ROODYLIB

# Table of Contents

# Introduction and Acknowledgements

Welcome to Roodylib! Roodylib exists to improve upon and fix some things in the original Hugo library and to hopefully add some extra functionality while we're at it. Roodylib would not exist without the contributions and suggestions from Kent Tessman, Mike Snyder, Jason McWright, Robb Sherwin, Rob O'Hara, Paul Lee, Juhana Leinonen, Nikos Chantziaras, and Tristano Ajmone.

Additional tsunamis of thanks go to Paul Lee for his invaluable suggestions for this document. The tiniest raindrop of gratitude goes to Marius Müller for his one suggestion.

Written with LibreOffice.

# Getting Started

First off, a note about flags (such as `USE_ROODYLIB`, `USE_DARK_ROOM`, or any of the other ones listed in this document): you'll always want to `#set` them before any grammar or library files are included. If you're using the Roodylib "new shell", you can set all flags in **flags.hug**; otherwise, you really can just put them before any libraries are included.

## Starting Fresh

Starting a new game? The best way to jump right in is to use one of the game stub files from the "shells" folder. The one in the "old" folder is one file with some of the most used switches and file inclusions available, while the one in the "new" folder splits all that up into several files. It's my intention that the new shell is a good start for a larger, more-complicated game where organization is important.

You'll want to make sure the `#set USE_ROODYLIB` line is not commented out—I believe that's the default—but I include the option to turn Roodylib off to make it easier to track down if a bug is due to Roodylib code.

### MakePlayer

Especially if your game is not in the usual second person, you might want to be aware of `MakePlayer`, a helper routine for setting up the player character object.

```
MakePlayer(<player character object>, <tense number>)
```

So, in a regular, second-person game where the player character is defined as the `you` object, you would call `MakePlayer(you,2)`. Besides setting the `player` variable, the routine also sets the `location` variable to the parent of the player (which Roodylib will still interpret correctly if it happens to be an object in a room).

### Completely New to Hugo?

If you are just starting out with Hugo and are using Windows, I recommend using my Hugo & Notepad++ bundle. Notepad++ is a highly-configurable text editor, and I've prepared it with syntax highlighting and toolbar buttons for easy file-creation and compilation (among other things). You can get it here.

# Updating an Older Game to Roodylib

Of course, the most important thing is to include Roodylib itself. To do this, you want to `#include "roodylib.g"` before **verblib.g** and `#include "roodylib.h"` *after* **hugolib.h**. Beyond that, you'll want to call the routines `Init_Calls` in the `init` routine and `Main_Calls` in the `main` routine.

```
routine init
{
!: First Things First
       SetGlobalsAndFillArrays
!: Screen clear section
#ifclear _ROODYLIB_H
       cls
#else
       InitScreen
       Init_Calls
#endif
!: Game opening
       IntroText
       MovePlayer(location)
}

routine main
{
       counter = counter + 1
       run location.each_turn
       runevents
       RunScripts
       if parent(speaking) ~= location
              speaking = 0
       PrintStatusLine
       Main_Calls
}
```

*example init and main routines*

If you're using any additional extensions that make use of Roodylib functionality, it'd be good to `#set USE_ROODYLIB` before any files are included.

```
Objects:       55 (maximum  1024)      Routines:   250 (maximum   320)
Attributes:    25 (maximum   128)      Events:       1 (maximum   256)
Properties:    41 (maximum   254)      Labels:       9 (maximum   256)
Aliases:       43 (maximum   256)      Globals:     59 (maximum   240)
Constants:    121 (maximum   256)      Arrays:       8 (maximum   256)
```

*a Roodylib shell file compiled with the Roodylib library included*

Roodylib adds a lot of extra routines so it's likely you'll have to raise your routine limit settings. To do this, add this to the beginning of your code:

```
$MAXROUTINES = [new limit]
```

*raising the maximum number of routines*

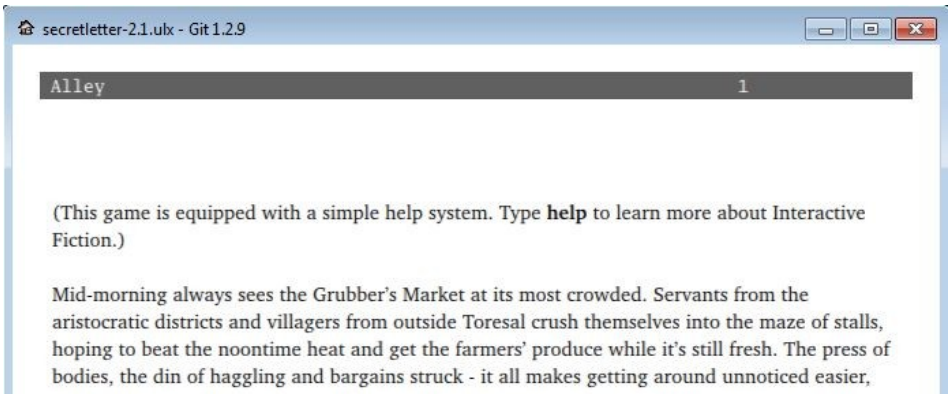
Depending on your game, you may need to change other limits as well (all are covered in the Hugo Book). Basically, if the compiler complains that you have gone over the limit for any particular thing, just keep raising the number of the max allowed until it works!

# Presentation

One of the biggest features Roodylib provides authors right out of the box is its attention to presentation.

## Status Lines

If you are new to IF, we refer to the top line of the window (the one that displays the room name and possibly a score and/or turn counter) as the "status line." In recent years, I've been disappointed with the attention given to status lines in many games. We may have moved past keeping score in many games, but I find the unmarked turn counter in some games a bit distracting.




The score counter is not even in an eye-pleasing location. No matter what you kind of information you want in status bar, this is something Roodylib will do automatically.

In Hugo, you select the status line type you want by changing the `STATUSTYPE` global. To make this simpler, I've set up some constants you can use for setting your status line type.

| | |
|---|---|
| **NO_STATUS** | No information displayed in top right |
| **SCORE_MOVES** | The abbreviated score/turn counter popular today ("0/0") |
| **TIME_STATUS** | Display turn counter as converted to clock time ("9:00 am") |
| **CUSTOM_STATUS** | Use the routine `STATUSTYPE4` to print the status |
| **INFOCOM_STYLE** | Print the old long version of score/turns ("SCORE: 0 TURNS:0") |
| **MILITARY_TIME** | Display turn counter as clock in military time ("22:00") |

To select the status type you want, just put a line like this in `init` or `SetGlobalsAndFillArrays` (depending on whether you are using one of the Roodylib shells):

```
    STATUSTYPE = INFOCOM_STYLE
```

### PrintStatusLineLocation

Roodylib now uses a dedicated routine for printing the room name in the status window.

```
routine PrintStatusLineLocation
{
	if not location
		print "\_";
	elseif not light_source
		print "\_In the dark";
	else
	{
		print "\_";
		print capital location.name;
	}
}
```

This can be easily replaced if there are instances where you want different behavior.

### Custom Status Fields

If you set `STATUSTYPE` to the `CUSTOM_STATUS` constant, replace the `STATUSTYPE4` routine to print your status information how you would like to see it (by "status information," I only mean the information in the top right side of the status window, where the score/turn count would normally go). This makes it easy to provide other kinds of information in your status line, such as moods, health, or whatever else you can think of.

```
replace STATUSTYPE4
{
	local a
	select player.mood
		case 4 : a = "Happy"
		case 3 : a = "Bothered"
		case 2 : a = "Distraught"
		case 1 : a = "Absolutely Crushed"
	print a;
}
```

Any colors you use in `STATUSTYPE4` will be properly displayed, too.

## Room Descriptions

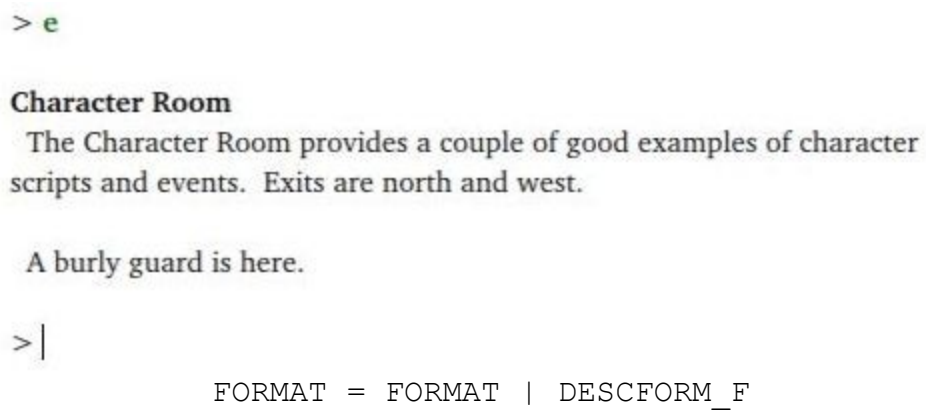
Roodylib also offers a variety of options for how room description text is presented. Like the original Hugo library, some of these settings are determined by the `FORMAT` global variable and whatever masks you apply to it, with a command like the following in `init` or `SetGlobalsandFillArrays`:

```
FORMAT = FORMAT | (mask constant)
```

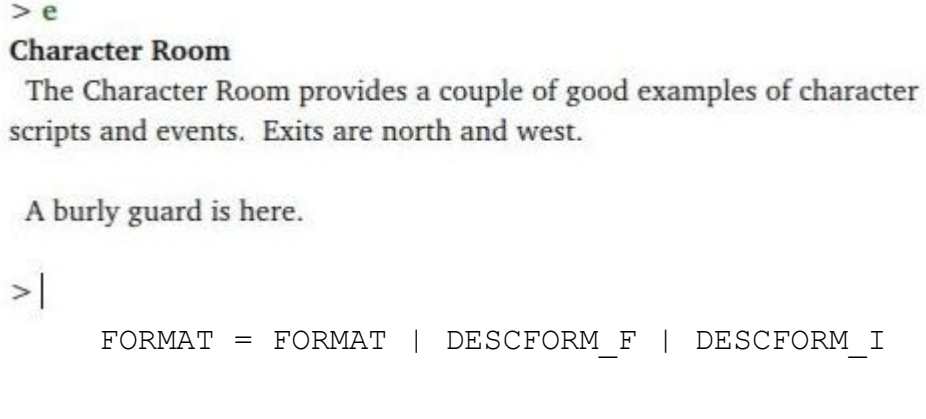
All of the available `FORMAT` masks are listed in the Hugo Book (and **hugolib.h**), but I think the ones most likely to be used by authors are:

| | |
|---|---|
| **LIST_F** | Contents of objects are given "tall" lists instead of listed in sentences (so, *Zork* style) |
| **NOINDENT_F** | If you disagree with Hugo's indentation style, this is a quick way to turn it off. |
| **DESCFORM_F** | This puts an extra new line in between a room's description and its contents. |

Roodylib slightly changes the behavior of how games that use that `LIST_F` mask look, but for the most part, you don't need to worry about any of that. Roodylib also adds a `DESCFORM_I` mask. If this is used, the `DescribePlace` routine does not automatically print a new line before a room description is printed.




```
FORMAT = FORMAT | DESCFORM_F
```




```
FORMAT = FORMAT | DESCFORM_F | DESCFORM_I
```

## PrintRoomName

Roodylib also now provides a routine for printing the location name in the current room description.

```
routine PrintRoomName(a)
{
	if not (FORMAT & DESCFORM_I)
		print ""
	else
		print newline
	Font(BOLD_ON)
	print capital a.name;

	! Print ", in <something>" if necessary
	if location = a and player not in a
	{
		if parent(player).prep
			print ", "; parent(player).prep; " ";
		elseif parent(player) is platform
			print ", "; ON_WORD; " ";
		else
			print ", "; IN_WORD; " ";
		print Art(parent(player))
	}
	print newline
	Font(BOLD_OFF)
}
```

Again, this is for ease of use. If authors want any different behavior (like added colors or more control over preposition usage), they won't have to dig too deep into DescribePlace figuring out how they should do it.

## "Relative Descriptions"

Roodylib has an option for special treatment when the player is inside a container in a room. To use it, just `#set USE_RELATIVE_DESCRIPTIONS` in your code before Roodylib is included.

> **Start Location, in the coffin**
> There is an exit to the east.
> A key is here inside the coffin.
> A couch, the ladder, the horse, and a portal are outside the coffin.

*example of "relative description" generated text*

**Start Location, in the coffin**
There is an exit to the east.
Inside the coffin is a key.
A couch, the ladder, the horse, and a portal are here.

*the same room without relative descriptions*

The above works automatically if the parent of the player is a container, but platforms are ignored by default (if the player is sitting on something like a chair, you wouldn't want everything else described as "off" the chair). Still, there may be a platform instance where you would want the "relative parent" behavior. To do this, first replace the `RelativeParent` routine to allow for the object you want it to work for:

```
replace RelativeParent(obj)
{
	if player not in location and parent(player) is container
		return true
	elseif parent(player) = monkey_bars
		return true
	else
		return false
}
```

*replacing RelativeParent*

And then you replace the `RelativeText` routine to print whatever text you want for objects that do or do not share the same parent as the object in question:

```
replace RelativeText(obj)
{
	if obj = location and player not in location
	{
		if parent(player) = monkey_bars
			print "off ";
		elseif parent(player) is container
			print "outside ";
	}
	elseif obj is container
		print "inside ";
	else
		print "on ";
	The(parent(player))
}
```

## Alternate Dark Room Behavior

Ok, not many games these days even have dark rooms, but somewhere along the way, I decided I didn't like the way dark rooms are handled in Hugo. Even though its behavior was based on classic games such as *Zork* and *Adventure*, I found it disorienting the way that dark rooms almost feel like non-rooms. I figured it'd be cool to make it look more room-like, so I added an option for this. To use it in your game, `#SET USE_DARK_ROOM` before **roodylib.h** is included.

**Outside a vault**
Kind of that 1930s, Bela Lugosi, graveyardy motif at work here. It's a pretty creepy place. Directly in front of you is the giant door to an even more giant vault. Above the door hangs a rusty sign.

> e

**Darkness**
It's pitch black in here. Stumbling around in the dark isn't such a hot idea: you're liable to be eaten by a grue.

>|

*with USE_DARK_ROOM*

**Outside a vault**
Kind of that 1930s, Bela Lugosi, graveyardy motif at work here. It's a pretty creepy place. Directly in front of you is the giant door to an even more giant vault. Above the door hangs a rusty sign.

> e
It's pitch black in here. Stumbling around in the dark isn't such a hot idea: you're liable to be eaten by a grue.

>|

*default behavior*

If you are using the `USE_DARK_ROOM` option and would like to configure any of its behavior, you can do the following:

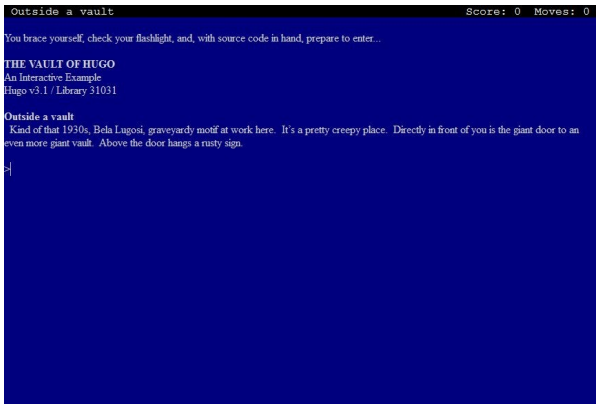
1. If you'd like to change the darkness "room" name, change `darkness.name` to the text of your choice ( `darkness.name = "OMG Can't See Anything"`) somewhere like `init` or `SetGlobalsAndFillArrays`. Alternatively, you could replace the darkness object and give it a new name that way.
2. If you'd like to change the rest of the text, as before, replace the `DarkWarning` routine:

```
replace DarkWarning
{
	PrintRoomName(darkness)
	Indent
	print CThe(player); " stumble"; MatchSubject(player); \
		" around in the dark."
}
```

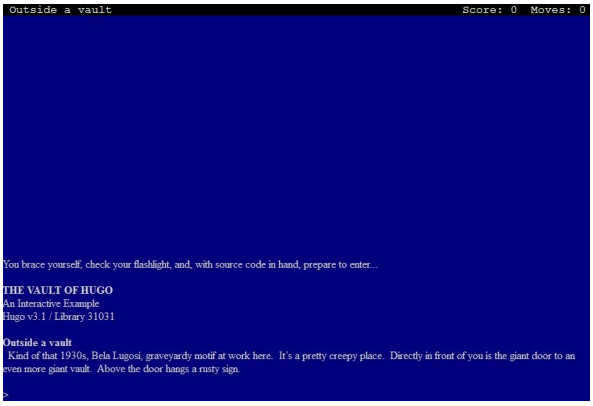
# Clearing the Screen

Roodylib has a few routines for making screen-clearing consistent and cool-looking. Personally, I prefer when IF games' text is drawn from the top of a window down. Roodylib tries to mimic this look by moving the cursor whenever the screen is cleared using one of its functions. Some routines like `PictureInText` pretty much depend on the cursor being at the bottom of the screen, though. In such a game, you'd want to force the game to always keep the cursor at the bottom. You do this by replacing the `LinesFromTop` routine.

```
replace LinesFromTop
{
	return display.windowlines
}
```




*default LinesFromTop value*




*replaced LinesFromTop*

Calling the `InitScreen` routine will completely clear the screen (getting rid of any existing windows), moving the cursor to wherever `LinesFromTop` determines it should be. The `ClearWindow` routine, though, only clears the current window (and then moves the cursor). This is for instances where you don't really need to redraw everything—just the current window.

# Accessibility

While it shouldn't affect most of you as authors, be aware that Roodylib has some accessibility features, like the ability to change the prompt to a "Your command…." phrase or to disallow screen-clearing (and forcing numerical menus as long as "newmenu.h" is used).

# Hugolib Object Class Improvements

I imagine one thing people will want to know is how Roodylib changes any object class behavior. Let's go cover some!

## Attachables

One thing that Roodylib does differently is it lists attachables held by the player *but attached to something in the room* in the room description. If the "new DescribePlace" system is on, it also changes when regular attachables are listed in the room description.

Roodylib has support for rollable objects— those that can be pushed from room to room. (I use the term "rollable" because I always think of the giant onion from *Beyond Zork*, but the concept applies to really anything that can be moved.) Roodylib's attachable code has been updated to accommodate such scenarios where an attachable is connected to a rollable object.

## Characters

In Roodylib, characters are automatically excluded from "all" commands (like **>GET ALL**). Additionally, while the Hugo library has always allowed for taking objects from friendly characters, in Roodylib, that gets its own message ("so-and-so allows you to take the <blank>"), whereas before it was just "Taken."

If you want to *disallow* taking items from friendly characters, stop it with a "parent(object) DoGet" before routine on the character object.

Roodylib's default also allows for **>GET ALL FROM <character>**. To disallow that, replace the `ExcludeFromAll` routine so it always returns true when the parent is living. You'll also want to provide `NewParseError` case 9 with a special message that checks that the parent is living and returns with something like "You'll have to specify one object at a time."

If you set the `LIST_CLOTHES_FIRST` switch, the player character will have worn clothing listed before other items when inventory is taken. For NPCs, worn items will also be listed first in descriptions if you add the following code to their objects:

```
      list_contents
            return ListClothesFirst(self)
```

`female_character` objects are now of class `character` (still with the female attribute, of course). Before, they were of the class `female_character` which made for extra work when checking if objects were characters.

Additionally, `player_character` objects are now of class `player_character` since that seemed like something an author might specifically want to check for.

# Checkheld Objects

This isn't so much an object class as it is an object-handling system. Normally, certain commands only work on held items (>**WEAR**, for instance). If `USE_CHECKHELD` is set, though, the game will first attempt to pick up the unheld item and then try to carry out the command. In the original Hugo library, it is advised to not use the checkheld system as it has some bugs. I *believe* I've fixed them for Roodylib, but it still needs lots of testing.

# Containers and Platforms

Containers and platforms are important to any game. Roodylib tries to add a couple things.

## Enterable Containers and Platforms

If the `SMART_PARENT_DIRECTIONS` flag is set, if the player is, say, sitting in a chair and tries to leave in a non-valid direction, the game responds with *"You can't go that way."* instead of *"You'll have to get up from the chair first."*

## Holding property

With the regular Hugo library, it is expected of authors to remember to add a `holding` property to containers or platforms to which children can be added. Since routines like `Acquire` recalculate the `holding` and `capacity` properties every time they're called, Roodylib uses a global variable whenever a `holding` property is missing. So, rejoice, it's no longer necessary.

# Doors

By default, locked doors in Roodylib are automatically unlocked when walked through (as long as the player has the applicable key). This can be turned off by setting `NO_AUTOMATIC_DOOR_UNLOCK`. Conversely, you can have the game act as if unlocked doors aren't even there (with no "(opening the door first)" text) by setting the `SKIP_DOORS` flag.

If a key for a locked door is given the `quiet` attribute, automatic door-unlocking won't work until the player has specifically used that key to unlock the door first.

# Rooms

If the `NEW_ROOMS` flag is set, the `room` object class is replaced with one with an extra property that will hold the counter value when a room is visited for the first time. This allows for consistent `initial_desc` behavior and other instances where the game can be thrown off by an >**UNDO** after the first turn in a room.

Also, if `NEW_ROOMS` is set, two additional routines are at your disposal. There is **`RoomTurnCount`**, which returns the number of moves the player has spent in the current room, and there is **`FirstVisit`**, which returns true if it's the player's first visit to the room. These exist so authors don't have to come up with their own time-keeping solutions when writing event-like functions using the room's `each_turn` property.

# Vehicles

The Hugo library makes assumptions about how vehicles can be exited. To make this more configurable, Roodylib replaces the vehicle class and relies on slightly different code.

A horse would have the following code to allow for >**DOWN** to get off the horse:

```
before
{
	parent(player) DoGo
	{
		if object = d_obj
			return object
		return false
	}
}
```

# Parsing

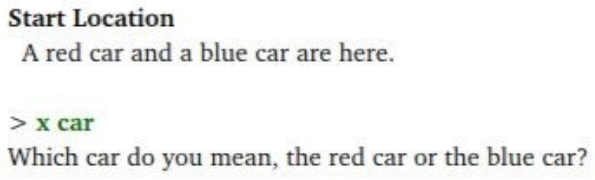
Having a well-implemented parser is one of the best ways to make your game seem polished. This section should help with that.

## extra_scenery

In the Hugo library, the `extra_scenery` property is available to give to rooms to hold words that will result in a *"You don't need to refer to that"* message when typed by the player. In Roodylib, you can also give the `extra_scenery` property to the player object, making those words always available. This is useful if your game happens to mention a body part or something that will otherwise not be referred to.

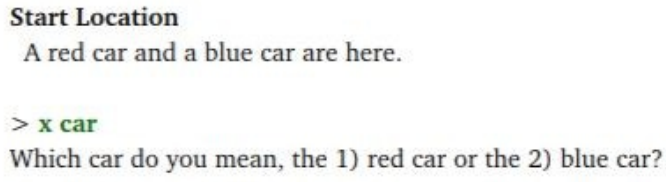
## Disambiguation Helper

Admittedly, the interactive fiction language Inform has had the most attention given to its development, so it's not much of a surprise that it has had many great ideas along the way. When I see ones I really like, I try to add them to Hugo. One such thing is additional parser help when disambiguating objects.




*traditional Hugo disambiguation*

Now, occasionally, situations arise where the adjectives and nouns for several of the objects being listed match and it's almost impossible for the player to choose the exact object he or she wants. To help with this, Roodylib also adds a numbering system to help out the player.




*Roodylib disambiguation*

The player can type "1" or "2" besides any of the adjectives. "Former," "latter," "first," and "second" are also accepted.

Roodylib will keep track of up to three objects to be disambiguated. If your game possibly has situations where even more might be needed, you can up the limit by declaring the `DISAMB_MAX` constant before Roodylib is included.

```
constant DISAMB_MAX 5
```

If, for some reason, you want to turn off the disambiguation helper completely, you can set the following before Roodylib is included:

```
#set NO_DISAMB_HELP
```

# Preparse

If your game has non-default verbs, it's likely that at some point, you'll need to modify certain player commands and make them play nice with what the game's grammar expects.

```
replace PreParse
{
	local i

	! Since "get off wing" or "exit wing" will cause a parser complaint
	! because the player isn't really "in" the wing, change either to
	! simply "exit" (i.e., to direct the library to out_to).
	!
	if (word[1] = "get", "climb") and word[2] = "off"
	{
		word[1] = "exit"
		DeleteWord(2)
	}
	if word[1] = "exit" and ObjWord(word[2], wing)
		DeleteWord(2)

	! Allow handing of, e.g., "ask girl about her mother", so that "her"
	! doesn't get mapped incorrectly
	!
	if word[1] = "ask", "tell"
	{
		for (i=2; i<=words and word[i]~=""; i++)
		{
			if word[i] = "his", "her", "your"
			{
				DeleteWord(i)
				break
			}
		}
	}
}
```

*PreParse replacement in Kent Tessman's Down*

## OrdersPreParse

Now, `PreParse` has always been in Hugo, but Roodylib adds a routine called `OrdersPreParse` specifically for parsing orders to characters. Here is a not-particularly-useful example:

```
!\ b is the word array element the command starts with and e is where it
ends \!
replace OrdersPreParse(b,e)
{
	if word[b] = "take" and word[(b+1)] = "break"
	{
		DeleteWord(b+1)
		word[b] = "wait"
		return true
	}
	return false
}
```

*changing "CHARACTER, TAKE A BREAK" to "CHARACTER, WAIT"*

## preparse_instructions objects

Additionally, Roodylib has a system in place for several `PreParse`-esque instructions to coexist peacefully. This is mainly because several of my Hugo library extensions use `PreParse` for various reasons; I wanted to save authors the time of having to copy and organize everything into one routine themselves. So, if you're writing a library extension that also uses `PreParse` instructions, make a `preparse_instructions` child instead!

```
! Among other things, Roodylib uses such an object to redraw the screen if it
! has changed
object parse_redraw
{
	in preparse_instructions
	type settings
	execute
	{
		if display.needs_repaint
		{
			if RepaintScreen
				RedrawScreen
		}
		return false
	}
}
```

*preparse_instructions object example*

As a general rule, have your code return true if the command needs to be reparsed and return false if everything is fine.

# Pronouns

At some point, I might try my hand at a deeper pronoun redesign, but in the meantime, you have the following updates at your disposal.

## AssignPronoun

`AssignPronoun` is from the standard Hugo library, but previously, it was tricky for authors as `Parse` would always reset the wanted pronoun without some specific hackery (to be precise, you had to set the `last_object` global to -1). While we're waiting for me to decide what future pronoun behavior should be, you can force pronoun setting by adding an extra `true` argument to its call.

```
AssignPronoun(<object getting a pronoun set to it>, true)
```

*changing a pronoun*

## ParsePronounFix

Roodylib has a `ParsePronounFix` routine which tries to find an applicable object or xobject from the previous command if the action in question doesn't apply to the current "it object". It is called by `Parse`. Just add whatever game pronoun-setting rules you want in your replaced version if you need more specific rules.

```
routine ParsePronounFix(count)
{
	local i
	select word[(count-1)]
		case "open","close": i = openable
		case "lock","unlock" : i = lockable
		case "wear" : i = clothing
		case "read","peruse" : i = readable
		case "activate","start","stop","deactivate" : i = switchable
		case "turn", "switch"
		{
			if word[(count+1)] = "off","on"
				i = switchable
		}
	if i and object is not i and it_obj = object
	{
		if xobject is i
			it_obj = xobject
	}
	elseif i and xobject is not i and it_obj = xobject
	{
		if object is i
			it_obj = object
	}
}
```

*ParsePronounFix*

## ExcludeFromPronouns

`ExcludeFromPronouns` limits what objects pronouns can be set to (I had an issue with pronouns occasionally being directed to direction objects, which I did not like). If you find pronouns being applied to objects you don't want, you can replace this and have it check for that, too.

```
routine ExcludeFromPronouns(obj)
{
	if obj = player:  return true
#ifclear NO_OBJLIB
	elseif obj.type = direction : return true
#endif
	return false
}
```

*ExcludeFromPronouns*

# Game Loop

I'm using this section to discuss things closely related to the process of each game turn that weren't already covered in "Parsing" (as that, of course, is also part of the game loop). Really, I had to make up *something* to call this section or else I'd really just be throwing a lot of information at you at once.

## main_instructions

Any object within the `main_instructions` object has its "execute" property run after each turn (called by the `Main_Calls` routine, which in turn is in the `main` routine in a Roodyized Hugo game shell). Certain library extensions are already set up to work this way, but if you are writing your own library extension, you might want to look at this example:

```
object footnotemain
{
#ifset _ROODYLIB_H
       type settings
       in main_instructions
       execute
       {
              FootnoteNotify
       }
#endif  ! _ROODYLIB_H
}
```

Having the above code in "footnotes.h" makes FootnoteNotify be called after every successful turn. In general, `Main_Calls` messages are good for the game's system messages that should be printed after, say, game events.

## NO_LOOK_TURNS

One not-often-implemented IF theory is that "look" actions (room descriptions, examining objects, etc.) should not use up a turn. Some people feel that looking should not take the same amount of game time as other actions and that it can become a frustration (especially in time-sensitive situations).

Setting `NO_LOOK_TURNS` in Hugo gives it this behavior, for the most part. >**LOOK UNDER OBJECT** still uses a turn as it implies an action along with looking.

```
#set NO_LOOK_TURNS
```

*setting NO_LOOK_TURNS*

# Scripting

Character scripting may not be used often in Hugo games in recent years, but when I took a look at it for Roodylib, I was dissatisfied with how looping scripts wasted a turn calling the `LoopScript` routine.

Roodylib replaces a couple routines so that adding a true value to a character script array that calls `&LoopScript` will restart the script on the same turn.

```
setscript[Script(northgoingzax, 2)] = &CharMove, n_obj,
                                      &LoopScript, true
```

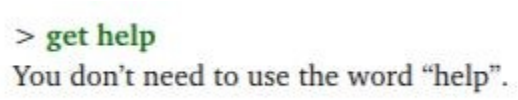
*the "northgoingzax" will move north every turn*

# Game Messages

Roodylib provides several new message-providing routines to help games look more polished. Also, adapting default messages to your game can be an important part of stylizing your game. This section covers those things.

## AMERICAN_ENGLISH

I've had at least one betatester complain about default error messages that follow non-US rules when it comes to quotation marks and full stops.




*default behavior*

Set `AMERICAN_ENGLISH` to have quotation marks in error messages follow American rules.

```
#set AMERICAN_ENGLISH
```

*turning on AMERICAN_ENGLISH*

## AUTOMATIC_EXAMINE

I'm a fan of games that give convenience to players—ideally, without spoonfeeding the entire experience to them. Michael Gentry's *Anchorhead* did a nice thing where unexamined objects automatically had their descriptions given when picked up the first time. I imagine it has shown up in other games since, but either way, I figured I'd make it easy for Hugo authors to have this behavior at the flick of a switch.

```
#set AUTOMATIC_EXAMINE
```

*turning on AUTOMATIC_EXAMINE*

## CoolPause

In-game pauses for narrative effect have increased a ton in modern IF games since the early 2000s. It always irks me when a game is waiting for a keypress but doesn't explicitly tell the player it is doing so. I created the `CoolPause` routine as a remedy for this. First off, in interpreters that support it, it uses a technique to hide the cursor so the screen just *looks nicer*. Secondly, it provides a *"press key to continue"* message to be modified to an author's whim.

```
CoolPause(pausetext)
```

*how to call*

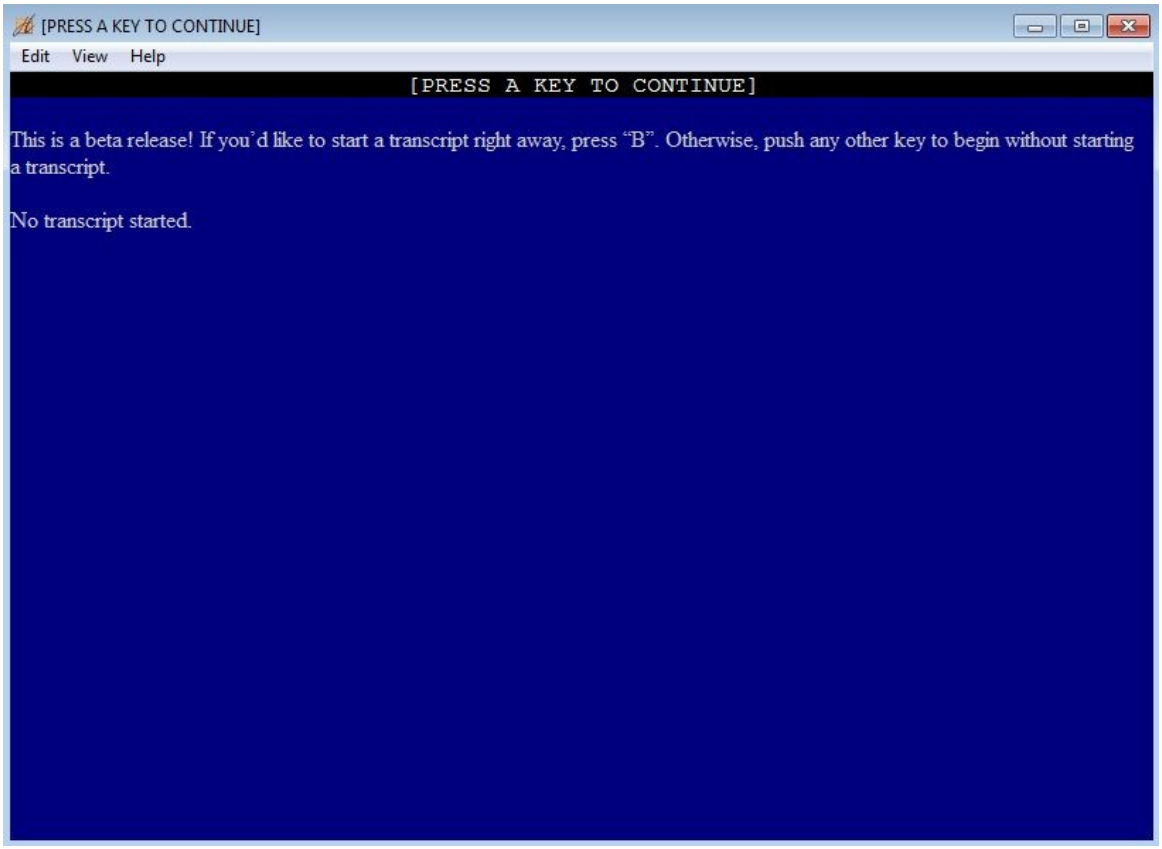
The `pausetext` argument is the string to be printed if you want a quick-and-easy non-default message (without replacing the `&CoolPause` response in the `RlibMessage` routine).

### TopPause

`TopPause` is a similar routine but differs in that it puts the pause text in the status bar so it doesn't break up the flow of the main game text.

```
TopPause(pausetext)
```

*how to call*




*TopPause() example*

# RoomSounds(location)

For a long time, I had difficulty coming up with a clean method of providing "room sounds" (where >LISTEN without an object would describe the room, if the location had any sounds to provide).

I eventually decided that a helper routine could solve the problem:

```
routine RoomSounds(obj)
{
	if verbroutine = &DoListen and not object
		return location
	else
		return false
}
```

In practice, this means adding this code to any room that has sounds in it.

```
	before
	{
		RoomSounds(location)
		{
			"No sounds but the wind." ! Or whatever the sounds are
		}
	}
```

Now, listening in that room without an object will provide the expected answer.

# DoVersion / GameTitle

I had a request at one point that Roodylib provide a default >**VERSION** response (which was a good idea since providing them largely reinventing-the-wheel for each game). If your game already provides a >**VERSION** response, just `#set NO_VERSION` to turn Roodylib's responses off.

```
routine DoVersion
{
	print GameTitle
#if defined BLURB
	 print BLURB ! "An Interactive Blahblahblah"
#endif
#ifclear NO_COPYRIGHT
	Copyright
#endif
	PrintBanner
	ReleaseAndSerialNumber
#if defined IFID
	print "IFID: "; IFID
#endif
#ifset BETA
	BetaNotes
#endif
#ifset DEMO_VERSION
	DemoNotes
#endif
	OtherNotes
}

! Roody's note: I changed TITLECOLOR to a global. Set it to something else in
! SetGlobalsAndFillArrays if you'd like to provide a special title color.

global TITLECOLOR = DEF_FOREGROUND

routine GameTitle
{
	color TITLECOLOR
	Font(BOLD_ON | ITALIC_OFF)
	print GAME_TITLE;
	Font(BOLD_OFF | ITALIC_OFF)
	color TEXTCOLOR
#ifset DEMO_VERSION
	print "\B (demo version)\b";
#endif
#ifset HUGOFIX
	print "\I (HugoFix Debugging Suite Enabled)\i";
#endif
}
```

*Replace these routines if you'd like to change the DoVersion text in any way*

# Standard Message Replacement

Roodylib continues the Hugo standard library's method of message replacement. Since it's such a useful thing to understand, I thought I'd give it a quick overview here. While you can always replace an entire routine in Hugo to make your changes, Hugo makes this simpler by keeping game messages in their own routine. If you want to change the *"Taken."* response when an object is picked up, you don't replace `DoGet`, you add a special case for `&DoGet` in `NewVMessages`.

Roodylib adds plenty of its own messages, too, kept in `RlibMessage` and `RlibOMessage` (for object class associated messages). In some instances, Roodylib adds message-routine calls to routines where there were none previously. In other cases, messages replace previous Hugo library messages, and some messages are entirely new. For the most part, you'll have to check the applicable routine's code to see if `RlibMessage`, `RlibOMessage`, `Message`, or `OMessage` is being called.

```
replace NewRlibMessages(r, num, a, b)
{
   select r
            case &DoHit : "You only hit grooves and mad beats."
            case else : return false
   return true ! this line is only reached if we replaced something
}
```

*example of Roodylib message replacement*

# Resources

Roodylib now has a few additional routines for checking for the existence of resources. This way, if your multimedia-enriched game somehow gets distributed without its resource files, the game can check for the resource and then behave accordingly.

## CheckResourceMusic

`CheckResourceMusic` checks for the existence of a particular song. If the interpreter's op code functionality supports it, it will check without playing the song. Otherwise, it attempts to play the song for a moment at volume 0 and returns whether the attempt was successful.

```
CheckResourceMusic(<resource file name>,<song file name>)
```

*how to call*

## CheckResourceSound

`CheckResourceSound` does the exact same thing for sound files.

```
CheckResourceSound(<resource file name>,<sound file name>)
```

*how to call*

## CheckResourceGraphic

`CheckResourceGraphic` only works with interpreters that support the op code check method, as it's kind of impossible to design a graphic-checking routine that will work for all scenarios.

```
CheckResourceGraphic(<resource file name>,<graphic file name>)
```

*how to call*

## Resource Treatment in Gargoyle

The multi-interpreter Gargoyle, while very pretty when it comes to games that don't rely upon graphics, music, or even text orientation, does this ugly thing where it rips each graphic, music, or sound file from the resource file and clogs up the game directory. Roodylib, by default, doesn't allow resources to be used at all with Gargoyle. If you want to be nice, you can set the `allow_gargoyle` global variable to `true`. Just remember that using the `PictureInText` routine will not work in Gargoyle (and `LoadPicture` will only work in the main window).

# HugoFix

HugoFix, an in-game suite of debugging commands, is immensely useful to any Hugo author. Besides the additional features we're about to get into, Roodylib adds a pregame splash screen for turning on different kinds of game monitoring before the game even begins. We're going to go over a couple of the newer features, but there are some we won't cover right now (type *$?* in-game to get the full list)

## OrganizeTree

Roodylib uses a lot of extra objects to keep track of settings and such, and the object tree can get to be somewhat of a mess and an eyesore. When HugoFix is turned on, at the beginning of the game, non-*game* objects are moved to applicably named objects so all of your rooms and game objects are all together, for the most part.

Roodylib also replaces the `DrawBranch` routine so things like display windows and fuses are easier to keep track of.

```
>$ot 0
[Skipped object numbers are replaced objects.]

[0] nothing
[1] (display)
[35] (audio)
[36] (replaced_objects)
[37] (object_classes)
. . [2] (fuse)
. . [3] (daemon)
. . [4] (room)
. . [13] (direction)
. . . . [14] north
. . . . [15] northeast
. . . . [16] east
. . . . [17] southeast
. . . . [18] south
. . . . [19] southwest
. . . . [20] west
. . . . [21] northwest
. . . . [22] above
. . . . [23] below
. . . . [24] in
. . . . [25] out
. . [27] (scenery)
. . [28] (component)
. . [32] (attachable)
. . [54] (character)
. . [55] door
. . [56] (female_character)
. . [57] (player_character)
. . [58] (self_class)
. . . . [59] himself
. . . . [60] herself
. . . . [61] itself
. . . . [62] themselves
. . [63] (vehicle)
. . [64] (plural_class)
. . . . [201] garbage bags
. . . . [263] cards
. . [65] (identical_class)
. . [66] <replaced object>
. . [82] (menu_category)
. . [83] (option)
. . [84] (hint_option)
```

*example HugoFix object tree listing*

## Recording Playback Helper

I use Hugo's playback feature quite a lot when testing code. Since saved games won't work over different compilations, there are just times when you need to repeat a lot of steps to get to a scene that you are testing. I created the recording playback helper commands to help speed up this process. Typing >**$rp** in a game with HugoFix on results in *"Keep waiting?"* prompts to be skipped. It also skips in-game pauses in anything that uses the `HiddenPause` routine.

# Finishing Touches

Hooray, you're almost done with your game!  Roodylib can help with that, too!

## Ending the Game

Just properly ending the game can involve several steps.

### CallFinish

Sometimes it's easy for new authors or authors who haven't recently looked at the Hugo Book (or example code) to forget that to actually end the game, no routine is called.  You just set the `endflag` global to the value you want and the game calls `EndGame` and prints the applicable ending text as determined by `PrintEndGame`.

I don't know if this will actually help anybody, but I provided Roodylib with a routine for ending the game just for the people who can't deny the part of themselves that says calling a routine just *feels right*.

```
CallFinish(<endflag value>)
```

*ending the game with CallFinish*

### SpecialKey / SpecialRoutine

I've always been a fan of games with additional options when a game is won (like >**AMUSING** things to try).  Hugo didn't make it easy to provide these options without replacing `EndGame` completely, so I rewrote it to call a couple extra routines for such situations.

First off, `SpecialKey` looks for the proper `endflag` / word combination for providing the extra option.

```
replace SpecialKey(end_type)
{
	if (word[1] = "amusing","a") and end_type = 1
		return word[1]
	return 0
}
```

*example SpecialKey replacement*

Then you replace `SpecialRoutine` to do whatever you want when the player selects that choice under the proper conditions.

```
replace SpecialRoutine(end_type)
{
	ShowPage(amusing_list) ! Example of using newmenu's ShowPage routine
	! alternatively, you could just print the AMUSING list right here
}
```

*example SpecialRoutine replacement*

### QuitGameText

After the player has decided he or she wants to quit the game, Roodylib provides a *"Thanks for playing"* message and waits for a keypress before letting the window close. I thought this was a cute effect in some Infocom games and figured it'd suit Hugo well, too.

If you don't like it, you can just replace `QuitGameText` with an empty routine!

## BETA system

Even if you don't think so, your game probably needs betatesting! Setting `BETA` in your code before Roodylib is included provides a splash screen to compiled games asking betatesters if they'd like to start a transcript before the game has even begun. It also reminds them that prefacing their commands with an asterisk will be interpreted as a note to the author.

```
#set BETA
```

*turning on the BETA system*

# Other Features

Roodylib has plenty of helpful routines not easily thrown under one categorization. We'll talk about some of them here.

## The New Opcode System

Nikos Chantziaras has been doing a wonderful job of making it a much more pleasurable experience to play Hugo games on Linux and MacOS computers; on top of that, all ports (including Windows) are feature-complete and improve upon the original interpreters in several ways. At some point, I'll probably include a section on making settings files to distribute along with a game to ensure it has the presentation you want. Right now, though, I'm going to talk about the new opcode system.

Nikos designed a clever way to use Hugo's configuration file system as a way to talk to Hugor and provided several opcode values for specific behaviors (with probably more coming in the future).

Roodylib provides a `opcodeterp` object that it automatically gives the attribute switchedon if it detects that the interpreter is opcode-capable. If your code has some features that depend on opcode functionality, you can test for it like this:

```
if opcodeterp is switchedon
```

Additionally, at the start of every session, Roodylib checks the entirety of known opcodes and marks each available one as `switchedon`, so you can check the availability of a particular opcode with something like:

```
if open_url is switchedon
```

### Using the Opcodes

For all of the opcodes, one calls the following routine:

```
ExecOpcode(opcode_file, str)
```

Roodylib provides several opcode objects to be used in the above routine to save authors the trouble of looking up opcode values. Additionally, some opcodes require a secondary string argument. The following list is not complete; I'm skipping some of the ones that I think are only useful for Roodylib behind-the-scenes (you can go look at the opcode section in "roodylib.h" for more information). Also, a couple will be discussed in the next section.

**Roodylib opcode objects**

| **Opcode** | Description | Example: |
|---|---|---|
| **open_url** | Opens the secondary string argument in the default web browser. | *ExecOpcode(open_url, "http://notdeadhugo.blogspot.com")* |
| **fullscreen** | Changes interpreter to fullscreen | *ExecOpcode(fullscreen)* |
| **windowed** | Changes interpreter to windowed mode | *ExecOpCode(windowed)* |
| **clipboard** | Copies the secondary string argument to the clipboard. | *ExecOpcode(clipboard, "roodyyogurt@gmail.com")* |
| **is_fullscreen** | Returns true if the interpreter is currently in fullscreen mode | *ret = ExecOpCode(is_fullscreen)* |

## Opcode object classes

Some opcodes can be used several ways, and I find it most useful to define an object class from which authors can define certain behaviors.

### *fade_screen*

As one might guess, the `fade_screen` opcode affects the transparency of all text on the screen. With it, it's possible to make text fade in and out as you wish. Roodylib provides one example:

```
fade_screen full_opacity "restore full opacity opcode"
{
	in fade_screen ! This isn't really necessary
	block 1 ! true if the fade should stop game code execution
		!  false if it should run in the backgound
	duration 1 ! duration of fade, in milliseconds
	start_opacity 255 ! beginning opacity of fade (0 = completely faded,
!			255 = full opacity
!		       -9999 = whatever current opacity is)

	end_opacity 255 ! opacity at end of fade
}
```

The above is sort of an "un-fade." Roodylib calls it automatically after game restarts or loads, so that if a game is currently "mid-fade", it doesn't get stuck in a situation where there is unreadable text. Anyhow, using those guidelines, you can create your own `fade_screen` objects so your fades can be as long or short as you'd like. Then you can call them like such:

```
ExecOpcode(full_opacity)
```

### *set_color*

Hugor now lets you define additional colors to be used in games, well beyond the 16 that we have made due with in the past. First, make a set_color object:

```
set_color burnt_sienna "burnt sienna"
{
	rgb 233 116 81	! the rgb values
	alpha_value 255	! the opacity, from 0-255
}
```

You can now apply this color to any of the color numbers between 100 and 254.

```
ExecOpCode(burnt_sienna,100)	! applies burnt sienna to 100
```

Finally, your game can check if the color opcode is supported and apply colors accordingly:

```
if set_color is switchedon
	color 100, BLACK	! burnt sienna on a black background
else
	color WHITE, BLACK
```

# Contact and Future Additions

To be honest, I did not cover *everything* yet in this release of the Roodylib documentation. At some point, I'd like to cover the following, too:

- "settings" objects and explanation of word array saving
- HiddenPause, GetKeyPress
- An explanation on how `SpeakTo` is basically a parsing routine and the steps Roodylib takes to improve its functionality

In the meantime, if you have questions about these or any other things, feel free to e-mail me at roodyyogurt@gmail.com or post a question at any of the following forums!

https://www.intfic.com/

https://intfiction.org/c/technical-development/development-systems/56

http://www.joltcountry.com/phpBB3/viewforum.php?f=8

(Let it be known that I check joltcountry.com more regularly than the others.)