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 its 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, and Juhana Leinonen.

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 it wherever.

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 also 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.

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.

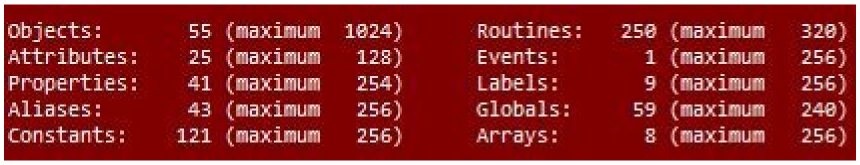
Updating an Older Game to Roodylib

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

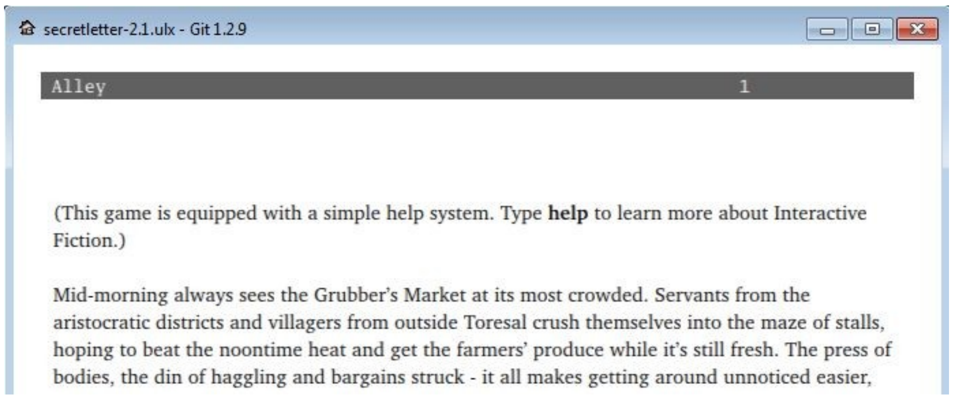
Depending on your game, you may need to change other limits as well (all are covered in the Hugo Book). Basically, if the compiler complains that you have gone over the limit for any particular thing, just keep raising the number of the max allowed until it works!

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. I think the worst offenders are scoreless games with ugly hanging turn-counters like this:



(This game is equipped with a simple help system. Type **help** to learn more about Interactive Fiction.)

Mid-morning always sees the Grubber's Market at its most crowded. Servants from the aristocratic districts and villagers from outside Toresal crush themselves into the maze of stalls, hoping to beat the noontime heat and get the farmers' produce while it's still fresh. The press of bodies, the din of haggling and bargains struck - it all makes getting around unnoticed easier,

I have to think that an unlabeled number like that makes no sense to anyone new to IF, and I don't think it's really a step forward in authors' attempts to get away from the game aspect of IF. Most importantly, *c'mon*, those games make absolutely no effort to position the score counter 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
```

Custom Status Lines

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. 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.

Expert Status Line Configuration

Sometimes, you might need to support completely different status line behaviors all in one game. For instance, I had to design a way for the Automap Hugo extension (which draws simple ASCII maps in the status line in Glk interpreters) to peacefully coexist with the NewConverse extension (which lists conversation options in the status window), on top of doing, ya know, regular status line stuff. To this end, I created a printstatuslib object, which Roodylib checks for children, using their find_height and status_override properties to determine which instructions should be followed on any given turn.

I won't go into the specifics of the system just now (it is all somewhat documented in **roodylib.h**), but here is an example printstatuslib object:

```
object mapwindow
{
	in printstatuslib
	find_height
	{
		return (call &FindMapHeight)
	}
	draw_window
	{
		return (call &DrawMapWindow)
	}
	status_override 0
}
```

Room Descriptions

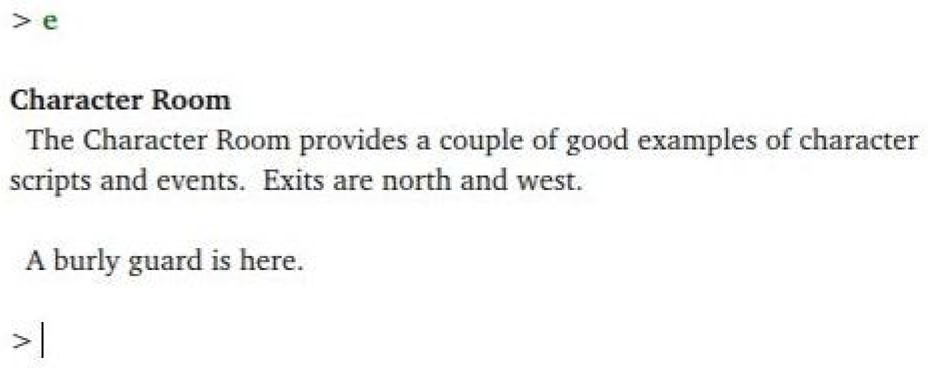
Roodylib also offers a variety of options for how room description text is presented. Like the original Hugo library, some of these settings are determined by the FORMAT global variable and whatever masks you apply to it, with a command like the following in `init` or `SetGlobalsandFillArrays`:

```
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:

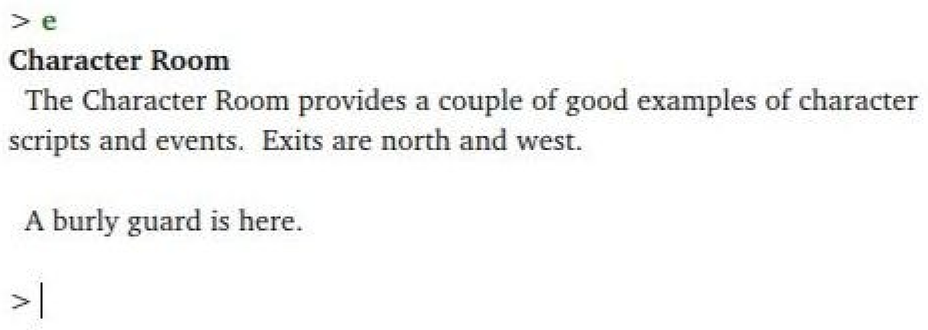
LIST_F	Contents of objects are given “tall” lists instead of listed in sentences (so, *Zork* style)
NOINDENT_F	If you disagree with Hugo’s indentation style, this is a quick way to turn it off.
DESCFORM_F	This puts an extra new line in between a room’s description and its contents.

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.



```
> e

Character Room
  The Character Room provides a couple of good examples of character
scripts and events.  Exits are north and west.

  A burly guard is here.

>
```

FORMAT = FORMAT | DESCFORM_F

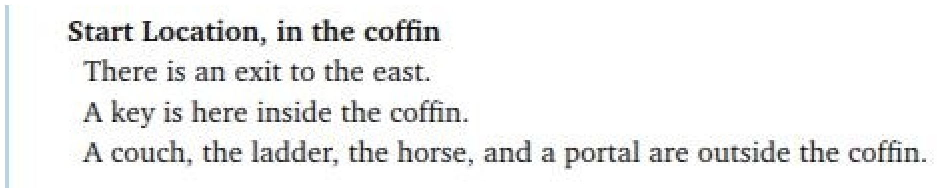


```
> e
Character Room
  The Character Room provides a couple of good examples of character
scripts and events.  Exits are north and west.

  A burly guard is here.

>
```

FORMAT = FORMAT | DESCFORM_F | DESCFORM_I

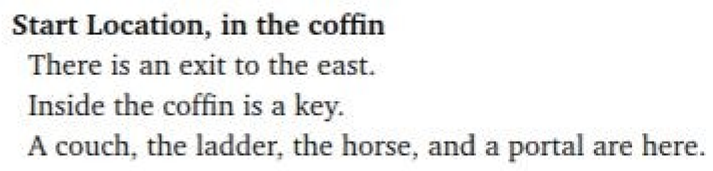
“Relative Descriptions”

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
}
```

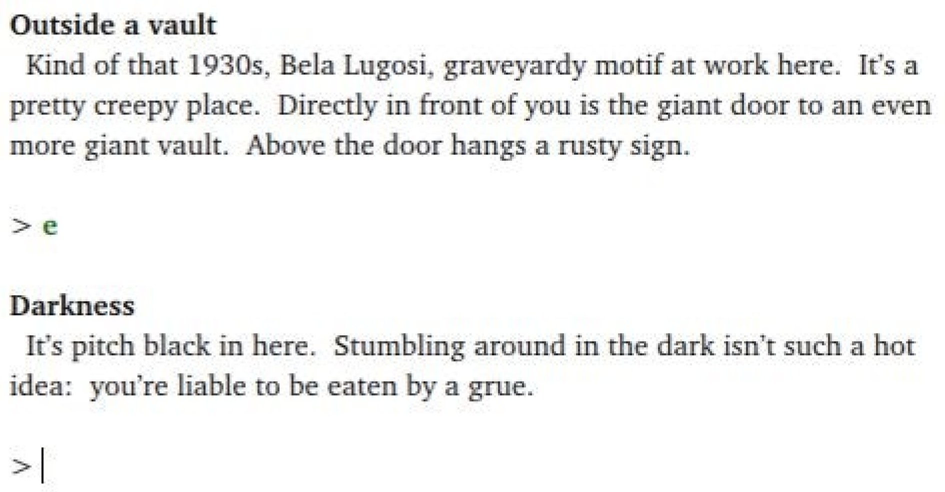
replacing RelativeParent

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))
}
```

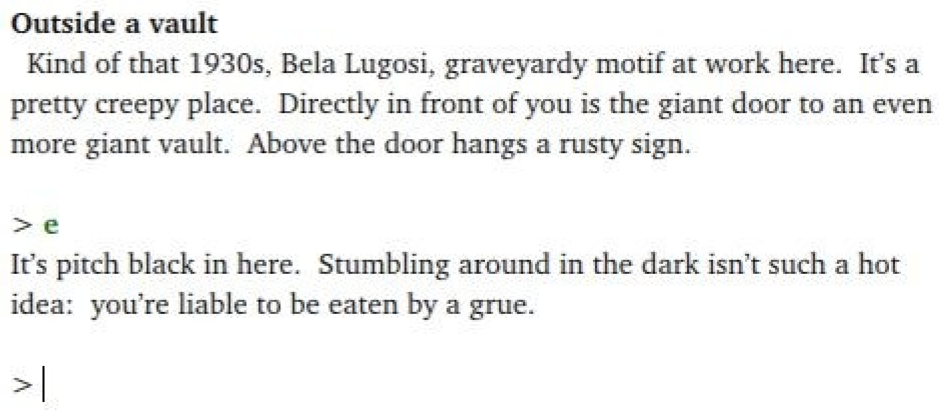
Alternate Dark Room Behavior

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:

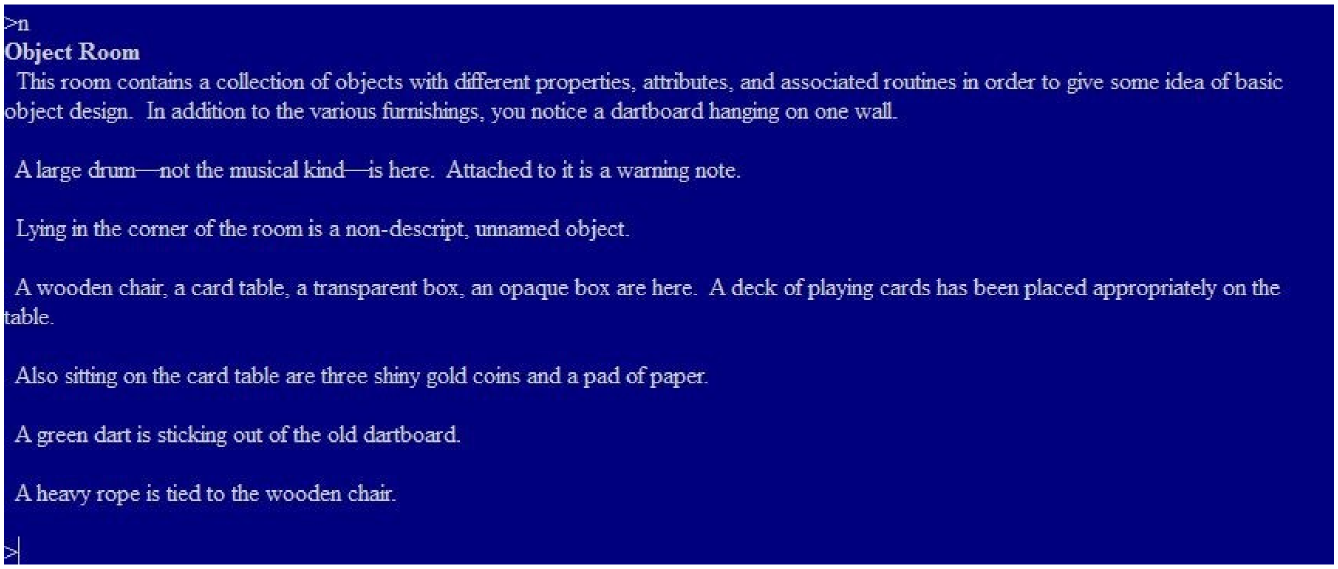
1. If you'd like to change the darkness "room" name, change `darkness.name` to the text of your choice (`darkness.name = "OMG Can't See Anything"`) somewhere like `init` or `SetGlobalsAndFillArrays`. Alternatively, you could replace the darkness object and give it a new name that way.

2. If you'd like to change the rest of the text, as before, replace the `DarkWarning` routine:

```
replace DarkWarning
{
	RLibMessage(&DescribePlace,1,darkness)
	Indent
	"[New text here]"
}
```

NEW_DESCRIBEPLACE

Besides everything already mentioned, one can configure room descriptions even further if he or she uses the `NEW_DESCRIBEPLACE` flag ("`#set NEW_DESCRIBEPLACE`"). For one thing, it allows the usage of the new `DESCFORM_D constant` ("`FORMAT = FORMAT | DESCFORM_D`"). If set, room descriptions get "double space" treatment, having an extra line in between every grouping of objects.



```
>n
Object Room
 This room contains a collection of objects with different properties, attributes, and associated routines in order to give some idea of basic
object design.  In addition to the various furnishings, you notice a dartboard hanging on one wall.

 A large drum—not the musical kind—is here.  Attached to it is a warning note.

 Lying in the corner of the room is a non-descript, unnamed object.

 A wooden chair, a card table, a transparent box, an opaque box are here.  A deck of playing cards has been placed appropriately on the
table.

 Also sitting on the card table are three shiny gold coins and a pad of paper.

 A green dart is sticking out of the old dartboard.

 A heavy rope is tied to the wooden chair.

>
```

DESCFORM_D in action

Another thing you can do with `NEW_DESCRIBEPLACE` is actually change the order in which things are listed. This feature was inspired by Robb Sherwin's games where NPCs never have `short_desc`'s (and it's nice to give them a priority over other objects in the room). Roodylib's `DescribePlace` lists objects in the following order:

1. Contents of the parent of the player (if he or she is not in the location itself) and contents of scenery items (as they were most likely mentioned in the room's `long_desc`)
2. Characters with descriptions
3. Characters without descriptions
4. Objects with descriptions

5. Objects without descriptions
6. Attachables in the room (and what they are connected to)
7. Attachables held by the player (but attached to something in the room)

You can change the order in which these are listed by rearranging the values in the `DescribePlaceArray` array:

```
DescribePlaceArray[0] = &ParentofPlayerScenery, \
       &CharsWithDescs, &CharsWithoutDescs, &ObjsWithDescs, \
       &ObjsWithoutDescs, &AttachablesChildren, &ListHeldAttachables
```

(Usually, you would do this in the `init` or `SetGlobalsAndFillArrays` routines, but you could do this mid-game, too, if a particularly room has a different behavior from the rest.)

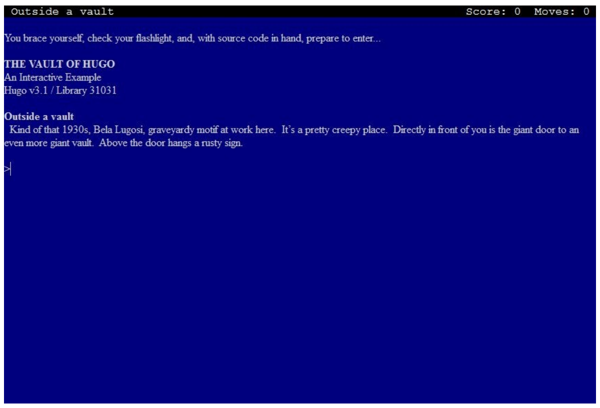
You can even add your own routines for listing objects based on other rules, but if it requires more array elements, you'll have to define the `DESCRIBEPLACE_ELEMENTS` constant before **roodylib.h** is included.

```
constant DESCRIBEPLACE_ELEMENTS 8 ! adds one extra array element
```

Clearing the Screen

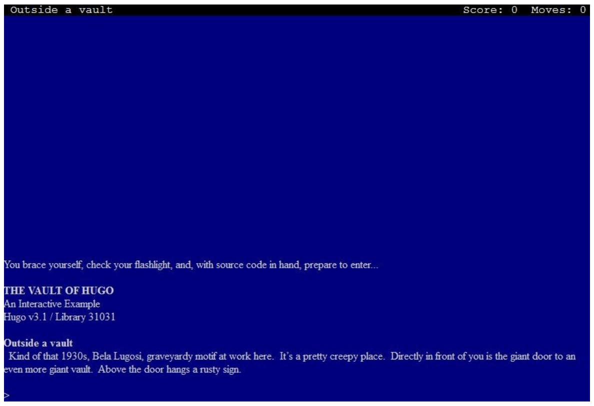
Roodylib has a few routines for making screen-clearing consistent and cool-looking. Personally, I prefer when IF games' text is drawn from the top of a window down. Roodylib tries to mimic this look by moving the cursor whenever the screen is cleared using one of its functions. Some routines like `PictureInText` 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
}
```



```
 Outside a vault                                   Score: 0  Moves: 0

You brace yourself, check your flashlight, and, with source code in hand, prepare to enter...

THE VAULT OF HUGO
An Interactive Example
Hugo v3.1 / Library 31031

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.

>
```

default LinesFromTop value



```
 Outside a vault                                   Score: 0  Moves: 0

You brace yourself, check your flashlight, and, with source code in hand, prepare to enter...

THE VAULT OF HUGO
An Interactive Example
Hugo v3.1 / Library 31031

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.

>
```

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.

"Repainting" the Screen

Adjusting the interpreter window mid-game can cause ugliness (depending on the interpreter being used). If Hugo detects a screen size change, Roodylib automatically clears and redraws the screen at the next turn, using the routine `RedrawScreen`:

```
routine RedrawScreen
{
	! if the screen size has changed, we'll clear the screen,
	! print the player's command, and redraw the status line before
	! proceeding to interpret the command

	InitScreen
	PrintStatusLine
	ShowCommand

}
```

If your game has other windows to re-draw, you'll want to replace `RedrawScreen` and make sure those are taken care of, too.

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).

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 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)
```

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

As important as containers and platforms are to any game, Roodylib has several optional ways of using them!

Emptying Containers

Roodylib has additional object classes so containers can empty in one of four different ways (held objects emptying to the room, held objects emptying to the player, unheld objects emptying to the room, unheld objects emptying to the player). This entails a surprising amount of grammar tinkering! Anyhow, to use this system, `#set NEW_EMPTY` in your code and have your container object inherit the applicable behavior (`unheld_to_player`, `held_to_player`, `held_to_ground`, or `no_empty`).

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.

Supercontainers

The supercontainer class (for those objects that are both containers **and** platforms) is built into Roodylib (just have to `#set USE_SUPERCONTAINER` in your code). It even improves upon the original code by checking for supercontainers in a couple places the original code missed.

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.

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.

Routine Grammar Helper

In the past, I've occasionally clashed with Hugo's grammar tokens, finding them too strict for specific scenarios. For instance, I wanted parsing success to be based on on object's `type` property, not its attribute, and in those cases, I wanted the game to disallow inapplicable phrases just like other disallowed-by-grammar instances. Alternatively, there might be a case where you want a verb's acceptance to rely on *multiple* attributes.

In any case, the routine grammar helper allows for these things.

Let's say you wanted >**SHOOT BADGUY WITH (object)** to only work with guns. First you would add the grammar, using a routine grammar token:

```
verb "shoot"
	* object "with"/"at" (CheckGun)        DoShoot

!\ (ok, this isn't going to be a working example of firing a gun since
there is really so many grammar phrases to account for) \!
```

Then you write the applicable routine:

```
routine CheckGun(obj)
{
	TOKEN = HELD_T ! (make sure the player is holding the gun)
	if CheckObject(obj)  ! checks to make sure object is held
	{
		if obj.type = gun   ! if it's a gun, we'll accept the command
			return true
		else
			ParseError(12, obj) ! "you can't do that with that."
	}
	return false  !  don't accept the command
}
```

AnythingTokenCheck

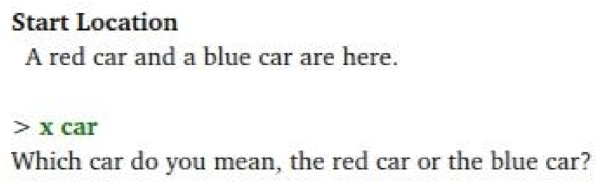
Unfortunately, the above example still runs into problems, as routine grammar is treated like an `anything` token, so without some help from us, a command like >**SHOOT BAD GUY WITH**

GUN will give a disambiguation message listing every gun in the game (*"Which gun did you mean, the revolver or the Uzi?"*). We can fix this by replacing the `AnythingTokenCheck` routine (called by `FindObject`) so all objects not in scope aren't even considered.

```
replace AnythingTokenCheck(obj,objloc)
{
      local failed_check
      select verbroutine
#ifclear NO_VERBS
            case &DoGet
            {
                  if obj in player
                        failed_check = true
            }
            case &DoShoot
            {
                  if not parent(obj) or not FindObject(parent(obj),location)
                        failed_check = true
            }
#endif ! ifclear NO_VERBS
            case else: failed_check = false
      return (not failed_check) ! return false if it failed
}
```

Disambiguation Helper

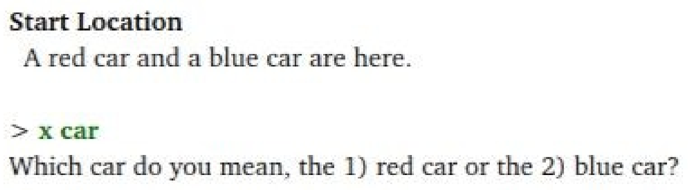
Admittedly, the interactive fiction language Inform has had the most attention given to its development, so it's not much of a surprise that it has had some great ideas along the way. When I see one I really like, I try to add it to Hugo, ha. One such thing is additional parser help when disambiguating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


Start Location
A red car and a blue car are here.

> **x car**
Which car do you mean, the red car or the blue car?

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.


Start Location
A red car and a blue car are here.

> **x car**
Which car do you mean, the 1) red car or the 2) blue car?

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 some 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

To be honest, I feel that Hugo needs a redesign as far as pronouns are concerned. I'll get to that eventually. In the meantime, you have the following updates at your disposal!

AssignPronoun

`AssignPronoun` is from the standard Hugo library, but previously, it wasn't very useful to 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

SetPronouns

I believe there are instances where pronouns should be set to the `xobject` instead of the `object`. For instance, >**PUT BOWL IN MICROWAVE** should probably set the microwave to "it" instead of the bowl, as >**CLOSE IT** should close the microwave.

I haven't committed to this theory in Roodylib yet, but the groundwork is already there with the included `SetPronouns` routine. It is called by `Perform`. Just add whatever game pronoun-setting rules you want.

```
replace SetPronouns
{
	if routine = &DoPutIn and xobject
		AssignPronoun(xobject)
	else
		AssignPronoun(object)
}
```

example SetPronouns code

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.

NEW_FUSE

Roodylib's `NEW_FUSE` system has fuses that determine timers by the game counter so calling them multiple times in one turn does not result in different behavior. The point of all of this is so that fuse/daemon text can still run after an >**UNDO** or >**RESTORE**; authors should just be aware that in those cases, the fuses/daemons *are* called an additional time automatically so their code should support that.

```
#set NEW_FUSE
```

to set NEW_FUSE

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

react_before/react_after for "scope objects"

Previously, Hugo only executed `react_before` and `react_after` properties for the player, location, and direct children of the location. It's now possible to also check those for scope objects (objects that are in scope because of a `found_in` or `in_scope` property). Be aware that this does **not** apply to components and the scope objects must have *nothing* as their actual parent in the object tree.

Also, since this method checks every scope object in the game which could be a lot of useless steps in a game that doesn't need it, it has to be explicitly turned on.

```
#set USE_SCOPE_REACT
```

turning on react_before/react_after for scope objects

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
```

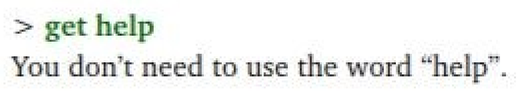
the "northgoingzax" will move north every turn

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.



```
> get help
You don't need to use the word "help".
```

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

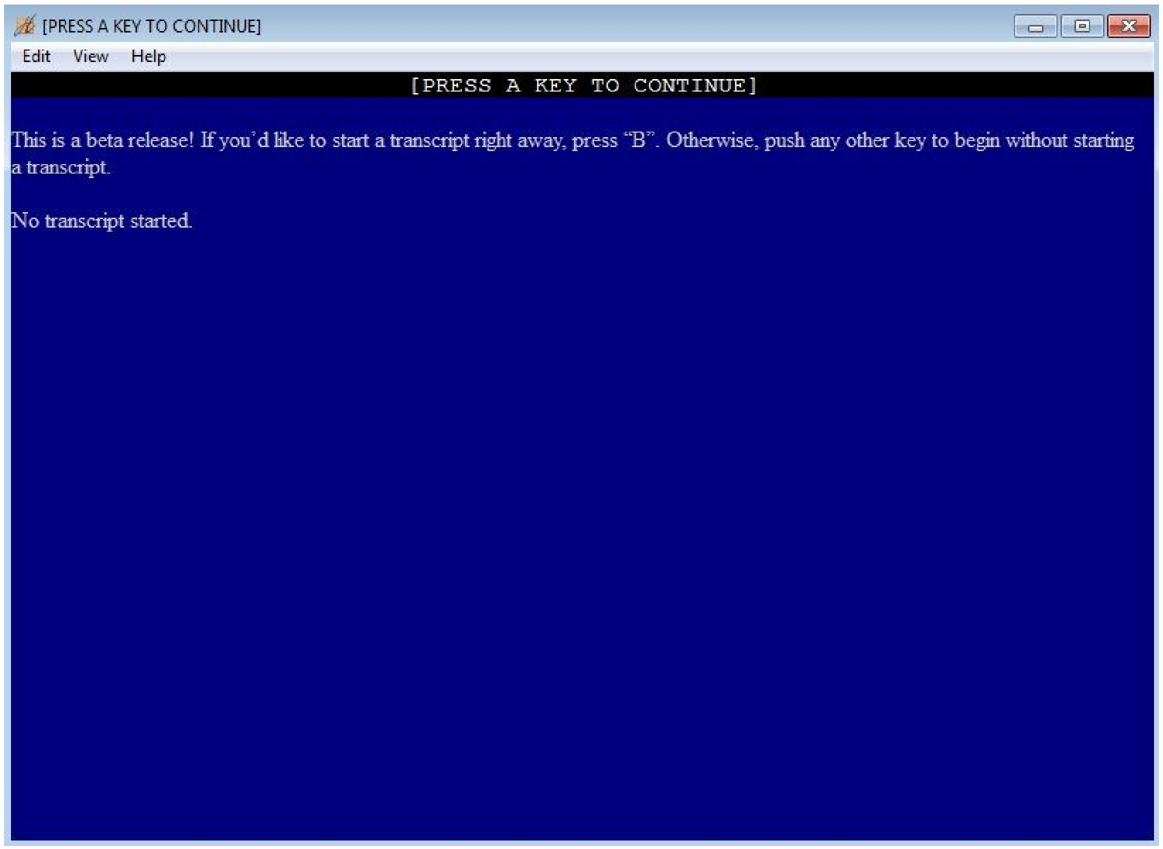
The `pausetext` argument is the string to be printed if you want a quick-and-easy non-default message (without replacing the `&CoolPause` response in the `RlibMessage` routine).

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



```
[PRESS A KEY TO CONTINUE]

This is a beta release! If you'd like to start a transcript right away, press "B". Otherwise, push any other key to begin without starting a transcript.

No transcript started.
```

TopPause() example

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

Rotating Object Descriptions

It was brought to my attention that Inform and TADS have "list managers" for quickly rotating between several descriptions for a given object. The `ROTATE_DESC` system tries to allow this.

```
#set ROTATE_DESC
```

turning the ROTATE_DESC system on

Then, you can just define an object like this:

```
door red_door "red door"
{
      noun "door"
      adjective "red"
      article "the"
      rotations 0
      long_desc
      {
            rotate( "It's a door.", "It's still a door.", "Stop looking at
it.", \
            "I mean it.", "I really do.")
      }
      between startlocation east_room
      !is recycle  ! uncomment if you want descs to cycle
}

! RandomDesc picks any of them at random
object rock "rock"
{
      article "a"
      noun "rock"
      in STARTLOCATION
      long_desc
      {
            RandomDesc("A rock.","2nd rock desc.", "3rd rock.","4th rock.")
      }
      rotations 0
}
```

Rotate code example

Now, this method is limited to 5 string arguments. If you need more than that, you'll have to use the `NEW_ROTATE` system instead. Instructions on its use are in **roodylib.h**.

Standard Message Replacement

Roodylib continues the Hugo standard library's method of message replacement, but 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

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.

DoScope / DoScopeRooms

Personally, as an author, I like to write the initial objects for a game often without descriptions, getting back to that task after having taken care of some of the basic mechanics. The downside to this is that I sometimes lose track of my own code and miss objects that still need descriptions. I created these debugging verbs as a way to remind the author what the player can see from any given room. Typing >**SCOPE** will list all of the objects within scope in the current location, while >**SCOPE ROOMS** will list everything currently in scope in every room of the game.

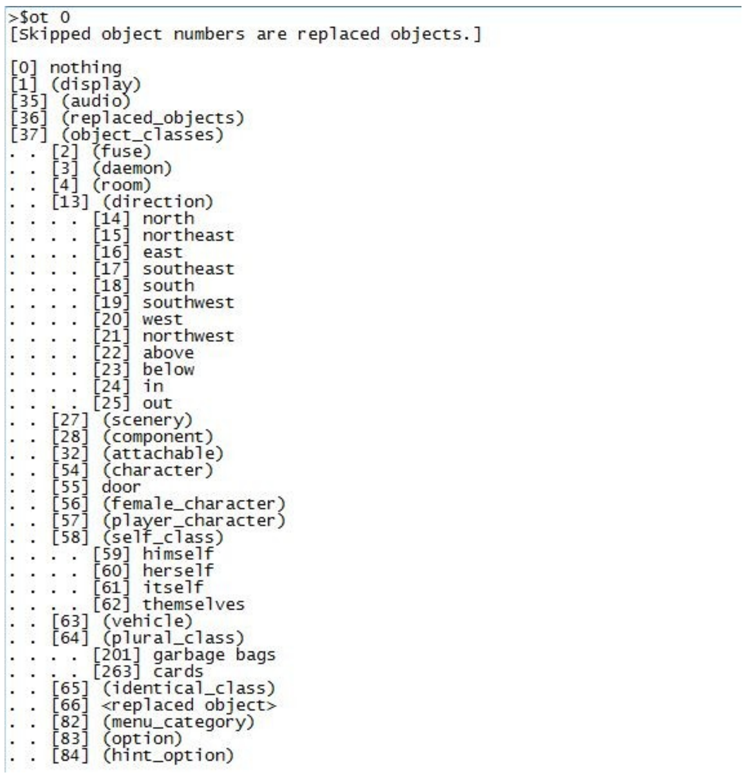
DoVerbTest

Another feature I borrowed from Inform 7 development (specifically, an extension by Juhana Leinonen), the >**VERBTEST** command shows the response given when any standard library verb is applied to an object. It's surprisingly easy to forget to supply an answer to things like eating a food item. To use it, just type >**VERBTEST <object name>**.

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!

Credits / Extension Crediting

One of the signs of a polished game is credit attributed to betatesters and, in some cases, the additional extensions used to write your games. Roodylib tries to accommodate the latter by providing some routines to do that all for you.

The `Roodylib_Credits` routine prints a list of everybody whose ideas, bug reports, and code contributions have helped Roodylib's development. If your game uses any extra extensions I have written or updated, setting `USE_EXTENSION_CREDITING` will make it easy to print a list of extension credits, also.

```
routine DoGameCredits
{
      "This game is by me. I worked on it hard!\n"
      Roodylib_Credits ! Print the Roodylib credits
      ""
      "Additionally, this game uses the following library extensions:"
      ListExtensions  !  Lists extension credits as a tall list
}
```

example credits 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.

Configuration Files

More fully-featured Hugo interpreters have the ability to write to an external file. Games like

Future Boy use this to save settings; my little joke game uses it to share information between two different compiled games. You could use the system for these reasons or plenty of others!

Along the way, I decided it was worthwhile to write a configuration file manager. For one thing, several of the extensions I wrote had the ability to write to configuration file, and it seemed wasteful to have each of them write to a separate file and unreasonable to expect every author to replace everything so it could be written to one file every time. Furthermore, it's easy to confuse reading from and writing to a configuration file; if you make a new compile that adds a new unexpected value to the configuration file, when it comes to read it, the game often misreads what is supposed to be what.

To this end, I created a configuration file system that stacks what instructions are to be followed and checks for certain values here and there. If something new has been added that messes up the order, it goes, "ok, this isn't right" and throws out the whole configuration file and starts anew.

```
#set USE_CONFIG_SYSTEM
```

turning on the configuration file manager

```
object jukebox_config "Next Day Jukebox v1"
{
      in config_instructions
      name_sum 0  ! we don't need a value here but we need the slot
      first_time 1 ! if you want a pre-game question/menu only the first time
                   ! the game is run, put a true value here
      load_info
      {
            play_music = readval
!           self.first_time = readval  ! uncomment if you want the setup to
                                       ! run only the first time ever
      }
      save_info
      {
            writeval play_music
!           writeval self.first_time   ! uncomment if you want the setup to
run only
                                 ! the first time ever
      }
      setup ! This property routine gets called at game start
      {
            if self.first_time
            {
                  self.first_time = 0
                  print "Do you want to play this game with music? ";
                  play_music = YesOrNo
                  return true ! returning true will cause InitScreen to be
called
            }
      }
}
```

example configuration file object

So, as you can see, the file-writing and file-reading instructions for your game go in the `load_info` and `save_info` properties. Any pre-game questions go in the `setup` property.

Footnotes

Games such as *Stationfall* and *Guilty Bastards* are fondly remembered for their footnote systems. Roodylib's optional footnote system makes it possible to add such footnotes to your game quickly and easily.

```
#set USE_FOOTNOTES
```

turning on the footnote system

First off, if your game has more than 10 footnotes, you'll have to first declare a new `MAX_FOOTNOTES` constant before **roodylib.h** is included.

```
constant MAX_FOOTNOTES 30
```

example MAX_FOOTNOTES declaration

If you want all footnotes available from the get-go (and not numbered in the order that they are "found"), you can set `HITCHIKER_STYLE`.

```
#set HITCHHIKER_STYLE
```

setting HITCHHIKER_STYLE

Then, let's actually write the footnotes somewhere. To do this, replace the `PrintFootnotes` routine.

```
replace PrintFootNote(num)
{
      select num
            case 1 : "This is the first footnote."
            case 2 : "This is the second footnote."
            case 3 : "Etc."
}
```

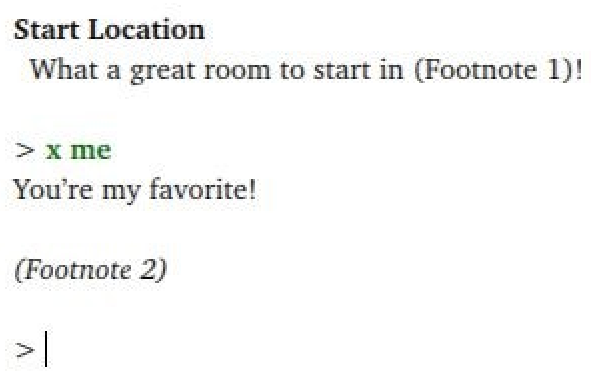
PrintFootnote replacement

Roodylib also has a couple ways of adding footnotes to your game's messages (and the applicable routines do the "unlocking" in footnotes systems where footnotes have to be found).

```
player_character you "you"
{
      long_desc
      {
            "You're my favorite!"
            AddFootnote(1)
      }
}

room STARTLOCATION "Start Location"
{
      long_desc
            print "What a great room to start in "; Footnote(2);"!"
}

replace PrintFootnote(num)
{
      select num
            case 1 : "No, really, you are!"
            case 2 : "Don't mind the dead body."
}
```

Calling footnotes

`Footnote(num)` will print "(Footnote #)" at that exact point in the text while `AddFootnote(num)` prints an italicized "(Footnote #)" after everything else.



```
Start Location
  What a great room to start in (Footnote 1)!

> x me
You're my favorite!

(Footnote 2)

>|
```

footnotes in action

Hugor 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 improves 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). I updated Roodylib with some easy-to-read methods to do this. Additionally, the Roodylib shells now default to showing the Hugor interpreter version along with everything else in the game's banner, so that's an easy way to see if the game has detected it's being played in Hugor.

Roodylib provides a `hugor` object that it automatically gives the attribute switchedon if detects that Hugor is being used (and clears it if it is not). If your code has some features that depend on Hugor functionality, you can test for it like this:

```
if hugor is switchedon
```

Using the Opcodes

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

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

Roodylib has several opcode objects to be used in the above routine. A couple require a secondary string argument. Let's go over those opcodes!

Roodylib opcode objects for Hugor

getversion	Returns the Hugor interpreter version being used. It is called by Roodylib automatically, storing the values in the `hugor` object's `version` property.
getos	Returns the OS Hugor is currently on. It is also called by Roodylib automatically, storing the value in the `hugor` object's `os` property, according to the following values: 1 = Windows, 2 = MacOS, 3 = Linux
op_abort	Aborts opcode execution. Mostly for debugging purposes and not something we need to worry about.
fade_screen	Allows screen fading in and out. Not to be called directly. See below (after the table) for more info.
open_url	Opens the secondary string argument in the default web browser. Example: *ExecOpcode(open_url, "http://notdeadhugo.blogspot)*
fullscreen	Changes interpreter to fullscreen
windowed	Changes interpreter to windowed mode
clipboard	Copies the secondary string argument to the clipboard. Example: *ExecOpcode(clipboard, "roodyyogurt@gmail.com")*
is_music_playing	Checks if a music file is currently being played. Roodylib's jukebox code calls it automatically. If you're not using the jukebox, it clears `audio.current_music` if a song is no longer playing.
is_sample_playing	Checks if a sound file is currently being played. It clears `audio.current_sound` if a sound is no longer playing.

The `fadescreen` opcode object mentioned above is a little trickier, as it's more of an object class to create other opcodes. 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 has been playing around with a fade, games don't get stuck like that in such situations.

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)
```

Multiple Player Characters

Not that it comes up often, but a game that changes the player character could be confusing to the player if those characters visit the same rooms and interact with the same objects. Ideally, when the second player visits a room the first character has been in, it should be treated as unvisited. Roodylib's `MULTI_PCS` system exists to help with this.

```
#set MULTI_PCS
```

turning on MULTI_PCS

Then, you'll need to create new attributes for your characters and replace the following routines:

```
! let's say this game consists of Laurel and Hardy
attribute laurel_moved
attribute laurel_known
attribute hardy_moved
attribute hardy_known

replace ObjectIsKnown(obj)
{
      local ret
      select player
            case laurel
                  if obj is laurel_known : ret = true
            case hardy
                  if obj is hardy_known : ret = true
      return ret
}

! continued on next page
```

```
replace ObjectIsMovedVisited(obj)
{
      local ret
      select player
            case laurel
                  if obj is laurel_moved : ret = true
            case hardy
                  if obj is hardy_moved : ret = true
      return ret
}

replace MakeMovedVisited(obj)
{
      select player
            case laurel: obj is laurel_moved
            case hardy: obj is hardy_moved
}

replace MakeKnown(obj)
{
      select player
            case laurel: obj is laurel_known
            case hardy: obj is hardy_known
}
```

example replacements

Music Jukebox

Robb Sherwin's *Cryptozookeeper* has the largest Hugo soundtrack to date. After each song ended, one would be picked at random to start at the next turn. I added a similar "jukebox" system to Roodylib.

```
#set USE_JUKEBOX
```

turning on the jukebox system

First, make `song` objects of all of the songs for your game.

```
song zombiecrap "Zombie Crap"
{
      artist "Ben Parrish"  ! optional property
      file "zombie"
      length 4 14
      in jukebox
}
```

song example

Notice the length property. The first field is the song length in minutes, and the second one is the seconds.

Assuming you're going to want all of your songs bundled in one resource file, you'll also have to define a `MUSIC_RESOURCE_FILE` constant.

```
constant MUSIC_RESOURCE_FILE "gamemus"

resource "gamemus"
{
      "zombie.mp3"
}
```

defining the MUSIC_RESOURCE_FILE constant

The way the jukebox system works is that it always plays the eldest child of the jukebox. Once played, it moves it back to the jukebox so it's the youngest child, continually looping. If you'd like the songs to be shuffled each time someone plays the game, add this code:

```
object music_shuffler
{
	in init_instructions
	execute
	{
		if not CheckWordSetting("undo")
			MixObjects(jukebox)
	}
}
```

music shuffling code

Now, all that's left is turning on the jukebox! You can control it with these routines:

`PlayJukebox` – Turns the jukebox on and plays songs continuously.

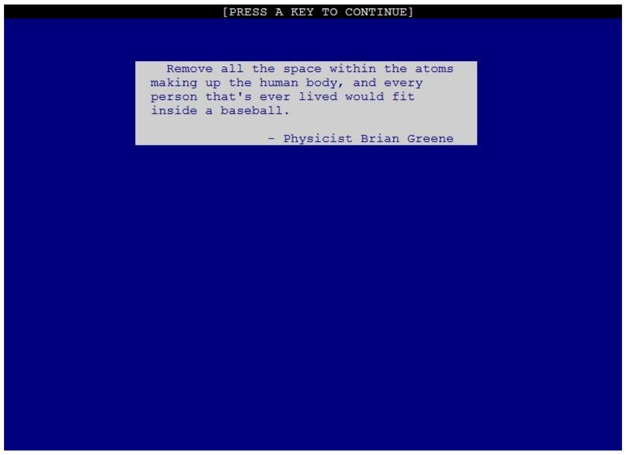
`StopJukebox` – Stops the jukebox.

`NowPlaying` – Says either "There is no song currently playing." or "<song name> by <artist> is currently playing."

`PlaySong( songfile , loop)` – This will play a song class object (as if it were in the jukebox) and update audio.current_music when it is over, in case there's any time you want to know when a song is over but aren't actually using the jukebox.

Quote Boxes

While never *especially* popular, there have been various IF games that used a "quote box" effect (usually to make some kind of literary allusion). Roodylib includes a modified extension written by Cardinal Teulbachs. Among other things, it takes some care to make sure the quoted text looks nice in the game transcript.



```
                    [PRESS A KEY TO CONTINUE]

          Remove all the space within the atoms
         making up the human body, and every
         person that's ever lived would fit
         inside a baseball.

                        - Physicist Brian Greene
```

boxdraw in action

```
#set USE_BOXDRAW
```

turning the quote box system on

First, you need to define your quotes that will be used:

```
quote baseball_intro
{
       line "\_ Remove all the space within the atoms " \
            "making up the human body, and every " \
            "person that's ever lived would fit\_ " \
            "inside a baseball.\_                 " 0 \
            "\_              - Physicist Brian Greene"
       simplefont ITALIC_ON  ! note: gargoyle will only honor italic OR bold,
not
                             ! both
!      is centered  ! add this attribute if the text should be centered
}
```

example quote

Then, there are several ways to call your quote within the game:

`Box(quotefile)` – Draws the quote at the current text position.

`Epigram(quotefile, pauseflag)` – Draws a quote box near the top of the screen at the end of the current turn. If the `pauseflag` value is true, it waits for a keypress before drawing the box.

`TitleEpigram(quotefile)` – Since quote boxes are often used for title pages, here is a routine that'll handle the screen clearing and such for you.

Score Notification

Not many games these days have scores, but when you do, it's nice to alert the player of score changes. This code helps with that!

```
#set SCORE_NOTIFY
```

turning SCORE_NOTIFY on

In your code, if the score goes up by 10 points, call `AddScore(10)`. If it goes down by 10 points, call `SubtractScore(10)`. Roodylib's code will handle everything else!

Time Manager

Now, Hugo can't do actual-real-time games like *Border Zone* or *Knight Orc* where NPCs wander around and things happen whether or not you are entering commands, but it *can* keep track of time. Kent Tessman wrote a number of time-keeping routines for *Future Boy!* (it kept track of how long the player has been playing the game). I've included them in Roodylib. They've been useful for some rather obscure purposes. For instance, the IF interpreter Gargoyle can read configuration files but can't write to them. Saving the current time to file and then reading the time right back (and checking the difference) is a good way to see if an interpreter *fully* supports configuration files. They're also used by Roodylib's "jukebox" to determine if a song has ended.

```
#set USE_TIME_SYSTEM
```

turning on the time manager

The time system uses a `time_object` class that keeps track of years, months, days, minutes, and so forth.

```
class time_object
{
      tm_year 0
      tm_month 0
      tm_day 0
      tm_hour 0
      tm_minute 0
      tm_second 0
}
```

the `time_object` class

Usually, to do anything, you'll need at least three `time_object` objects.

```
time_object movie_start
{}

time_object current_time
{}

time_object difference_in_time
{}

time_object movie_length
{}
```

example `time_objects`

With the above, you store the time you started a video in `movie_start`, periodically checking the time and storing it in `current_time`, determine the difference between those two and store it in `difference_in_time`, and finally compare *that* to `movie_length` to determine if the movie is over.

Let's go over some time-management routines:

`GetCurrentTime(timefile)` - Saves the current time to the `time_object` given as an argument.

`CalculateTimeDifference(current, previous, result)` - Determines the difference between the current-time `time_object` and the earlier-time `time_object`, saving to result `time_object`.

`IsTimeLonger(first, second)` - Returns true if first `time_object` is longer than second.

`AddTimes(time1, time2, result)` - Adds two `time_objects`, saving to result `time_object`.

`CopyTimeValue(time_orig, time_copy)` - Copies one `time_object` to another.

`PrintTimeValue(time, no_seconds)` - Prints a `time_object` as "# years, #

months, # days, # hours, # minutes, # seconds" (if the `no_seconds` argument is true, the seconds are skipped).

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:

- object sorting
- string manipulation
- "settings" objects and explanation of word array saving
- HiddenPause, GetKeyPress
- Accessibility enhancements for the visually impaired and limited-function interpreters
- MakePlayer
- NEW_STYLE_PRONOUNS
- Plural-handling improvements
- BeforeParseError

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/

http://www.intfiction.org/forum/viewforum.php?f=16

http://www.joltcountry.com/phpBB2/viewforum.php?f=8