 | – | 14 | Stab2 | 25 | Melee | 36 | Heal |
| 4 | Blowhorn | 15 | Tthrust | 26 | – | 37 | Calm |
| 5 | Flip | 16 | Jab | 27 | Invoke | 38 | Light |
| 6 | Punch | 17 | Parry | 28 | Slash | 39 | Window |
| 7 | Kick | 18 | Hack | 29 | Cleave | 40 | Spit |
| 8 | Warcry | 19 | Berzerk | 30 | Bash | 41 | Brandish |
| 9 | stab1 | 20 | Fireball | 31 | Stun | 42 | Throw |
| 10 | Climbdown | 21 | Dispell | 32 | Shoot | 43 | Fuse |

Table 3.12: Built-in Attack Types

## 3.9 Equip Filter

When an two items are swapped, is the order insured to be: remove followed by add?

This filter is triggered whenever an object is added to or removed from any of the 30 locations on a Character's body, quiver, backpack, or whatever, the DSA at that location will be called. This feature might be used, for example, to modify a Character's strength whenever a Corbum is added to his possessions or to warn him of the consequences of carrying the object. The DSA will be called:

| message | when triggered | |
|---|---|---|
| S0 | Object is added | |
| C0 | Object is removed | |

Table 3.13: Equip Filter Messages

NOTE: that the filter does NOT get called when objects are placed in or removed from a chest.

There are four parameters provided to the Filter:

| | | | |
|---|---|---|---|
| 0 | *charID* | character effected | r |
| 1 | *equipPos* | LINK TO TABLE | r |
| 2 | *objectID* | object add/removed | r |
| 3 | flags | always zero | r |

Table 3.14: Equip Filter Parameters

## 3.10 Party Move Filter

The party move filter is entered with the following structure as the parameters. In general, you can inhibit the movement by setting a bit in the flags parameter.

enum PARTYMOVE_CONSTANTS { PM_BEGINTURN = 1, PM_STAIRWAY = 2, PM_ATTEMPTMOV
= 3,

PM_INHIBITMOVE = 0x0001, PM_SETDELAY = 0x0002, PM_ADDDELAY = 0x0004, };

struct PARTYMOVEDATA { ui32 moveType; // PM_BEGINTURN // The party is about to turn. // 'fromLocationType' is valid. // if locationType = 3 = stairwell then if you do not // inhibit the movement the filter will be called // again when the party is about to traverse the // stairwell. // 'toLocationType' is valid. // 'fromLocaation' is valid. // 'toLocation' is valid unless locationType is stairwell. // 'direction' is 0, 1, 2, 3 for right, left, up, down. // flags is 0. // Set flag PM_INHIBITMOVE to cancel the turning movement // PM_STAIRWAY // The party is about to traverse a stairway. // 'relDirection' = 'absDirection' = 0 for down, 1 for up // 'flags' = 0 // 'fromLocation' is valid // 'toLocation' is valid // 'fromLocationType' is valid // 'toLocationType' is valid and is equal to 3. // Set flag PM_INHIBITMOVE to cancel the movement // PM_ATTEMPTMOVE // The party is about to attempt a move forward, backward, slide left, slide right // 'relDirection' 0, 1, 2, or 3 for forward, right, backward, left // 'absDirection is 0, 1, 2, or 3 for north, east, south, west // 'flags' is zero // 'fromLocation' is valid // 'toLocation' is valid // 'fromLocationType' is valid // 'toLocationType is valid // 'staminaAdjustments are valid and you can change them. They will // not be applied if the move is inhibited by PM_INHIBITMOVE. // You can set delay to be the delay before the party can move again. // If you set PM_SETDELAY then this value will be used whether or not // the party moves. They may be inhibited by a monster, for example. // If you set PM_INHIBITMOVE then the delay will be ignored. // If you set PM_ADDDELAY then this value will be added to the computed // delay only if the party actual moves. // Set flag PM_INHIBITMOVE to cancel the movement. ui32 flags; ui32 delay; ui32 staminaAdjustment[4]; // decrement to stamina; -1 if

character non-existent ui32 relDirection; // ui32 absDirection; ui32 fromLocation; // location with pos = facing. ui32 toLocation; // location with pos = facing. ui32 fromLocationType; // cellType (or roomType) 0=stone, etc. ui32 toLocationType; };

## 3.11 Monster Attack Filter

Allows modifying the standard actions when a monster is about to attack the party.

This filter has 20 parameters:

| | | | |
|---|---|---|---|
| 0 | objectID | | |
| 1 | type | | |
| 2 | index in group | 0-3 | |
| 3 | level | | |
| 4 | position-x | | |
| 5 | position-y | | |
| 6 | cell position | | |
| 7 | missile origin position | | |
| 8 | missile range | | |
| 9 | missile damage | | |
| 10 | missile friction | | |
| 11 | direction (monster to party) | (2.3) | |
| 12 | distance to party | (2.3) | |
| 13 | | | |
| 14 | | | |
| 15 | shouldSteal | | |
| 16 | | | |
| 17 | | | |
| 18 | | | |
| 19 | | | |

Table 3.15: Monster Attack Parameters

[0] Monster ID. This is the "Indirect Pointer Index" of the monster group that is attacking the party. It is a unique identifier among all objects in the dungeon. It can only be reused if the group of monsters is killed. Then it can be reused for any type of object that is created during gameplay.

[1] Monster Type. Giggler, Screamer, etc. See Monster Types.

[2] Monster Index. The index of the particular monster within the group of monsters that is carrying out the attack. 0-3.

[3] Monster Level. This is the dungeon level of the monster.

[4] Monster X. This is the X-coordinate of the monster.

[5] Monster Y. This is the Y-coordinate of the monster.

[6] Monster Position. This is the position of the monster within the cell. 0=NW, 1=NE, 2= SE, 3= SW.

[7] Missile Origin Position. The position within the cell at which any missile will be launched. This is not the same as the Monster Position because if the monster is standing in the rear of the group (relative to the party) then the missile would hit his own companion in front. So the Missile Origin Position is moved to the front of the group (relative to the party).

[8] Missile Range. This is computed by the following sequence of operations: Range = Monster Descriptor Byte8[4] / 4 + 1 Range = Range + Random(Range) Range = Range + Random(Range)

[9] Missile Damage. This is set to Monster Descriptor Byte8[4].

[10] Missile Friction. This is the amount subtracted from Missile Range and Missile Damage each time the missile moves. Standard value is 8;

[13] Missile Type. An integer specifying whether Fireball, PoisonCloud, etc. See Missile Types. This is computed from the Monster Type as follows: Vexirk or Lord Chaos - Random (Fireball 50%) (Dispell, Lightning, Zo, Poisoncloud each 12.5%) Slime Devil - Poison Flying Eye - Random (Lightning 87.5%) (Zo 12.5%) Zytaz - Random (Fireball, Poisoncloud each 50%) Demon or Dragon - Fireball Anything else - No Missile

[14] Monster Should Launch Missile - Non-zero if the monster should attack with a missile spell. Compute with the following sequential steps: Set to 0. Set to 1 if Distance To Party is greater than 1. Set to 1 with 50% probability. Set to 0 if bits 12-15 of word 14 of the Monster Descriptor are less than 2.

[15] Monster Should Steal - Non-zero if the monster should try to steal an item from the party. Set to 1 if the monster is a Giggler.

[16] Index of hero to damage. 0 to 3. This index is computed in one of two ways depending on the Monster Descriptor word 2 bit 4. If word 2 bit 4 is non-zero then the index is the index of a random, live character. If word 2 bit 4 is zero then this is the index of the hero closest to the monster. Closest means closest front-to-back (relative

to the monster). If two heroes are equally close front-to-back then choose the one of the two closest left-to-right.

[17] The ordinal of the Attacking Sound. Zero means silence.

[18] Disable Time - Set to -1 to mean unused. Perhaps someday we can do this.

[19] SupressPoison Set to -1 initially. Set this to +1 to suppress poisoning.

The Attack - or - How the Parameters are used.

The sound of the attack is queued. Then we chose the Type-of-Attack and carry it out. There are three different Types-of-Attack:

1. Missile
2. Theft
3. Physical attack

We will describe how the Type-of-Attack is selected and then how each type is carried out.

**Selecting the Type-of-Attack** First we look to see if the monster should launch a missile (parameter Monster Should Launch Missile is non-zero). If so, the monster attempts to launch the missile and we are done. Note that if the Monster Should Launch Missile then the parameter named Monster Should Steal is ignored. Note that if the Monster Should Launch Missile but the Missile Type is NONE then nothing is launched and the attack totally fizzles (you can completely abort an attack this way).

Second we look to see if the monster should steal an item from the party (parameter Monster Should Steal is non-zero). If so, the monster attempts to steal an item and, whether or not it is successful, the attack is complete.

Lastly, the monster attempts to inflict physical damage.

**Launching a Missile** If the Missile Type is NONE (or illegal) then no missile is launched and the attack is complete. Otherwise we launch the missile specified by parameter Missile Type from the location indicated by parameters Monster X, monster Y at Missile Origin Position. The missile will travel in the direction indicated by parameter Direction to Party. Its range and damage will be set to parameters Missile Range and Missile Damage and its friction will be set to the parameter Missile Friction.

**Stealing an Item** The monster will attempt to steal an item from the hero designated by the parameter Index of Hero to Damage. The theft may or may not be successful, but in either case the attack is complete. Someday we may use one of the unused parameters to specify which item or kind of item should be stolen.

**Inflicting Physical Damage** The standard mechanisms are used to cause damage to the hero specified by Index of Hero to Damage. The attack may or may not be successful. In any case, the attack is complete.

## 3.12 Monster Movement Filter

This filter is called twice whenever a monster is making some sort of move or an attack.

| message | when triggered | |
|---|---|---|
| S0 | About to move or attack | |
| C0 | Action is complete | |

Table 3.16: Monster Moving Filter Messages

Because DSAs are rather CPU hungry it is best to select the options that only activate the DSA if the monster is on the same level as the party and is close by. It is possible to specify a different DSA to be used on each level of the dungeon. If no DSA is specified for a level then the 'Global' one is used.

The DSA receives parameters that provide information about the monster, its position, and the party's position. The DSA can then affect how the monster moves by using the &MONBLK (Monster Block) operation. This instruction has no effect whatsoever outside of a *Monster Movement Filter* and it only applies to the one monster that is about to move.

Here are the parameters:

| | | | |
|---|---|---|---|
| 0 | Monster's dungeon level | | |
| 1 | Monster's X location (relative to level's x offset) | | |
| 2 | Monster's Y location (relative) | | |
| 3 | Object ID of monster | | |
| 4 | Party's dungeon level | | |
| 5 | Party's X location (relative) | | |
| 6 | Party's Y location (relative) | | |
| 7 | Clear 0 message only - Action flags 0-31 | | |
| 8 | Clear 0 message only - Action flags 32-63 | | |

Table 3.17: Monster Moving Parameters

Note that the Monster's level-, x-, and y-coordinates may be different on the two calls to the Filter. If the monster moves to a different cell then its new position will be indicated. If the monster is removed from the dungeon then its level will be -1. In either of these two cases, the flag MonMove_differentCell will be set.

The DSA is activated with a 'Set 0' whenever the monster is about to do something. There is no way to tell what it is up to. It may turn, move, attack, or flee. The code is impenetrable, at least to me. During the processing of the code we set 'Action Flags' to provide some information about what the code is doing and, therefore, what action the monster has taken.

enum MONSTEREVENTS { MDF6TI_turnMonsterTowardParty = 0, TMAG_tunrMonsterGroup = 1, TMAG_turnMonsterGroup = 2, PIn_turnMonsterGroup = 3, PIn_moveTwoSquaresSucceeded = 4, PIn_moveTwoSquaresFailed = 5, SF_greaterThanSmellingDistance = 6, SF_return = 7, SF_DeleteTimersMaybeSetFear = 8, SF_timeFuncMinusThree = 9, SF_TurnMonstersAsGrou = 10, SF_IncrementTime = 11, PIAF_processInvincible = 13, PIAF_standardFinish = 14, T524_possibleMove = 15, T524_moveSucceeded = 16, T524_moveFailed = 17, T524_processInvincibleAndFinish = 18, T524_mabeDeleteTimersFear6TurnIndividuals = 19, T524_ProcessInvincibleAndFinish = 20, IC29to41_atLeastOneMemberAlreadyDead = 21, IC29to41_damageMonsterSucceeded = 22, IC29to41_atLeastOneMemberAlreadDead = 23, IC29to41_damageMonsterFailed = 24, IC29to41_randomMoveSucceeded = 25, IC29to41_randomMoveFailed = 26, IC29to41_doNothing = 27, IC29to41_exitFalse = 28, TT29to41_TT31 = 29, TT29to41_fear5or6 = 30, TT29to41_TT30 = 31, TT29to41_fear5or6doN = 32, TT29to41_TT30turnAsGroup = 33, TT29to41_TT30deleteTimersMaybeSetFear = 34, TT29to41_TT30IncrementTimeByW52PlusRandom = 35, TT29to41_TT30TryDirectionsD5toI = 36, TT29to41_TT29 = 37, TT29to41_TTmonsterA3 = 38, TT29to41_blocked = 39, TT29to41_fearNot5or6 = 40, TT29to41_MaybeDeleteTimersFear6TurnIndividuals = 41, TT29to41_fear0or3 = 42, TT29to41_fear6 = 43, TT29to41_w30w32GretaterThan3 =

44, TT29to41_TTnot37 = 45, TT29to41_fear5 = 46, TT29to41_attacking = 47, TT29to41_notAttac = 48, IC29to41_Not32_33_37_38 = 49, MoveObject_NotAllowedOnLevel = 50, MonMove_differentCell = 51, };

## 3.13 Monster Delete Filter

This filter is triggered when a monster is being deleted from the dungeon. The parameters are:

| | | | |
|---|---|---|---|
| 0 | monsterID | | r |
| 1 | level | | r |
| 2 | x | | r |
| 3 | y | | r |
| 4 | type | | r |
| 5 | drop residue | (3.13) | rw |
| 6 | death cloud | (3.13) | rw |
| 7 | index | index in group, -1 for all | r |

Table 3.18: Monster Delete Parameters

Where the parameter *type* is:

| | |
|---|---|
| 0 | unknown |
| 1 | fusion |
| 2 | make room |
| 3 | damage |
| 4 | movement 1 |
| 5 | movement 2 |

Table 3.19: Monster Delete Type

**Drop Residue** The pre-defined items (like worm round, screamer slices, etc): 0 = default, 1 = drop and 2 = noDrop.

**Death Cloud**

## 3.14 Sound Filter

This filter is triggered whenever one of the standard internal sounds is internally initiated. It has the following parameters:

| | | | |
|---|---|---|---|
| 0 | Sound Number | | |
| 1 | Volume | ( -1=no sound, 0=low, 1=high ) | |
| 2 | Distance to party squared | | |
| 3 | Sound Source Level | | |
| 4 | Sound Source X ( relative ) | | |
| 5 | Sound Source Y ( relative ) | | |
| 6 | Game Volume setting | | |
| 7 | No Sound flag | | |
| 8 | Distance to party in units of 0.01 | | |

Table 3.20: Sound Filter Parameters

Example: 1 north and 1 west gives distance = 141.

If a Sound Filter is configured to receive sound notifications then the information will be passed to the Filter and the sound itself

** WILL NOT BE SENT TO THE SPEAKERS **.

This means that the Filter MUST re-initiate the sound with an '&SOUND' command using the first two parameters (Don't forget to negate the 'Sound Number'}. Of course, the filter may wish to change the sound or volume or not make the sound at all, depending on whether or not the party has recovered from the Deafness spell cast by that Vexirk on level 5. The sounds produced by the &SOUND command are not passed to the Sound Filter, so there is no fear of infinite recursion.

This behavior is a bit different from the other filters which allow the filter to modify parameters before the action occurs. The built-in sounds of the game do not pass by the Sound Filter on their way to the speakers. These sounds actually pass THROUGH the Sound Filter.

## 3.15 Missile Encounter Filter

This filter is called when a missile event occurs.

A Missile consists of the missile itself and projectile within the missile. For example, a missile might contain an arrow or a FUL BOMB.

The filter receives four parameters:

| | | | |
|---|---|---|---|
| 0 | type | (see below) | r/w |
| 1 | location | *cellPosition* | |
| 2 | missile | *objectID* | |
| 3 | projectile | *objectID* | |

Table 3.21: Missile Encounter Parameters

0 - The type of encounter 1 - Location of the encounter 2 - The object ID of the containing Missile 3 - The object ID of the contained projectile

The parameter *type* indicates the type of the event:

| | meaning | Effect on cancel |
|---|---|---|
| 1 | monster | will miss |
| 2 | door | pass through |
| 3 | solid trick wall | pass through |
| 4 | solid wall (fall to floor) | N/A (fall to floor) |
| 5 | max distance reached (fall to floor) | N/A (fall to floor) |

Table 3.22: Missile Encounter Types

The Filter should set the *type* field to zero and return the parameters in order to cancel the encounter. The effect of doing this varies depending on the original type.

## 3.16 Indirect Actions

Certain instructions are not allowed in *DSA* filters. Generally, these are instructions that could result in dungeon state changes. An example is &MOVE (4.16.10) which causes an object to be deleted from one location and added to another. Conversely there are instructions which may only be called within a particular filter, such as &MONBLK. Each instruction with any such a limitation will noted at the beginning of its description.

Instructions where are not allowed in filters have an *indirect* version which can be used. These *indirect* versions have the same name with the percent character inserted immediately after the initial ampersand (e.g. &%MOVE). These *indirect* versions gather together all the parameters needed to perform the operation and sends them off to another *DSA* using the &MESSAGE instruction. When the filter is completed, the message will be delivered to this *Indirect Action Filter* which can use the instruction &%INDIRECT (4.21.1) to reconstruct the necessary parameters from the message and perform the operation that was initially intended.

The Indirect Action DSA need not use the &%INDIRECT instruction if it chooses to manipulate the parameters themselves. For example, it may want to see if an object that it is about to delete still exists. The parameters have this format:

- The action to perform - The number of stack entries - . . . the stack entries . . . Top of stack comes first - Flag — 0 if no array variables. 1 if array variables. Top of stack is number of variables, next to top is index of first array variable - . . . Array variables . . . - Number of parameters - . . . parameters . . .

The action to perform is from this list: Del = 79 Add = 80 CreateCloud = 81 Cast = 82 TeleportParty= 83 MonsterStore = 84 CharStore = 85 Move = 86 Copy = 87 CellStore = 88 Throw = 89

The message that delivers this information has a delay of 0 and therefore the operation will not be delayed in game time. But it will be delayed in realtime and you need to know that when you say &%MOVE the actual move will not take place until the current DSA is completed. Moreover, multiple *indirect* operations within a single DSA will not necessarily be performed in the same order that the commands were issued. For example, if you do an &%ADD followed by an &%DEL, the delete may actually be performed before the add.

# Chapter 4

# Instruction Reference

## 4.1 Terminology

| | | |
|---|---|---|
| operand | | an input taken from the stack |
| | | |
| | | |

## 4.2 Expressions

*foo* ::= *bar*

"..." literal expression

| | |
|---|---|
| *foo bar* | *foo* concatenated with *bar* |
| *foo* \| *bar* | either a *foo* or *bar* expression |
| [ *foo* ] | zero or one *foo* expressions |
| {*foo*}+ | one or more *foo* expressions |
| {*foo*}* | zero or more *foo* expressions |

## 4.3 Foo

EXPRESSION .....DEFAULT VALUE
*integer* ::={0-9}+
*signedInteger* ::=["+" | "-"] *integer*
*level* ::=*integer* .....(current level)
*position* ::="N" | "E" | "S" | "W" .....("N")
*absLocation* ::=*level* "(" *column* "," *row* ")" [*position*] .....(current cell)
*delay* ::={ *integer* | "X" | "Y" } ..... (0)
*localParameter* ::= "A" | "B"
*dsaParameter* ::= *localParameter* | {"C" - "Z"}

## 4.4 Interpreting Instruction Descriptions

| ... items removed ⇒ ... items added |
|---|

## 4.5 General

### 4.5.1 NOP

| N |
|---|
| ... ⇒ ... |

No operation. Useful as a single instruction sequence which merely modifies the state.

1N Next state is set to '1' (assuming first instruction)

### 4.5.2 Load

| |
|---|
| "L" *value* |
| ... ⇒ ... *value* |

Pushes *value* onto the stack, where:

*value* ::= *integer* | *dsaParameter* | *absLocation*

| | |
|---|---|
| 6L33 | Next state is '6' (assuming first instruction) and push '33' onto the stack |
| L1 | Push '1' onto the stack |
| LA | Push the value of parameter A onto the stack |
| L5(4,3)E | Push " onto the stack (2.6.1) |

$$(((64pos) + level)\,32 + x)\,32 + y \tag{4.1}$$

### 4.5.3 Load DSA Location

| |
|---|
| L$ |
| ... ⇒ ... *cellLocation* |

Pushes the location of the current executing *DSA* instance.

### 4.5.4 Set State

| |
|---|
| &SETNEWSTATE |
| ... $n$ ⇒ ... |

Sets the next state of the instance to *n*.

### 4.5.5 Random Number

&RAND

... $n$ ⇒ ... *result*

Returns a random number between 0 and n-1 inclusive, or formally: $[0, n)$.

As an example: `L6 &RAND` will return one of the following values: 0,1,2,3,4,5

### 4.5.6 Read Time

&TIME@

... ⇒ ... *result*

Returns game clock tick in $\frac{1}{6}$ second increments.

Is the tick game time? (save/loaded with savegame?)

### 4.5.7 Global Information Get

&Global@

... *n* ⇒ ... *value*

Currently returns the party *location* if $n = 1$, and zero for any other value.

### 4.5.8 Override

"O" *overrideType* [*integer*]

... ⇒ ...

*overrideType* ::="P"

The only currently defined *overrideType* is currently "P", which overrides the *position* of the following instruction.

OP3

Zyx – thinks good for M as well...look-up

### 4.5.9 DSA Query

&DSAINFO@
... *objectID* ⇒ ... *dsaType dsaState*, *paramA*, *paramB*

Return information about the specified *DSA* instance.
If not a valid DSA, the *dsaType* is -1.

What are the valid values of *dsaType?*

## 4.6 Message Passing

*messageDelay* ::={ *integer* | "X" | "Y" } .....(default is 0)

- X = parameter A
- Y = parameter B

*messageAction* ::="S" | "C" | "T"
*messageType* ::="N" | *messageAction* ..... (default is "S")

- N = do nothing
- S = set
- C = clear
- T = toggle

*messagePos* ::={0-3}
*messageColumn* ::=*messageAction messagePos* ..... (default is "S0")
*messageTarget* ::="A" | "B" | *absLocation*.....(default is "A")

### 4.6.1 Standard Message

| |
|---|
| "M" [*messageDelay*] *messageType* [*messageTarget*] |
| ... ⇒ ... |

Sends a message, after the specified *delay*, of the given type to *messageTarget*.

M

### 4.6.2 Standard Message (Indirect)

| |
|---|
| "M" [*messageDelay*] *messageType* "*" |
| ... *cellLocation* ⇒ ... |

Sends a message with the target *cellLocation* specified as an operand (i.e. read from the stack).

### 4.6.3 Message

| |
|---|
| &MESSAGE |
| ... *cellLocation*, *messageType*, *num*, *delay* ⇒ ... |

Sends a message with all inputs as operands. Additionally allows passing of integer values along with the message to the target.

The number of these additional values is specified by *num* (up to 29). Before passing the message, these values are set-up via &PARAM! (4.21.3). The receiver fetches them via &PARAM@ (4.21.2).

Zyx Notes:
"The number of these additonal values is specified by num (up to 29). Before passing the message, these values are set-up via &PARAM!" they must be set from index 0 of &PARAM!, so the user must be careful not to erase conflctualy the parameters of the filter if the Message is sent from a filter. Solutions: make a copy of the filter's parameters and restore them after the Message is sent, or directly store the filter's parameters with an index > 0.

## 4.7 Stack Manipulation

### 4.7.1 Drop

&DROP

$\ldots x \Rightarrow \ldots$

Discards the top element of the stack.

### 4.7.2 Drop 2

&2DROP

$\ldots x,\ y \Rightarrow \ldots$

Discards the top two elements of the stack.

### 4.7.3 Swap

&SWAP

$\ldots\ x,\ y \Rightarrow \ldots\ y,\ x$

Swaps (reverses) the order of the top two stack elements.

### 4.7.4 Over

&OVER

$\ldots\ x,\ y \Rightarrow \ldots\ x,\ y,\ x$

Push a copy of *TOS(1)*.

### 4.7.5 Duplicate

&DUP

$\ldots\ x \quad \Rightarrow \quad \ldots\ x,\ x$

Duplicates the top element of the stack.

### 4.7.6 Duplicate 2

&2DUP

$\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x,\ y,\ x,\ y$

Pushes copies of the top two stack elements.

### 4.7.7 Pick

&PICK

$\ldots\ n \quad \Rightarrow \quad \ldots\ TOS\,(n)$

Pushes a copy of the $n^{th}$ stack element.

### 4.7.8 Pick 2

&PICK2

$\ldots\ x,\ y,\ z \quad \Rightarrow \quad \ldots\ x,\ y,\ z,\ x$

Push a copy of *TOS(2)*.

### 4.7.9 Poke

&POKE

$\ldots\ value\, n \quad \Rightarrow \quad \ldots$

Pops *n* and *value*, then replaces $TOS\,(n)$ with *value*: $TOS(n) = value$

### 4.7.10 Roll

&ROLL

$\ldots\ n \quad \Rightarrow \quad \ldots$

Moves the $n^{th}$ stack element to the top, all elements between *n* and *TOS* are shifted down one position.

### 4.7.11 Unroll

&-ROLL

$\ldots\ n \quad \Rightarrow \quad \ldots$

Moves the element at *TOS* to the $n^{th}$ stack element, all elements between *n* and *TOS* are shifted up one position.

### 4.7.12 Rotate

&ROT

$\ldots\ x,\ y,\ z \quad \Rightarrow \quad \ldots\ y,\ z,\ x$

Rotates the top three elements by bring the third to the top and moving the other two down one position.

### 4.7.13 Unrotate

| &-ROT |
| --- |
| $\ldots\ x,\ y,\ z \quad \Rightarrow \quad \ldots\ z,\ y,\ x$ |

Rotates the top three elements.

## 4.8 Array Access

Instructions which directly manipulate the *Array* (2.4.3).

### 4.8.1 Array Get

| &@ |
| --- |
| $\ldots\ n \quad \Rightarrow \quad \ldots$ *array*[*n*] |

Returns the $n^{th}$ array element: pushes *array*[*n*]

### 4.8.2 Array Set

| &! |
| --- |
| $\ldots\ value,\ n \quad \Rightarrow \quad \ldots$ |

Sets the $n^{th}$ array element to *value: array*[*n*] = *value*

## 4.9 Parameter Access

Instructions which directly manipulate *Parameters* (2.4.1).

### 4.9.1 Parameter Set

| |
|---|
| "S" *parameter* |
| ... *value* ⇒ ... |

Sets the specified parameter (2.4.1) to the given *value.*

### 4.9.2 Parameter Get

Pushing the value of a parameter (2.4.1) is accomplished with load instruction (4.5.2).

## 4.10 Global Variables

Instructions which directly manipulate *Global Variables* (2.4.4).

### 4.10.1 Global Get

| |
|---|
| "GV" *n* "@" |
| ... ⇒ ... *value* |

Pushes the value of the $n^{th}$ global variable (*global*[n]) on the stack.

### 4.10.2 Global Set

| |
|---|
| "GV" *n* "!" |
| ... *value* ⇒ ... |

Sets the value of the $n^{th}$ global variable to *value*: $global[n] = value$

## 4.11 Temporary Variables

Instructions which directly manipulate *Temporary Variables* (2.4.5).

### 4.11.1 Variable Get

| "V" *n* "@" |
| --- |
| $\ldots \quad \Rightarrow \quad \ldots\ value$ |

Pushes the value of the $n^{th}$ variable no the stack: PUSH$(tempVar[n])$

### 4.11.2 Variable Set

| "V" *n* "!" |
| --- |
| $\ldots\ value \quad \Rightarrow \quad \ldots$ |

Sets the value of the $n^{th}$ variable to *value*: $tempVar[n] = value$

## 4.12 Binary Operators

*Binary Operators* consume two *operands* from the stack and return (push) one result.

### 4.12.1 Arithmetic

#### 4.12.1.1 Addition

| &+ |
| --- |
| $\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x + y$ |

#### 4.12.1.2 Multiplication

&*

$\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x * y$

#### 4.12.1.3 Division

&/

$\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ \frac{x}{y}$

NOTE: division by zero will cause the game to terminate with an error message box.

#### 4.12.1.4 Remainder

&%

$\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x\%y$

NOTE: division by zero remainder will cause the game to terminate with an error message box.

### 4.12.2 Bitwise

Bitwise

#### 4.12.2.1 Shift Right Arithmetic

&RSHIFT

$\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x>>y$

Performs a *signed* (or *arithmetic*) right shift of $x$ by $y$ bit positions. If $y$ is negative, a left shift is performed by $-y$ positions.

NOTE: For hardware portability the number of bit positions ($y$) is masked by 31.

#### 4.12.2.2 Shift Left

| &SHIFT | | |
|---|---|---|
| $\ldots\ x,\ y$ | $\Rightarrow$ | $\ldots\ x<<y$ |

Performs a left shift of $x$ by $y$ bit positions. If $y$ is negative, a *signed* right shift is performed by $-y$ positions.

NOTE: For hardware portability the number of bit positions ($y$) is masked by 31 (SEE: (4.12.2.1))

#### 4.12.2.3 Bitwise AND

| &AND | | |
|---|---|---|
| $\ldots\ x,\ y$ | $\Rightarrow$ | $\ldots\ x\&y$ |

Performs a bitwise AND on the two operands.

#### 4.12.2.4 Bitwise OR

| &OR | | |
|---|---|---|
| $\ldots\ x,\ y$ | $\Rightarrow$ | $\ldots\ x\|y$ |

Performs a bitwise OR on the two operands.

#### 4.12.2.5 Exclusive OR

| &XOR | | |
|---|---|---|
| $\ldots\ x,\ y$ | $\Rightarrow$ | $\ldots\ x\hat{}y$ |

Performs an XOR on the two operands.

### 4.12.3 Logical

#### 4.12.3.1 Logical AND

&LAND

$\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x\&\&y$

#### 4.12.3.2 Logical OR

&LOR

$\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x||y$

Foo

## 4.13 Unary Operators

Unary operators modify the value at the top of the stack.

### 4.13.1 Negate

&NEG

$\ldots\ x \quad \Rightarrow \quad \ldots\ -x$

### 4.13.2 Logical NOT

&NOT

$\ldots\ x \quad \Rightarrow \quad \ldots\ !x$

### 4.13.3 Decrement

&1-

$\ldots\ x \quad \Rightarrow \quad \ldots\ x-1$

The value at *TOS* is decreased by one.

### 4.13.4 Increment

&1+

$\ldots\ x \quad \Rightarrow \quad \ldots\ x+1$

The value at *TOS* is increased by one.

### 4.13.5 Ones Complement

&COMP

$\ldots\ x \quad \Rightarrow \quad \ldots\ \sim x$

Performs a bitwise not.

### 4.13.6 Bit Count

&BITCOUNT

$\ldots\ x \quad \Rightarrow \quad \ldots$ bitcount($x$)

Returns the number of bits in $x$ which are set (are "1"). Also known as the Hamming weight.

## 4.14 Comparison Operators

*Boolean* results map to $1$ for *true* and $0$ for *false*.

### 4.14.1 Equals

| &= |
|---|
| $\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x{==}y$ |

### 4.14.2 Not Equals

| &!= |
|---|
| $\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x \neq y$ |

### 4.14.3 Signed Less-Than

| &< |
|---|
| $\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x < y$ |

Returns the *signed* comparison.

### 4.14.4 Unsigned Less-Than

| &U< |
|---|
| $\ldots\ x,\ y \quad \Rightarrow \quad \ldots\ x < y$ |

Returns the *unsigned* comparison.

## 4.15 Flow Control

*Flow Control* instructions allow changing location of where instructions are...

When execution is passed by any of these flow-control instructions, any *nextState* specifier at the target location is ignored.

More details on *Flow Control* instructions are provided ... (5.6).

### 4.15.1 Explicit Branch

"G" [*state*] [*column*]

... ⇒ ...

Transfers control to the specified *state* and *column*.

### 4.15.2 Explicit Subroutine

"J" [*state*] [*column*]

... ⇒ ...

Transfers control to the specified *state* and *column*. Upon completion, control returns to the next instruction in the this sequence.

### 4.15.3 Branch

&J*

... *message, state* ⇒ ...

Transfers control to the specified.

### 4.15.4 Call Subroutine

&G*

... *message, state* ⇒ ...

Control is transferred to the specified. Upon completion, control returns to the next instruction in the current sequence.

### 4.15.5 Multi-Target

#### 4.15.5.1 Case

| |
|---|
| "??{" *CASE* "}" |
| ... $n$ $\Rightarrow$ ... |

The value *n* is popped from the stack and compared with values specified by *CASE*.

Where:

*CASE* ::={*caseElement*}+ ..... (one or more *caseElement*)

*caseMatch* ::=*integer*

*caseElement* ::= "(" *caseMatch* "," *state column* ")"

If there is a matching value (*n* equals one of the *caseMatch*), then control is transferred to the specified place, otherwise execution continues with the following instruction.

??{}

SEE: XXX

#### 4.15.5.2 If Then Else

| |
|---|
| "?" *IfElse* |
| ...$n$ $\Rightarrow$ ... |

The value *n* is popped from the stack, if its value is zero it is considered *false* and otherwise it is consider *true*.

*IfElse* ::={*onTrue* | *onFalse* | *onTrueOrFalse*}

*FlowTarget* ::={"J" | "G"} [*state*] [*column*]

*onTrue* ::=*flowTarget*

*onFalse* ::=":" *flowTarget*

*onTrueOrFalse* ::=*flowTarget* ":" *flowTarget*

?JC0:GT1
?:J3T2

SEE: XXX

# 4.16 Object

| | |
|---|---|
| | (4.17.4) |
| | (4.19.8) |
| | |
| | |

## 4.16.1 Charges

Objects blah, blah...

| | |
|---|---|
| Weapon | remaining charges |
| Clothing | remaining charges |
| Potion | strength |
| Miscellaneous | value |

Table 4.1: Object Charges

### 4.16.1.1 Charges Get

&GETCHARGES

... *objectID* ⇒ ... *charges*

Returns the number of *charges* associated with the specified item. Silently returns zero for invalid input.

SEE: 4.1

### 4.16.1.2 Charges Set

&SETCHARGES

... *objectID*, *charges* ⇒ ...

Sets the number of *charges* associated with the specified item. Silently does nothing for invalid input.

SEE: 4.1

## 4.16.2 Broken

### 4.16.2.1 Broken Get

&GETBROKEN
... *objectID* ⇒ ... *flag*

Returns the *broken* flag associated with the specified item. Silently does nothing for invalid input.

### 4.16.2.2 Broken Set

&SETBROKEN
... *objectID*, *flag* ⇒ ...

Sets the *broken* flag associated with the specified item. Silently returns *false* for invalid input.

## 4.16.3 Cursed

### 4.16.3.1 Cursed Get

&GETCURSED
... *objectID* ⇒ ... *flag*

Returns the *cursed* flag associated with the specified item. Silently does nothing for invalid input.

### 4.16.3.2 Cursed Set

&SETCURSED
... *objectID*, *flag* ⇒ ...

Sets the *cursed* flag associated with the specified item. Silently returns *false* for invalid input.

## 4.16.4 Poisoned

### 4.16.4.1 Poisoned Get

&GETPOISONED
... *objectID* ⇒ ... *flag*

Returns the *poisoned* flag associated with the specified item. Silently does nothing for invalid input.

### 4.16.4.2 Poisoned Set

&SETPOISONED
... *objectID*, *flag* ⇒ ...

Sets the *poisoned* flag associated with the specified item. Silently returns *false* for invalid input.

## 4.16.5 Sub Types

### 4.16.5.1 Subtype Get

&GETSUBTYPE
... *objectID* ⇒ ... *subtype*

Returns the *subtype* .

#### 4.16.5.2 Subtype Set

| &SETSUBTYPE |
| --- |
| ... *objectID*, *subtype* ⇒ ... |

Sets the *subtype*

### 4.16.6 Object Type

| &TYPE |
| --- |
| ... *objectID* ⇒ ... *typeOfObject* |

Foo.

### 4.16.7 Fetch

| F |
| --- |
| ... *depth*, *absLocation*, *posMask*, *typeMask* ⇒ ... *objectID* |

- *depth*: The depth of the object in the 'virtual' pile of objects. The 'virtual' pile consists only of those objects that meet the *posMask* and *typeMask* criteria. All other objects are ignored. The object at the top of this 'virtual' pile is at depth zero.
- *location*: The absolute location of the cell to be examined. This includes the offsets for the level.
- *positionMask*2.6.4.1: There are four positions in each cell where objects might be placed. North, East, South, and West for a wall cell. Northwest, Northeast, Southeast, and Southwest for items in an open cell. These directions are respectively numbered 0, 1, 2, and 3. The position mask should have a bit set for each position to be considered when building the 'virtual' pile of objects. Bit zero for position zero and so on. If the *positionMask* is set to the value negative-one ( -1 ) then only the one position as described by the location parameter will be considered.

- *typeMask:* There are several kinds of objects - weapons, potions, chest, etc. See objectTypes The typeMask should have a bit set for each type of object to be considered when building the virtual pile of objects. For example, bit 5 ( 32 ) for weapons objects.
- *objectID* Result - This is the unique object ID for the object found in the virtual pile of objects. If no object exists at the specified depth then the resulting *objectID* will be 0xffff (65535).

"Fetch Object ID", " F(<depth>,", " <absolute location>,", " <position mask>,", " <object type mask> ... <objectID>)", " Discussion:", " If <position mask> == -1 then use position of <absolute location>." " <depth> applies only to objects of the proper type at the" " given positions. Other objects are totally ignored" "—",

### 4.16.8 Object Spawn

&ADD

... *posMask*, *location*, *objectID* ⇒ ...

This instruction may not be called within a filter (Chapter 3).

This instruction duplicates an object and places it at the specified *location* (2.6.2).

### 4.16.9 &OBJECTID

&OBJECTID@

... ⇒ ... *objectID*

Returns the most recently manipulated object by a previous instruction. As the list of supported instruction may be extended in the future, this instruction should always be placed immediately after it for future compatibility.

| | | |
|---|---|---|
| &ADD | | |
| &ADD | | |

- add
- indirect add

This list of instructions may be expanded in future versions, so to insure forward compatibility always use this instruction directly after one of those listed.

This information is not carried over from one DSA execution to another.

### 4.16.10 Object Move

&MOVE

... *SOURCE,* DESTINATION ⇒ ...

This instruction may not be called within a filter (Chapter 3).

Moves an object from one location in the game to another.

The DSA operation '&MOVE' can be used to transfer an object form one location in the game to another. The syntax is:

&MOVE ( <source> <destination> . . . )

Where <source> and <destination> have the same syntax, namely:

<source> | <destination> = ( locationType objMask positionMask location depth )

Therefore, ten parameters must be placed on the stack before attempting an &MOVE operation; five for the source and five for the destination. The use of each of the five parameters is summarized in the following table:

locationType objMask positionMask location/object depth 1 = dungeon cell types of objects NESW (-1 = use location) location of cell depth in list 2 = Cursor N/A N/A N/A N/A 3 = Monster possession types of objects N/A Object ID of Monster depth in list 4 = Character possession N/A character index possession index N/A 5 = Chest types of objects N/A ObjectID of chest depth in list

Some discussion of the various parameter types:

objMask - This is the mask of the object types to be considered for the move operation. Bit numbers are (5=weapon) (6=clothing) (7=scroll) (8=potion) (9=chest) (10=miscellaneous). More than one bit may be set. positionMask - When used to designate a

position within a dungeon cell then bit 0=north, bit1=east, bit2=south, and bit3=west. If more than one bit is set when used as a cell destination then the position of the object will be selected at random from the possibilities allowed. If source posMask is set to negative one then it means that the position associated with the location parameter should be used. More than one bit may be set. If destination posMask has no bits set then the position associated with the location parameter is used. When used as a character index it specifies which character to use; zero is the leftmost character and three is the rightmost character in the character portrait area of the playing screen.

location - this is generally the location within the dungeon where the object can be found. It can be a character index, the ID of a container, or the level, x, y, and position coordinates of a cell in the dungeon. depth - When an object resides in a list such as the list of monster possessions then the depth parameter specifies which of the objects in the list is to be referenced. When searching the list, only objects of the type specified by objMask and objects in the correct positions as specified by the positionMask are considered. It is as if the objects not meeting these criteria were non-existent. For example if a list contained a key, a sword, a potion, and an apple and the objMask specified only miscellaneous objects then depth zero would pick the key and depth one would pick the apple. In the destination parameters, depth must be specified as zero.

Some discussion of the operation: dungeon cell Location is an 18-bit integer; (bits0-4 = Y) (bits5-9 = X) (bits10-15 = level) (bits16-17 = position). You can specify the same location as both the source and destination. If you do, the depth parameter of the destination is interpreted after the object is removed from the dungeon cell. cursor This is the very simplest case. As a source it succeeds if the cursor is full and as a destination it succeeds if the cursor is empty. You should not specify the cursor as both the source and the destination. monster or chest A chest can hold only eight objects and a monster can hold any number. You should not specify the same chest and monster as both the source and destination. character The location parameter specifies the location on the character's body, in his quiver, backpack, etc. You should not specify the same location on the same character as both the source and destination. As a source it succeeds if the location is full and as a destination it succeeds if the location is empty and will accept the object being moved. You cannot put a helmet on a character's feet. Sorry.

If the &MOVE fails the Timer Trace will show the status: enum ERRORCODES { SER_OK = 0, SER_UnknownLocType = 1, SER_NoObjInCursor = 2, SER_IllegalCharacterIndex = 3, SER_IllegalCarryLocation = 4, SER_NoObjectAtCarryLocation = 5, SER_InvalidCellLocation = 6, SER_NoSuchObjectInCell = 7, SER_IllegalMonsterID = 8, SER_MonsterIsNotMonster = 9, SER_NoSuchObjectOnMonster = 10, SER_IllegalChestID = 11, SER_ChestIsNotChest

= 12, SER_NoSuchObjectInChest = 13, SER_ChestBelongsToCharacter = 14, DER_OK = 0, DER_UnknownLocType = -1, DER_IllegalCharacterIndex = -2, DER_IllegalPossessionIndex = -3, DER_PossessionSlotNotEmpty = -4, DER_IllegalObjectForCarryLocation = -5, DER_CursorNotEmpty = -6, DER_InvalidCellLocation = -7, DER_IllegalDepth = -8, DER_IllegalChestID = -9, DER_ChestIsNotAChest = -10, DER_ChestBelongsToCharacter = -11, DER_ChestIsFull = -12, DER_IllegalMonsterID = -13, DER_MonsterIsNotAMonster = -14 };

### 4.16.11 Cloud Create

&CREATECLOUD

... *cellLocation*, *type*, *size* ⇒ ...

This instruction may not be called within a filter (Chapter 3).

The operand *type* may be any of the following:

| | |
|---|---|
| FIREBALL | 0 |
| DISPELL | 3 |
| OPENDOOR | 4 |
| POISON | 7 |
| DEATH | 40 |
| FLUXCAGE | 50 |

Table 4.2: Cloud Types

### 4.16.12 Missiles

#### 4.16.12.1 Missile Info Get

&MISSILEINFO@

... *objectID* ⇒ ... *contents*, *range*, *damage*, *direction*

Foo

#### 4.16.12.2 Missile Info Set

| &MISSILEINFO! |
| --- |
| ... *range damage direction, objectID* ⇒ ... |

Foo

## 4.17 Monster

### 4.17.1 Monster Delete

| &DELMON |
| --- |
| ... *cellLocation, index* ⇒ ... |

This instruction may not be called within a filter (Chapter 3).

Deletes a single monster within a group. Cannot delete the last.

The operand *index* is index of the monster in the group. (0 to *numberOfMonsters*-1)

### 4.17.2 Monster Insert

| &INSMON |
| --- |
| ... *cellLocation, monPosMask* ⇒ ... |

This instruction may not be called within a filter (Chapter 3).

Inserts a single monster into an existing group. It cannot create a new group.

The operand *monPosMask* is a mask of positions that the new member can occupy. Use 15 to allow any position.

### 4.17.3 Monster Variables

| | | |
|---|---|---|
| 0 | Number of monsters in the group | read-only |
| 1 | Monster type | |
| 2 | Hit points $1^{st}$ in group | |
| 3 | Hit points $2^{nd}$ in group | |
| 4 | Hit points $3^{rd}$ in group | |
| 5 | Hit points $4^{th}$ in group | |
| 6 | flags: 1=invisible, 2=drawAsSize4, 4=Unique, 8=Poisoned | |
| 7 | Alternate Graphic number | |

Table 4.3: Monster Variables

#### 4.17.3.1 Monster Variable Get

&MONSTER@

... *ID, index, num* ⇒ ...

Get/Set information about a group of monsters. (ID index num ... )

- *ID* is the identification number of the monster group.
- *index* is the address within the array where the information should be placed.
- *num* is the number of words of information that should be retrieved.

NOTE: These flags only work if you have '*Enabled Extended Monster Flags*' in the '*Edit/Global*' dialog.

#### 4.17.3.2 Monster Variable Set

&MONSTER!

... *monsterID, index, num* ⇒ ... *objectID*

This instruction may not be called within a filter (Chapter 3).

&MONSTER! cannot change the number of monsters or the monster type. Only the HitPoints and flags will be modified.

### 4.17.4 Monster Possession

&MONPOSS

... *monsterID, index* ⇒ ... *objectID*

Gets a monster possession.

### 4.17.5 Monster Movement Filter

#### 4.17.5.1 Monster Block Move

&MONBLK

... *directionMask* ⇒ ...

This instruction may only be called within a *Monster Moving* (3.12) filter.

#### 4.17.5.2 Monster Location and Distance

&MONL&D

... $n$ ⇒ ...

This instruction may only be called within a *Monster Moving*(3.12) filter.

Gets the monster's location and distance to the party and puts the two numbers in the Temporary Variable Array (2.4.5).

| | | |
|---|---|---|
| *tempVar*[n+0] | monster's location | $1024\,level + 32\,x + y$ |
| *tempVar*[n+1] | *neighborDistance* to party | (2.3) |

Is the storage location correct?

# 4.18 Cells

## 4.18.1 Cell Flags

These are variable length based on the type of the cell. The first value contains the ordinal of the type and is read-only.

### 4.18.1.1 Cell Flag Types

#### Type 0 - Wall (Closed Cell)

1. bits flags (writeable): *posMask* (2.6.4.1) for wall random decorations.

#### Type 1 - Floor (Open Cell)

1. bit flags:

| | | |
|---|---|---|
| 1 | random floor decorations | rw |

#### Type 2 - Pit

1. bit flags:

| | | | |
|---|---|---|---|
| 1 | false pit | cannot fall through | rw |
| 4 | obscure | hard to see even when open | rw |
| 8 | open | the pit is open | rw |

#### Type 3 - Stairs

1. bit flags:

| | | | |
|---|---|---|---|
| 4 | up | they lead up. | r |
| 8 | north/south | enter from north or south | r |

**Type 4 - Door**

1. bit flags:

| | | | |
|---|---|---|---|
| 1 | | north/south | r |
| 2 | | opens up/down | rw |
| 4 | | switch | rw |
| 8 | | fireball damages | rw |
| 16 | | axe damages | rw |

2. current open/closed state:

| | | | |
|---|---|---|---|
| 0 | open | | |
| 1 | $\frac{1}{4}$ closed | | |
| 2 | $\frac{1}{2}$ closed | | |
| 3 | $\frac{3}{4}$ closed | | |
| 4 | closed | | |
| 5 | bashed | | |

3. door type index (add link)

4. door decoration ordinal (add link)

**Type 5 - Teleporter**

1. bit flags (w):

| | |
|---|---|
| 4 | visible |
| 8 | active |

2. orientation change on teleport (w):

| | |
|---|---|
| 0 | none |
| 1 | turn right |
| 2 | turn to opposite |
| 3 | turn left |
| 4 | set to N |
| 5 | set to E |
| 6 | set to S |
| 7 | set to W |

3. what gets teleported (w):

| | |
|---|---|
| 0 | items |
| 1 | monsters |
| 2 | party & items |
| 3 | everything |

4. destination cell (r).

**Type 6 - Removeable Wall**

1. bit flags (w):

| | |
|---|---|
| 1 | trick wall (passable) |
| 4 | invisible |

#### 4.18.1.2 Cell Flags Get

`&CELL@`

... *cellLocation*, *index*, *num* ⇒ ...

Gets *num* cell flags and stores the values starting at *index* in the *array*.

The sequence:

```
LA L10 L3 &CELL@
```

therefore inspects the cell at the location stored in parameter-A and stores the first three parameters in the array starting at index 10.

#### 4.18.1.3 Cell Flags Set

`&CELL!`

... *cellLocation*, *index*, *num* ⇒ ...

This instruction may not be called within a filter (Chapter 3).

This is used to modify the specified cells flags.

### 4.18.2 Extended Cell Flags

What are the extended cell flags?

#### 4.18.2.1 Extended Cell Flags Get

&ECF@

... *cellLocation* ⇒ ... *flags*

Gets the extended flags of the specified cell (2.6.1).

#### 4.18.2.2 Extended Cell Flags Set

&ECF!

... *flags, cellLocation* ⇒ ...

Sets the extended flags of the specified cell (2.6.1).

### 4.18.3 Teleporter Copy

"CT"<to><from>

... ⇒ ...

Foo

### 4.18.4 Generator Delay

The delay (*generatorDelay*) is the amount of time that a Monster Generator is disabled after being triggered. When a Monster Generator is triggered, it creates a monster, a Timer message and then disables itself. When the Timer message arrives, it turns

the disabled actuator back into a Monster Generator. So, during that time there is no Monster Generator present. If you want a DSA at the same location as the Monster Generator to determine whether or not the generator is present then you must be sure that the DSA comes before the Monster Generator in the list of items in the Cell. Otherwise, the generator will have disabled itself (and disappeared from view) before the DSA has had a chance to perform its magic.

The value *generatorDelay* is 8 bits in length and is encoded so as to allow for very long delays. Values of 1 through 127 are mapped to the delays of 1 through 127. The delay associated with values of 128 through 255 are determined by the function:

$$generatorDelay = 64(value - 126) \tag{4.2}$$

So that the maximum delay is 64(255 - 126) = 8256 = about 23 minutes.

#### 4.18.4.1 Generator Delay Set

`&GENERATORDELAY!`

... *generatorDelay*, *cellLocation* ⇒ ...

Sets the delay of the generator at the specified location (2.6.1).

If the location contains no generator then it searches for an actuator that has been disabled and, if one is found, we assume that it is a disabled Monster Generator and set its delay value. If either are found, then the instruction silently does nothing.

#### 4.18.4.2 Generator Delay Get

`&GENERATORDELAY@`

... *cellLocation* ⇒ ... *generatorDelay*

Gets the delay of the generator at the specified location (2.6.1). Returns -1 if there is no generator.

### 4.18.5 Neighbors Inspect

&NEIGHBORS

... *cellLocation*, *criteriaMask* ⇒ ... *result*

Determine type and contents of cell's neighbors.

This instruction examines the four neighbors of a cell and sets a bit in the result for each that meet the criteria described by the *criteriaMask*.

This instruction might be useful to determine an escape route for the party, for example.

Bits in the result are: bit 0 - North bit 1 - East bit 2 - South bit 3 - West

The *criteriaMask* is divided into three sections:

1 - type of cell - bits a through b 2 - contents of cell - bits c through d

In order for a bit to be set in the result, the cell must match at least one bit in the 'type of cell' section and at least one bit in the 'contents of cell' section.

The 'Type of Cell Bits' are

bit 0 - Cell outside the dungeon (seen as a stone wall)

bit 1 - Solid stone wall

bit 2 - Real pit - open

bit 3 - Real pit - closed

bit 4 - False pit - cannot fall through

bit 5 - Door - open

bit 6 - Door - mostly open

bit 7 - Door - half open

bit 8 - Door - mostly closed

bit 9 - Door - closed

bit 10 - Door - smashed

bit 11 - Teleporter (objects only) active

bit 12 - Teleporter (objects only) inactive

bit 13 - Teleporter (party/objects) active

bit 14 - Teleporter (party/objects) inactive

bit 15 - Teleporter (monsters only) active

bit 16 - Teleporter (monsters only) inactive

bit 17 - Teleporter (anything) active

bit 18 - Teleporter (anything) inactive

bit 19 - Open

bit 20 - Trick Wall visible but passable

bit 21 - Trick Wall visible and impassible

bit 22 - Trick wall invisible and passable

bit 23 - Stairs up

bit 24 - Stairs down

The 'Contents of Cell' bits are:

bit 31 - Empty of monster or party

bit 30 - Occupied by monster

bit 29 - Occupied by party

Example. Suppose you want to know in which direction the party could proceed without falling through a pit or being teleported or moving to a different level. The criteriaMask should have the following bits set: bits 3 and 4 - a safe pit. bits 5 - an open door bits 11, 12, 14, 15, 16, and 18 - inactive/nonparty teleporters bit 19 - an open cell bits 20 and 22 - a passable trick wall bit 31 - No monster is there

### 4.18.6 Cell Inspect

&THISCELL

... *cellLocation*, *criteriaMask* ⇒ ... *result*

This instruction is the same as (4.18.5), except that it only examines the specified cell (2.6.1) location. As such only bit 0 of the result is used.

### 4.18.7 Location Decode

&LOC2ABSCOORD

... *cellLocation* ⇒ ... *level, x, y, pos*

Decodes a *cellLocation* (2.6.1) into absolute coordinates, including level offsets.

### 4.18.8 &THROW

&THROW

... ⇒ ...

This instruction may not be called within a filter (Chapter 3).

Throw a missile, either an object or a spell. (type objectLocation launchLocation direction range damage delta ... ) If 'type' is zero then an object is fetched from 'objectLocation'. Other values of type are:

-128 Fireball -127 Poison -126 Lightning -125 Dispell -124 Zo Spell -122 Poison Bolt -121 Poison Cloud -88 Monster Death (whatever that is!) 'objectLocation' includes a cell location and position 'launchLocation' includes a cell location and position 'direction' 0=N; 1=E; 2=S; 3=W 'range' is how far the missile will travel before falling 'damage' is the amount of damage done by the missile 'delta' is what is subtracted from range and damage each step.

## 4.19 Party and Characters

### 4.19.1 &ISCARRIED

&ISCARRIED

... *charAnyID, objectID* ⇒ ... *result*

( char# objectID . . . (location or number> ) Searches one or more characters for an object.

&ISCARRIED ( char# ID . . . <number or position> )

" &ISCARRIED ( <char num> <id or -type> ... <location or number> ) ; Is item being carried?",

Makes a complete search a character's inventory. The search is recursive. That is chests are searched and any monsters found in the chests are searched and any possessions of those monsters are searched, etc. The search includes the cursor if the leader is included in the search.

char#: 0-3 to choose one of the character, 4 for the leader, 5 for the whole party

Non-negative ID is the identifier for a particular object. For example, a particular apple....the one found in the northern corner of the supplies area. In this case the return value is -1 or the character number and the location on that character where the item is carried. These are encoded as 256*char + pos. If the object is found in the cursor then the position is 255. See BODY POSITIONS at ObjectTypeConstants.

If ID is negative then it represents an object type. For example any apple. In this case the return value is the number of items of type -ID being carried by that character. The object types are those defined by the enumeration OBJECTTYPE at ObjectTypeConstants. Two caveats:

- Some objects have more than one type number. Torches are an example. There are several types of torches, depending on the brightness of the torch. Compass is another example having several directions. In such cases only the 'Basic Object Type' (the first listed) is used. &ISCARRIED will never find a 'torch-b'. ALL TORCHES are of type torch_a for the purposes of &ISCARRIED.
- Notice that the 'basic object type' of a compass is zero! Negative zero does not look very negative in twos-complement arithmetic. THEREFORE . . . This is a special case. You must use negative one to search for compasses.

### 4.19.2 Party Management

&SWAPCHARACTER

. . . *charID, fingerprint* ⇒ . . . *result*

This instruction allows adding, removing and swapping of the characters in the party.

A character is removed by specifying a *fingerprint* of -1 and setting *charID* to the slot of the character to remove. The last member of the party cannot be removed. The removed character is placed in the *wings* and the rest of the characters slide left to fill in the empty space.

To add a character to the party from the *wings* one specifies an *charID* of -1 and the *fingerprint* of the desired character. The character will be put into the first empty slot in the party. A fifth character may not be added.

To swap an active party member with one in *wings* one specifies the desired *charID* of the outgoing member and the appropriate *fingerprint* of the incoming.

When a character is placed in the *wings*, all his possessions and statistics are saved with him.

The following table specifies the returned *result*:

| | |
|---|---|
| 0 | success |
| 2 | Attempt to remove last party member. |
| 3 | Attempt to add a fifth party member. |
| 4 | Reference to a character not in the wings. |

Table 4.4: Party Management Results

For all error results (those other than zero), the state of the game is left unmodified.

SEE: fingerprint (2.6.5.2)

### 4.19.3 Party Distance

&PARTYDISTANCE

... *cellLocation* ⇒ ... *distance*

Returns the distance of the party from the specified location (2.6.1).

If the location is on the same level as the party then the result is the *Manhattan distance* (2.6.3), otherwise it is the negative number of level differences.

### 4.19.4 Party Variables

| n | Variable | Notes |
|---|---|---|
| 0 | Party size, including dead | |
| 1 | Party level | |
| 2 | Party position x | 1 |
| 3 | Party position y | 1 |
| 4 | Party facing. | (2.6.4) |
| 5 | Party is sleeping | 2,3 |
| 6 | Can see through walls | 2,3 |
| 7 | Has magic footprints active | 2,3 |
| 8 | Index of the party Leader | |
| 9 | Invisible | 2,3 |
| 10 | Fireshield value | |
| 11 | Spellshield value | |

Table 4.5: Party Variables

1. Does not include the offset.
2. Read-only value ????
3. Values other than zero indicate *true*.

#### 4.19.4.1 Party Variable Get

&PARTY@

... *index*, *n* ⇒ ...

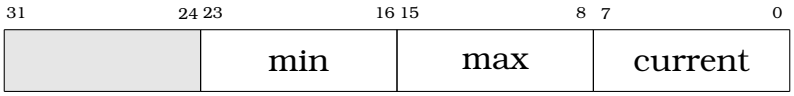
Reads the first *n* party variables (4.5) and places the results in the array starting at offset *index*.

### 4.19.5 Character Variables

| n | Variable | Store range | | Notes |
|---|---|---|---|---|
| 0 | Facing | - | | |
| 1 | Food | $[-1023, 2048]$ | | |
| 2 | Hit Points | $[0, Max]$ | | |
| 3 | Load | - | | |
| 4 | Mana | $[0, 900]$ | | |
| 5 | Ouches | - | | |
| 6 | Position | - | | |
| 7 | Shield Strength | - | | |
| 8 | Stamina | $[0, Max]$ | | |
| 9 | Water | $[-1023, 2048]$ | | |
| 10 | Luck | - | | 1 |
| 11 | Strength | - | | 1 |
| 12 | Dexterity | - | | 1 |
| 13 | Wisdom | - | | 1 |
| 14 | Vitality | - | | 1 |
| 15 | Anti-Magic | - | | 1 |
| 16 | Anti-Fire | - | | 1 |
| 17-56 | Skills | | | (2.6.5.3) |
| 57 | fingerprint of character | - | | (2.6.5.2) |
| 58 | 32-bit mask of talents | full range | | |

Table 4.6: Character Variables

1. Encoded as four eight-bit values: zero, minimum, maximum, current:




results [10] through [16] are encoded in four 8-bit bytes: zero, minimum, maximum, current result[10] = Luck

results[17] through [56] are two words for each of the 20 skills. 0 and 4-7 =Fighter 1 and 8-11 =Ninja 2 and 12-15=Priest 3 and 16-19=Wizard

result[57] = fingerprint of character (bottom 16 bits of location of defining text)

result[58] = *32-bit mask of talents (actually can be used for any purpose the designer pleases!)

#### 4.19.5.1 Character Variables Get

&CHAR@

... *partyID, index, n* ⇒ ...

&CHAR@ (char# index n ... ) char# 0 to 3 ( or 4 to indicate the 'Lead Character' ) index = where to put result in array n = number of variables to fetch Fetch variables associated with a single party member. See Character Fetch

char# 0 to 3 ( or 4 to indicate the 'Lead Character' ) index is where to put results in array (or where the new values are) n is number of variables to fetch (or number to store)

Stores the first *n* character variables (4.19.5) of

#### 4.19.5.2 Character Variable Set

&CHAR!

... *partyID, index, n* ⇒ ...

This instruction may not be called within a filter (Chapter 3).

Exactly the same a &CHAR@ except that the values you provide are used to modify the character's attributes. Only a few of the values are actually used but the format and order is the same so that you can do the &CHAR@, modify some of the variables, and then do the &CHAR!.

The table indicates which variables are modified by this instruction.

### 4.19.6 Character Location

&WHEREISCHAR

. . . *fingerprint* ⇒ . . . *result*

Returns the location of the character with the specified *fingerprint* (2.6.5.2).

If there is an active party member with the given *fingerprint*, then the index (2.6.5.1) of the first found is returned. Otherwise if there is a matching character within the *wings* (4.19.2) then the result is 5 and finally a result of 4 indicates that none was found.

### 4.19.7 Character Name

&CHARNAME@

. . . *fingerprint, index* ⇒ . . .

Sets *textString*[*index*] to the name of the character with the specified *fingerprint* (2.6.5.2). If no such character exists, the string is set to empty.

### 4.19.8 Character Possessions

&CHPOSS@

. . . *partyID, index* ⇒ . . . *objectID*

<char num> = 0 to 3; <index = 0 to 29>", " <char num> == 4 means 'Active Character'", " <char num> == -1 means object in cursor",

### 4.19.9 Character Skills

SEE: Skills (2.6.5.3)

#### 4.19.9.1 Skill Adjustement Parameters

&SETADJUSTSKILLSPARAM

... *p4, p5, p6, p7, p8* ⇒ ...

Sets the designer configurable parameters for next *Skill Adjust Filter* (3.2) call.

IS THIS CORRECT?
When a dungeon is loaded, the values which will be sent to the filter are initialized to zero. After a call to this instruction the values specified will be retained until another call of this instruction is issued.

#### 4.19.9.2 Give XP

&EXPERIENCE+

... *charID, skillID, amount* ⇒ ...

Increases the character's (2.6.5.1) XP total of the specified skill (2.6.5.3) by *amount*. If the specified skill is a secondary (hidden) skill, then its associated primary skill is increased by the same amount. If the primary skill moves over a mastery level boundary, level gains are applied.

Silently does nothing if invalid character or skill.

#### 4.19.9.3 Level of Mastery

&MASTERY

... *charID, skillID, flags* ⇒ ... *level*

Returns the character's level of mastery in the specified skill .

The operand *flags* is composed of the following bit values:

| | |
|---|---|
| 1 | don't include item effect |
| 2 | don't include temporary effects |

The sequence:

```
L0 L5 L3 &MASTERY
```

will return the left most character's (L0) mastery in thrust (L5) excluding any modifiers from items and temporary effects (L3, which is 1 logically OR 2).

### 4.19.10 &WHOHASTALENT

&WHOHASTALENT

. . . *talentMask* ⇒ . . . *characterIndexMask*

Find all characters within the party who have the given combination of talents. ( talentMask . . . characterIndexMask ) Each character has an associated 32-bit mask of talents that is defined when the character is reincarnated or resurrected. This mask can be examined/changed using &CHAR@ and &CHAR!.

This function sets the associated bit (bit 0 to bit 3 ) of the result for each character who has all the talents specified by the talent parameter. So you can ask which characters have a combination of 'TreeClimbing' and 'AxeSpell', for example. If you want to find which characters have EITHER of two talents, you could form the bitwise 'or' of the result of two &WHOHASTALENT function calls.

### 4.19.11 Poison

&CAUSEPOISON

. . . *poisonValue*, *charID* ⇒ . . .

This instruction may not be called within a filter (Chapter 3).

### 4.19.12 Teleport Party

&TELEPORTPARTY

. . . *cellLocation* ⇒ . . .

This instruction may not be called within a filter (Chapter 3).

Moves the party to the specified location (2.6.1).

### 4.19.13 Level XP Multiplier

&MULTIPLIER@

... *level* ⇒ ... *result*

Returns the experience point multiplier of the specified level. If the level does not exist, the result is one.

## 4.20 Effects

### 4.20.1 Color Palette

&PALETTE

... *unused*, *unused*, *overlayNum*, *density* ⇒ ...

sfasdf

The *density* operand takes an integer on [0,100].

### 4.20.2 Sound Play

&SOUND

... *soundNum*, *attenuation*, *flags* ⇒ ...

Plays an internal or custom sound.

In the case of *custom* sounds it must be a standard wave file in 8-bit mono at 11025 samples per second. The operand *soundNum* is the graphic number of the file in

CSBgraphics.dat. The *attenuation* is a divisor. Set it to 1 for full volume. Setting it to 100 makes it almost inaudible.

The operand *flags* is not used and should be zero for forward compatibility.

To play an internal sound you specify the negative of the internal sound number (2.12). In this case, volume is 0 for low volume and 1 for high volume. This is normally used to make the sounds quieter when the source of the sound is far from the party.

The global sound volume setting is always applied.

### 4.20.3 Text

WAY TOO LITTLE INFORMATION HERE

#### 4.20.3.1 Display Cell Text

&SAY

... *cellLocation, color* ⇒ ...

Displays the text from the specified cell location (2.6.1) in the scrolling text area.

#### 4.20.3.2 Display FOO text

&TEXTSAY

... *index, color* ⇒ ...

Displays *textString*[*index*] in the scrolling text area.

#### 4.20.3.3 Clear Text

| |
|---|
| &DISCARDTEXT |
| ... ⇒ ... |

Clears the text scrolling area.

#### 4.20.3.4 Text Get

| |
|---|
| &TEXT@ |
| ... *object, index* ⇒ ... |

C

#### 4.20.3.5 Global Text

| |
|---|
| &GLOBALTEXT! |
| ... *index, globaIndex* ⇒ ... |

globalText[*globaIndex*] = textString[*index*]

#### 4.20.3.6 Describe Object

| |
|---|
| &DESCRIBE |
| ... *location, index, color* ⇒ ... |

This instruction may only be called within a *Viewing Filter* (3.6).

The instruction is used to manipulate the *PhraseMask* and the text associated with any of the eight phrases that can be printed when an object is 'viewed'. The syntax is: &DESCRIBE ( location index color. . . ) 'index' is zero through seven.....which of the eight phrases is being modified. 'location' can be one of three things:

minus 1 - This means that the phrase should be printed using the current text. The appropriate bit in the PhraseMask will be set. minus 2 - This means that the phrase should be disabled. The appropriate bit in the PhraseMask will be cleared. a dungeon location (including the position within the cell). This means that the text at that location and position in the dungeon will be displayed. The appropriate bit in the PhraseMask will be set. 'color' is the color that the text will appear. The parenthesized list of phrases all will appear in the same color. There are six possible phrases that can be printed within the parentheses. The last of these that is actually printed will determine the color of the entire list. The last two of the eight phrases ( Index 6 and index 7 ) can each be set to its own individual color. So, altogether, only three colors can appear at one time.

### 4.20.4 Savegame Control

| &DISABLESAVES<br>... *value* ⇒ ... |
|---|

Saving of games is enabled if *value* is zero, otherwise disabled.

## 4.21 Indirect

### 4.21.1 &%INDIRECT

&%INDIRECT

| &%INDIRECT<br>... ⇒ ... *objectID* |
|---|

### 4.21.2 Parameters Get

&PARAM@

... *num*, *index* ⇒ ...

Fetches *num* parameters from *tempVar*[*index*].

### 4.21.3 Parameters Set

&PARAM!

... *num*, *index* ⇒ ...

Stores *num* parameters *tempVar*[*index*].

### 4.21.4 Delay

&%DELAY

... *time* ⇒ ...

Delays the next executed *indirect action*.

### 4.21.5 Cast

&CAST

... ⇒ ...

Cast spell using the previously stored parameters.

### 4.21.6 x

```
&FILTEREDCAST
... ⇒ ...
```

Cast spell (with filtering) using the previously stored parameters.

# Chapter 5

# Explanations

The goal of this chapter is to provide some “brush strokes”....

## 5.1 Basics

asdf

### 5.1.1 Understanding Integers

The introduction states that *“DSAs* operate solely on 32-bit integer data”. Let us break that down.

#### 5.1.1.1 Switches and Bits

One of the fundamental components used to build computers is a switch, which is really no different from a simple mechanical switch, and has two positions or states: *on*/*off*, *true*/*false* and *0*/*1* are possible ways to describe that state.

Now let us imagine that we group together three sets of these switches:

The first with one switch, the second with two and the third with three, and will come up with all possible combinations of *off*/*on* that each of them can collectively have. We will write *off* as *0* and *on* as *1*:

| Switches | Each possible state | | | |
|---|---|---|---|---|
| 1 | 0, 1 | 2 | 2 | $2^1$ |
| 2 | 00, 01, 10, 11 | 4 | 2*2 | $2^2$ |
| 3 | 000, 001, 010, 011, 100, 101, 110, 111 | 8 | 2*2*2 | $2^3$ |

Table 5.1: Number of switch combinations

Notice that with one switch, it's either *off* or it's *on*, so it can have two different states (or values). Increasing the number of switches to two increase the number of possible combinations to four. With three switch the number of combinations increases to eight. This generalizes to each time we increase the number of switches in the group by one, the number of possible combinations double.

It is exactly in this manner than computers store (and also manipulate) information. The switches are called bits. Going back to our original statement: "*DSAs* operate solely on 32-bit integer data", the "32-bit" part means we are grouping together 32 of these bits to consider collectively. Notice that this collection can form $2^{32} = 4294967296$ unique combinations!

#### 5.1.1.2 Integers

Integers to mathematicians are the *natural numbers* (1,2,3...), the negative of the *natural numbers* and zero. Computers however cannot deal with this mathematical ideal since we only have a limited number of bits at our disposal, specifically for us we are limited to collections of 32 bits.

The normal

$$1942 = 1 \times 10^3 + 4 \times 10^2 + 9 \times 10^1 + 4 \times 10^0$$

NOTE: XXX. For example, under Windows the standard Calculator accessory allows XXX. Click the view tab and select Scientific. XXX (The GNOME accessory *gcalctool*)

#### 5.1.1.3 Negative integers

Up to now we have only considered positive numbers. So what about negative numbers? Two's complement.

#### 5.1.1.4 Hexidecimal numbers

Generally our brains do not deal well looking at numbers in binary form and they make for very long sequences when written out, consider the decimal number 456085838 in binary form is:

11011001011110101000101001110

blah, blah, blah:

0001 1011 0010 1111 0101 0001 0100 1110

blah, blah:

0x1B2F514E

The first part "0x" is simply one of the standard methods to inform the reader that the number is in hexidecimal format and has no meaning outside of that.

| Dec | Hex | Bin |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |

| Dec | Hex | Bin |
|---|---|---|
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |

| Dec | Hex | Bin |
|---|---|---|
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |

| Dec | Hex | Bin |
|---|---|---|
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

Table 5.2: –

#### 5.1.1.5 Enumerations

These XXX

#### 5.1.1.6 Bit flags

Let us not forget that these 32-bit integers are just a collection of 32 switches. XXX

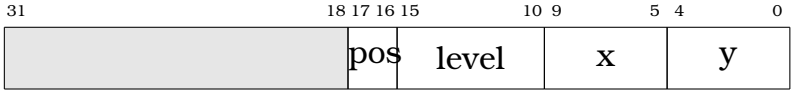



Table 5.3: Cell Location Format

### 5.1.2 Instances

One of the standard items in the game is a dagger. Now there may be more than one dagger in the game, but they all share the same basic properties, such as the icon used and the set of actions the hero can perform with it. Inside the game is a defination of all the common properties that all daggers have. The actual dagger item that a hero can use and manipulate is called an *instance* of that defination. This instance knows that it is a dagger and stores all the particular information which makes it unique from every other dagger in the game.

*DSAs* are logically game items which the hero cannot see or move around. Designers make their definitions by the programming model and places instances of these definations inside the dungeon like any other game item.

## 5.2 Memory

asdf

### 5.2.1 Parameters

asdf

### 5.2.2 Manipulating the Stack

asdf

## 5.3 Messages

## 5.4 Bitwise Operations

The instructions which are called *bitwise XXX*

## 5.5 Instruction Reference

Each instruction in the reference chapter starts with box which contains two pieces of information:

1. How the instruction is actually written.
2. How the instruction modifies the stack.

An example of an easy instruction is a follow:

&2DROP

$\ldots x,\ y \quad \Rightarrow \quad \ldots$

So to tell the *DSA* to perform this operation one simply writes: &2DROP in the appropriate place in the sequence.

XXXX

### 5.5.1 Understanding Expressions

Instructions which are not written in a fixed manner XXX

.....

*messageDelay* ::={ *integer* | "X" | "Y" } .....(default is 0)

This reads that valid *messageDelay* element is exactly one of the following: an integer, the letter "X" or the letter "Y". It also states than when *messageDelay* is an optional element and it that element is not specified, then it is equivalent to having using the integer zero.

So for an imaginary instruction defined as:

"BAR" [*messageDelay*]

The all of the following are valid instructions:

BAR BAR0 BAR123 BARX BARY

where the first two are identical.

## 5.6 Flow Control

The various flow control instructions ...

The sequence:

```
LA L0 G23 ...
```

# Appendix A

# Foo

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |

Table A.1: Pressure Pad Types

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | | 10 | | 20 | | 30 | | 40 | |
| 1 | | 11 | | 21 | | 31 | | 41 | |
| 2 | | 12 | | 22 | | 32 | | 42 | |
| 3 | | 13 | | 23 | | 33 | | 43 | |
| 4 | | 14 | | 24 | | 34 | | 44 | |
| 5 | | 15 | | 25 | | 35 | | 45 | |
| 6 | | 16 | | 26 | | 36 | | 46 | |
| 7 | | 17 | | 27 | | 37 | | 47 | |
| 8 | | 18 | | 28 | | 38 | | 48 | |
| 9 | | 19 | | 29 | | 39 | | 49 | |

Table A.2: –

# Appendix B

# ASDF

# List of Figures

# List of Tables

# Nomenclature

filter a DSA to modify built-in engine behavior

Fingerprint Blah

ordinal a specified number in a series