RPGsh User Manual

TheMohawkNinja

May 5, 2024

Ver. 0.8.2

DISCLAIMER: This project is entirely developed independantly. I am not associated in anyway with Wizards of the Coast, Paizo, or any other game development company. To any lawyers eyeing me up, I will not be adding information that would negate the need for players to purchase your products.

Contents

1 Introduction and Basic Usage

The Roleplaying Games Shell, `rpgsh`, is an interactive and extensible shell purpose-built for augmenting player and DM gameplay for table-top RPGs like Dungeons & Dragons®, Pathfinder®, and more!

`rpgsh` provides users with capabilities similar to those found in conventional shells like `bash` or `PowerShell` like command execution and variable assignment/modification, while also adding functionality more unique to shell environments, like varying data types and variable scopes.

1.1 The Prompt

When interacting with the shell directly, you will be presented with a prompt that will look similar to the following:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$
```

The prompt contains the currently loaded character's name (*<NO_NAME>*) along with their **current/max** *(temp)* hitpoints.

As with any command line interface, you interact with the prompt by entering in either a variable or a program, along with any operators or parameters. For example, if you want to roll a 20-sided die, you would enter the following:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ roll d20
```

The maximum size of the input buffer for the prompt is 256 characters. Exceeding this will throw an error.

2 Program Listing

As of version 0.8.2, the following programs are available to the user when interacting with the `rpgsh` prompt:

`banner`

Displays the ASCII art logo for `rpgsh` along with a one-line description of the program and the author's signature.

`help`

Lists all applications available to `rpgsh` along with a brief description.

`list` *

Lists all the variables in one or all scopes.

`roll` *

Dice-rolling program which supports custom lists and result counting.

`setname` *

Sets which variable is used for displaying the character's name.

`variables`

This is ***NOT*** to be explicitly called by the user, but is instead implicitly called when the user enters a variable as the first parameter in the prompt.

`version`

Prints `rpgsh` version information.

*Program contain additional parameters. Additional information can be found by running the program with the `-?` or `--help` flag.

3 Variables

`rpgsh` allows the user to set, get, and modify variables. Variables are arranged in a nested hierarchy through three different scopes, and through an arbitrary number of levels within each scope. Operations may be performed on variables, with the following operators currently supported:

Binary: `=`, `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`

Unary: `++`, `--`

The components of an operation must be space delimited as shown below:

`$v/four = 2 + 2` **Correct**

`$v/four=2+2` **Incorrect**

Variables in `rpgsh` follow the below syntax:

$$<scope><type>[<character>]/<level_1>/<level_2>/ \ldots /<level_n>$$

Below describes each part in detail:

3.1 scope

A single character representing which level of the overall hierarchy is being referenced. There are three total scopes for use in `rpgsh`. These are defined below:

`@` Character attributes. This scope encompasses all variables specific to a given character. If the <*character*> attribute is omitted, this references the currently loaded character.

These are stored in `~/.rpgsh/campaigns/<campaign>/characters/<charactername>.char`

`#` Campaign variables. This scope encompasses all variables in the current campaign, and are available while any character in the current campaign is loaded.

These are stored in `~/.rpgsh/campaigns/<campaign>/.vars`

`$` Shell variables. This scope encompasses all campaigns and is the broadest scope in `rpgsh`.

These are stored in `~/.rpgsh/.vars`

3.2 type

A single character representing the data type of the variable. When calling an existing variable, this parameter may be omitted, in which case `rpgsh` will find the matching variable. If more than one match is found, the first match will be used. As of version 0.8.2, the following data types have been implemented: `v` (Var), `d` (Dice), `c` (Currency), `s` (Currency System), and `w` (Wallet). Note that in all operations, the data type of the returning value will always be the same as the left-hand side (LHS) of the operation.

For each data type described in this section, the following subsections describe various attributes associated with the given data type:

Constructors:

These describe the ways in which each data type can be created while using `rpgsh`. These necessarily include a *explicit* constructor, which is in the format of *c*{*Properties*}, where *c* is some lower-case character that defines which data type is being constructed, and *Properties* which are one or more numbers and/or strings of characters that make up the constructed object. Additionally, constructors may include one or more *implicit* constructors, which do not have a universal format, but make for a more human-readable means of interacting with data types.

Properties:

These describe each property as declared in the explicit constructor definition, along with noting whether or not it is optional. If more than one properties are available to be defined, they must be in a comma-delimited list, and all commas must be entered even if a given optional property is omitted.

When calling a variable, the properties of a variable can be accessed by appending `.Property` to the end of the variable.

Examples:

Examples of possible ways to construct the given data type.

Operations Table(s):

These describe what happens when you operate on a variable with a given operator and a given right-hand side (RHS) data type. The left-most column describes a particular operator, and each cell in the columns to the right describe the operation performed when the RHS is the data type defined by the column header, with the given property affected printed in *italics*. Cells marked with an **ERR** result in an error being thrown and no change being made to the LHS.

3.2.1 Var

These are generic, lazily-evaluated variables that may contain either a string or an integer, similar to how variables in many scripting languages operate. Operations performed on var-type variables are thus dependant on whether or not the current value stored is evaluated to be a string or an integer.

Constructors:

- An integer
- A string of characters
- A string of characters wrapped in quotation marks
- v{<*Value*>}

Properties:

Value:
The value of the var-type variable. This can be any number or string of text. When used in an explicit constructor, quotation are not necessary for strings containing spaces.

Examples:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @v/MyVar = 3
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @v/MyVar = three
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @v/MyVar = "The number three"
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @v/MyVar = v{The number three}
```

Operations Tables:

LHS evaluates to Integer						
Op (Bin.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`s`	`w`
`=`	*Value* Assignment	*Value* Assign.*	**ERR**	**ERR**	**ERR**	**ERR**
`+`	*Value* Addition	**ERR**				
`-`	*Value* Subtraction					
`*`	*Value* Multiplication					
`/`	*Value* Division					
`+=`	*Value* Addition Assign.					
`-=`	*Value* Subtraction Assign.					
`*=`	*Value* Multi. Assign.					
`/=`	*Value* Division Assign.					
Op (Un.)	—					
`++`	*Value* Increment					
`--`	*Value* Decrement					

LHS evaluates to String						
Op (Bin.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`s`	`w`
`=`	*Value* Assign.*	*Value* Assignment	**ERR**	**ERR**	**ERR**	**ERR**
`+`	**ERR**	*Value* Concatenation				
`-`		**ERR**				
`*`						
`/`						
`+=`		*Value* Concatenation				
`-=`		**ERR**				
`*=`						
`/=`						
Op (Un.)	—					
`++`	**ERR**					
`--`	**ERR**					

*A warning will be thrown to indicate that the evaluated data type has changed.

3.2.2 Dice

These are variables which not only can be constructed and printed in the standard RPG dice format, but operations performed on dice are meant to allow users to more intuitively interact with the dice they may need to roll throughout gameplay.

Constructors:

- <*Count*>d<*Faces*>[+|−]<*Modifier*>
- d{<*Count*>,<*Faces*>,<*Modifier*>}

Properties:

Count (optional):
The count (quantity) of dice. If omitted, assumes a value of 1.

Faces:
The number of faces of the di(c)e.

Modifier (optional):
A modifier value which affects the total roll value. If omitted, assumes a value of 0.

Examples:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @d/MyDice = d20
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @d/MyDice = 2d8
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @d/MyDice = 3d6+1
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @d/MyDice = d{1,20,-5}
```

Operations Table:

Op (Bin.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`s`	`w`
`=`	**ERR**	Assignment*	Assignment	**ERR**	**ERR**	**ERR**
`+`	*Modifier* Addition	**ERR**	*Count* Addition**			
`-`	*Modifier* Sub.		*Count* Sub.**			
`*`	*Count* Multi.		**ERR**			
`/`	*Count* Division					
`+=`	*Modifier* Add. Assign.		*Count* Add. Assign.**			
`-=`	*Modifier* Sub. Assign.		*Count* Sub. Assign.**			
`*=`	*Count* Multi. Assign.		**ERR**			
`/=`	*Count* Div. Assign.					

Op (Un.)	—
`++`	*Modifier* Increment
`--`	*Modifier* Decrement

*If and only if the string is formatted appropriately, otherwise an error will be thrown.

**If and only if both dice have equal faces. Additionally, in the event that both dice have different modifiers, a warning will be thrown indicating that only the LHS modifier will be preserved.

3.2.3 Currency

These are variables which are used to handle monetary values. If the currency is part of a currency system, `rpgsh` can automatically calculate change and merge smaller denominations into larger denominations as needed.

Constructors:

- c{<*CurrencySystem*>,<*Name*>,<*SmallerAmount*>,<*Smaller*>,<*Larger*>}

Properties:

CurrencySystem (optional):
The name of the currency system that the currency is a part of. If the game only has one currency system, this property may be omitted.

Name:
The name of the currency. It must be unique within the scope that the currency is being declared within. In most cases, this should be the Campaign scope, as currencies usually cover more than one character in a given game, whereas the Shell scope would cover all games, which would be non-ideal unless you know for a fact you will only ever play the same game.

SmallerAmount (optional):
The amount of the larger denomination needed to equal this denomination. If the game only has one currency, this property may be omitted.

Smaller (optional):
The name of the smaller denomination. If the game only has one currency, this property may be omitted.

Larger (optional):
The name of the larger denomination. If the game only has one currency, this property may be omitted.

Examples:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @c/nickel = c{US,nickel,5,penny,dime}
```

Operations Table:

Op (Bin.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`s`	`w`
`=`	**ERR**	**ERR**	**ERR**	Assignment	**ERR**	**ERR**
`+`				**ERR**		
`-`						
`*`						
`/`						
`+=`						
`-=`						
`*=`						
`/=`						

Op (Un.)	—
`++`	**ERR**
`--`	

3.2.4 Currency System

These are variables which allow users to group related Currency-type variables into functioning currency systems, thus allowing `rpgsh` to know what Currencies to convert to and from when needing to calculate change or when compressing large quantities of a smaller denomination into smaller quantities of a larger denomination.

Constructors:

- A string of characters
- s{*Name*}

Properties:

Name:
The name of the currency system, mainly for use as a reference when constructing Currency-type variables.

Examples:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @s/US = US
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @s/US = s{US}
```

Operations Table:

Op (Bin.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`s`	`w`
`=`	**ERR**	*Name* Assign.	**ERR**	**ERR**	*Name* Assign.	**ERR**
`+`		**ERR**			**ERR**	
`-`						
`*`						
`/`						
`+=`						
`-=`						
`*=`						
`/=`						

Op (Un.)	—
`++`	**ERR**
`--`	

3.2.5 Wallet

A Wallet-type variable. Like a real, physical wallet, instances of this data type contain quantities of one or more Currency-type variables. These are both meant to be used as the wallet or coin purse of given character, but also as a formalized way of defining the cost of items.

Constructor:

- w{$CurrencyName_1$,$Quantity_1$,$CurrencyName_2$,$Quantity_2$, ... ,$CurrencyName_n$,$Quantity_n$}

Properties:

$CurrencyName_x$
The name of a Currency-type variable. It must be prefixed with a scope sigil so `rpgsh` knows which currency to load.

$Quantity_x$
The amount of $CurrencyName_x$ in the Wallet-type variable.

Examples:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @w/MyWallet = w{#Gold,10}
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @w/MyWallet = w{#Gold,10,#Silver,5}
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @w/MyWallet = w{#Gold,10,#Silver,5,#Copper,3}
```

Operations Tables:

Due to the fact that, unlike other data types, Wallet-type variables contain an arbitrary number of its' two properties, references to a given property (unless stated otherwise) in the below table refer to each instance of said property in a given wallet. This can be thought of as "for each *Property* in the wallet, do *something*."

Op (Bin.)	`v` *(Integer)*	`v` *(String)*	`d`
`=`	**ERR**	**ERR**	**ERR**
`+`			
`-`			
`*`	*Quantity* Multiplication		
`/`	*Quantity* Division*		
`+=`	**ERR**		
`-=`			
`*=`	*Quantity* Multiplication		
`/=`	*Quantity* Division*		

Op (Bin.)	`c`	`s`	`w`
`=`	**ERR**	**ERR**	Assignment
`+`	*Quantity* of RHS Increment		*Quantity* Addition
`-`	*Quantity* of RHS Decrement		*Quantity* Subtraction
`*`	**ERR**		**ERR**
`/`			
`+=`	*Quantity* of RHS Increment		*Quantity* Addition
`-=`	*Quantity* of RHS Decrement		*Quantity* Subtraction
`*=`	**ERR**		**ERR**
`/=`			

Op (Un.)	—
`++`	*Quantity* of smallest *CurrencyName* Increment
`--`	*Quantity* of smallest *CurrencyName* Decrement

*Just as in real banking, the division of currencies may be lossy, as currencies are not infinitely divisible. Thus, when dividing the quantities in a wallet, any remainder after making change down to the lowest denomination is lost. For example, suppose we have a currency system containing Dollars and Pennies, where 1 Dollar equals 100 Pennies, and we have a wallet containing 1 Dollar. If I divide that wallet by 3, I would end up with 33 Pennies remaining in my wallet, with the remaining 1/3 of a Penny being lost.