The Onyx Ring rLibrary for I6

A User's Guide

Jim Fisher

LAST UPDATED: AUG 12, 2025

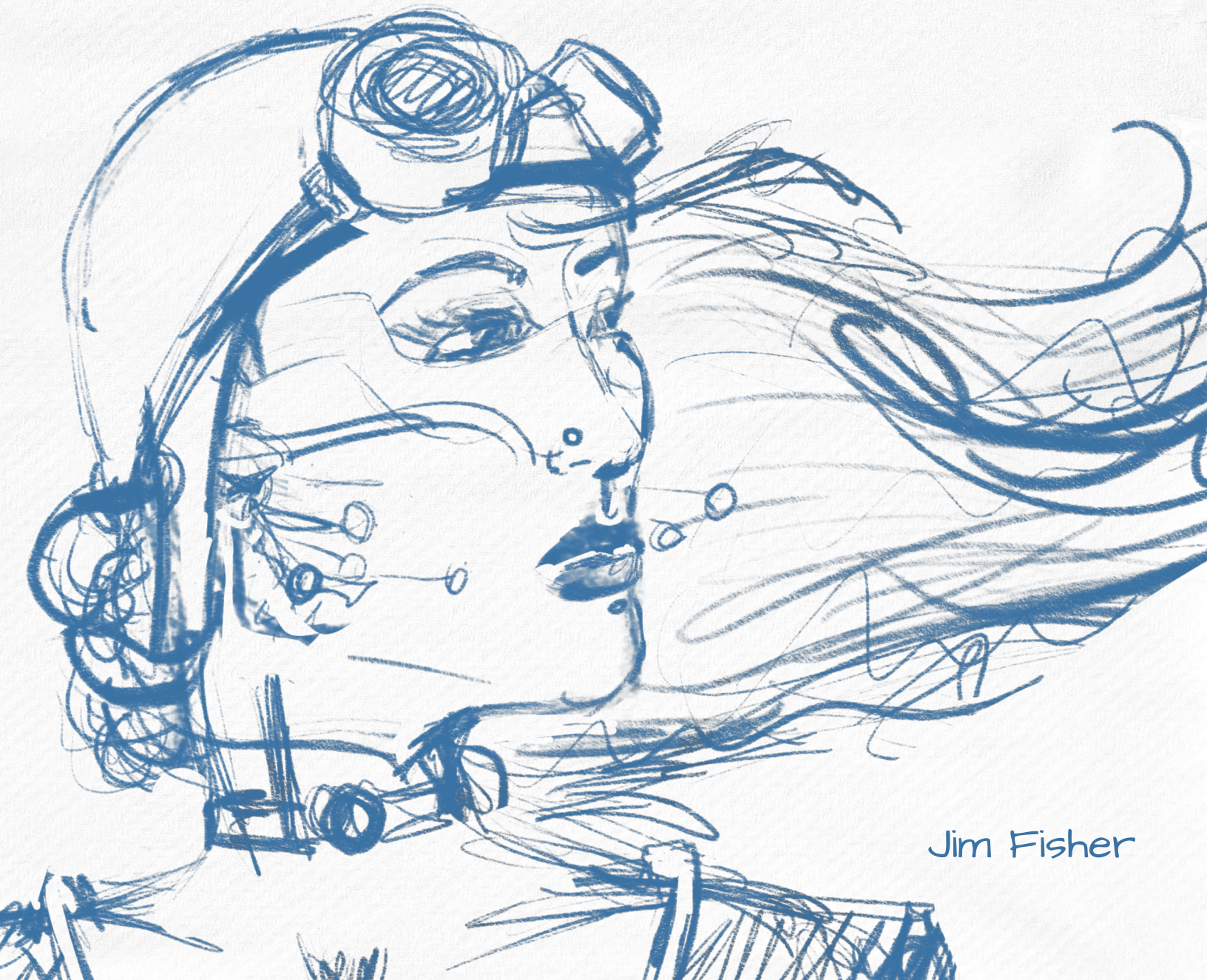
My original sketch

The finished image (more or less)

The cover overlays the original sketch on the finished image.

The orLibrary for I6

A User's Guide



Jim Fisher

Let's get introductions out of the way: I'm Jim, and I like to make stuff.

Online, some people know me as OnyxRing or, less commonly, as CodingMonk. I'm a serial Maker who gets a dopamine hit from bringing new things into the world. I've worked in software solutions development for more than half of my life. I'm also a casual writer and a casual artist.

No surprise, I game: card games, table-top games, video games, RPGs, and, once upon a time, Interactive Fiction. I design these as well as play all of these because, as I said, I like to make stuff.

When I do manage to finish a project, I like to put it out into the world, as I did with the original ORLibrary and the document which came with it. So I'm doing this again with this updated version of the library and this guide. I hope some people find it a useful aid in their efforts to make their own stuff.

Credits

Written by

My left brain ~~Jim Fisher~~

Art

~~Jim Fisher~~ My right brain

Editing/Proof Reading

Sterling Fisher, Elizabeth Fisher

Used Public Domain Assets:

"Watercolour background with leaves" texture
VecMes from www.freepik.com

FFF Tusj font
Magnus Cederholm

Architects Daughter font
Kimberly Geswein

This work is FREE, but even free things have to have legal agreements!

Here's all the official licensing text:

ORC Notice	This product is licensed under the ORC License held in the Library of Congress at TX-307-067 and available online at various locations including: www.azoralaw.com/orclicense All warranties are disclaimed as set forth therein.
Attribution	This product is an original creation. If you use this Licensed Material in your own published work, please provide credit in your product *for the used material* to: Jim Fisher
Reserved Material	Reserved Material elements in this product include the images it contains which may not be reused except with special permission from the author/licensor. This includes the OnyxRing logo, which is wholly reserved.
Expressly Designated Licensed Material	All other content contained within this document is owned by the Licensor and hereby designated as Licensed Material.

And here's what the license means, loosely translated:

This guide is meant to be freely shared. The licensing is intentionally open, allowing you to copy, adapt, and distribute this content as long as you credit me as the author of any sections you copy.

Please don't use the OnyxRing logo on your stuff, make it clear that your work is not affiliated with any "official" OnyxRing line (as if there were such a thing.)

The images used in this document should not be reused in other products except with special permission.

You are free to profit from your own works, even if you include text from this guide, and you may license that work as you see fit; however, the material you copy from this document will retain this open license.

"A nasty-looking troll, brandishing a bloody axe, blocks all passages out of the room."

Table of Contents

Introduction to the orLibrary Version 2.0

The ORLibrary was a personal contribution I started in 2001 for the Inform 6 community. It gained some traction and a few games, not just my own, were built on it. I enjoyed working on the ORLibrary quite a bit, but I set it aside shortly after the Natural Language version of Inform surfaced and, until now, it's remained relatively untouched.

Despite I7, the I6 community has persisted and its members periodically seek to preserve I6 contributions before they cease to exist altogether. In 2024, I stumbled upon a forum request for the last, original archive of the ORLibrary's codebase. I still had it, being a bit of a digital hoarder, and contributed it. I also took a look at its final state, curious about how it held up and if it was still usable in modern versions of Inform.

I had mixed emotions about what I found.

I was somewhat happy to see many extensions were no longer relevant, their functionality having been pulled into the standard library proper. I was disappointed that a handful of extensions no longer worked as designed. Finally, I was horrified to find the documentation, which the younger version of myself considered substantial, was terribly lacking. Unsatisfied with these findings, I chose to address them by fashioning a shiny new update, one free of deprecated modules and brought in line with changes made to Inform 6 since the ORLibrary was last released.

The orLibrary version 2.0, is the result. It's been refactored into something cleaner, easier to understand, and simpler to use. This guide, too, is a significant reworking of the original "ORLibrary Developer's Guide," and bares some small similarity to its predecessor. It is my hope that this updated version will be useful to I6 developers for years to come.

A Few Changes...

Aligning the orLibrary with the current version of I6 afforded the opportunity to make some improvements to its structure and usability. Some of these changes are small, others more significant. Here's a summary of the major changes:

Note the case change in the "or" prefix. As of 2.0, I call this the "orLibrary" rather than follow the former convention of "ORLibrary". This is stylistic since Inform is case-insensitive, but I try to reflect this change in all extensions and this documentation. I reserve "ORLibrary" for referencing older versions of the library.

It is Easier to Adopt

Previously, the orLibrary was added to a project by including it four times, wrapping each of the standard library's three required **`#include`**s. In practice, this technique continues to work; however, going forward it is better to include the orLibrary's wrappers for the standard library and be done with it.

See page 5.

#Includes instead of Constants

Older versions of the library pulled in extensions using constants, such as **`USE_ORDoor`**. This was, I've always thought, a bit messy. It required game authors to know the constant and provided no error messages when they got the constant wrong. In this update, you add extensions to your project with single **`#include`** directives at the top of your source and the framework takes care of the rest. If you mistype the filename, normal file-not-found compiler errors will let you know.

So is it a "library" or is it a "framework"?

It's both! Its a framework for managing extensions, and a library of extensions which the framework manages.

I touch upon this on page 4 and in more detail on page 109.

Breaking Changes

The changes associated with this version of the orLibrary are significant. Although a few of the extensions mirror their turn-of-the-millennium ancestors, most have been extensively refactored. This version eschews backward compatibility, throwing the principle aside entirely, and instead favors a fresh, clean product. It incorporates learnings from the past without all the baggage.

Works built upon a previous version of the ORLibrary will need to be updated.

Notes on the Style of this Guide

The following conventions are used in this document:

1. Code examples, compiler output, and in-game text, are all rendered with fixed-width font. Code is emboldened; game-text and compiler output is not. Often, these are intermingled, with interpreter output following the code which produced it:

```
orPrint("$IAm:actor '$name:actor.'");
I am 'Legion.'
```

   If you are following along, don't type the output lines, these are unlikely to compile.

2. Sometimes you'll see the same name rendered in two different styles which, at first glance, may seem inconsistent (e.g. **`orPrint`** vs. orPrint). The former represents actual code, such as routines, objects, or classes defined in the extension, the latter is used when referring to the extension itself.

Throughout this guide I scribble notes in the margins. I do this to clarify concepts, and to point out other material which may be related to the topic at hand.

3. There are annotations.

4. As a loose rule, excluding the Introduction on the previous page, the primary material of this document is written in a casual, semi-formal style, from a shared perspective (e.g. "Let's see" and "We go over..."). We're on this journey together, after all.

I write annotations in a more relaxed voice, using first person ("I did this" and "I recommend").

5. There are also doodles.

It may not make sense, including a bunch of my random sketches in a guide for programming text adventures; however, my work on the orLibrary, and this guide, has been a labor of love. I include in it things which make me happy, even if they aren't quite on brand. These scribbles do that.

I consider such sketches to be alternative expressions of "me". Many come into existence unplanned and without forethought, products of a collaboration between my fingers and subconscious. They are suggestive of the creativity I hope my tools enable in other.

Part One:
Getting Started

In this first part of the guide, we go over the orLibrary itself, covering the basics: what it is and how to use it.

01

Getting Started

Welcome to the orLibrary

Welcome to the orLibrary User's Guide (ORLUG). In this section we'll cover what the orLibrary is, provide context for the rest of this document, and explain how to use the library in your own Inform 6 projects.

What is the orLibrary?

The original ORLibrary documentation described the library as:

> "a collection of objects, routines and enhancements which sit on top of the standard library"

See my note on page 1 regarding the "or" vs "OR" prefix.

And this description remains true today. It is NOT a replacement for the Inform standard library, rather its an augmentation. It provides a collection of extensions which you can add to your own projects, but it's actually more...

...An Extensions Framework

The foundation upon which the entire orLibrary rests is the extensions subsystem, named the "orExtensionsFramework." This provides the plumbing used to manage extensions and trivializes their inclusion in a project. The general details, seen through the lens of the orLibrary are covered on the next page. A deeper dive into the framework, how it works, and how you can use it to manage your own collection of extensions, independent of the orLibrary, appears later in this guide.

I go over the orExtensionFramework in detail on page 109.

...A Collection of Extensions

As a group of extensions, the orLibrary provides a rich, diverse smorgasbord to choose from. These range between general-purpose utilities and deep, functional enhancements to the standard library. The extensions let you perform mundane world-building tasks with less code or unlock complex automatons, governed by the same rules as the player. Regardless of what you are looking for, you'll likely find extensions here to help.

Assumptions of this Guide

This document is written for those who are already fluent in the Inform 6 language and have a working knowledge of its corresponding Standard Library. It contains code examples, some little more than snippets, which may be difficult to follow if this is your first foray into I6. If it is, don't despair, there are a number of free resources available for you. In addition to the last published official standard to the language, the Designer's Manual, fourth edition (DM4), I highly recommend searching online for "the Inform Beginner's Guide" written by Roger Firth and Sonja Kesserich to establish this foundation.

Setting it Up and Using it

02
Getting Started

Adding the orLibrary to your own project is easy-peasy. Download the latest version, from:

```
https://github.com/onyxring/orLibrary-for-I6/blob/main/orLibI6-latest.zip
```

Then unzip it to a folder of your choosing which the Inform 6 compiler can access.

Two Steps to Add It

Once you've downloaded it, there are two steps to add the orLibrary to your personal work:

- Include the orLibrary's path in your ICL settings in the same way you include the Inform Standard Library's path. This *could* be a command line parameter, but most often appears at the top of your source code, à la:

```
!% +include_path=./inform6/lib,./orLibraryI6
```

Of course, paths are OS-specific. This example is from my Mac and would look similar on Linux. In the ORLib 1.x days, on my Windows machine, it looked more like this:

```
!% +include_path=C:\inform6\lib,
C:\inform\orLibraryI6
```

- Replace the three Standard Library includes with the orLibrary *wrappers*. Specifically, this means including **`#parser`** instead of **`parser`**, **`#verblib`** instead of **`verblib`**, and **`#grammar`** instead of **`grammar`**.

These alternative include files, prefixed with #, do NOT replace the standard library! They actually pull it in for you. I use this technique to ensure the orLibrary's Extension Framework is included at all the right places and lines of code cannot accidentally worm their way between the #includes which must be grouped together.

Verifying the Install

Confirming the orLibrary is correctly installed is done by compiling source code. Use your own, or the following shortest possible program:

```
!% +include_path=./inform6/lib,./orLibraryI6
#Include "#parser";
#Include "#verblib";
[Initialise;];
#Include "#grammar";
```

Assuming your paths are correct, something similar to the following will be produced during compile time:

```
Inform 6.42 (10th February 2024)
orExtensionFramework: pre-PARSER section.
   orHookBanner...
----
orExtensionFramework: AFTER PARSER section.
   orHookBanner...
----
orExtensionFramework: AFTER VERBLIB section.
   orHookBanner...
----
orExtensionFramework: AFTER GRAMMAR section.
   orHookBanner...
----
```

Correctly installed!

Note that simply including the orLibrary's extension framework does NOT automatically include the library's extensions. To do so, you must #include them individually (except orHookBanner which we cover on page 112.)

Notice the orHookBanner extension is included by default. When you run your game, the game banner, which typically announces the version of the standard library, will also include the current version of the orLibrary:

```
Release 0 / Serial number 000000 / Inform v6.42 Library 6.12.6 S
orLibrary release 2.0 (2024.04.01)
```

Including Extensions in Your Project

The mechanics which drive the "framework" part of the orLibrary make the task of adding extensions trivial: Simply **#include** the filename *before* **#parser** and the extension will be pulled in. To demonstrate this, add the **orPrint** extension to your source code...

Again, include this before #parser.

```
#include "orPrint";
```

...and compile. The output will now declare the new extension, attesting that it has been successfully added to the compilation process. In this case, orPrint pulls in other extensions, like orString, as dependencies. Some of these extensions depend on still other extensions and include those. The resulting compiler output shows all included extensions, including orHookBanner which was included by default:

Don't let the number of extensions scare you. As a matter of practice I like to decompose my code into reusable chunks resulting in an increased number of smaller extensions.

```
Inform 6.42 (10th February 2024)
orExtensionFramework: pre-PARSER section.
   orInfExt...
   orHookBanner...
   orHookInformLibrary...
   orPrint...
   orString...
   orStringPool...
   orUtil...
   orUtilBuf...
   orUtilChar...
   orUtilNum...
   orUtilRef...
   orUtilStr...
----
orExtensionFramework: AFTER PARSER section.
   orInfExt...
   orHookBanner...
   orHookInformLibrary...
   orPrint...
      (English detected. Including English print patterns.)
   orPronoun_English...
   orPrintPatterns_English...
   orString...
   orStringPool...
   orUtil...
   orUtilBuf...
   orUtilChar...
   orUtilNum...
   orUtilRef...
   orUtilStr...
----
orExtensionFramework: AFTER VERBLIB section.
   orInfExt...
   orHookBanner...
   orHookInformLibrary...
   orPrint...
   orPrintPatterns_English...
   orPronoun_English...
   orString...
   orStringPool...
   orUtil...
   orUtilBuf...
   orUtilChar...
   orUtilNum...
   orUtilRef...
   orUtilStr...
----
orExtensionFramework: AFTER GRAMMAR section.
   orInfExt...
   orHookBanner...
   orHookInformLibrary...
   orPrint...
   orPrintPatterns_English...
   orPronoun_English...
   orString...
   orStringPool...
   orUtil...
   orUtilBuf...
   orUtilChar...
   orUtilNum...
   orUtilRef...
   orUtilStr...
----
```

A Quick Tour of the Content

03
Getting Started

Take a look at the files in the orLibrary folder. There are quite a few, most ending with the .h file suffix. The majority of these are extensions, but not all...

The Extension Framework

The orLibrary uses a framework to manage extensions. The files making up this framework are prefixed with the pound symbol (#). There are just a handful of these:

These **wrappers** pull in the unmodified standard library too.

```
#grammar.h
#parser.h
#verblib.h
```

The actual **Extension Framework entry point** the wrappers pull in.

```
#orExtensionFramework.h
```

Where extensions are **registered** in the framework. All managed extensions appear here so the framework knows about them.

```
#orExtensionRegistry.h
```

An empty place holder, providing a default for optional files.

```
#orExtensionRegistryStub.h
```

All remaining files in the root folder which do not begin with the # prefix are extensions or packages.

We discuss packages later in this section.

Primary vs. Supporting Extensions

There are dozens of extensions in the orLibrary, and some early feedback was this: it is easy to get overwhelmed and lost in all the library contains. To help keep things clear, extensions are divided into two groups:

Primary extensions are those which typical game authors are most likely to use. These add features which directly influence the resulting game behavior. This guide covers all primary extensions.

Supporting extensions are prefixed with an underscore (_). These provide low-level plumbing, usually in the service of other extensions. Although it is perfectly safe to use these in your own work, we do not cover every supporting extension in this guide.

Extension Categories

Extensions are further grouped into the following loose categories:

Enhancements – Tweaks to how the Standard library normally works, including new verbs and parsing improvements.

Widgets – Objects with pre-defined behaviors you can drop into your game and use.

Utilities – Extensions which don't alter default behavior. Instead, these provide tooling to make "doing things" in code a bit easier.

Systems – Whole new, pre-built feature sets, not really present in the standard library. These tend to be larger than widgets and require configuration to get what you want out of them.

The line between these classifications can be blurry and a few extensions fit multiple groups. In such cases I simply chose one that seemed the closest fit to me.

The _extras Folder

There is also an `_extras` folder which contains, as you might expect, extra files. These include templates for creating your own extensions and games, unit tests for the orLibrary itself, and the version of this very same user's guide which was distributed with the release.

Packages

Some of the files begin with the letters "**pkg**". Strictly speaking, these are *not* extensions, these are just a convenient way to include multiple extensions which are complimentary, and likely to be used together. The follow is a list of default packages:

I cover Enhancements in Part Two of this guide, starting on page 10.

pkgOrEnhancements includes most of the extensions in the "enhancement" category. Although the orLibrary is *not* an alternative to the standard library, this package can make it feel that way with all the new or upgraded functionality.

I list all the new hooks introduced by orLibrary extensions on page 123

pkgOrHooks includes all the supporting extensions which introduce new "hooks" into the standard library (but not necessarily hooks which are separate from the standard library). Although we don't cover them in this guide, "hooks" are extensions which inject new entry routines in the standard library's **LibraryExtenstions** object. This approach enables multiple extensions to patch the same standard library routine, without conflict.

I cover the util object, and the tools it brings to the table on page 100 .

pkgOrUtils includes all utility extensions. By including this package, the entire **util** object is fully hydrated with all utility helpers.

You can include a package in the same way you include an extension:

```
#include "pkgOrUtils";
```

With the above, all extensions defined in the package will be included, as will their dependencies.

Excluding Individual Extensions

I discuss the "stage constant" on page 120, later in this guide.

If you decide to exclude one or more of the extensions specified in a package, you may do so by **undef**ining the "stage constant" for that extension, after the package has been included. The following includes all extensions listed in the Enhancements package, sans the orUniqueMultiMessage extension:

```
#include "pkgOrUtils";
undef orUniqueMultiMessage_STAGE;
```

Note that this is equivalent to not including it directly. If another extension which you are including specifies orUniqueMultiMessage as a dependency, it will still get pulled in through normal framework resolution.

Part Two:

Enhancements to the Standard Library

In part two we cover extensions classified as "enhancements". Enhancements are changes to the default behavior of the standard library. These either expose new properties to enable enhanced functionality or tweak the standard behavior in ways which are benign unless used.

04

Standard Library Enhancements

Narrative Flexibility

This collection of extensions adds to the standard library's flexibility when describing things. Since virtually all games written in Inform are text-based, the ability to do this is especially important.

orFirstImpressions

When describing a room or object for the first time, it can be useful to toss in a some initial, passing thoughts from the player character. This technique can help flesh out the character's personality, or communicate knowledge the character has, which the player doesn't.

Normal Behavior

The default behavior of the standard library does not provide once-only description text.

Revised Behavior

The orFirstImpressions extension provides two properties, **preImpression** and **postImpression**, which can be defined on any room or object. Below demonstrates this with an object, seen through the **examine** verb, but the same format applies to a room and the **look** verb:

```
object candlestick "candlestick" with name 'candlestick'
    description "It was copper, or perhaps brass, polished to a
        dull gleam.  Even free of a candle it was
        heavy and hard enough to make a formidable
        weapon.",
    preImpression "A sense of unease settled on me as I
        looked at the candlestick.",
    postImpression "I hefted it, calculating the force it would
        exert across the back of someone's head. Definitely
        lethal.";
```

Notice there are no linefeeds separating the **preImpression** and **postImpression** text. You can add these yourself if you want them.

```
>x candlestick
A sense of unease settled on me as I looked at the candlestick. It
was copper, or perhaps brass, polished to a dull gleam.  Even free
of a candle it was heavy and hard enough to make a formidable
weapon. I hefted it, calculating the force it would exert across
the back of someone's head. Definitely lethal.

>x candlestick
It was copper, or perhaps brass, polished to a dull gleam.  Even
free of a candle it was heavy and hard enough to make a formidable
weapon.
```

By default, once the first impressions are described, they are not displayed again; however, players can revisit their first impressions with the **review** verb if the orReview extension is included.

I cover the REVIEW verb in the orReview extension on page 41.

orStateDescriptors

The orStateDescriptors extension enables developers to modify the text which describes the state of objects.

Normal Behavior

During inventory and room descriptions, the standard library interrogates objects and prints qualifying text such as "(providing light)" or "(closed, empty and providing light)":

```
object startingRoom "Room" with description "The place where you
    are.";

object -> box "magic box" with name 'magic' 'box',
    description "The box is impossible to ignore."
    has container openable light;
```

Assuming the player character is in **startingRoom**, the following transcript plays out:

```
Room
The place where you are.

You can see a magic box (closed and providing light) here.

>take box
Taken.

>i
You're carrying:
  a magic box (providing light and closed)
```

Inform does not provide a simple way to add new state descriptors.

Revised Behavior

This module implements an extensible framework for defining new state descriptors as well as redefining or removing existing ones. Simply including it produces output almost identical to the above:

```
Room
The place where you are.

You can see a magic box (providing light and closed) here.

>take box
Taken.

>i
You're carrying:
  a magic box (providing light and closed)
```

Adding a new state descriptor is done by first creating a routine to print the new descriptor if the object qualifies...

```
[sdBlessed obj suppress;
    if(suppress==false && obj has blessed)
        print "glowing with purity";
    return obj has blessed;
];
```

Note the first parameter, **obj**, is the object being considered. It's the routine's responsibility to interrogate the object and determine if qualifying text should print, returning **true** or **false** in either case.

The second parameter, **suppress**, indicates whether the routine should actually print the qualifying text. If **true**, then the parser is simply checking to see if text will be printed when it is ready to do so and the text should be suppressed.

To actually put this routine handler into place, it must be registered as a handler for either room descriptions, inventory descriptions, or both, usually from **initialise()**:

```
orStateDescriptors.addRoomDescriptionHandler(sdBlessed);
orStateDescriptors.addInventoryHandler(sdBlessed);
```

Now objects with the **blessed** attribute will show the new descriptor:

```
object -> box "magic box" with name 'magic' 'box',
    description "The box is impossible to ignore."
    has container openable light blessed;

You can see a magic box (glowing with purity, providing light and
closed) here.
```

Besides the one's we create, there are six default handlers, which orStateDescriptors uses to describe the various object states:

```
sdWorn
sdClosedcontainer
sdOpencontainer
sdLockedcontainer
sdEmptycontainer
sdLight
```

Removing existing descriptors is as easy as adding new ones. The following, code, usually in **initialize()**, turns off the "providing light" message from room descriptions, while keeping it for inventory listings:

```
orStateDescriptors.removeRoomDescriptionHandler(sdLight);

You can see a magic box (glowing with purity and closed) here.

>take box
Taken.

>i
You're carrying:
  a magic box (glowing with purity, providing light and closed)
```

One final note: the orStateDescriptors extension eliminates the use of **ListMiscellany** messages, numbered 7-17. These can be repurposed.

orCantGoOdd

The orCantGoOdd extension causes the standard library to distinguish typical directions (including the cardinal and inter-cardinal directions) from non-typical directions, like up, down, in, and out.

Normal behavior

Without special treatment, players attempting to move in a direction which is not defined receive the default response:

```
You can't go that way.
```

This can be overloaded with the **cant_go** property...

```
object forest "Forest" has light
    with description "You are in a small forest.",
    cant_go "Trees block travel in that direction.";
```

...but if not crafted correctly, **cant_go** can be a bit odd when applied to non-traditional directions:

```
Forest
You are in a small forest.

>s
Trees block travel in that direction.

>up
Trees block travel in that direction.

>in
Trees block travel in that direction.
```

Revised behavior

Simply include the orCantGoOdd extension and your game will treat **up**, **down**, **in**, and **out** differently:

```
Forest
You are in a small forest.

>s
Trees block travel in that direction.

>up
It is not possible to go that way.
```

Define the **cant_go_odd** property to provide the same location specific behavior as **cant_go**, but for the four odd directions:

```
object forest "Forest" has light
    with description "You are in a small forest...",
    cant_go "Trees block travel in that direction.",
    cant_go_odd "Reality blocks travel in that direction.";

Forest
You are in a small forest, surrounded by trees.

>s
Trees block travel in that direction.

>up
Reality blocks travel in that direction.
```

A new default message (#go, 13) is used when **cant_go_odd** is not defined.

Note: most of the time, properties introduced by the orLibrary are named using the camelCase convention; however, since **cant_go_odd** is so closely related to the standard library's **cant_go** property, I opted to keep the same convention for consistency.

orUniqueMultiMessage

When interacting with multiple objects, the standard library sometimes generates a series of identical messages. OrUniqueMultiMessage consolidates these into unique lines.

Normal behavior

Consider the following:

```
class marble "marble" with name 'marble' 'marbles//p'
    plural_name "marbles";
class candy "candy" with name 'candy' 'candies//p'
    plural_name "candies";
class rock "rock" with name 'rock' 'rocks//p'
    plural_name "rocks";

object bowl "bowl" has open container;
    marble ->;
    marble ->;
    marble ->;
    marble ->;
    marble ->;
    candy ->;
    candy ->;
    rock ->;
```

The following transcript demonstrates the default behavior of the standard library:

```
>take all from bowl
marble: Removed.
marble: Removed.
marble: Removed.
marble: Removed.
marble: Removed.
candy: Removed.
candy: Removed.
rock: Removed.
```

Revised Behavior

The orUniqueMultiMessage extension analyzes the responses made by multi-object commands and consolidates them.

Simply include this extension and the above transcript plays out more concisely:

```
>take all from bowl
5 marbles: Removed.
2 candies: Removed.
rock: Removed
```

No additional code need be written to make this functionality occur.

orVagueQuantity

This extension allows groups of identical objects to be described with a vague adjective rather than a specific number.

Normal behavior

By default, when describing a group of identical objects, the standard library states the exact number:

```
class marble "marble" with name 'marble' 'marbles//p'
    plural "marbles";

object bowl "bowl" has open container;
    marble ->; marble ->; marble ->;
    marble ->; marble ->; marble ->;
    marble ->; marble ->; marble ->;
    marble ->; marble ->; marble ->;

>x bowl
In the bowl are twelve marbles.
```

Revised Behavior

By simply including the orVagueQuantity extension, the standard behavior adopts vaguer language which scales with the actual quantity:

```
>x bowl
In the bowl are numerous marbles.

>take 6 marbles
...
>x bowl
In the bowl are several marbles.

>take 3 marbles
...
>x bowl
In the bowl are a few marbles.

>take 1 marble
Taken.

>x bowl
In the bowl are a couple of marbles.
```

By default, "numerous" is used to describe quantities of ten or more.

Snipped output.

Counts from 5 to 9 are rendered as "several."

3-4: "a few".

2, of course. Values of 1 are rendered as "one."

The above defaults can be overridden per type by defining the **vagueQuantity** property array, like so:

```
class fish "fish" with name 'fish'
    plural "fish",
    vagueQuantity 4 "some";
```

This is the most basic of definitions: Quantities of 1-3 are printed as exact numbers, 4 and above are rendered as "some".

Additional ranges can be appended to the same array:

```
class fish "fish" with name 'fish',
    plural "fish",
    vagueQuantity 4 "some" 6 "a school of" 9 "far too many";
```

Here, an object count between 1 and 3 is described with precise numbers (e.g. **two fish**), 4 and 5 come out as **some fish**, 6, 7, and 8 as **a school of fish**, and 9 or more is rendered as **far too many fish**. The quantities must be specified in ascending order.

Singular Groups

The previous example isn't quite right. Although most of the ranges are generated correctly, one is not:

```
In the bowl are some fish.
In the bowl are a school of fish.
In the bowl are far too many fish.
```

Wrong!

Sometimes, plurality is not easy to distinguish. Consider:

In the bowl...
...**are** a couple of fish.
...**are** a handful of fish.
...**is** a school of fish.

It often helps to reverse the subject and verb to test the way it sounds...

A couple of fish **are**...
A handful of fish **are**...
A school of fish **is**...

...in the bowl.

While "some" and "far too many" are both ***quantity*** phrases, considered plural for the purposes of noun/verb conjugation, "a school of" is considered a singular ***group***, so "*are* a school of..." is incorrect.

To force a vague quantity to be singular, specify **`true`** after the word/phrase :

```
class fish "fish" with name 'fish'
    plural "fish",
    vagueQuantity 4 "some" 6 "a school of" true 9 "far too many";
```

This corrects the output:

```
In the bowl are some fish.
In the bowl is a school of fish.
In the bowl are far too many fish.
```

05
Standard Library Enhancements

Parsing and Disambiguation

Members of this group of Library Enhancements change the way Inform translates input from the player into commands.

orAdjective

I first ran across the idea of adding adjectives in the DM4, exercise 75. The solution to that example is incomplete and has a resolution bug, but it was the original inspiration for this orAdjective extension.

orAdjective adds the **adjective** property to help the parser more clearly make choices.

Normal Behavior

Consider the following items. In some cases, the words "glass" and "marble" serve as adjectives, in others they are nouns:

```
object -> glass "drinking glass" with name 'drinking' 'glass',
    description "for drinking";
object -> marble "glass marble" with name 'glass' 'marble',
    description "for rolling";
object -> glass_table "glass table" with name 'glass' 'table',
    description "flat and glass";
object -> marble_table "marble table"
    with name 'marble' 'table',
    description "flat and marble";
```

Because there is no distinction between nouns and adjectives, the player cannot simply refer to the **marble**, or the **glass** objects without the parser asking for clarification:

```
>x glass
Which do you mean, the drinking glass or the glass marble?

>x marble
Which do you mean, the glass marble or the marble table?
```

Revised Behavior

Using this extension, authors can distinguish adjectives from nouns:

```
object -> glass "drinking glass" with name 'glass',
    adjective 'drinking', description "for drinking";

object -> marble "glass marble" with name 'marble',
    adjective 'glass', description "for rolling";

object -> glass_table "glass table" with name 'table',
    adjective 'glass', description "flat and glass";

object -> marble_table "marble table" with name 'table',
    adjective 'marble', description "flat and marble";
```

This gives the parser enough information to decide correctly...

```
>x glass
for drinking

>x marble
for rolling
```

...while still asking for clarification when the player's input is too vague:

```
>x table
Which do you mean, the glass table or the marble table?
```

By default, the parser will still match objects, even if only adjectives are provided:

```
>x drinking
for drinking
```

But the parser can instead be made to require at least one noun by changing the **orAdjective** object's **mode** property, usually in **Initialise()**:

```
orAdjective.mode=REQUIRE_NOUN; !--is PREFER_NOUN by default
```

Which makes the parser just a little less tolerant of ambiguity:

```
>x drinking
You can't see any such thing.
```

orRecogName

This extension eliminates the requirement of specifying the **name** property. More specifically, it allows Inform to recognize objects without requiring dictionary words.

Normal Behavior

By default, the **name** property must contain all possible words used to match against an object, including those which duplicate the text in the object's display name:

```
object -> brownDog "great brown dog"
  with description "Happy and brown.",
       name 'great' 'brown' 'dog';

object -> smellyCat "dirty smelly cat"
  with description "It stinks.";

> x cat
You can't see any such thing.

> x dog
Happy and brown.
```

Names required to match

No names, can't match

Revised Behavior

With orRecogName, the **name** property is no longer required, but is still honored. The following code demonstrates:

```
object smellyCat "dirty smelly cat"
   with description "It stinks.";

object brownDog "great brown dog" with name 'Spot',
     description "Happy and brown.";

> x cat
It stinks.

> x dog
Happy and brown.

> x spot
Happy and brown.
```

Note that we can still specify synonyms in the name property. They work like they always have.

See that the cat and dog words still match the object, even though they are not present in the name property.

With this extension in place, the words in the printed name of the object do not need to be repeated in the **name** property; however, if this behavior ever conflicts with the way an object's name is to be parsed, you can turn this off by giving the object the **noRecogName** attribute which restores the default behavior for a specific object:

```
object brownDog "great brown dog" has noRecogName
    with name 'Spot', description "Happy and brown.";

> x spot
Happy and brown.

> x dog
You can't see any such thing.
```

orPrefixSuffix

This extension allows the use of prefixes and suffixes which are followed by periods (for example, "Col. Mustard" or "Mr. Anderson").

Normal Behavior

Consider an object named "Mrs. Robinson"...

```
object -> mrsRobinson "Mrs. Robinson" has proper
    with description "Loved by Beatles. And Jesus.",
        name 'mrs' 'robinson';
```

In normal English, we might refer to the character like so:

```
>examine Mrs. Robinson
```

But the default behavior of the standard library is to stop parsing the input at the period and treat the text as two separate commands ("examine mrs" and "robinson"), producing the following:

```
Loved by Beatles. And Jesus.
That's not a verb I recognize.
```

Revised Behavior

This extension scans user input and removes periods which follow common general prefixes and suffixes (e.g. "Mr.""Mrs." "Dr." "Col.", "Jr."...). This addresses the parser's confusion:

```
>examine Mrs. Robinson
Loved by Beatles. And Jesus.
```

Simply include this extension to enable this behavior. No additional changes are needed.

orBetterChoice

OrBetterChoice adds a bit more common sense to the parser's ability to decide between objects. Specifically...

When executing the TAKE action, the parser will...

- Give less priority to objects which are already carried
- Give less priority to objects which are immovable (static)

When executing the DROP action, the parser will...

- Give less priority to objects which are *not* carried

When Processing the `EAT` command, the parser will...

- Prefer edible objects to non-edible ones

For any action specifying a second object, the parser will...

- Give less priority to objects which have already been selected as the subject of the command

Normal Behavior

Under normal conditions, the default behavior of the standard library can struggle with the above use cases, failing to identify what was obviously intended by the player, or including items which are clearly not appropriate when asking the player for clarification.

Consider the following dispenser, with an attached handle, supporting candy:

```
object -> dish "big candy dispenser"
   has supporter static
   with name 'candy' 'dispenser', add_to_scope handle;

object handle "candy dispenser's handle" has static
   with  name 'candy' 'dispenser^s' 'handle';

object chocolate "chocolate candy" dish
   with name 'chocolate' 'candy' has edible;

object toy "plastic candy toy" dish
   with name 'plastic' 'candy' 'toy';
```

The following, confused responses are default from the standard library:

```
>take candy
Which do you mean, the big candy dispenser, the chocolate candy,
the plastic candy toy or the candy dispenser's handle?

>take all candy
chocolate candy: Taken.
plastic candy toy: Taken.

>drop candy
Which do you mean, the big candy dispenser, the chocolate candy,
the plastic candy toy or the candy dispenser's handle?

>drop all candy
plastic candy toy: Dropped.
chocolate candy: Dropped.

>eat candy
Which do you mean, the big candy dispenser, the chocolate candy,
the plastic candy toy or the candy dispenser's handle?
```

Here, the immovable dispenser and the handle cannot be taken. Why list them for disambiguation?

Especially since TAKE ALL CANDY knows to ignore them.

Notice the parser asks about dropping things which the player isn't even carrying.

Again, DROP ALL CANDY gets it right.

There is only one edible object. The parser should be able to make this assumption.

Normally, changing this behavior requires taping into the standard library's **`ext_chooseobjects`** hook.

Revised Behavior

After installing orBetterChoice, the standard library is just a little smarter:

Now the parser stops asking about things which can't be taken or dropped.

And assumes we are only trying to eat things which are actually edible.

```
>take candy
Which do you mean, the chocolate candy or the plastic candy toy?

>take all candy
chocolate candy: Taken.
plastic candy toy: Taken.

>drop candy
Which do you mean, the plastic candy toy or the chocolate candy?

>drop all candy
plastic candy toy: Dropped.
chocolate candy: Dropped.

>eat candy
(the chocolate candy)
(first taking the chocolate candy)
You eat the chocolate candy. Not bad.
```

Simply include the orBetterChoice extension for the above behaviors to become active.

Note that this extension doesn't stop attempts to take static objects, nor does it keep the player from trying to eat things which aren't edible. It just considers such items less likely when compared to other, more suitable, options.

A final convenience introduced by orBetterChoice brings is a new property, **chooseObject**, which the library consults during disambiguation. This lets developers break apart monolithic **ChooseObjects** routines into bite-sized pieces of logic which are attached to the objects they evaluate. Here's an example of a torch which will be ignored during TAKE/REMOVE ALL commands:

```
object fixed "fixed torch" with name 'torch' 'fixed',
    description "It's a torch on the wall.",
    chooseObject[code;
        if (code == 1 && action_to_be == ##Take or ##Remove)
            return 2;
    return -1;
];
```

The function of the **chooseObject** property closely mimic exactly the standard library's **ChooseObjects** entry point, but there are two differences:

First, the initial parameter, **obj**, is not provided since it is attached to the **object** under consideration. Code should **self** when making such considerations.

Second, because **0** can be a legitimate return value from **ChooseObjects**, returning **-1** instructs the library to follow default behavior.

06 Modeling and Mechanics

Standard Library Enhancements

Extensions in this group of Library Enhancements adjust our options for defining the game world and enhance some of the basic rules governing how that world operates.

orPlayerCommandQueue

The orPlayerCommandQueue extension allows you to specify a chain of commands to be performed, one per turn, by the player character.

Normal Behavior

Vanilla Inform does not support queued commands.

Revised Behavior

This extension introduces the **playerCommands** queue object. When empty, the standard library behaves normally; however, once commands have been added to the queue, they are automatically executed and removed, one per prompt, as though the player were entering them.

Note: For this discussion, I define "commands" as text which the player types in, and "actions" as the resolved verb and noun combinations which the game ultimately executes.

Adding a command to the queue is easy. The following adds three which execute over three turns:

```
playerCommands.pushCommand("go down");
playerCommands.pushCommand("teke banana");
playerCommands.pushCommand("give banana to monkey");
```

As soon as commands are added to the queue, they play out as though typed by the player:

Notice the misspelling!

```
>>go down
Underground cave
A cave, under the ground.

You can see a monkey and a yellow banana here.

>>teke banana
That's not a verb I recognize.

>>Give banana to monkey
You aren't holding that.
```

As you can see above, there is nothing restricting illegal commands. Verbs and nouns which don't exist in the game simply result in the same parser errors a typo would:

```
playerCommands.pushCommand("teke banana");
That's not a verb I recognize.

playerCommands.pushCommand("examine bannana");
You can't see any such thing.
```

Allowing the parser to decide what is being referenced at runtime has some practical uses. For example, when the same pushed command could reference different game objects based on the currently surroundings.

The Meta and Silent Parameters

Following the command string, **pushCommand** also accepts two optional parameters. The first, **meta**, causes the command to take no game time, so daemons and **each_turn** routines do no fire.

The second, **silent**, causes the command and the game prompt, to ***not*** print when executed. The standard library continues to behave as though the player typed the command, it just isn't output.

The interpreter probably won't fall for this ruse, though.

Inferring Commands from Actions

Instead of queueing raw command text, **pushCommand** can also take a well-formed action:

```
playerCommands.pushCommand(##examine, banana);
```

Specifying actions in this way causes the extension to generate the command text for you. For example, depending on the printed name of the **banana** object, the above might be rendered as:

```
>>examine yellow banana
```

It would be easy to confuse the above form of **pushCommand** with the execute action syntax:

```
<examine banana>;
```

However these are not the same. Because **<examine banana>** completely defines the action and skips parsing entirely, normal scope rules are ignored and the **banana** will be described even if hidden away in another room entirely.

Since the **pushCommand** syntax uses the standard library's parser, scope rules require an object matching the YELLOW BANANA text to be nearby for resolution. If not, the standard error message is displayed:

```
You can't see any such thing.
```

Changing the Generated Command Prompt

By default, all generated commands are printed at a double prompt, to clearly distinguish them from player input:

```
playerCommands.pushCommand(##jump);

>>jump
```

This can be changed by defining the **orQueuedActionPrompt** constant before **#verblib**:

```
constant orQueuedActionPrompt "AutoAct:";

playerCommands.pushCommand(##jump);

AutoAct:jump
```

Clearing the queue

Sometimes an in-game event occurs which renders queued actions pointless. In such cases, its easy to empty the queue of all unprocessed actions:

```
playerActions.clear();
```

This halts autonomous behavior entirely.

orImplicitCommands

The orImplicitCommands extension changes the way implicit actions are handled, promoting these to full-fledged actions which each take game time and allow background processes to run.

Normal behavior

Consider the following piece of fruit with a recurring notification:

```
object -> banana "banana" has edible with name 'banana',
    each_turn[; "A fruit fly hovers nearby."];
```

Normal, explicit actions take one turn each:

```
You can see a banana here.

A fruit fly hovers nearby.

>take banana
Taken.

A fruit fly hovers nearby.

>eat banana
You eat the banana. Not bad.

>score
You have so far scored 0 out of a possible 0, in (2) turns.
```

Player-input commands each take a turn.

Incongruently, implicit actions are performed in the same turn as the original action, taking no time at all:

```
You can see a banana here.

A fruit fly hovers nearby.

>eat banana
(first taking the banana)
You eat the banana. Not bad.

>score
You have so far scored 0 out of a possible 0, in (1) turn.
```

This assumes Inform's **no_implicit_actions** variable is set to **false**.

Implicit actions occur out of time.

Since time does not advance between the implicit TAKE and the explicit EAT, no other game timers elapse, including **each_turn** routines and daemons.

Revised Behavior

Instead of performing implicit actions immediately, the orImplicitCommands extension causes implicit commands to be generated and executed one at a time, each at its own prompt, as through the player had typed them. This allows "between prompt" processes to run:

```
You can see a banana here.

A fruit fly hovers nearby.

>eat banana
(first taking the banana)

>>take banana
Taken.

A fruit fly hovers nearby.

>>eat banana
You eat the banana. Not bad.

>score
You have so far scored 0 out of a possible 0, in 2 turns.
```

See page 25 for more on generated command prompts.

Notice that two whole turns pass, as well as events which happen between commands.

Adding New Implicit Commands

Adding new implicit commands TODO

TODO: orImplicitOpen

orParentChildRelationship

The orParentChildRelationship extension reworks the standard library's approach to game objects which can *contain* other objects (e.g. **containers** and **supporters**). This extension provides three independent capabilities which we'll cover separately.

This extension rolls together multiple extensions from previous versions of the orLibrary, specifically "orExamWithContents" and "orSupporterContainer." Additionally, it enables creation of new relationships, such as "behind" or "under."

1: LISTING CHILDREN DURING EXAMINATION

The first, most basic, feature of this extension enables you to easily configure the EXAMINE command to describe, or not, the contents of supporters and containers.

Normal Behavior

The standard library's default behavior of the EXAMINE verb treats containers differently than supporters, listing the visible contents of one but ignoring the visible contents of the other. Consider this:

```
object room "Room" has light with description "Here.";

object bowl -> "bowl" has container with name 'bowl';
object apple -> -> "apple";

object Table -> "table" has supporter;
object book -> -> "book";
```

The following transcript demonstrates the default behavior:

```
Room
Here.  You can see a bowl (in which is an apple) and a table (on
which is a book).

>x bowl
In the bowl is an apple.

>x table
There's nothing special about the table.
```

← But it also has a book on it!

To see the contents of the table, the player must perform an alternative command like SEARCH.

Vanishing Container Listings

Strangely, this default behavior for containers changes once we provide a description:

```
object bowl -> "bowl" has container with name 'bowl',
    description "Can contain things.";
```

Now the behavior of listing contents abruptly vanishes:

```
>x bowl
Can contain things.
```

← Like the undescribed book above.

Revised Behavior

This extension introduces two "controllers" which enable you to redefine the game's handling of parent-child relationships. The objects **orPcrSupporter** and **orPcrContainer** govern the supporter and container relationships respectively, each providing the **includeContentsInExamine** property, which determines this behavior.

The following line causes EXAMINE to list the contents of supporters:

```
orPcrSupporter.includeContentsInExamine=true;
```

With this setting, the in-game behavior for supporters matches that of open containers:

```
>x table
On the table is a book.
```

Alternatively, you could choose to align things in the opposite way, opting to never list children from EXAMINE, even for **container** objects:

```
orPcrContainer.includeContentsInExamine=false;

>x bowl
Can contain things.
```

Suppressing Container Listings by Parent

With this extension, the presence of a **description** property does *not* eliminate the practice of listing contents. If a relationship is configured to list its children during **Examine**, it will do so even if a description is defined.

You can, however, suppress content lists, regardless the of relationship's configuration, by returning **false** from the description routine:

```
object mysteriousBowl -> "basin" has container
    with name 'basin',
    description[; print "The basin is mysterious, its
                  contents unclear."; rfalse;];

object orange -> -> "orange";

object bowl -> "bowl" has container with name 'bowl',
    description "Can contain things.";

object apple -> -> "apple";

>x bowl
Can contain things.
In the bowl is an apple.

>x basin
The basin is mysterious, its contents unclear.
```

Container with listings suppressed.

Another, with listings not suppressed.

In this example, assume the orPcrContainer's includeContentsInExamine property is set to true.

2: SIMULTANEOUS CONTAINERS/SUPPORTERS

The second capability introduced by orParentChildRelationship enables objects to be both containers and supporters.

Normal Behavior

By default, the standard library does not handle objects with both the **supporter** and **container** attributes:

```
object room "Room" has light with description "Here.";

object -> cubby "cubby" with name 'cubby'
    has supporter container;

object -> -> book "book" with name 'book';
object -> -> lunchbox "lunchbox" with name 'lunchbox';
```

The game output clearly demonstrates the broken behavior:

```
Room
Here.  You can see a cubby (on which (in which are a book and a
lunchbox).

>x cubby
On the cubby are a book and a lunchbox.

>take all from cubby
book: Removed.
lunchbox: Removed.

>put book on cubby
You put the book on the cubby.

>insert lunchbox into cubby
You put the lunchbox into the cubby.

>x cubby
On the cubby are a book and a lunchbox.
```

Yuck!

Here the normal behavior **looks** good...

...but its not, obviously.

Distinguishing Contained Items from Supported Items

(Still NORMAL behavior.)

Since the default standard library restricts each parent object to just one type of "containment," we normally look at the parent to infer the child's relationship to it. If the parent is a **`container`**, we interpret the child as being "in" it. Likewise if the parent is a **`supporter`**, then we assume the child is "on" it:

```
if(parent(book) has container) "is in";
if(parent(book) has supporter) "is on";
```

Revised Behavior

Instead of being limited to *either* **`container`**, or **`supporter`**, objects can be both. Simply include this extension and the previous code produces a more meaningful transcript:

```
Room
Here.  You can see a cubby (on which are a book and a lunchbox).

>x cubby
On the cubby are a book and a lunchbox.

>take all from cubby
book: Removed.
lunchbox: Removed.

>put book on cubby
You put the book on the cubby.

>insert lunchbox into cubby
You put the lunchbox into the cubby.

>x cubby
On the cubby is a book.
In the cubby is a lunchbox.
```

Right away, this is better.

Everything looks the same.

Except that it is actually working.

Notice the lunchbox and book are both interpreted as initially being **on** the cubby. Since we did not define the type of relationships with the cubby, the extension makes some assumptions, preferring support over containment if both are possible but neither specified.

To specify the relationship explicitly, objects can be declared with either the **contained** attribute, or the **supported** attribute:

```
object -> -> book "book" has supported with name 'book';

object -> -> lunchbox "lunchbox" has contained
    with name 'lunchbox';
```

Which produces the expected output:

```
Room
Here.  You can see a cubby (on which is a book, in which is a
lunchbox).
```

Distinguishing Contained Items from Supported Items (Revised behavior)

Since an object's parent may provide multiple types of containment, testing that parent is no longer a reliable way to determine the relationship. Instead, we have two methods for determining the type of containment:

```
if(isContained(book)) "is in";
if(isSupported(book)) "is on";
```

You might be tempted to check for the existence of the **contained** or **supported** attributes with tests like **book has contained**; however, this approach is unreliable. This extension supports the long-standing practice of assuming children are contained/supported when their parent is only a **container** or only a **supporter**. That is, game authors are not forced to specify the **contained** or **supported** relationship when the parent only provides one option. Using **isContained()** and **isSupported()** will make the appropriate tests for you.

Unsupported Supported (and Contained)

There are a couple of restrictions which may or may not be obvious, but which are called out here to keep things clear:

Undefined, and probably wonky things will happen if you ignore these two common sense rules.

- Although parent objects may support multiple types of containment (e.g. be *both* **container** and **supporter**), child objects can have only one relationship at a time. This is by design. In our previous example, the book cannot be both *in* and *on* the cubby; therefore it is not legal for the **contained** and **supported** attributes to coexist on the same object. Don't do this.

- It makes little sense to declare a child as **contained** when its parent is not a **container**. Ditto for **supported** and **supporter**. So don't do this either.

3: CREATING NEW RELATIONSHIPS

The last feature this extension provides is the ability to define and use new parent-child relationships which can exist side-by-side with **container** and **supporter**.

Normal Behavior

There is no equivalent to this capability provided by the standard library.

Revised Behavior

Adding a new parent-child relationship is easy, as we'll see. In the following example, we enable objects to be "under" their parents, in the same way they can be "in" or "on" them.

To add a new form or containment, we start by defining two attributes: one to show that the parent can *provide* our new relationship (like **container** and **supporter** do), and the other for children when they *have* the relationship with their parent (like **contained** and **supported**). For this example, we'll use the following attributes:

```
attribute under; !when a child is under its parent
attribute cover; !When the parent can have things under it
```

Next we define the relationship itself by creating an instance of the **orParentChildRelationship** class. It is here we specify which of our new attributes applies to the child and which applies to the parent. We also declare the **preposition** used when referring to the relation (in this case "under"):

```
orParentChildRelationship orPcrCover
    with   childAttribute under,
           parentAttribute cover,
           preposition "under";
```

Take note that the **preposition** property is a one-word string. Its a string, because it is printed; however, it is also matched against user input by the parser (even though there may not be a correlating dictionary word). So it should be limited to one word.

That's really all it takes to get things working, since creating an **isUnder()** routine isn't absolutely necessary (in fact, the **isContained()** and **isSupported()** routines we covered previously are just syntactical sugar.) It's still a good idea do this and keep things consistent, though:

```
[isUnder c; return orPcrUnder.isAppliedTo(c);];
```

Now, armed with our new relationship, let's tweak our running example to put shoes under the cubby:

```
object -> cubby "cubby" with name 'cubby'
    has supporter container cover;

object -> -> shoes "shoes" with name 'shoes'
    has pluralname under;
```

With that in place, we can see the relationship in action:

```
Room
Here.  You can see a cubby (on which is a book, in which is a
lunchbox, and under which are shoes).

>put book under cubby
(First taking the book)
You put the book under the cubby.

>x cubby
In the cubby is a lunchbox.
Under the cubby is a book and some shoes.

>take shoes
You take the shoes from under the cubby.

>inventory
You are carrying:
    shoes
```

Notice we didn't need to create a special-purpose verb/grammar rule to get the "put this under" syntax to work. This extension matches user input against the preposition property of any orParentChildRelationship instance to determine how to behave.

As you might expect, there's a fair bit more nuance to **orParentChildRelationship** than I've described here, but these examples demonstrate the most common use cases.

If you'd like to dig deeper, I recommend looking at the orPcrSupporter and orPcrContainer objects to see additional capabilities.

orBackdrop replaces the orProps extension in the original ORLibrary.

orBackdrop

The orBackDrop extension provides a streamlined means of adding scenery, items which the player may **examine** but do little else with.

Normal Behavior

Default Inform supports two approaches for creating scenery in a game. The first is accomplished by defining dictionary words in the **name** property of the location:

```
object clearing "Clearing" has light
  with  description "Thick plants surround you in all
    directions. A hand, rising from the bushes
           to the west beckons to you.",
        name 'plants' 'bushes' 'hand';

Clearing
Thick plants surround you in all directions.  A hand, rising from
the bushes to the west beckons to you.

>x plants
You don't need to reference that in this game.

>x hand
You don't need to reference that in this game.
```

The benefit of this approach is its simplicity and the absence of a need for additional objects to be created; however, responses are limited to a single global message (Miscellany #39) and it lacks overall flexibility.

A bit more customization can be made by leveraging scenery objects:

```
object clearing "Clearing" has  light
  with  description "Thick plants surround you in all
    directions. A hand, rising from the bushes
    to the west, beckons to you.";

object -> hand "hand" has scenery
      with name 'hand',
        description "It gestures for you to follow it to
    the west.",
        before [;
            Examine, Search:
            follow: <<##go w_obj>>;
            default: "The hand probably wouldn't go
            along with that.";
        ];

object -> plants "plants" has scenery
      with name 'plants'
        description "The plants are thick, mostly
            impenetrable.",
        before [; enter: <<##go w_obj>>; ];

Clearing
Thick plants surround you in all directions.  A hand, rising from
the bushes to the west, beckons to you.

>x hand
The hand beckons you to following it westward.

>take hand
The hand probably wouldn't go along with that.
```

```
>x plants
The plants are thick, mostly impenetrable.

>take plants
You don't need to reference that in this game.
```

As shown, defining a **before** property lets you redirect specific commands. Both of the following...

```
>enter plants
>follow hand
```

...translate into:

```
>go west
```

Revised Behavior

orBackdrop implements the **backDrops** property, which simplifies the previous examples. The most basic implementation looks quite a bit like the previous example, but using **backDrops** instead of **name**:

```
object clearing "Clearing" has light
  with description "Thick plants surround you. A hand, rising
    from the bushes to the west beckons to you.",
       backdrops 'hand' 'plant' 'plants';
```

The resulting transcript is identical to the first example under "Normal Behavior" where examining either the plants or the hand results in the same response:

```
>x plants
You don't need to reference that in this game.
```

Unlike the default behavior, you can specify alternative descriptions immediately following the appropriate dictionary words:

```
object clearing "Clearing" has light
  with description "Thick plants surround you. A hand, rising
    from the bushes to the west beckons to you.",
       backdrops 'hand'
           "It gestures for you to follow it the west."
           "The hand is elusive." bdEnd
           'plant' 'plants'
           "The plants are thick, mostly impenetrable."
           bdDefault bdEnd;
```

There aren't any commas used here. These are all entries in the backdrops property array.

Notice the archetypical pattern here: some number of dictionary words, followed two strings, and **bdEnd**. We see this pattern repeated above, effectively creating two scenery objects with the first string serving as our description:

```
>x plants
The plants are thick, mostly impenetrable.

>x hand
It gestures for you to follow it the west.
```

Attempting any action other than examine results in the second string:

```
>take hand
The hand is elusive.

>take plants
You don't need to reference that in this game.
```

As you can see, **bdDefault**, may be substituted for a string, which will use the standard library's default message (**##Miscellany, 39**).

Verbs Directly Referring to Backdrops

Both the description and "you can't do that" strings can be replaced with routines. For the latter, this allows us to redirect actions which wouldn't normally be allowed against scenery:

```
object clearing "Clearing" has light
  with description "Thick plants surround you. A hand, rising
          from the bushes to the west beckons for you to
          follow.",
       backdrops 'hand'
          "It gestures for you to follow it to the west."
          [; if (action==##Follow) <<##go w_obj>>;
              "The hand probably wouldn't go along with that."
          ]  bdEnd
          'plant' 'plants'
          "The plants are thick, mostly impenetrable."
          [; if (action==##Enter) <<##go w_obj>>; ]  bdEnd;
```

As we saw in the final "Normal Behavior" example, this translates **enter plants** and **follow hand** into **go west.** Note the return value of this handler is relevant. If **false**, the typical new line is printed as normal, but is suppressed if the return value is **true** (as is the case with the **<<…>>** syntax). In the above example, we expect the **GoSub** routine to handle the new line as it normally would.

Alternate Backdrop Properties

The 32 entry cap is a Z-Machine limitation. If you are compiling to Glulx, this does not apply; however the variants of the **backdrop** property are still available.

Since the **backdrops** property is a property array, it is limited by the virtual machine's maximum number of entries for a single property. For the Z-machine, this upper limit is 32 entries on a single property. The above example takes up 9 of these for the two scenery items it defines. For locations with many scenery items, the additional properties **backDrop1**, **backdrop2** and **backDrop3** are supported, providing a total of four lists with 32 individual entries each (on the Z-machine).

The orBackdrop Class

Finally, as an alternative to defining the **backdrop** properties on the location itself, these may be defined on instances of the **orBackdrop** class, which can be placed in specific locations or float around via **found_in**:

```
orBackdrop with found_in clearing northPath southPath,
    backdrops 'plant' 'plants'
        "The plants are thick, mostly impenetrable."
            bdDefault bdEnd
        'air' 'wind' 'breeze'
        "The slight breeze is warm and smells of spring."
            bdDefault bdEnd
        'light' 'moon'
        "Luna shines down at you, lighting your path."
        "The moon is unreachable." bdEnd;
```

07 Miscellaneous Enhancements

Standard Library Enhancements

Here we cover extensions serving a variety of purposes which don't quite fit into the buckets we've defined so far.

orMenu

The orMenu extension provides a menu which presents choices to the player, allows them to navigate through a tree of options, and allows them to make selections.

Normal Behavior

The standard library does not provide equivalent menu functionality.

Actually, there's a whole section in the DM4 (§44) covering the optional **Menus.h** file. It's described as though the file is included with the standard library itself, and may once have been, but all archived versions of the standard library in the IF-archive do *not* include it. You can find it at the IF-archive alongside other I6 extensions, though.

Revised Behavior

Instances of the orMenu class, each representing a menu option, are arranged in the same parent-child relationship game world objects are. Here's a simple example:

```
orMenu topMenu "Choose from the following";
    orMenu -> "Do stuff with sticks.";
        orMenu ->-> takesticks "Pick up sticks." ;
        orMenu ->-> burnsticks "Set sticks on fire.";
    orMenu -> "Look at stuff";
        orMenu ->-> "Look at the fire ash."
            with description “Soot and grime.”;
        orMenu ->-> "Look in the river."
            with description “There’s a fish in there.”;
        orMenu ->-> "Look at the fish."
            with description “There’s a fish in there.”;
```

As a pattern, **orMenu** objects are arranged in a hierarchal fashion, with a single top-most parent presiding over a collection of children and grand children. The above demonstrates this, with **topMenu** acting as the entry point to the menu itself.

Activating the menu is done with a call to **show**:

```
result=topMenu.show(orMenuFullScreen);
```

The parameter specifies how the menu should be displayed. It can be either:

orMenuFullScreen, to cause the menu to take up the entire display, clearing it first. This is often used in help systems.

orMenuTopOnly, to keep the game text on the screen and restrict the menu to a minimum number of lines at the top. This is especially useful when prompting the player to choose from several options like, for example, in the CYOA model of play.

"CYOA," or "Choose Your Own Adventure," was a series of choice-based stories made popular in the 70's and 80's.

When an **orMenu** object is **show**n, its children are displayed, but not any of its grand children. Players are expected to navigate the displayed list with the arrow keys and select one to show its children. The Players may back out to the previous menu using a number of synonymous keystrokes, including the ESC and X keys.

Return Value

The show() routine's return value depends on how the menu is defined, and how the player interacts with it, but it can be summed up with two rules:

- When an **orMenu** option is selected which: a) has no children and b) does not define a **description** property, the menu will exit that menu selection is returned.
- If the user backs out of the menu with no selection, a zero result is returned.

the **description** property is ignored for menu objects which also have children.

Differences by Display Mode

There are a few nuances which differ between the two display mode:

In full screen mode, childless menus can define the **description** property. If defined, this will be displayed, inset on the screen, and the menu will continue to run, allowing the player to back out and read other menus. In this mode, childless menus which do not define the **description** property will end the menu session and return the selected menu for the calling process to handle.

In non-full screen mode, childless menus end the menu session and are returned, regardless of the presence of the the **description** property.

Also, the **description** property is only used in full screen mode.

orStatusLine

The orStatusLine extension simplifies modifying the game's status bar, allowing for independently defined left-aligned, centered, and right-aligned text elements.

Normal Behavior

By default, changing the standard library's default status bar requires developers to **replace** the **DrawStatusLine** routine with their own version. There's a fair amount of nuance involved with this approach, usually requiring the use of assembly opcodes, and screen width calculations.

I opted to *not* give examples of custom DrawStatusLine implementations here, since there are several to be found in the DM4 §42.

Even the shortest of these is messy, IMHO.

Revised Behavior

orStatusLine makes typical status line modifications simpler by introducing three entry points, usually defined as text printing routines, but which just as easily could be string constants:

orStatusLeft usually prints the name of the room. This text is left-aligned in the status bar.

orStatusMiddle text is printed in the middle of the status bar.

orStatusRight - text is aligned to the right of the status, usually the score.

By default, this extension mimics the standard library's behavior, so including it is transparent, but you can define the above routines before **#verblib** to change this:

Actually, this isn't 100% true. While this extension right-aligns the score text, Inform's default status bar instead reserves space for the highest possible score + turn combination. This is why the status of most Inform games end in several unused spaces.

```
[orStatusRight;
    if(score==0) print "Peace keeps no score";
    else print "Score: ",score;
];
[orStatusMiddle; print "GOAL:",(string)currentTask; ];
```

Depending on the value of various game variables, this would produce something like:

Notice we did NOT define **orStatusLeft**, so the default behavior for the room area remains unchanged.

```
Temple                    GOAL:Find alter          Peace keeps no score
```

08

Standard Library Enhancements

Additional Verbs

This collection of extensions adds, or modifies, simple verbs which crop up occasionally in modern works of IF.

Such as **ASK/TELL** which are revised in the orDialogue extension.

Several complex verbs introduced in the orLibrary aren't discussed here but are included with other, more advanced extensions and are covered with those.

orDistinctRead

The orDistinctRead extension draws a conditional distinction between the EXAMINE and READ verbs.

Normal Behavior

By default, READ and EXAMINE are treated as synonyms.

```
object count "couch" with name 'couch', description "A brown
    leather couch.  Comfortable.";

>examine couch
A brown leather couch.  Comfortable.

>read couch
A brown leather couch.  Comfortable.
```

Revised Behavior

orDistinctRead attempts to strike a better balance between the EXAMINE and READ commands. With this extension, READ doesn't automatically mean EXAMINE:

```
>examine couch
A brown leather couch.  Comfortable.

>read couch
You cannot read the couch.
```

Be cautious when defining different responses for **READ** and **EXAMINE**. Many players feel the general form of examination ought to provide the same detail as more-specific forms (such as **READ**, or even **SEARCH**). Requiring the player to use alternate words when the original intention is clear can foster annoyance.

That said, such guidelines rarely apply all the time and there are cases where this approach makes sense.

To implement the questionable practice of using different EXAMINE and READ responses, define the **readDescription** property:

```
object count "couch" with name 'couch', description "A brown
    leather couch with words scratched into the fabric.",
    readDescription "It says ~Kilroy was here.~";

>x couch
A brown leather couch with words scratched into the fabric.

>read couch
It says "Kilroy was here."
```

Making READ and EXAMINE synonymous for individual objects is done by setting **readDescription** to the property value **description**...

```
object poster "poster" with name 'poster',
    description "It depicts an acrobat just over the text ~Come
        see the greatest show on Earth!~",
    readDescription description;
```

orExits

The EXITS command is a simple verb which scans the world map for locations adjacent to the current location, including those separated by doors, and list them as exits:

The EXITS verb will also consider doors when listing exits (assuming they aren't CONCEALED).

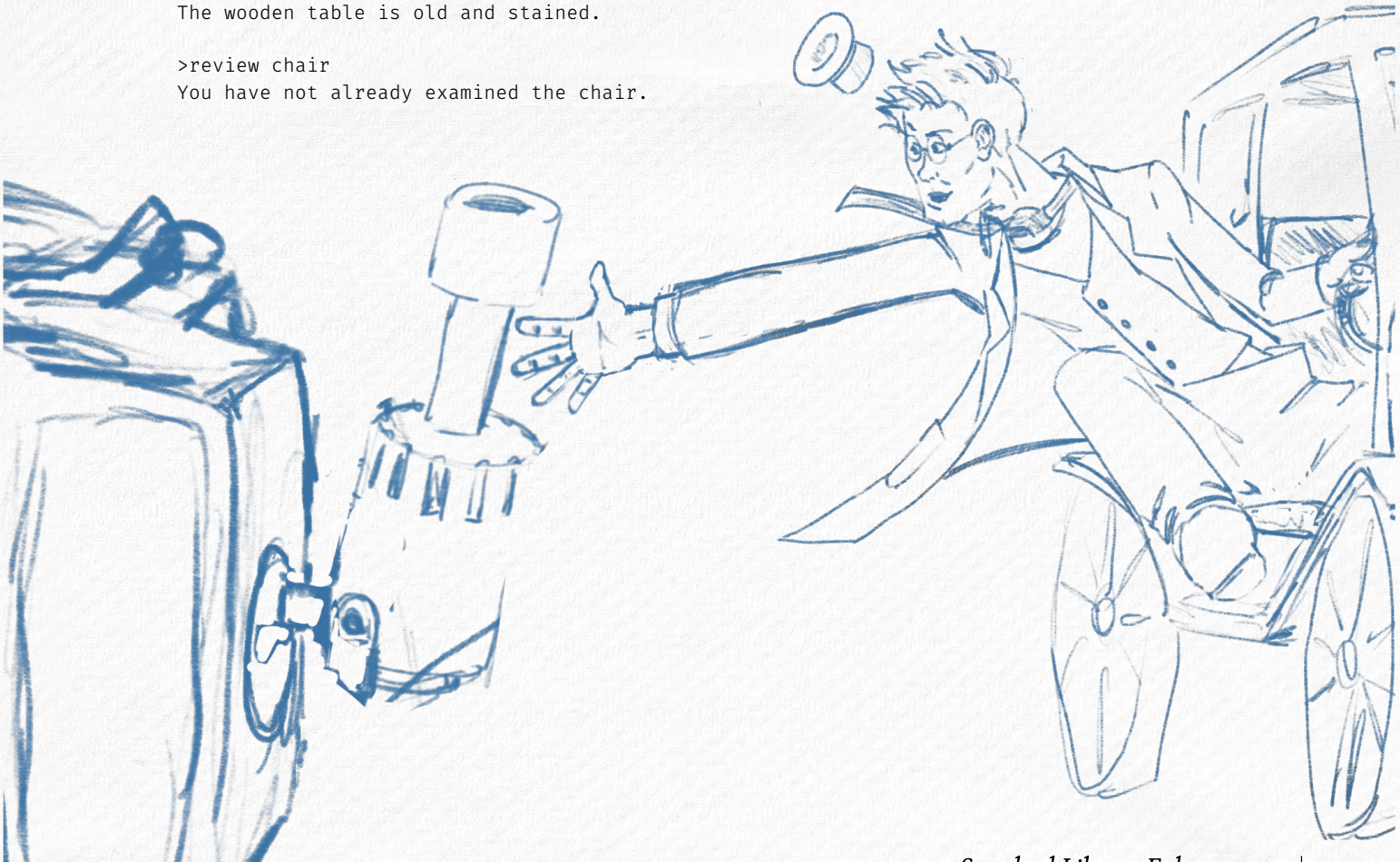
```
object diningRoom "Dining Room" has light
    with description "...",
    n_to laboratory, w_to study, e_to kitchen;

>exits
The only exits lay north, east, and west.
```

Alternatively, a location may define the **exitsText** property which, if it exists, will be printed instead.

orReview

The REVIEW command acts as a **meta** version of LOOK and EXAMINE. That is, it enables the player to review descriptions, which they have previously seen, without the passages of time (i.e., without advancing the turn counter). REVIEW applies to both objects and room descriptions. The following transcript, occurring without increasing the turn count, demonstrates. It assumes the player has already **examine**d the table and the room, but not the chair.

```
>review
Room
A basic room.

You can see a table and chair here.

>review table
The wooden table is old and stained.

>review chair
You have not already examined the chair.
```



orGotoLocation

The orGotoLocation extension enhances the GO verb, allowing the player to specify, by name, visited locations. The character then follows the path to get there, autonomously, one turn at a time, until they arrive.

To make this possible, each location must supply the **name** property with dictionary words which identify the room:

(Actually, the **name** property can be skipped with the **orRecogName** extension.)

```
object laboratory "Laboratory" has light
    with name 'lab' 'laboratory',
    'description "...",
    s_to livingRoom;

object kitchen "Kitchen" has light
    with name 'kitchen',
    'description "...",
    w_to livingRoom;

object livingRoom "Living Room" has light
    with name 'living' 'room',
    'description "...",
    w_to study, n_to laboratory, e_to kitchen;

object study "Study" has light
    with name 'study' 'office',
    'description "...",
    e_to livingRoom, w_to bathroom, n_to closet;

object closet "Closet" has light
    with name 'closet' 'wardrobe'
    'description "...",
    w_to study;

object bathroom "Bathroom" has light
    with name 'bath' 'bathroom',
    'description "...",
    e_to study;
```

The following example transcript assumes the player has previously visited the laboratory and the path between the bathroom and laboratory locations is clear and traversable:

```
Bathroom
>Go to laboratory
You decide to return to the Laboratory…

>>go east
Study

>>go east
Living Room

>>go north
Laboratory
```

The double prompts denote generated commands. See page 25 for more on this and how to change the prompt to your liking.

By default, the GO TO LOCATION verb expects the character's knowledge of the game map to be discovered during play. This translates into the following rules:

- The player character must have previously visited the destination. If the player specifies a location which has not been visited yet, the extension responds with:

```
You don't recall that location.
```

- The player character must remember the path to the destination. In practice, this means the character must have previously visited every location on the path to get there.

The above rules avoid characters automatically knowing the path to locations which have moved. Dr. Who's Tardis, for example, might have teleported to an unexplored portion of the map and would have to be found again before the `GO TO LOCATION` command would work. They also mean the player character will automatically retrace a known path, rather than take a shorter route he is unfamiliar with.

These rules may be turned off to simulate characters already familiar with their environment with the following command:

```
orPlayerPathFinder.avoidUnvisitedLocations = false;
```

There are also a few limitations associated with this extension:

- There is a finite amount of space used by the extension for calculating paths. Path tracing will fail if the length of the resulting path is more than 32 rooms in length. (The max size of Z-machine property arrays.)
- The path tracing routine uses temporary workspace. The calculation will fail if the complexity of the map requires more space than is allocated by the **`PATHDEPTH`** constant (32 by default, but this can be overridden).

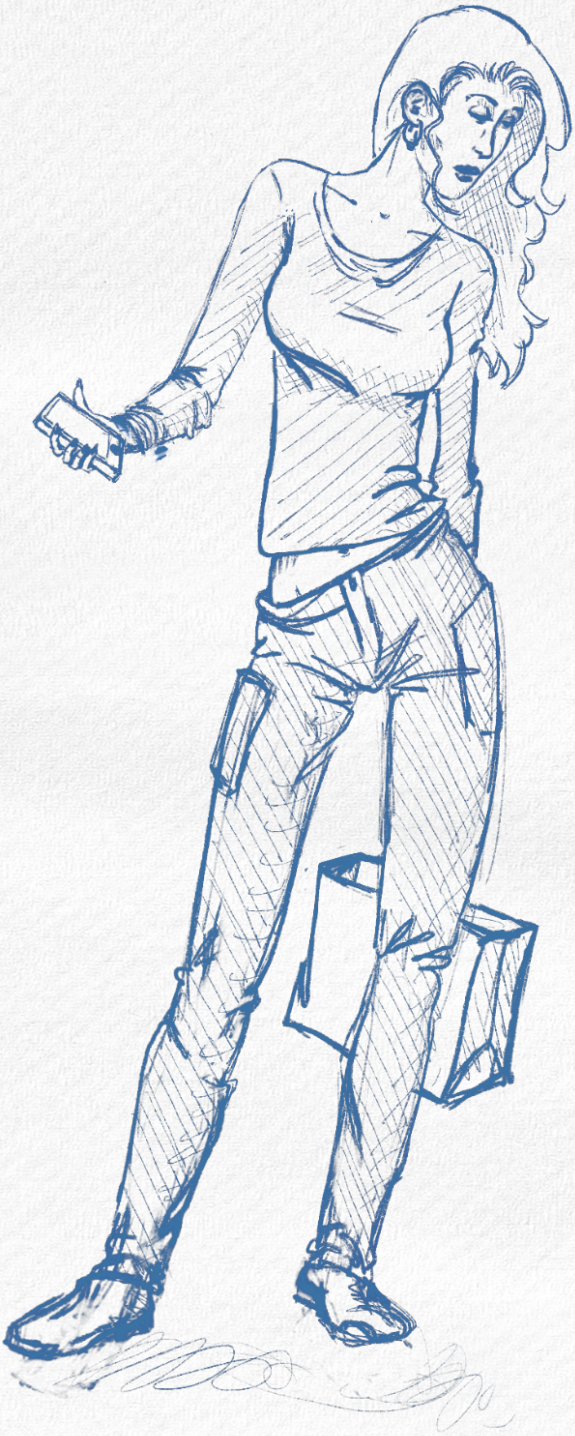
If either of these above limitations are broken, the extension responds with the following:

```
You can't recall how to return to that location.
```

You can override this by redefining the **`L__M(##GotoLocation,1)`** message.

- This extension maps paths through doors; however the actions to open doors do not happen automatically. As such, attempting to move through a closed door will fail and abort the series of queued actions.

(That is, unless you are also using the orImplicitOpen extension which automatically opens doors for the player character as they attempt to move through them.)



Part Three:
Widgets

In this part we cover a few extensions containing purpose-built objects which you can just drop into a project.

09 World Mapping

Widgets

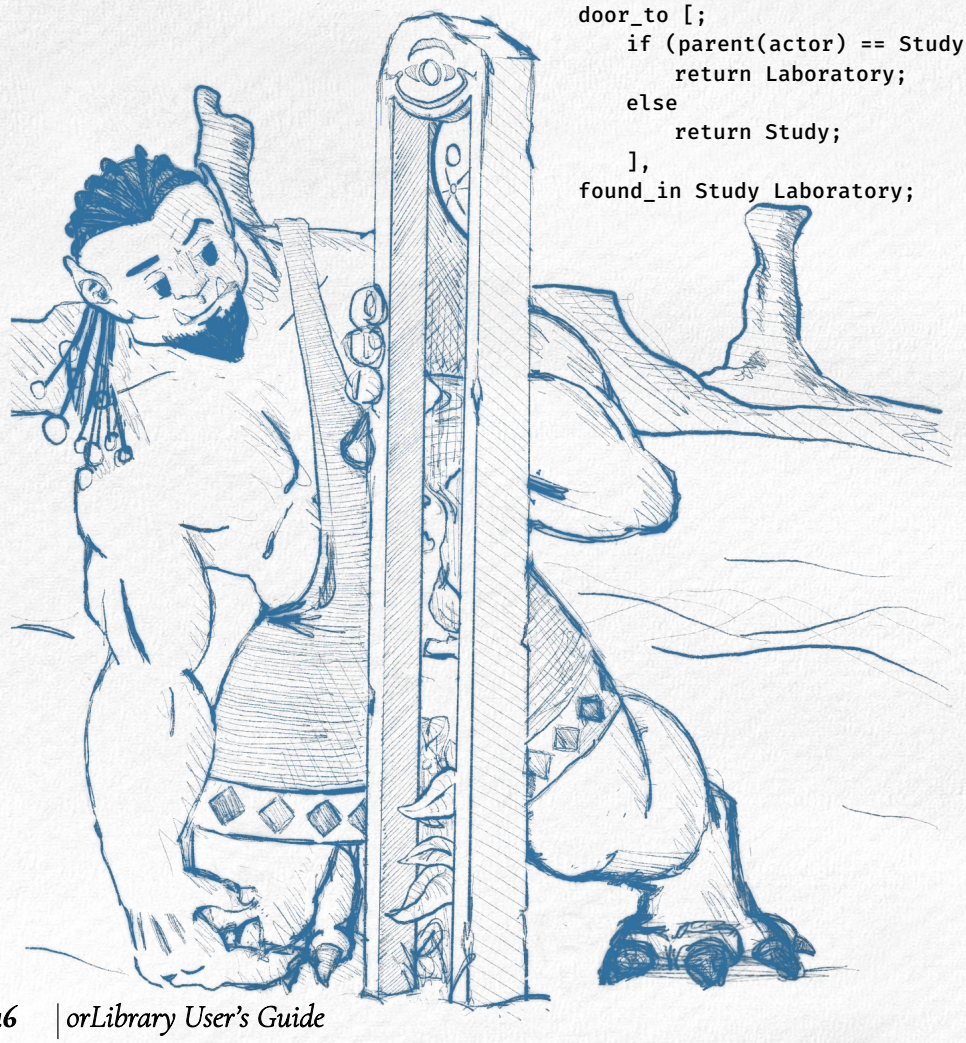
orDoor

The **orDoor** class provides a reusable implementation of the standard door which may sit between game-world rooms. It simplifies the creation of these doors for game developers.

Normal Behavior

The traditional approach for placing a door between two rooms uses location-specific code to tie locations together:

```
object Study "The Study" with w_to SteelDoor;
object Lab "The Laboratory" with e_to SteelDoor;

object SteelDoor "steel door"
    has open openable door static scenery
    with name 'steel' 'door',
        door_dir [;
            if (parent(actor) == Study)
                return w_to;
            else
                return e_to;
            ],
        door_to [;
            if (parent(actor) == Study)
                return Laboratory;
            else
                return Study;
            ],
        found_in Study Laboratory;
```



Note the **door_dir**, **door_to**, and **found_in** properties have knowledge to where the door connects explicitly coded to make everything work as expected:

```
The Study
>w

The Laboratory
```

Revised Behavior

The orDoor extension interrogates the world to automatically populate **door_dir**, **door_to**, and **found_in**. Doors inheriting from the **orDoor** class get this automatically. The **SteelDoor** in our "normal behavior" example is reduced to just a single declarative line:

```
orDoor SteelDoor "steel door" with name 'steel' 'door';
```

orReferByContents

The **orReferByContents** class is used to create containers which can be referred to by the names of their contents.

Normal Behavior

Consider the following, where the player is holding a bottle of marbles:

```
object bottle1 "bottle" selfobj with name 'bottle'
    has open container;

object "marshmallows" bottle1 has edible pluralname
    with name 'marshmallows';

object bottle2 "bottle" selfobj with name 'bottle'
    has open container;

object "marbles" bottle2 has pluralname with name 'marbles';
```

It is perfectly natural for the player to refer to the bottle by its contents:

```
>i
You're carrying:
  a bottle
    some marshmallows
  a bottle
    some marbles

>drop marbles
(first taking the marbles out of the bottle)
Dropped.

>undo
...

[Previous turn undone.]

>drop bottle of marbles
You can't see any such thing.

>drop bottle
Dropped.

>i
You're carrying:
  a bottle
    some marbles
```

Here, the player's intention was to drop the whole bottle of marbles.

Of course, this is a challenge, since the parser doesn't consider what the bottle contains.

Note also that since the bottles aren't unique, the parser can't tell the difference. It makes an arbitrary choice.

The wrong one in this case.

Revised Behavior

By default, any object which inherits from the **orReferByContents** class can be referred to by its contents:

```
orReferByContents bottle1 "bottle" selfobj with name 'bottle'
    has open container;

object "water" bottle1 with name 'water' has edible;

orReferByContents bottle2 "bottle" selfobj with name 'bottle'
    has open container;

object "marbles" bottle2 has pluralname with name 'marbles';
```

This resolves the behavior shown above. Now the parser chooses the correct container for verbs like **Drop**, **Take** or **Throw**:

```
>i
You're carrying:
  a bottle
    some marshmallows
  a bottle
    some marbles

>Drop marbles
Dropped.

>i
You're carrying:
  a bottle
    some marshmallows
```

For example, you wouldn't expect EAT MARSHMALLOWS to refer to the bottle, you'd instead eat the sweets inside of it.

This revised behavior is not appropriate for all verbs, however. Other actions are correctly handled by the default behavior of referring to the content, rather that the container. To cause the container to not inherit its contained object's names for specific verbs, the **ignoreActions** property list can be used:

Note the orReferByContents class does not automatically imply the container attribute since it could just as easily have been a supporter.

```
orReferByContents "bottle" selfobj has open container
    with    name 'bottle'
    ,       ignoreActions ##Eat ##Drink;
```

In this case, verbs like **Drop** and **Take** refer to the bottle, but **Eat** and **Drink** refer to the contents.

Alternatively, when more verbs should be disqualified than allowed, the **referActions** property list can be defined instead.

orDynamicMap

orDynamicMap was originally named orDynaMap in the original ORLibrary.

The orDynamicMap extension enables you to reveal locations to the player in a specific order, regardless of the direction they travel. It fills in the game map as the character moves, making the revealed during exploration permanent. This is a useful technique for generating a sense of expansiveness in a game, while keeping the number of locations relevant and meaningful.

Normal Behavior

This extension was inspired by behavior demonstrated in the "mars" scene of Adam Cadre's "Photopia."

There is no standard library equivalent to this capability.

Revised Behavior

To implement this experience, create an instance of the **orDynamicMap** class with the **found_in** property listing the rooms which are to be arranged, in the order they are to appear.

```
orDynamicMap with cant_go "The trees are too dense.",
    found_in wateredge forestmidst treasuretrove;
```

The first entry in the list, wateredge in this example, is the player's starting point in the dynamic map.

Each of the rooms to be dynamically arranged inherit from **orDynamicMapRoom**.

The following example allows the player to explore in virtually any direction from the water's edge:

```
orDynamicMapRoom wateredge "Edge of water"
    with s_to "Your boat is that way, but lets go explore.",
        description "A small strip separating the south ocean
            from the surrounding forest.";

orDynamicMapRoom forestmidst "Midst of Forest"
    with description "In the forest. Lots of trees.";

orDynamicMapRoom treasuretrove "Pile of Treasure"
    with description "At last! The long-lost treasure!";
```

Once the dynamic allocation has run its course, the **cant_go** message comes into play. All attempts by the player to veer from the already established map will display this, unless the room defines it itself.

Let's assume the player character starts at in **wateredge** location and see the above code play out:

```
Edge of water
A strip separating the south ocean from the surrounding forest.

> s
Your boat is that way, but lets go explore.

> n
Midst of Forest
In the forest. Lots of trees.

> n
Pile of Treasure
At last! The long lost treasure!
```

South is already defined, so it doesn't fill in dynamically.

Notice the rooms play out in the order defined in the orDynamicMap's found_in property.

```
> undo
Midst of Forest

>undo
Edge of water/forest

>w
Midst of Forest
In the forest. Lots of trees.

>w
Pile of Treasure
At last! The long lost treasure!

>w
The trees are too dense.

>e
Midst of Forest
```

If we undo the state-machine and rewind these moves....

...exploration in different directions will reveal the same locations in the same order.

When the list of rooms is exhausted, the cant_go message kicks it.

But the generated map remains in effect.

Miscellaneous Game Objects

10

Widgets

Here we cover pre-built objects and object classes, ready to drop into a game to meet specific use cases.

orNameable

The **orNameable** class is used to create objects which can be named by the player. Subsequently, the parser will accept that name as a valid reference to the object.

Normal Behavior

By default, the standard library does not support player-assigned names.

Revised Behavior

Use of orNameable is straightforward: simply inherit an object from the **orNameable** class to give it this functionality:

```
orNameable sword "sword" with name 'sword',
    description "Sharp and pointy.";
```

During the game, the player may use any of a handful of new syntaxes to provide custom names:

```
> name the sword "Excalibur"
You begin to think of the sword as "Excalibur."

> x Excalibur
Sharp and pointy.
```

The quotes around the new name are optional in either case.

Nameable objects currently support only one player-defined name at a time, so a subsequent naming replaces the previous:

The **NAME** command and **REFER TO** syntaxes are synonymous. They resolve to the same verb.

```
> refer to the sword as Durandal
You begin to think of the sword as "Durandal."

> x Excalibur
You can't see any such thing.

> x Durandal
Sharp and pointy.
```

The player character can also forget the assigned name:

```
> unname sword
You forget any name you may have given the sword.

> x Durandal
You can't see any such thing.
```

orNumberedContainer

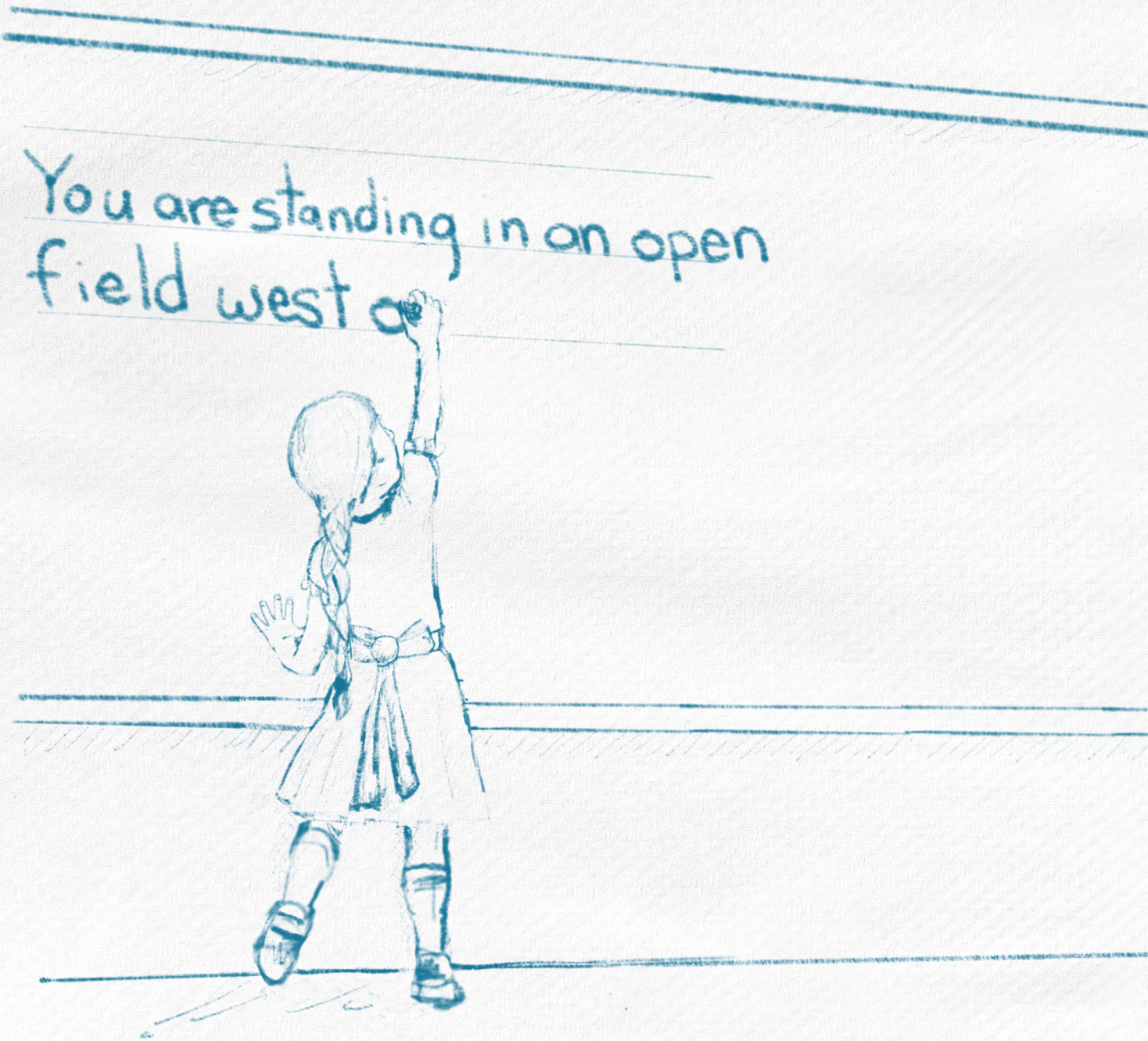
TODO

Normal Behavior

By default, the standard library does not have an object with multiple compartments.

Revised Behavior

Isn't there a box too?


You are standing in an open
field west o

Part Four:
Advanced Text

It's ironic that Inform 6, a product dealing almost exclusively in text, offers minimal support for text manipulation. Extensions in this section focus primarily on text, enhancing features which exist in Inform today, or providing features common in other languages, but missing in Inform.

11 Text Routines

Advanced Text

This section contains simple routines which improve and build upon what is currently available in the standard library.

orCenter

The orCenter extension provides a routine, **orCenter**, which centers text on the screen. At first blush, this routine seems the same as the **centre** routine; however, **orCenter** is more sophisticated than its standard library counterpart.

Normal behavior

The standard library's **centre** routine works well for centering short bits of text but leaves some room for improvement when dealing with more complex use cases. The following intermingled code and transcripts demonstrate:

Imagine these lines are the window boundaries of your interpreter...

```
centre("How shall the burial rite be read?");
```

```
            How shall the burial rite be read?
```

```
centre("How shall the burial rite be read?^The solemn song be
    sung?");
```

```
     How shall the burial rite be read?
The solemn song be sung?
```

Fail!

```
centre("How shall the burial rite be read? The solemn song be
    sung? The requiem for the loveliest dead...");
```

When text exceeds the screen width, centre() just gives up.

```
How shall the burial rite be read? The solemn song be sung? The
requiem for the loveliest dead...
```

```
centre("How shall the burial rite be read?^The solemn song be
sung?^The requiem for the loveliest dead,^That ever died so young?
^^-Edgar Allan Poe ~A Paean~");
```

Yikes!

```
** Library error 14 (160, 21311) **
How shall the burial rite be read?
The solemn song be sung?
The requiem for the loveliest dead,
That ever died so young?

-Edgar Allan Poe "A Paean"
```

Revised Behavior

The **orCenter** routine handles longer strings of text than **centre** and correctly supports embedded linefeeds. All lines are centered appropriately, even when the text length exceeds the screen width.

Compare the following output with the same calls to **centre**, made above:

```
orCenter("How shall the burial rite be read?");
```

```
            How shall the burial rite be read?
```

(Same.)

```
orCenter("How shall the burial rite be read?^The solemn song be
    sung?");
```

```
                How shall the burial rite be read?
                     The solemn song be sung?
```

```
orCenter("How shall the burial rite be read? The solemn song be
    sung? The requiem for the loveliest dead,");
```

```
 How shall the burial rite be read? The solemn song be sung? The
                  requiem for the loveliest dead,
```

```
orCenter("How shall the burial rite be read?^The solemn song be
    sung?^The requiem for the loveliest dead,^That ever died so
    young?^^-Edgar Allan Poe ~A Paean~");
```

```
                How shall the burial rite be read?
                     The solemn song be sung?
               The requiem for the loveliest dead,
                     That ever died so young?

                     -Edgar Allan Poe "A Paean"
```

For the Z-Machine, the orCenter() routine relies upon mono-spaced fonts to achieve accurate centering calculations and will temporarily assume that mode when centering text. This is not true for Glulx.

Look at all the centered beauty-ness which **centre()** doesn't do!

There are two optional parameters which can be used to affect the output. The first is **maxwidth**, which determines the maximum width of each line and inserts word-wrapped line breaks accordingly. This allows box-like sections of text to be centered. If not supplied, the screen width, as provided by the interpreter, is used. Adding a smaller **maxwidth** to the example above produces narrower text lines, still centered:

```
orCenter("How shall the burial rite be read?^The solemn
song be sung?^The requiem for the loveliest dead,
^That ever died so young? ^^-Edgar Allan Poe ~A Paean~",23);
```

```
                    How shall the burial
                        rite be read?
                   The solemn song be sung?
                     The requiem for the
                       Loveliest dead,
                   That ever died so young?

                       -Edgar Allan Poe
                          "A Paean"
```

The second, optional, parameter is the **highlight** parameter. This can be passed as either **NO_HIGHLIGHT** (the default value if unspecified), **HIGHLIGHT_TEXT_ONLY**, or **HIGHLIGHT_ALL**. Specifying a value of **HIGHLIGHT_TEXT_ONLY** centers the text in reverse:

```
orCenter("I'm just a Poe boy; I need no sympathy.");
```

```
                I'm just a Poe boy; I need no sympathy.
```

A value of **HIGHLIGHT_ALL** does the same, but also reverses the spaces which proceed the text:

```
                I'm just a Poe boy; I need no sympathy.
```

Generally, a value of HIGHLIGHT_ALL is useful when printing on the status line.

In addition to the **orCenter** routine, this extension also provides the **orInset** routine. Like **orCenter**, **orInset** will wrap and center text; however, instead of centering each line independently, **orInset** centers a block of left justified text.

Here's an example:

```
orInset("How shall the burial rite be read?^The solemn
song be sung?^The requiem for the loveliest dead,
^That ever died so young? ^^-Edgar Allan Poe ~A Paean~",23);
```

```
How shall the burial
rite be read?
The solemn song be sung?
The requiem for the
Loveliest dead,
That ever died so young?

-Edgar Allan Poe
"A Paean"
```

orTransition

The **orTransition()** routine creates a transitional effect, appropriate for separating sections of text, like chapters. It prints separating text and prompts the user to press the space bar. Once acknowledged, it clears the screen.

```
orTransition("You've finished part one.^Continue at your own risk...");
```

Notice how the text is centered, compensating for the embedded new_line. This extension uses orCenter() to do that (see page 57).

```
You've finished part one.
Continue at your own risk...

Press SPACE to continue.
```

Optionally, the **text** and **skipErase** parameters can be supplied (in that order) to change the separation text and avoid clearing the display, respectively. Here's an example using both:

```
orTransition("You've arrived at the Space Bar & Pub.", "Only the
    SPACEBAR bars this space.",true);
```

It's hard to demonstrate clearing the display in a static guide, so you'll have to take my word for what the **skipErase** parameter does.

```
You've arrived at the Space Bar & Pub.

Only the SPACEBAR bars this space.
```

Dynamic Text

12

Advanced Text

Here we deep dive into some of the most advanced text manipulation routines available for Inform.

orGibberish

The orGibberish extension prints random, meaningless, human-pronounceable words of varied length. Making gibberish pronounceable isn't as straightforward as stringing together random characters; there are phonetic principals to be honored. This extension wraps these principals into a single, straightforward call taking one parameter to specify the number of ***syllables*** the gibberish word should contain:

```
for(t=0:t<4:t++) orGibberish.printWord(2),” “;
zaquask squeesle akrou rekril
```

At first glance, it may not be obvious how to make effective use of this in a work of IF (printing gobbledygook can only get you so far, after all) but pairing orGibberish with other extensions opens up some interesting possibilities. For example, the **orString** object lets you capture and reuse dynamic text, and the orRecogName extension lets players refer to game objects by their printed names, even if there are no correlating dictionary words.

I cover the orRecogName extension on page 20.

So random passwords, monster names, and spell incantations become potential uses for orGibberish.

Example Use Case

By way of demonstration, here's an sample object which takes on a different random name every time the game begins:

```
#include "orGibberish";
#include "orRecogName";
#include "orUtilStr";
```

Include these before #parser.

...

```
object start "startRoom" has light, description "Here.";
object -> myThing with name 0
    ,   short_name[;
            if(self.name==0){
                self.name=util.orStr.new().capture();
                orGibberish.printWord(2);
                self.name.release();
            }
            self.name.print();
            rtrue;
        ];


startRoom
Here.

You can see a drouskroo here.

>x drouskroo
You see nothing special about the drouskroo.
```

orString

Also, the Z-machine was designed to work on low-powered hardware built in the 80s. String manipulation can be resource intensive.

Most modern, object-oriented languages formally include "strings" as a core data type for holding and manipulating text. Vanilla Inform does not.

Inform's lack of native string objects is likely due to a restriction in the Z-machine which Inform historically compiled to: The Z-machine does not allocate memory at run-time. In modern languages, memory allocation is a cornerstone of text functionality: creating a string, then modifying it, invariably produces multiple memory allocations and deallocations under the covers.

The orString extension adds the concept of string objects, enabling authors to declare strings and perform advanced operations with them. There's a lot to the orString extension and it can be difficult to know where to start. For context, here's a quick sampling of some basic things you can do with an **orString** object, presented as an example:

See page 108 for more information on the util.orStr utility object.

```
str=util.orStr.new("this is a whole bunch of text.");

str.left(7).print();
this is

str.substring(8,16).append(" written words.").print();
a whole bunch of written words.

str.right(5).format("I'm going to write some $0").print();
I'm going to write some text.

str.free();
```

Note: orStrings are zero-indexed. The first character is index zero.

Dynamic Allocation vs. Static Allocation

In the above example, note the call to the **new()** routine:

```
str=util.orStr.new("this is a whole bunch of text.");
```

This line pulls an **orString** object from the string pool and initializes it with a value. Conceptually, this is similar to allocating memory in other languages (for example, with **malloc** or **new**) and requires the developer to release the allocated **orString** when finished using it:

```
str.free();
```

Failing to do this constitutes a memory leak which, given enough iterations, will lead to the following out of memory error:

```
[ERROR: Unable to allocate a new instance.]
```

As an alternative, **orStrings** may be declared statically, at the global scope. The following creates a static **orString** and the underlying buffer it uses:

```
array myBuf buffer orStringDefaultSize;
orString str with primaryBuf myBuf;
```

The advantage of static declarations is the lack of memory management responsibilities. In fact, because neither the **orString** nor **buffer array** come from the **orString** memory pools, you *cannot* **free** them or unexpected things will happen.

A disadvantage of this approach is that the **orString** and **buffer array** both reserve memory for the entire duration of the program, even when unused. In cases where **orStrings** are used routinely, it is usually more efficient to create and dispose of them explicitly.

Ephemeral Return Values

With a few exceptions, **orString** operations do not modify their underlying values. You can see this in the following example, where **replace** returns the result of the operation but leaves the original value unchanged:

```
str.set("this is a whole bunch of text.");
str.replace("whole bunch", "little bit").print();

this is a little bit of text.

str.print();

this is a whole bunch of text.
```

Sets the value of an already existing string named **str**.

Unchanged.

To pull this off, **orString** operations return temporary, short-lived **orString** objects, called "ephemeral" strings. ***Ephemerals are guaranteed safe only in the moment they are returned.*** Beyond this, they are unsafe for use and will change in mysterious and unpredictable ways.

As a general guideline, be cautious of assigning **orString** return values to variables:

```
revStr = str.reverse();
```

Danger!

This practice, which would be perfectly appropriate other languages, assigns an ephemeral **orString** to the **revStr** variable, overwriting the reference to whatever object was previously assigned. Subsequent **orString** operations will eventually reinitialize and return the same **orString** object now referenced by **revStr**. (Because it's ephemeral)

If you need to persist a returned ephemeral, you have a couple of options. The first is to copy the ephemeral result to an existing "safe" (non-ephemeral) **orString** using the **set()** method:

```
safeStr=util.orStr.new("feebie");

ephemStr = safeStr.upper().reverse().right(4).append(" it's
    what's for dinner.");

safeStr.set(ephemStr);
```

Notice that orString operations can be chained together, with each operation applying to the results of the previous.

Copies the value of the ephemeral string into a safe string.

As a result, the **safeStr** string now contains the previously returned value, safe from behind-the-scenes disruption. No additional memory management efforts are necessary.

A second approach is to call the memory management routine **lock()** on the ephemeral object itself which removes it from the pool of recyclable objects:

```
ephemStr.lock();
```

Of the two approaches, **lock()** is faster, since it bypasses allocation of a new **orString** object and the subsequent string copy to initialize it.

In this case, **ephemStr** ceases to be ephemeral, and must eventually be released:

```
ephemStr.free();
```

If you lock it, you must free it.

Remember that the Z-machine allocates memory at compile time. Changing these constants will have a very real impact on the size of your compiled game.

Reserving More Space

As mentioned briefly above, the orString extension reserves a pool of recyclable **orString** objects and pool of memory buffers. The size of these pools is defined by the constants **orStringPoolReserve**, (the number of **orString** objects to reserve, 5 by default), and **orBufferPoolReserve** (the number of buffers to reserve, 2 more than orStringPoolReserver, by default).

If you find yourself running out of memory, and you are confident there are no memory leaks, you can expanded these memory pools by defining the appropriate constant before you include **#verblib**. At the time of this writing, each pool can be expanded to a max size of 32.

Right-Sizing orStrings

In most languages, strings are abstractions which sit atop character arrays, or "buffers". **orString** is no exception to this. Under the covers of each **orString** object lies a fixed-length buffer.

Since operations, such as **append()** and **replace()**, can grow the length of the string value, there must be enough fixed space in the buffer to handle increases in size to avoid buffer overflow conditions. Buffer sizes for **orString**s are defined by the constant **orStringDefaultSize**, which is 500 characters by default.

If you need more space to handle larger strings, or want to save space by utilizing smaller strings, simply define this constant with a different value before you include **#verblib**. Remember, just like changing the two pool sizes, increasing this constant will grow your story file, since all **orString** buffers will be affected by this change.

Capturing Printed Text

As we've seen, you can initialize an orString object using the **.set()** method. Alternatively, calling **capture()** will redirect all printed text to the **orString**, until the **release()** routine is called.

```
str.capture();
print "Beetlejuice. ";
str.release();

print "how do you summon him? Repeat...^";
str.print().print().print();

How do you summon him? Repeat...
Beetlejuice. Beetlejuice. Beetlejuice.
```

Member Routines of orString

The following routines are exposed on all **orString** instances:

append(text) returns an ephemeral **orString**, set to the result of adding the **text** property to the end of the current string.

capture() directs output to the string buffer. This is almost always paired with a call to **release** and returns a reference to itself.

clone() returns a copy of the current string as an ephemeral **orString** object.

contains(searchText) returns **true** or **false** if the current value contains the text specified by the **searchText** parameter.

delete(pos, count) returns an ephemeral **orString**, set to the result of removing **count** number of characters from the current string, starting at position **pos**.

`equals(altBuf, caseInsensitive)` returns **`true`** if the current value is equal to the parameter **`altBuf`**, otherwise **`false`**. This comparison will be made without regard to case if the **`caseInsensitive`** parameter is **`true`**.

`equalsOneOf(str1, str2, str3, str4)` returns **`true`** if the current value is equal to one (or more) of the parameters; otherwise **`false`**.

`equalsOneOfCaseInsensitive(str1, str2, str3, str4)` returns **`true`** if the current value is equal to one (or more) of the parameters, independent of case; otherwise **`false`**.

`free()` removes locks applied to the **`orString`** object, making it ephemeral and returns a reference to itself.

`format(pattern, p1, p2, p3)` returns an ephemeral **`orString`**, set to the resolved value of the pattern parameter. Resolution occurs with the following default rules:

> **`$0`** is replaced by the current **`orString`** value.
> **`$1`** is replaced by the first parameter following **`pattern`**.
> **`$2`** is replaced by the second parameter following **`pattern`**.
> **`$3`** is replaced by the third parameter following **`pattern`**.

In addition to the default resolution rules, if the orPrint extension is included, **`format`** will also resolve print rules embedded in **`pattern`**.

`getLength()` returns the length of the current string.

`getBuf(suppressAllocation)` returns the buffer underlying this object, allocating a new one if necessary (unless **`suppressAllocation`** is **`true`**).

`getChar(pos)` returns the character at the **`pos`** position in the string.

`indexOf(searchText, startingIndex)` searches the current value for the first occurrence of **`searchText`**, starting at the **`startingIndex`** position, and returns the index where it was found, or **`-1`** if not found.

`indexOfFirstFalse(routine, startingIndex)` passes every character from **`staringIndex`** forward to the routine specified by **`routine`**, until the result is **`false`**, returning the position or **`-1`** if **`false`** is never returned.

`indexOfFirstTrue(routine, startingIndex)` passes every character from **`staringIndex`** forward to the routine specified by **`routine`**, until the result is **`false`**, returning the position or **`-1`** if **`true`** is never returned.

`insert(pos, source, count)` returns an ephemeral **`orString`**, set to the result of inserting **`count`** number of characters from source into the current string, starting at position **`pos`**.

`isBufferLocked()` returns **`true`** if the underlying buffer of the **`orString`** object has been removed from the free memory allocation pool, otherwise **`false`**.

`isEmpty()` returns **`true`** if the string is empty, otherwise **`false`**.

`isLocked()` returns **`true`** if the **`orString`** object itself has been removed from the free memory allocation pool, otherwise **`false`**.

I don't call it these, but you could think of isLocked as isSafe() or isNotEphemeral().

`left(count)` returns an ephemeral **`orString`** equal to the **`count`** left-most characters of the current string.

`lock()` applies a lock to the **`orString`** object so it will not be allocated and reused by subsequent allocation requests. Returns a reference to itself.

`lower()` returns an ephemeral **`orString`** equal to the current string, converted to lowercase.

`mid(start, count)` returns an ephemeral **`orString`**, set to a string of **`count`** characters from the current string, starting at position **`start`**.

`numOccurances(searchText, startingIndex)` returns the number of times **`searchText`** appears in the current string, starting at position **`startingIndex`**.

`print()` prints the current string and returns a reference to itself.

`prepend(text)` returns an ephemeral **`orString`**, set to the result of adding the **`text`** property to the front of the current string.

`release()` redirects output to the screen, following a previous call to **`capture`**. Returns a reference to itself.

`replace(searchText, replaceText)` returns an ephemeral **`orString`**, set to the result of replacing the first occurrence of **`searchText`** with **`replaceText`** in the current string.

`replaceAll(searchText, replaceText)` returns an ephemeral **`orString`**, set to the result of replacing all occurrences of **`searchText`** with **`replaceText`** in the current string.

`reverse()` returns an ephemeral **`orString`**, set to the value of the current string with all characters in reverse order.

`right(count)` returns an ephemeral **`orString`** equal to the **`count`** right-most characters of the current string.

`set(str)` sets the current string to the value contained in the **`str`** parameter and returns a reference to itself.

`setLength(newLength)` forcibly sets the length of the current string to the value of **`newLength`**. This does not modify the underlying buffer, so whatever is there from previoius operations will become part of the current string.

`setChar(pos, char)` changes the value of a single character at position **`pos`** in the current string to **`char`** and returns a reference to itself.

`trim()` returns an ephemeral **`orString`** equal to the current string, but with leading and trailing spaces removed.

`trimLeft()` returns an ephemeral **`orString`** equal to the current string, but with leading spaces removed.

`trimRight()` returns an ephemeral **`orString`** equal to the current string, but with trailing spaces removed.

`upper()` returns an ephemeral **`orString`** equal to the current string, converted to uppercase.

orPrint

The orPrint extension provides a powerful tool for generating text which changes based on game state. Traditionally, we do this with **print** statements, and printing rules, combined with if-then logic. That approach continues to work; orPrint simply expands your options.

Core Concepts

Here we cover the basic concepts of the orPrint extension. Additional features of orPrint build upon these.

Print Patterns

Über geeks can find a formal definition of a print pattern, described in Augmented Backus-Naur Form (ABNF), on page 133.

At the heart of orPrint is the idea of a "print pattern." Print patterns are expressions embedded in your text which are evaluated at run-time; the result of the evaluation is printed instead of the pattern text itself.

Very loosely, print patterns follow this syntax:

For the rest of us philistines, this is roughly correct.

```
$patternName:object(paramString);
```

Because the constituent pieces of the syntax are generally optional, and in some cases move around, the structure of two resulting print rules may look very different.

Simple Example

$patternName:object(paramString);

Print patterns support advanced use cases and can quickly become complex. Before we jump into the deep-end, let's start with a basic, but instructive, example:

orPrint *always* returns true.

```
return orPrint("Weller says: ~$bold;Now$roman is the time for all
    $underline;good men$roman to come to the aid of their
    country.~");
```

(the above is a lot shorter when using the pattern synonyms/abbreviations, which we'll see in a bit.)

Which prints...

```
Weller says: "Now is the time for all good men to come to the aid
of their country."
```

The above example contains four embedded patterns, all defined just by their **patternName**; none define **object**s or **paramString**s.

Non-Printing Terminator

$patternName:object(paramString);

If you look closely at previous examples, you'll see the semicolon is not always present:

```
~$bold;Now$roman_is
```

This is not a typo. In fact, if the terminating semicolon is not encountered, orPrint will consider the pattern complete *before* the first character which is not a number, letter, underscore, dash, or part of the **paramString**.

Since a space follows **$roman**, orPrint takes that to mean the pattern is finished, terminating it and printing the space. However, had we done this with **$bold**, we would have printed that space improperly:

```
~$bold_Now$roman is

"_Now is
```

Our original use of the semicolon, the *non-printable terminating character*, solves this for us. It terminates the pattern, but does not print.

The paramString

$patternName:object(paramString);

Pattern **paramString**s are optional strings of text, wrapped in parentheses. These can contain most printable characters, including those which would otherwise cause the pattern to terminate. The few characters which are not allowed in **paramString**s, the colon and close-parentheses symbols, can be "escaped" as **$:** and **$)**.

The following demonstrates this using the **$upper** rule:

```
orPrint("$upper(~7~ is truly prime. (~2~ and ~3~ are only
    honorarily so$))");
```

Here we escape the closing parentheses.

```
"7" IS TRULY PRIME. ("2" AND "3" ARE ONLY HONORARILY SO)
```

The colon is reserved because it is commonly used to delineate multiple parameters.

The value of **paramString** is completely dependent on what the specific print rule expects. Some patterns expect a **paramString** of raw text, others expect multiple values, delimited by colons.

Specifying Objects

$patternName:object(paramString);

Many of the prebuilt orPrint patterns accept objects. Which we can provide in one of two ways. First, and most commonly, we specify one of the standard library's four canonical object variables by name:

```
player
actor
noun
second
```

Note that $actor and $noun are **not** patternNames, they are objects. Since a pattern **isn't** specified, orPrint uses the **$default** pattern. I talk more about $default in the "Core Patterns" section on the next page.

For example:

```
orPrint("$actor blesses $noun.");
```

This would produce something like the following, depending on the values of **actor** and **noun**:

```
Father Naru blesses Gotie.
```

The second way is by referencing a parameter to the orPrint routine...

Passing Parameters to orPrint()

We can pass up to four parameters to the **orPrint()** routine and refer to them in patterns by specifying their sequence number:

These examples use the **$default** pattern too.

```
orPrint("$1 materializes to stop $2", spirit, scrooge);

Jacob materializes to stop Ebenezer.
```

Above we pass objects; however, we are not limited to this. We could just as easily have passed other types. Individual patterns handle varying parameter types differently.

Don't confuse passing parameters with the paramString. These are different things.

This distinction about the LDF is more meaningful to authors writing games in languages other than English.

Core Patterns

The following are the orPrint extension's "core patterns." Core patterns mimic, and in some cases rely on, functionality established by the compiler or the standard library, independent of the Language Definition File.

$default

In the previous examples, we supplied objects but did not specify the **patternName**:

```
$actor
$1
```

When **patternName** is not supplied, orPrint will assume the **$default** pattern. The above two lines are exactly equivalent to the following, which we never need to write, but could if we ever felt the urge:

```
$default:actor
$default:1
```

$default applies some meager logic:

orStrings are covered on page 43.

- If the object value is a literal string, or an **orString** object, the text is printed.
- If the object value is a game object (as has been the case for all our examples so far), it is printed with Inform's **name** printing rule.
- If the object value is a routine, the routine is called, without parameters, and any text it prints is output. The return value of the routine is discarded and not printed.
- Anything else, **$default** renders as a number.

We already saw an example of style patterns, in the "Simple Example" section on page 66.

$bold, $underline, $reverse, $roman

The most basic of print patterns are the *style patterns* which let you to embed Inform **style** commands directly into your text. In practice, these usually occur in pairs. The first pattern changes the font; the second returns things to normal:

```
orPrint("~Shut up and $bold;dance$roman!~ said the princess.");
```

```
"Shut up and dance!" said the princess.
```

All four of the z-machine styles have supported patterns, shown in the following table:

Inform print style	orPrint pattern equivalent
style bold	$bold, $b
style underline	$underline, $italics, $i
style reverse	$reverse, $r
style roman	$roman, $normal, $n

Notice these all have pattern **synonyms**, or variant names which function identically, provided to accommodate developer tastes.

$name

Behind the scenes, orPrint's **$name** pattern translates into Inform's **name** printing rule:

```
orPrint("$name:actor $name:1",friend);
```

Remember, **$name:1** means "name of the first parameter," in this case the **friend** object.

Since the **$default** rule automatically resolves to **$name** for objects, we seldom have need to specify it explicitly. The following is identical to the above, but expressed more concisely:

```
orPrint("$actor $1", friend);
```

Both of these equate to the following in vanilla Inform:

```
print (name)actor, " ", (name)friend;
```

$a and $the

Inform's article printing rules have orPrint pattern analogs:

```
orPrint("$The:actor $the:player $A:noun $a:second");
```

This is equivalent to:

```
print (The)actor, " ",(the)noun," ",(A)second," " ,(a)second;
```

Note orPrint patterns are case sensitive: **$A** and **$The** match the capitalized versions of Inform's printing rules; **$a** and **$the** match the lowercase variants.

Also, print patterns are often more powerful than their equivalent printing rules. This is true for **$a** and **$the**: these also support language-specific verb conjugation, which we'll cover in the next section.

(Specifically, on page 71.)

$upper, $lower

The **$upper** and **$lower** print rules, abbreviated as **$up** and **$lo**, print the **paramString** portion of a pattern in uppercase and lowercase, respectively. For example:

```
orPrint("$up(upper) and $lo(LOWER).");
UPPER and lower.
```

I showed an example of **$upper** on page 67.

English Additions

At compile time, the orPrint extension attempts to recognize the language your game is written in and include print patterns specific to that language. In this section, we cover the basic print patterns which only load if the English LDF is detected.

(You can see this detection in action in the compiler output on page 6.)

This section would seem to imply there is currently support for languages other than English. There is not.

Despite a couple of semesters of Spanish and my on-again-off-again relationship with DUO Lingo, the number of languages I can claim fluency in remains firmly pegged at one. The orPrint extension was designed to support languages other than English, but at this time there are none.

$mineYoursIts, $mine

The **$mineYoursIts** pattern, abbreviated as **$mine**, covers possessive pronouns. This pattern will render as "mine", "ours", "yours", "his", "hers", "its", or "theirs".

```
The sword is $mine:actor.
The sword is his.
```

The capitalized versions **$MineYoursIts** and **$Mine** are also available.

$meYouIt, $me

The **$meYouIt** pattern, abbreviated as **$me**, covers accusative pronouns, or those pronouns acting as the *object* of a sentence. This pattern resolves to "me", "you", "him", "her", "it", "us", or "them."

```
You give the baseball to $me:noun.
You give the baseball to her.
```

The capitalized versions **$MeYouIt** and **$Me** are also available.

$myselfYourselfItself, $self

The **$myselfYourselfItself** pattern, abbreviated as **$self**, translates the provided object into the appropriate reflexive pronoun, as dictated by plurality, person, and gender. It renders as "myself", "ourselves", "yourself", "yourselves", "himself", "herself", "itself", or "themselves".

Assuming third-person narrative:

```
orPrint("The sorcerous casts the spell on $self:actor.");
```

Would output something akin to:

```
The sorcerous casts the spell on herself.
```

The capitalized variants **$MyselfYourselfItself** and **$Self** are also available.

$this and $that

The **$this** and **$that** patterns cover demonstrative pronouns, automatically deciding between "this" or "these" and "that" or "those", based on object plurality:

```
orPrint("$This:noun, $that:second.");
This hat, those shoes.
```

I discuss the "object suppression syntax" on page 72 and "verb conjugation" on page 71.

Notice that the name of the object is also printed. This can be disabled using the "object suppression syntax."

The capitalized variants **$This** and **$That** are also available.

$iYoult , $i

Nominative pronouns, or those which act as the subject of a sentence, include "I", "you", "he", "she", "it", "we", and "they".

Remember that the $italics style pattern also supports the synonym $i. This is not a conflict since this pronoun version MUST be qualified with an object, and the style version CANNOT be.

So:
$i:noun prints the pronoun.
$i changes the style

```
orPrint("~$I:actor am '$name:actor.'~");
"I am Legion."
```

As you can see above, the **$iYouIt** pattern may be abbreviated as **$i**. Additionally, since nominative pronouns act as the subject of the sentence, the synonyms **$nom** and **$subj** have been provided. Capitalized versions for all of these synonyms, **$IYouIt**, **$I**, **$Nom**, and **$Subj**, are all supported.

This pattern also supports verb conjugation. (Which I cover on page 71.)

$myYourIts; $my

The **$myYourIts** pattern, which can be abbreviated as **$my**, is used to qualify possession. In English, such words (technically "possessive determiners") proceed the noun they refer to, so this pattern renders as "my", "our", "your", "his", "her", "its", or "their" followed by the name of the object. For example:

```
$my:actor(noun)
her book
```

Capitalized versions, **$MyYourIts** and **$My**, are also available.

Conjugating Verbs

One of the more advanced features of orPrint is its ability to conjugate verbs. To utilize this feature, simply define the **paramString** with one or more verb forms:

I know what you're thinking, but NO. I reject any suggestion that Sir Mix-A-Lot's lyrics are too distasteful to find a home in this guide.

```
$I:noun(like) big butts and $i:noun(cannot) lie.
```

Depending on the combination of tense, person, and object definition, this could render as...

```
The choir likes big butts and they cannot lie.
```

This example deserves a bit of scrutiny. Specifically, notice that only one verb form, present-tense, was supplied. This is usually enough for orPrint, but you can explicitly define the verb variants as a list, separated by colons:

Here, the last verb form, which looks like the third, past-tense form, is "perfect" tense, as in "would have liked." These are not always the same.

```
$I:noun(like:likes:liked:liked)
```

Why didn't we need to specify all forms in the verb we used? The reason is simple: the orPrint module does a good job at filling in the conjugation blanks. It knows the most common irregular verbs, and their forms, plus it understands the standard conjugation rules for regular verbs. As such, most of the time only the first form of the verb is required.

The verb forms, if supplied, must appear in the following order: present-tense, third-person-singular present-tense, simple-past-tense, and perfect-tense. In the case where **orPrint** fails to conjugate the verb appropriately, or if the game designer chooses to leverage an alternate form of verb conjugation, we can specify the correction, leaving the satisfactory default conjugations blank:

```
$a:noun(fly::flied)
```

This is automatically expanded at runtime to:

```
$a:noun(fly:flies:flied:flown)
```

(Obviously this is wrong, but purposefully so.)

$iAmYouAreItIs; $iAm

The verbs of being conjugate differently than typical irregular verbs. Instead of three present and past tense forms, there are five: "am", "is", "are", "was", and "were." This logic is exposed through the **$iAmYouAreItIs** pattern, abbreviated as **$iAm**:

```
$iAm:actor
```

For a plural actor in past tense the following is produced:

```
They were
```

The capitalized versions, **$IAmYouAreItIs** and **$IAm**, are also available.

$imYoureIts; $im

The pattern **$imYoureIts**, abbreviated as **$im**, performs the same conjugation, as **$iAmYouAreItIs**, except contractions are used if available in the current narrative settings:

```
$im:actor
he's
```

The capitalized versions, **$ImYoureIts** and **$Im** are also available.

$imYoureItsNot; $imNot

The pattern **$imYoureItsNot**, abbreviated as **$imNot**, is like the previous **$imYoureIts** pattern, but includes "not" in its consideration to produce contractions:

```
$imNot:actor
He isn't
```

The capitalized versions, **$ImYoureItsNot** and **$ImNot** are also available.

Conjugating Would, Could, and Should

The orPrint extension will also conjugate the auxiliary verbs "would," "could," and "should," simply by including them with the primary verb:

```
$That(would accomplish) little.
```

When appropriate, orPrint conjugates with the "perfect-tense" form of the verb. For example, in a past-tense narrative, the following is produced:

```
That would have accomplished little.
```

Negative forms of these words are also supported, including contractions. The following are all valid possibilities, depending on tense, object plurality, and gender:

```
$The:noun(should not go) to the store.
The runners should not go to the store.

$The:actor(couldn't swing) the axe.
The troll couldn't have swung the axe.
```

The auxiliary verb itself need only be prepended on the first verb form if other forms are to be defined. Extending our previous example of a purposefully wrong conjugation:

Again, wrong with intention.

```
$The:noun(would fly:::flied)
The fairy would have flied to the store.
```

Object Suppression Syntax

Most rules which change text to agree with a provided noun, support "object suppression." That is, they can print the results of the adaptive text, without printing the name of the object.

Although this established sequence of **rule:object** vs. **object:rule** may *seem* arbitrary, it reads intuitively, at least to me. When I write such patterns, the voice in my head reads $this:noun as "this noun." $noun:that just doesn't align grammatically, and so comes across as "that."

The convention to implement this is simple: place the noun before **patternName** instead of after. Recall our previous example under the section covering **$this** and **$that**: (page 70)

```
orPrint("$This:noun, $that:second.");
This hat, those shoes.
```

Using the object suppression syntax, this becomes:

```
orPrint("$noun:This, $second:that.");
This, those.
```

The $default pattern is covered on page 68.

A possibly useful technique combines the object suppression syntax and the **$default** pattern:

```
$noun:(run)
runs
```

The placement of the colon is important. If we leave it out entirely, the normal form **$default:noun(run)** would be assumed, printing the object.

Since **$default** simply prints the object name in this case, and object suppression hides the object name, only the conjugated verb is output.

Using orPrint as a Printing Rule

In cases where parameters are not needed, the **orPrint** routine can also be used as an Inform printing rule:

```
print (orPrint)"$i;Hello.$n $upper(thought) $the:noun.";
Hello. THOUGHT the monkey.
```

Inform English Printing Rules Equivalency

If you scrutinize the English.h file, you'll find a collection of printing rules which are needed to create the dynamic library messages. We don't need specialized orPrint patterns to cover every one of these; the patterns discussed so far cover most. Below is a list of traditional English printing rules and routines used to create adaptive text, next to their orPrint equivalents:

```
ThatOrThose(noun)                       $that:noun
CthatOrThose(noun)                      $That:noun
ItOrThem(noun)                          $it:noun
CSubjectVerb(noun,true, false, verb     $actor(verb)
    verb2, verb3, verbPast)
CtheyreOrThats(noun)                    $Im:noun
IsOrAre(noun)                           $noun:IAm
CsubjectIs(noun)                        $actor:IAm
CsubjectIsnt(noun)                      $actor(isn't)
CSubjectHas(noun)                       $actor(have)
CSubjectWill(noun)                      $actor(will)
CSubjectCan(noun)                       $actor(can)
CSubjectCant(noun)                      $actor(can't)
CSubjectDont(noun)                      $actor(don't)
OnesSelf(noun)                          $self:actor
Possessive(noun)                        $actor:my
PossessiveCaps(noun)                    $actor:My
theActor(noun)                          $the:actor
Tense(verb1, verb2)                     $noun:(verb)
```

If you specify a pattern name orPrint doesn't recognize, it will disregard the entire pattern, printing it out unmodified, dollar sign and all.

Extending orPrint With New Print Patterns

If one of the pre-defined print patterns doesn't suit our needs, we can easily create new ones. To do this, we define an instance of the **orPrintPattern** class, with the **runRule** property routine defined. Here's an example:

```
orPrintPattern orkanLanguage with runRule[;
    if(self.patternName.equals("orkan")) {
        if(self.getParam(1).equals("hello")) "Na-Nu Na-Nu";
        if(self.getParam(1).equals("dang!")) "Shazbot!";
    }
    rfalse;
];
```

With the above in place, we are free to write the following:

```
orPrint("~$orkan(dang!)~ $actor exclaims, then smiles and extends
    his hand.  ~$orkan(hello),~ he greets.");

"Shazbot!" Mork exclaims, then smiles and extends his hand.  "Na-
Nu Na-Nu," he greets.
```

The following properties of the **orPrintPattern** class are useful with writing new orPrint patterns:

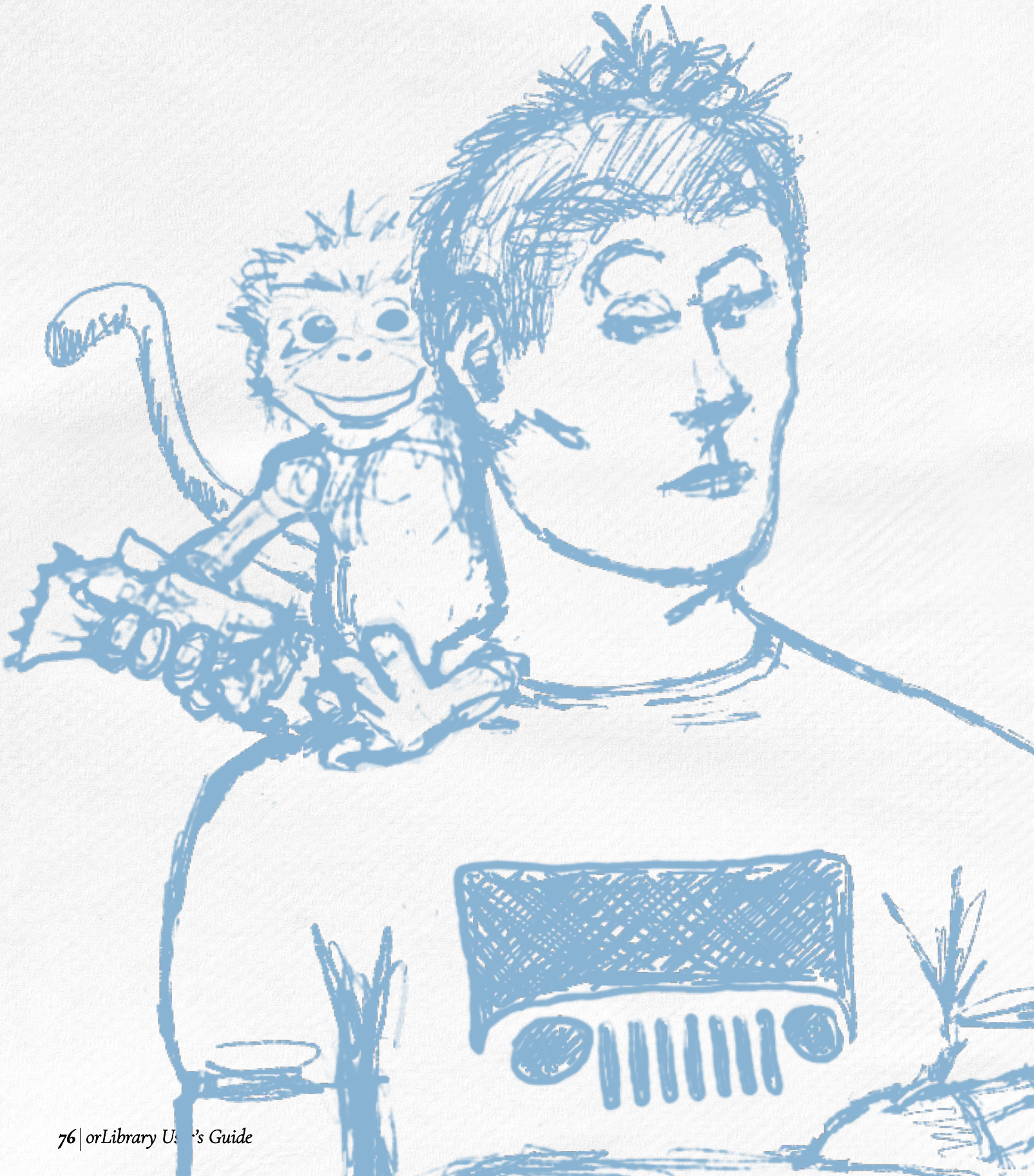
patternName - an **orString** containing the name of the pattern

contextObject - the resolved game object, if one was provided

objSpecifiedFirst - **true** if the object precedes the **patternName** in the expression, otherwise **false**

numParams() - a routine returning the number of parameters in the **paramString**, separated by colons

getParam(number**)** - A routine returning the specified parameter (zero-based index), defined in the **paramString**, separated by colons



Part Five:
Advanced Non-Player Characters

Non-Player Characters (NPCs) are often the most complex objects in a work of interactive fiction. Since they typically represent other people, NPCs should, in theory, be able to do all of the things the player character can do. Depending on the author's vision, they might hold a conversation with another character or perform a chain of actions such as searching for a key before unlocking a door.

The Inform standard library's support for NPCs includes a few staples. For example, it defines the **`animate`** *and* **`talkable`** *attributes to indicate something is alive. In this way, players may* `TALK` *to an NPC, but not a book. Additional support is supplied by the* **`life`** *and* **`orders`** *property routines which provide hooks for developers to code reactions to player commands.*

The orLibrary builds upon this core, providing a framework for crafting NPCs with a spectrum of sophistication ranging from those which simply wait for the player to ask them questions to those which walk around in pursuit of their own agendas.

13

Advanced NPCs

Knowledge and Conversation

Conversation with NPCs is a thoroughly discussed topic in game design forums and there are many approaches to modeling dialogue in Interactive Fiction. No single paradigm is "the best," rather each lends itself to the author's objectives in different ways. In this section, we go over the orLibrary's approach, which includes a variety of ways characters can share and learn knowledge.

_orTopic (An overview)

The **orTopic** class is foundational to multiple conversation-based extensions and is automatically pulled in by these as a dependency. As such, you seldom, if ever, need to **#include** this extension yourself. Here are a few commonly used extensions which are dependent on _orTopic:

Much of the material in this section is covered in others. I collect it here, in one place, for easier reference.

orDialogue Enables the PC to talk to NPCs, either asking them specific questions or telling them specific things.

orNpcSkillDialogue Extends the above to allow autonomous NPCs to ask things of, and tell things to, the PC and/or other NPCs. This extension also introduces a topic pool for tracking relevant topics for the player and NPC.

orDialogueMenu Presents a menu over the PC's topic pool, allowing the player to select from a list what they want the PC to say.

Since conversations can be nuanced, **orTopic** configurations are as well. The chart on the facing page summarizes the most common configuration properties and how to set them.

Additionally, here are a few points which clarify the behavior of **orTopic**s:

- Topics may be declared in a parent/child hierarchy. Most properties, **quip** and **relatedTopics** excluded, will resolve to their parent's value if not explicitly defined; hence the wording "unresolvable" in the facing chart means "not defined on the **orTopic** instance, nor any of its ancestors."

I talk about the ANSWER command in the section covering **orNPCSkillDialogue** on page 90.

- By default, when the player character is asked about a topic by an NPC, they will *not* automatically answer. Instead, if the player chooses to answer, they must do so at the prompt, either with an appropriately constructed TELL command or the ANSWER command.

  Alternatively, if the global variable **orDialogueAutoAnswer** is set to **true**, the PC will automatically answer questions as they are asked, in the same turn.

Quick Reference: Common orTopic Properties

Property	Value	Meaning
`knownBy`	List of in-game characters	Only listed characters may speak the topic.
	`ORDIA_ALL`	All characters in the game can speak the topic.
	routine	A routine returning true or false if the passed in character parameter knows, and therefore may speak, the topic.
	unresolvable value	Only the PC can speak the topic.
`context`	List of in-game characters	The topic can be told only to listed characters.
	`ORDIA_ALL` or unresolvable	No restriction on who the topic may be spoken to: topic can be told to all characters in the game.
	routine	A routine returning `true` or `false` if the passed in character parameter can be told the topic.
`quip` (single value)	text/routine	Text is output when the speaker of a topic `TELL`s it. Topic may not be `ASK`ed about.
`quip` (`ASK`: 1st of 2 entires)	text/routine	The `ASK` quip: Text is output when the inquirer of a topic `ASK`s about it. If a routine, the two conversing characters are passed in as parameters.
`quip` (`TELL`: 2nd of 2 entires)	text/routine	The `TELL` quip: Text is output when the answerer of a question `TELL`s the topic. If a routine, the two conversing characters are passed in as parameters.
	`ORDIA_RESPONSE_ONLY`	The topic cannot be spoken, except when asked about.
`learnable`	`true`	When a topic is spoken to a character that is not listed in the `knownBy` property, the character is added to it. Effectively, the character learns the topic and can subsequently speak it.
	`false` or unresolvable	Characters being told a topic cannot learn it and are not added to the `knownBy` list.
	routine	A routine returning `true` or `false` if it can be learned in the exchange between the two passed in character parameters.
`relatedTopics`	List of topics	When the topic is spoken, all topics listed in this property are added to the conversation pools of both participants if appropriate.

← See previous page for what "unresolvable" means.

"**Appropriate**" for a character that **knows** it means all of the following are true:

The **other character** does NOT also know it.

The topic has not already been spoken **BY** this character, **TO** the other character.

The topic is not designated with **ORDIA_RESPONSE_ONLY** in the second (TELL) entry of the **quip** property.

"**Appropriate**" for a character that does NOT know it, means:

The topic has not already been spoken to the character (regardless of who told him or her).

orDialogue

The orDialogue extension implements the classic ASK/TELLparadigm. Specifically, it enables the following syntaxes for the player:

```
> ask NPC about topic
> tell NPC about topic
> say topic to NPC
> NPC, topic
```

In addition to providing this conventional conversation pattern it also provides support for advanced features, such as learnable topics and the ability to track who said what to whom. More advanced conversation extensions in the orLibrary build upon orDialogue.

Normal Behavior

There are several approaches to modeling conversations in Inform but the DM4 §18 describes using the **life** property to handle **ASK/TELL**. Here's an example:

```
object -> magicMirror "magic mirror" has talkable
    with name 'magic' 'mirror'
       description "It reflects back the truth. Verbally.",
       life[;
           Ask:if(second=='beauty' or 'handsome') {
                  print "~Mirror, Mirror, in my hand, who's most
                      handsome in this land?~";
              }
        ];
```

Which enables...

```
>ask mirror about beauty
"Mirror, Mirror, in my hand, who's most handsome in this land?"
```

This approach is straightforward, but limited. For example, it requires specialized parser handling for topics which are more than one word and doesn't lend itself well to menus, to NPCs which can initiate conversations, or to information which can be "learned".

I cover NPCs which can initiate conversations on their own, with the orNpcSkillDialogue extension on page 89.

Revised Behavior

To power conversations, orDialogue wraps pieces of dialogue into **orTopic** objects which NPCs may know and talk about when asked. Consider the following...

```
object -> magicMirror "magic mirror" has talkable
    with name 'magic' 'mirror',
       description "It reflects back the truth. Verbally.";

orTopic tMom with knownBy selfObj, name 'mom',
    context magicMirror,
    quip "~You know,~ you say to... That's YOUR mom.~";

orTopic tBeauty with knownBy magicMirror,
    name 'handsome' 'beauty',
    quip "~Mirror, Mirror... in this land?~"
         "~Your mom... quite remarkable.~";
```

To the best of my knowledge, Emily Short, was the first to associate the term "quip" with the text of a conversational exchange in Interactive Fiction.

Notice the **quip** property can contain two different strings. One for ASK, the other for TELL. I discuss this more on page 82.

...which plays out as...

```
>ask mirror about beauty
"Mirror, Mirror, in my hand, who's most handsome in this land?"

"Your mom," the mirror replies. "Seriously, her mustache is quite
remarkable."
```

```
>tell mirror about mom
"You know," you say to the mirror. "You are really just a piece of
a larger, more famous mirror. That's YOUR mom."
```

Establishing Who Knows a Topic

As a rule, a character must *know* a topic to **TELL** someone about it, either as a response to **ASK**, or as an independent **TELL** action.

Although the parser relies on the **name** property to determine which topic is being referenced, **orTopic** objects themselves aren't in the game world at all. Instead, the scope of an **orTopic** object is determined by which characters "know" it (i.e., which characters are listed in the **knownBy** property).

```
>tell mirror about beauty
I'm not sure what you want to say.
```

Because only the mirror knows the **tBeauty** topic...

```
>ask mirror about mom
The magic mirror does not appear to know the answer to that.
```

There are a few nuances about the **knownBy** property worth clarifying:

- If **knownBy** is *not* defined, a topic uses its parent's **knownBy** property, if it has one. This enables topics to be grouped together in code:

...and only the player knows the **tMom** topic.

Notice that this is not an orTopic and it has no name. It can never actually be referenced.

```
object topicsKnownByMirror with knownBy magicMirror;

    orTopic -> tBeauty with name 'beauty',
        quip "~Mirror, Mirror...~";

    orTopic -> tPoisonBananas  with name 'monkey' 'problem',
        quip "~How can I get rid of this little monkey problem
        I have? ...~";
```

  In the above, the **tBeauty** and **tPoisonBananas** topics are both known by the **magicMirror** because their parent is.

- Instead of a list of NPCs, a solitary routine may be defined in the **knownBy** property. This routine accepts an NPC as a parameter and should return **true** if that character knows the topic:

```
    orTopic -> tPoisonBananas  with name 'monkey' 'problem',
        knownBy[npc; return (npc has zoology or herbology); ],
        quip "~How can I get rid of this little monkey problem I
            have? ...~";
```

  Defining a routine which always returns **true** effectively makes a topic "common" knowledge. That is, every character in the game knows it. As a shortcut to this, you may simply define the **knownBy** property with a single **true** value:

```
    orTopic -> tMoon with name 'moon' 'luna',
        knownBy true,
        quip "~Luna's magic is subtle, but powerful.~";
```

Context

While the **knownBy** property lists the characters who can *say* a topic, the **context** property determines *who* they can say it to. Remember our **tMom** topic from the previous page, which our player (**selfObj**) knows. This topic was defined with **magicMirror** listed as the only member in its **context** which keeps it from being spoken to anyone else. A nearby servant, for example:

```
orNpc -> servant "servant" with name 'servant',
       description "Waiting quietly to attend to your needs";

>tell servant about mom
I'm not sure what you want to say.
```

Syntactically, **context** is very much like **knownBy**: it will inherit its parent's **context** values, if any, and a routine may be specified instead. Unlike **knownBy**, if no context can be resolved, it is assumed to be a valid topic for anyone.

Differentiating "ASK" From "TELL"

The previous example shows the **quip** property containing the text to print during successful ASK and TELL commands; however, observant readers will notice that the **tBeauty** topic contains two separate strings.

The first of these strings is the ASK quip. It is printed when a character successfully ASKs about the topic. The second is the TELL quip, which is printed when the topic is actually communicated. This distinction is not always important. For most games, where only the PC can initiate actions and NPCs are unable to speak unless spoken to, only a single string is useful and serves both ASK and TELL purposes; however, as NPCs begin to act of their own accord, distinguishing a topic being ASKed about from one being volunteered is important.

I'll discuss this more when I cover the **orNpcSkillDialogue** extension on page 89.

Learnable Topics

In real-world conversations, information is learned. If you share a secret with another, that person then knows it and could choose to tell others. The orDialogue extension implements this with the **learnable** property. Topics which are learnable are remembered by the characters they are told to. Mechanically, they are added to the **orTopic**'s **knownBy** list.

For reasons which may or may not be obvious, the ability to learn topics breaks when the topic's **knownBy** list is defined as a routine. As a work around, consider overriding the **isKnownBy** property routine instead.

Basic Dissemination Tracking

The orDialogue extension tracks the exchange of information between characters. A simple use of this, where all conversations include the player, can be powered by the **hasBeenTold** property routine on **orTopic**. Remember our magic mirror from previous examples:

```
object -> magicMirror "magic mirror" has talkable
    with name 'magic' 'mirror',
       description[; if(tBeauty.hasBeenTold())  "Etched in the
        glass is a taunting shape, seemingly a mustache.";
            "It reflects back the truth. Verbally.";
        ];
```

With this change, the mirror now alters its description after the question of beauty is answered:

```
>x mirror
It reflects back the truth. Verbally.

>ask mirror about beauty
"Mirror, Mirror, in my hand, who's most handsome in this land?"

"Your mom," the mirror replies. "Seriously, her mustache is quite
remarkable."

>x mirror
Etched in the glass is a taunting shape, seemingly a mustache.
```

Advanced Dissemination Tracking

When NPCs gain the ability to converse among themselves, more advanced tracking techniques are needed to record which character has spoken a topic and to whom. The orDialogue extension supports this with with some conditions:

- All characters participating in the conversation must inherit from **orNpc**.
- Space for tracking dissemination must be allocated on the **orTopic** itself, in the **dissemTrack** property. (discussed on page 84)
- The maximum number of **orNPC** instances allowed in a game is 255.

Until now, all features of the orDialogue extension work as well with **animate** (or **talkable**) objects as they do with **orNpcs**.

Advanced tracking is enabled with the **hasBeenToldTo** and **hasBeenToldBy** property routines off of **orTopic**s. These take an NPC (or the player) as a parameter and return **true** or **false**.

Consider the following example, where both the player and the servant can talk to the mirror. In addition to making the mirror a true NPC, we also setup the servant to relay questions:

Being able to track that an NPC has been told something is different than the NPC having **learned** it.

Hearing and learning are two separate things; as any school teacher will attest.

Just an object no more!

```
orNpc -> magicMirror "magic mirror"
    with name 'magic' 'mirror';

orNpc -> servant "servant" name 'servant',
    description "Waiting quietly to attend to your needs",
    orders[;
           Ask: <Ask noun second, self>;
    ];
```

Here is a topic, which both player and servant can ask of the mirror:

Because there is no **context** specified, any character in the game can ASK about this topic.

```
orTopic tBestNumber with knownBy magicMirror,
    name 'best' 'number',
    dissemTrack 0 0,
    quip "~What's the best number?~"
        [teller talkingTo;
            if(self.hasBeenToldTo(talkingTo)){
                print "~I told you already...~ the mirror says.";
            }
            "~Two is the best. It's the only prime number that is
                even.~";
        ];
```

Notice that the teller of the topic, (in this case the mirror) now remembers its past conversations. If asked more than once by the same character, it will grumble about repeating itself:

```
>ask mirror about number
"What's the best number?"

"Two is the best. It's the only prime number that is even."

>ask mirror about number
"What's the best number?"

"I told you already," the mirror says. "Two is the best. It's the
only prime number that is even."

>servant, ask mirror about number
"What's the best number?"

"Two is the best. It's the only prime number that is even."

>servant, ask mirror about number
"What's the best number?"

"I told you already," the mirror says. "Two is the best. It's the
only prime number that is even."
```

As coded, the mirror only grumbles about repeating itself, when actually repeating itself to the same character.

Similar to the **hasBeenToldTo** property routine used in the above example, the **hasBeenToldBy** routine also takes the PC or an NPC as the first parameter and returns **true** if that character has ever spoken of the topic. Additionally, it *can* accept an optional second character as the second parameter. If provided, **hasBeenToldBy** returns **true** *only* if the first character has told the topic to the second. A character hearing of the topic from a different character does not meet this criterion.

Reserving Space for Tracking

In the previous example, the **dissemTrack** property reserves two entries, each providing room for a single telling (one for the mirror TELLing the topic to the player, the other for the mirror TELLing the servant). Calculating the space required to hold all tellings is as straightforward as that: each unique, possible telling requires a single entry in the array.

Technically, each telling requires 2 bytes. The Z-machine defines **WORDSIZE** as 2 bytes, so one entry tracks one telling. Glulx, with a **WORDSIZE** of 4 bytes, requires half the number of entries to be allocated to track the same number of tellings. That is, for Glulx, each entry in the property array can track 2 TELLings of a topic.

the ASK and TELL Verbs

The orDialogue extension provides two enhancements to the traditional ASK/TELL implementations. The first includes the A and T abbreviations for ASK and TELL respectively.

The second enhancement is the extension's attempt to determine with whom the player is intending to speak if otherwise unspecified. Combined, these two make the following possible:

```
>a beauty
(Asking magic mirror)
"Mirror, Mirror, in my hand, who's most handsome in this land?"

"Your mom," the mirror replies. "Seriously, her mustache is quite
remarkable."

>t mom
(Telling magic mirror)
"You know," you say to the mirror. "You are really just a piece of
a larger, more famous mirror. That's your mom."
```

Other ways of TELLing

As mentioned previously, the orDialog extension enables two additional variants for saying things to characters in the game:

```
>say mom to mirror
"You know," you say to the mirror. "You are really just a piece of
a larger, more famous mirror. That's your mom."
```

Like ASK and TELL, SAY can be abbreviated with S. Additionally, the SAY will also attempt to reason out who is being addressed if unspecified making the following possible:

```
>s mom
(to magic mirror)
"You know," you say to the mirror. "You are really just a piece of
a larger, more famous mirror. That's your mom."
```

Finally, the orDialog extension enables a command-like syntax for speaking to characters:

```
>mirror, mom
"You know," you say to the mirror. "You are really just a piece of
a larger, more famous mirror. That's your mom."
```

orDialogueMenu

The orDialogueMenu extension builds upon the foundations laid by the orDialogue and orMenu extensions to display potential conversation topics in a menu for players to choose from.

Normal Behavior

There is no ready-made equivalent to orDialogueMenu in the standard library.

Revised Behavior

TODO

```
verb meta 'ms' 'menuspeak' * -> MenuSpeak;

[MenuSpeakSub;

        dialogueMenu.show(orMenuTopOnly);

];
```

14
Advanced NPCs

Autonomous NPCs

In this section we cover the individual components of the orLibrary's NPC framework. These offer a solid starting point for constructing your own NPCs.

orNpc

The orNpc extension provides a foundation upon which more sophisticated NPCs are built. It provides a plug-in architecture to create and apply NPC skills, "teaching" them new abilities.

Normal Behavior

Below is an example non-player character, built from the standard library:

```
object -> butler "butler" has animate
    with name 'man' 'butler',
        description "The butler ignores you, preoccupied with
            other things.",
        each_turn[;
            print random("The butler clears his throat.",
                "The butler gives a small sigh.",
                "The butler looks around as though searching for
                    something.");
        ];
```

The above example prints a random reminder each turn that the NPC is nearby:

```
>z
Time passes.
The butler looks around as though searching for something.

>z
Time passes.
The butler gives a small sigh.
```

Revised Behavior

The code below demonstrates the above example, derived from the **orNpc** class:

```
orNPC -> butler "butler"
    with name 'man' 'butler',
        description "The butler ignores you, preoccupied with
            other things.",
        doNothing[;
            if(canPlayerWitness()==false) rfalse;
            print (string)random("The butler clears his throat.",
                "The butler gives a small sigh.",
                "The butler looks around as though searching for
                    something.");
        ];
```

A little experimentation shows the **orNpc**-derived butler behaves identically to the vanilla-Inform example, but there are noteworthy differences:

- Behind the scenes, the **orNpc** class searches for "registered skills" to perform. Since there are none in this example, it falls back to the **doNothing** default handler.
- Unlike the standard library's **each_turn** property which, despite the name, only executes when the player character is nearby, **orNpc** skills execute every turn. This enables NPCs to act, even when unobserved.

The canPlayerWitness Routine

Since an NPC may be in another part of the map altogether, we usually need to verify the player is nearby before describing what the NPC does. This is done with the **canPlayerWitness** routine, shown in the previous **doNothing** handler. This routine returns **true** if the player and current actor (the acting NPC) are in close proximity. We used this previously with the following line:

```
if(canPlayerWitness()==false) rfalse;
```

Here we choose to simply exit the handler if the NPC is not nearby; however, more complete examples would simply perform NPCs actions quietly.

Initialization

At the start of the game, each NPC is checked for the **npcInit** property. If defined, the routine is run, affording it the opportunity to "self configure." This occurs *before* the standard library's **Initialise** routine.

Turning NPCs Off

An **orNpc** can be turned off altogether by settings its **isEnabled** property to **false**, in which case the NPC does nothing, not even the behaviors defined in the **doNothing** handler:

```
butler.isEnabled=false;
```

To turn *all* NPCs in the game off, we use the **areEnabled** property of the global **orNpcControl** object:

```
orNpcControl.areEnabled=false;
```

Turning off all NPCs in this fashion overrides the NPC-specific **isEnabled** setting. All NPCs will lie dormant, even those which are enabled.

Sequencing NPC Actions

The order in which NPCs act is determined by their **priority** property. ***NPCs with higher priority values act before those with lower values.*** If two characters have the same priority, they will act, one after the other, with no guarantee of which character acts first, even between turns.

NPC Skills

A core feature of the orNpc extension is its extensible framework for NPC skills. There are several skills included as part of the orLibrary but it's also easy to make your own. Doing so starts with the **orNpcSkill** class. Here's an example of a skill enabling characters to break out in dance when the player is nearby:

```
attribute dancer;

orNpcSkill orNpcDanceSkill
  with perform[npc;
       print (The)npc, " dances in a way that impresses
       everyone present.";
  ],
  defaultPriority 1,
  canPerform[npc; return canPlayerWitness(); ],
  doesApplyToCharacter[npc;
      return (npc has dancer);
  ];
```

For every instance of **orNpcSkill**, the **doesApplyToCharacter** routine is called once, at the start of the game, to determine if the skill applies to a given NPC. It's up to this routine to determine if it does, using whatever criteria is appropriate. For example, it *might* be the NPC in question inherits from a given class or, as is the case above, that it has a given attribute. The **doesApplyToCharacter** routine returns **true** if the NPC *should* know the skill; **false** if not.

The **canPerform** routine is responsible for determining if the skill is possible given the current state of the game. In the example above, the **orNpcDanceSkill** skill is only considered when in the presence of the player, otherwise the NPC will choose a different skill to perform. (or call doNothing.)

When choosing between two or more possible skills, NPCs consider the skills' priority with higher priority skills selected over those with lower priorities. The **defaultPriority** property of the skill is used unless it is overridden. Our above example defaults the skill priority to 1, which is below the "default" **defaultPriority** of **5**, so most characters will generally prefer other activities over dancing.

Individual NPCs may prioritize their actions differently, by overriding **getSkillPriority**. You can see this in the below example of an NPC which has our new "dance" skill, and likes to do it whenever possible:

```
orNPC -> boogieMan "Boogie Man" has dancer
    with description "Altogether different than his cousin,
        the Boogey Man.",
        name 'man' 'boogie',
        getSkillPriority[skl;
            if(skl==orNpcDanceSkill) return 99;
            return skl.defaultPriority;
        ];
```

When everything is resolved, when the **orNpc** determines that an applicable skill can be carried out and has the appropriate priority, the skill's **perform** routine will be called with the acting NPC passed in as a parameter.

orNpcSkillDialogue

The orNPCSkillDialogue extension is an NPC skill which grants NPCs the ability to initiate conversations with other characters in the game. It takes the ASK/TELL abilities, introduced for the PC by the orDialogue extension, and extends these to **orNPC**s. Including this extension automatically includes orDialogue.

To Enable This Skill...

Making an **orNpc** capable of starting a conversation means inheriting it from the **orNpcSkillDialogue** class:

```
orNpc medusa "woman" elevator has female
    class orNpcSkillDialogue
    with name 'woman' 'medusa' 'gorgon' 'redhead',
        description "Dressed in a toga with a dozen red, rubber
        snakes intertwined with her long red curls.  They bounce
        around her face as she moves.";
```

A Review of Possible Topics

The same conditions introduced in orDialogue, which govern what a character can say, remain in effect for NPCs:

If you're not familiar with the orDialogue extension yet, stop now and turn to

- A character must ***know*** a topic before they can TELL someone about it.
- The topic must be appropriate (in ***context***) for the character to whom they are speaking.

For example, the following topic, known by the PC, is one which the Medusa character can ASK about:

I discussed this in "Differentiating ASK from TELL" on page 82.

```
orTopic -> medusa_costume "Medusa costume"
with quip "~I've always been fascinated … experience working with
    costumes.",
    knownBy medusa,
    context selfObj,
    relatedTopics theater_history dracula_costume;
```

And with this skill, she will talk about it, once the topic becomes appropriate...

Topic Pools

Normal conversations tend to follow a thread of relationships. One person speaks; the other responds with something related. Introducing random topics, unrelated to the current subject, can be off-putting.

To mimic the pattern of relevancy, the orNpcSkillDialogue extension provides NPCs with "conversational awareness." Mechanically, this is a pool of potential **orTopic**s which have come up in the conversation. Characters select entries from this pool to converse about topics they are aware of. This not necessarily the same as *knowing* a topic, since characters may ask about things they are aware of, but do not know. The following code brings the **medusa_costume** topic to Medusa's mind:

```
medusa.takeNoticeOfTopic(medusa_costume);
```

Assuming she has no other topics to choose from, Medusa will volunteer this information via the **orTopic**'s **quip**:

```
"I've always been fascinated with Greek mythology." She glances at
her reflection in the mirror. "My roommate helped me make this.
She's a theatrical major and gets a lot of experience working with
costumes."
```

After topics are spoken, they are removed from the topic pools of both characters. Unspoken topics, listed in the **relatedTopics** property, are added in to open up additional conversation options. Used appropriately, this allows for long, winding discussions to be crafted.

NPCs Asking Questions

In the previous example, the topic Medusa spoke of is one which she knew, so when she chose to speak of it, she did so using the TELL command; however, she does ***not*** know the **dracula_costume** topic, which is now in her conversation pool. Here's how it looks, for reference:

```
orTopic dracula_costume "Dracula costume"
with quip "~You chose Dracula?~ she asks, arching an eyebrow."
    "~Dracula was always my favorite monster,~ you say,
    glancing at your reflection.",
    name    'dracula' 'costume',
    knownBy selfObj,
    context medusa,
    relatedTopics medusa_costume medusa_fear_of_vampires;
```

This topic is known by the PC, so instead she will ASK about it on her next turn using the ASK **quip:**

```
"You chose Dracula?" she asks, arching an eyebrow.
```

PC Answering Questions

There are multiple ways to answer NPC questions. The first leaves control, whether to answer or not, in the hands of the player. This choice is made easier with the ANSWER verb:

```
>answer medusa
"Dracula was always my favorite monster," you say, glancing at
your reflection.
```

ANSWER is just short hand to TELL the topic just asked about. Alternatively, the player could type...

```
>tell medusa about dracula costume
```

...with identical results.

The second way is to set the global variable **orDialogueAutoAnswer** to **true**. This causes the player to automatically answer the question.

```
"You chose Dracula?" she asks, arching an eyebrow.

"Dracula was always my favorite monster," you say, glancing at
your reflection.
```

Topics Which Must Be Asked or Must Be Told

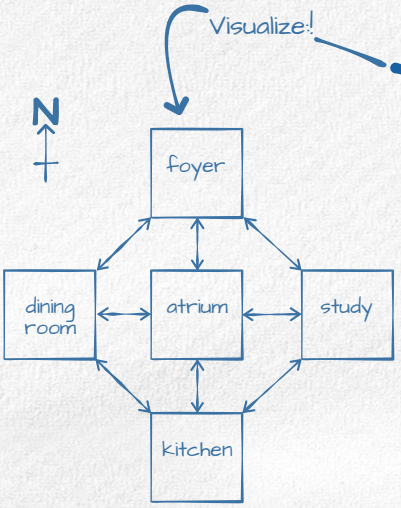
As covered when we discussed in the orDialogue extension, **orTopics** may contain two **quip**s: the ASK **quip**, and the TELL **quip**. This makes it possible to distinguish when a character is answering a question from when they volunteer it on their own.

If an **orTopic** does not have an ASK **quip** (i.e., there is only one **quip** defined), then it cannot be asked about.

Similarly, if the second quip is the value **ORDIA_RESPONSE_ONLY**, then it cannot be volunteered, only asked about. In such cases, the first **quip** should contain both the ASK and TELL text.

orNpcSkillMove

The orNpcSkillMove extension grants NPCs the ability to move around the game world. It supports several models of movement, from wandering aimlessly to targeted pursuit, but before we discuss these let's create a simple map to house our examples:




```
object foyer "Foyer" has light
    with description "This is a clean, simple entry hall.  Exits
        lead southwest, southeast, and due south.",
    sw_to diningroom, s_to atrium, se_to study
;
object diningroom "Dining Room" has light
    with description "This is really more of a breakfast nook.
        Exits lead northeast, southeast, and due east.",
    ne_to foyer, e_to atrium, se_to kitchen
;
object kitchen "Kitchen" has light
    with description "The place where food is prepared. Exits
        lead northwest, northeast, and due north.",
    nw_to diningroom, n_to atrium, ne_to study
;
object study "Study" has light
    with description "A library without books. Exits lead
        northwest, southwest, and due west.",
    nw_to foyer, w_to atrium, sw_to kitchen
;
object atrium "Atrium" has light
    with description "Light filters down from above. Exits lead
        north, south, east, and west.",
    n_to foyer, e_to study, w_to diningroom, s_to kitchen
;
[Initialise; location=atrium; ];
```

To Enable This Skill...

Bestowing movement abilities on an **orNpc** is as simple as inheriting it from the **orNpcSkillMove** class:

```
orNpc maid "maid" atrium has female
    class orNpcSkillMove
    with name 'maid';
```

Wandering is also the default behavior when all other methods for determining a path for the NPC fail:

Wandering...

With no other configuration, an NPC deciding to move will wander. That is, they will follow the map in a random direction from which they did not just come:

```
Atrium
Light filters down from above. Exits lead north, south, east, and
west.

You can see a maid here.

>z
Time passes.
The maid departs to the east.
```

After a few turns, the maid will eventually wander back:

```
The maid arrives from the south.
```

Wandering, as a fallback behavior, can be turned off by changing the **moveDoesWander** property:

```
maid.doesWander = false;
```

Following a Contiguous Path

You can define a specific path for the NPC to follow with the **movePath** property. Usually this is a list of rooms the character is to visit, in order:

```
orNpc butler "butler" atrium
    class orNpcSkillMove
    with name 'butler',
        movePath foyer diningRoom study;
```

The **movePattern** list isn't limited to locations. It can also contain game objects. This can include other NPCs, which are, themselves, moving.

Adding the above to our example introduces a new NPC which will go to each of the listed rooms, one per turn. The rooms need not be contiguous. In our example, you can see the dining room and study are not adjacent. The butler will infer a path between these and also cross the atrium since it connects the two listed items.

As a character arrives at each of the rooms specified in the **movePath** list, the NPC's **moveArrived** routine, if it has one, is called. Inferred rooms between two entries in the list do not trigger this.

Handling Path Completion

By default, once the butler reaches the last room in the list (the study), he will retrace his steps. That is, he will turn around and follow the same path, in reverse order, until he arrives in the starting room (the atrium), then start over.

You can change this behavior by setting the **movePattern** property. The following are the possible values:

MOVE_PATTERN_REVERSE (the default) once the path end is reached, it is followed in reverse.

MOVE_PATTERN_REPEAT once the path end is reached, it starts over again at the beginning.

MOVE_PATTERN_SINGLE once the path end is reached, the NPC stops and waits (at this point, the **movePattern** value changes to **MOVE_PATTERN_HALT**).

MOVE_PATTERN_HALT the NPC does not move at all, effectively turning off NPC movement.

MOVE_PATTERN_HALT is different than setting the **moveDoesWander** property to **false**, which stops NPCs from wandering aimlessly, but keeps all other intentional movement active.

Following a Specific Character

Having an NPC follow something else in the game is accomplished with the the **followTarget** property:

```
maid.followTarget = butler;

You can see a maid and a butler here.

>z
Time passes.

The butler departs to the north.
The maid departs to the north (following the butler).
```

Following a character is different than following a path in two ways:

- It requires characters to be in an adjacent room. If the NPC is cannot see their target, then movement is decided using other means, either by following **movePath** or by wandering around.
- It supersedes other ways of determining a direction. Once the NPC stumbles upon their target, they will follow it until the **followTarget** parameter is changed.

Alternatively, you can use the **movePattern** property to track other NPCs over distances (see my note at the top of this page).

orNpcFollowOrdersSkill

TODO

15
Advanced NPCs

Additional NPC Extensions

Here we cover a few additional extensions, separate from the orNPC framework covered in the previous section. These are also usable in works which build their own NPCs from scratch.

orDistinctMeSelf

The orDistinctMeSelf extension adds a distinction between the words ME and SELF when giving NPCs directions.

Normal Behavior

By default, the standard library equates all "me" words, including SELF, with the player. This is usually fine; however, when giving commands to NPCs, SELF takes on a different meaning than ME.

Consider the following **Bless** verb, designed for NPCs and players alike:

If you are trying this out yourself, don't forget to register the new grammar with:

verb 'bless' * noun -> bless;

```
[BlessSub;
    CSubjectVerb(actor,0,0,"administer",0,"administers");
    print " a blessing upon ";
    if(actor==noun) OnesSelf(noun);
    else ThatOrThose(noun);
    ".";
];
```

And an NPC which can be persuaded to hand out blessings:

```
object -> friar "friar" has animate male
 with name 'friar',
   orders[;
     Bless:
        <Bless noun, friar>;
        rtrue;
   ];
```

Since SELF and ME are treated the same, getting the friar to bless himself requires the player to perform some lexical gymnastics:

Expected behavior.

```
>friar, bless me
The friar administers a blessing upon you.
```

Not the behavior we were looking for.

```
>friar, bless self
The friar administers a blessing upon you.
```

```
>friar, bless yourself
There is no reply.
```

Weird phrasings to get this to work as expected.

```
>friar, bless friar
The friar administers a blessing upon himself.
```

Revised Behavior

No additional configuration need be made. Simply include this extension and orDistinctMeSelf causes the parser to correctly interpret `SELF` as the NPC, when the player is ordering him about, while continuing to interpret `ME` correctly as the player. Additionally, it adds the `YOURSELF` word, which is missing in the standard library, as a variant of `SELF`:

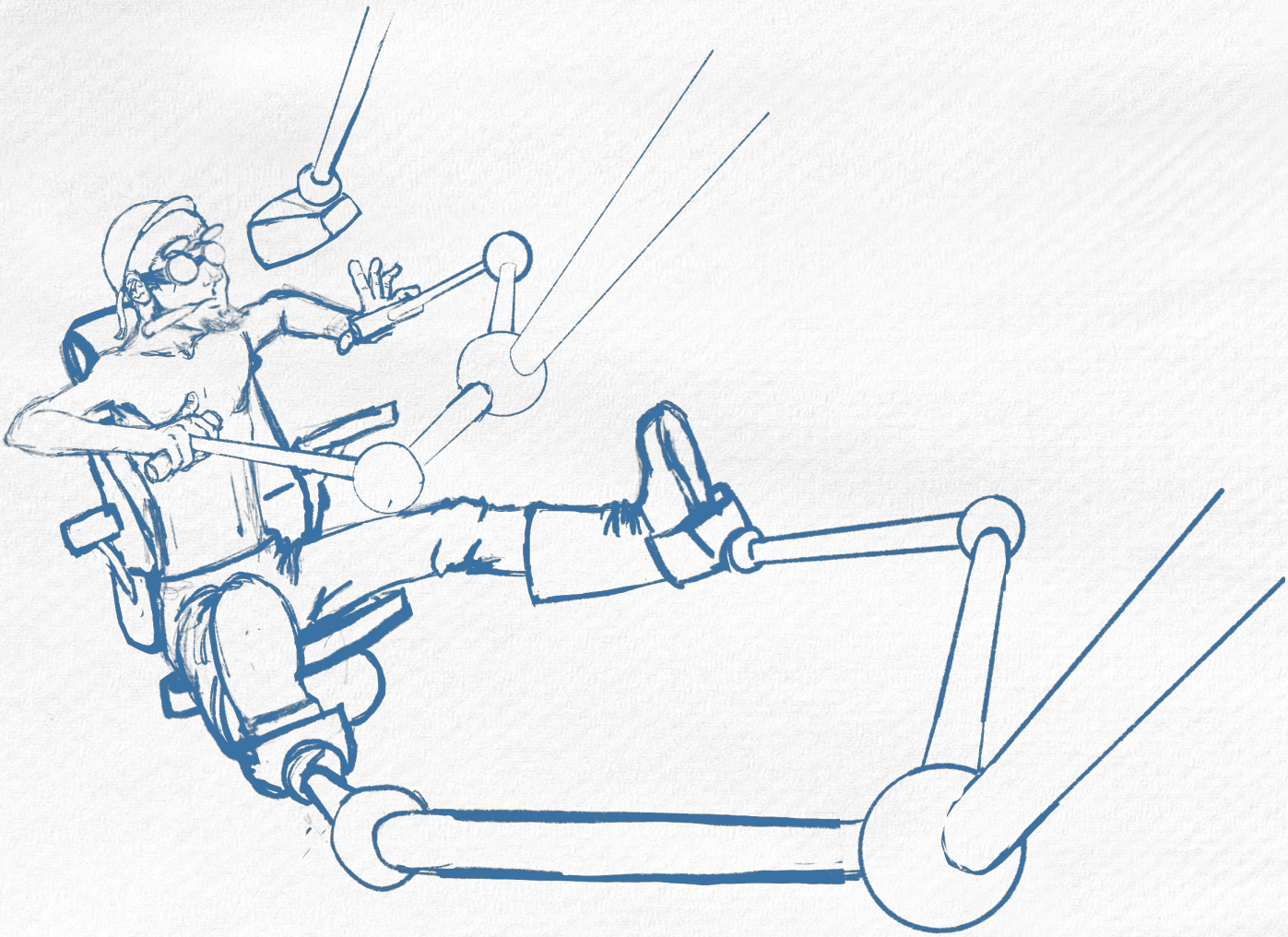
```
>friar, bless me
The friar administers a blessing upon you.

>friar, bless self
The friar administers a blessing upon himself.

>friar, bless yourself
The friar administers a blessing upon himself.
```

This is what we were expecting.

orFollowVerb

TODO

Follow



Part Six:
Select Utilities

Here we'll cover a handful of topics which are more technical in nature than the rest of this guide.

16 The util Object

Select Utilities

The orLibrary's **util** *object is a single place to reference general-purpose tooling. From one perspective it's like a keyring onto which some utility extensions install themselves. In other ways, it's like a Swiss Army Knife. Regardless of your preferred metaphor, it brings order to a variety of routines and is used extensively by the library itself.*

Although the **util** object is contained in the **_orUtil.h** file, you almost never need to include this. Instead, include one of the associated utility files, prefixed with "**orUtil**." Each of these will install the **util** object, if it isn't already in place, and attach their respective functionality to it. For example:

```
#include "orUtilBuf";
```

Makes the following code legal...

```
print util.orBuf.getLength(myBuf);
```

...but *not*...

```
print util.orArray.find(myArray, value);
```

If you choose, you can include the pkgOrUtils package to include all **util** extensions at once.

To compile the above line you would first need to include the **orUtilArray** extension:

```
#include "orUtilArray";
```

orUtilArray

The orUtilArray extension exposes routines from the **util.orArray** object which support manipulation of both standalone table arrays and property arrays, using the same syntax regardless of the array type.

Inform defines arrays which reserve the first element for length as "tables."

For purposes of contrast, note how Inform's native syntax for accessing the **n***th* element differs depending on the type of array:

```
print obj.&prop-->n;
print arr-->(n+1);
```

Notice how property arrays use zero-based indexing, but table arrays use one-based indexing.

The syntaxes for retrieving array size is equally disparate:

```
print obj.#prop;
print arr-->0;
```

With orUtilArray, the syntax for accessing both array types is nearly identical:

```
print util.orArray.get(arr,n);
print util.orArray.get(obj,prop,n);
```

For table arrays, pass in the array itself.

```
print util.orArray.getLength(arr,n);
print util.orArray.getLength(obj,prop,n);
```

For property arrays, pass in both the object and the property.

Note: all util.orArray routines use zero-based indexing, regardless of the array type they operate against.

The following list describes the routines exposed by the **orArray** object. As a rule, each of these routines accept *either* the object and a property name to identify a property array, *or* the array name to identify a standalone table array, supplying one less variable.

append(obj, prop, val) assuming there is unused space on the array, this routine adds **val** to the end of the array (i.e., the first empty value), thereby increasing the length of the array.

clear(obj, prop) removes everything from the list, setting all entries, and therefore the calculated length, to zero.

find(obj, prop, val) searches the array for the first occurrence of **val**, returning the index of the element, or -1 if not found.

get(obj, prop, n) returns the value of the **n***th* element in the array, or -1 if **n** is greater than the size of the array. Note that the length of the array is not considered, so it is possible to read past the current array length.

getLength(obj, prop) returns the number of entries up to, but not including, the first element with a zero value. If all elements in the array have values, the length will equal the size.

getSize(obj, prop) returns the number of entries reserved for the array itself, regardless of if they are used.

insert(obj, prop, n, val) inserts a value at the **n***th* index, shifting values to make room if necessary.

prepend(obj, prop, val) adds **val** to the front of the array, likely increasing the length; dropping the last entry if it is alreay full.

remove(obj, prop, n) eliminates an entry at the specified index, **n**, and shifts subsequent values to remove the gap.

removeValue(obj, prop, val) searches the array for all occurrences of **val** and removes them.

set(obj, prop, n, val) sets the value at the at the **n***th* element in the array.

Size vs. Length
Across the library I distinguish *size*, which is the number of array elements reserved at compile time, from *length*, which is the number of contiguous entries, starting at position zero, containing a value. This definition is used in several extensions, including orString, orUtilBuf, and orUtilArray.

The orUtilArray extension determines length using the (often false) assumption that a zero value means "no value" and therefore can be used to calculate free space.

If this is not how you are using your array, if zero is a legitimate value, the **append()** and **getLength()** routines will not be reliable for your purposes and you might need to code this logic yourself.

orUtilBuf

The orUtilBuf extension provides the **util.orBuf** utility object for managing arrays of characters, similar the capabilities supplied by low-level languages such as C.

The routines follow a couple of conventions:

- All **orBuf** routines use zero-based indexing, regardless of the presence of the length **word**, which prefixes Inform buffer arrays. This means position zero always represents the first character.
- When two buffer parameters are required, the first parameter is the one which is modified, the second is the one which is read from.

The following list details the routines provided by the **orBuf** utility object:

append(toBuf, fromBuf) appends the content of the **fromBuf** buffer to the end of the **toBuf** buffer. The length of **toBuf** is expanded appropriately.

capture(buf) redirects the interpreter's output stream to the supplied buffer.

convertToSizedBuffer(buf) takes a normal buffer, converts it to a Sized Buffer, and returns the adjusted pointer. The **buf** paramter is returned unchanged if it is already a Sized Buffer.

See page 110 for a discussion on Sized Buffers.

copy(toBuf, fromBuf, numCharsToCopy, fromPos, toPos) copies a number of characters (**numCharsToCopy**) the source buffer (**fromBuf**) over the characters in the destination buffer (**toBuf**). By defining the **fromPos** and **toPos** parameters, which default to **0**, you can change where the target text starts in each buffer. For example, you can copy characters starting at pos **12** in **fromBuf**, into **toBuf** at position **3**. The length of **toBuf** is adjusted appropriately.

delete(buf, pos, count) removes **count** characters from the **buf** buffer, starting at position **pos**. The right most characters are shifted left. The length of **buf** is adjusted appropriately.

equals(buf1, buf2, caseInsensitive) compares two buffers, returning **true** if they are identical, **false** if not. If **caseInsensitive** is **true**, the upper and lowercase characters are considered equal.

getChar(buf, pos) returns the character in position **pos** from the buffer **buf**.

getLength(buf) returns the length of the passed in buffer.

See the "**Size vs. Length**" side bar on page 101 for the difference between size and length.

getSize(buf) when **buf** is a Sized Buffer this returns the allocated size of the buffer, regardless of the length of the text it contains. If **buf** is *not* a Sized Buffer, then **-1** is returned. (Because Inform does not maintain size for buffers.)

indexOf(buf, searchTextBuf, startingIndex) returns the first occurance of the text supplied in **searchTextBuf** in the **buf** buffer, at or following the **startingIndex** position.

indexOfFirstFalse(buf, routine, startingIndex) passes each character in the **buf** buffer, starting at position **startingIndex**, to the supplied routine, until it returns **false**. Returns the position number which triggered the **false** result, or **-1** if **false** is never returned.

indexOfFirstTrue(buf, routine, startingIndex) passes each character in the **buf** buffer, starting at position **startingIndex**, to the supplied routine, until it returns **true**. Returns the position number which triggered the **true** result, or **-1** if **true** is never returned.

insert(toBuf, fromBuf, toPos, charCount) inserts **charCount** characters from the **fromBuf** buffer into the **toBuf** buffer at the position specified by **toPos**. The rightmost characters are copied out to make room. The length of **toBuf** is expanded appropriately.

Again, see "Sized Buffers", page 110.

isSized(buf) returns **true** if the passed in buffer is a Sized Buffer, otherwise it returns **false**.

left(toBuf, fromBuf, numCharsToCopy) copies the leftmost number of characters (**numCharsToCopy**) from the **fromBuf** buffer into the **toBuf** buffer. The length of **toBuf** is set appropriately.

mid(toBuf, fromBuf, fromPos, numCharsToCopy) copies **numCharsToCopy** characters from the **fromBuf** buffer, starting at position **fromPos**, into the **toBuf** buffer. The length of **toBuf** is set appropriately.

prepend(toBuf, fromBuf) inserts the content of the **fromBuf** buffer to the front of the **toBuf** buffer, pushing existing characters right and expanding the length appropriately.

print(buf) prints the passed in buffer.

release() restores the interpreters output stream to the screen, returning the buffer which was passed in from **capture()**.

replace(buf, searchTextBuf, replaceTextBuf) modifies the **buf** buffer by replacing the first occurrence of the text supplied in the **searchTextBuf** parameter with the text suppled in the **replaceTextBuf** parameter. The length of **buf** is adjusted appropriately.

replaceAll(buf, searchTextBuf, replaceTextBuf) replaces all occurrences in the **buf** buffer of the text supplied in the **searchTextBuf** parameter with the text suppled in the **replaceTextBuf** parameter. The length of **toBuf** is adjusted appropriately.

reverse(buf) reverses the string contained in **buf**.

right(toBuf, fromBuf, numCharsToCopy) copies the rightmost number of characters (**numCharsToCopy**) from the **fromBuf** buffer into the **toBuf** buffer. The length of **toBuf** is set appropriately.

set(buf, stringVal) copies the string in **stringVal** into **buf**. The **stringVal** parameter may be a literal string or another buffer. This routine interrogates the **stringVal** parameter to determine its type and calls **setFromLiteral** or **setFromBuffer** as appropriate.

setFromLiteral(buf, stringVal) copies the string in **stringVal** into **buf**. The **stringVal** parameter is a literal string.

setFromBuffer(buf, stringVal) copies the string in **stringVal** into **buf**. The **stringVal** parameter is a reference to a buffer.

setChar(buf, pos, val) sets the character at position **pos** in the buffer **buf** to **val**.

setLength(buf, newLength) sets the *length* of the buffer content (not the allocated size).

toLower(buf) converts the string contained in **buf** to lowercase.

toUpper(buf) converts the string contained in **buf** to uppercase.

trim(buf) removes any leading and trailing spaces from **buf**, adjusting the length appropriately.

trimLeft(buf) removes any leading spaces from **buf**, adjusting the length appropriately.

trimRight(buf) removes any trailing spaces from **buf**, adjusting the length appropriately.

orUtilChar

The orUtilChar extension provides routines for working with text characters. These are exposed off the **util.orChar** object. The following list describes these:

isAlpha(c) returns **true** if the passed in character is a letter of the alphabet, either upper or lower case; otherwise **false**.

isNumeric(c) returns **true** if the passed in character is numeric digit; otherwise **false**.

isAlphaNumeric(c) returns **true** if the parameter is either a letter of the

alphabet or a numeric digit; otherwise **false**.

isLower(c) returns **true** if the parameter is a lowercase letter of the alphabet; otherwise **false**.

isUpper(c) returns **true** if the parameter is an uppercase letter of the alphabet; otherwise **false**.

toLower(c) returns a lowercase version of the character passed in, if it was a letter of the alphabet; otherwise it returns the value which was passed in.

toUpper(c) returns an uppercase version of the character passed in, if it was a letter of the alphabet; otherwise it returns the character which was passed in.

orUtilLoopArray

The orUtilLoopArray extension provides a mechanism to loop over elements in a specifically-defined array (called a "loop array"), without repeating. Below is a list of the routines exposed off the **util.orLoopArray** object:

getAny(obj, prop, onceOnly) returns a random element in the array, marking it as returned so subsequent calls to **getAny()** do not return it again, until all elements have been returned. The **onceOnly** parameter determines the behavior of subsequent calls after all elements have been returned.

getNext(obj, prop, onceOnly) returns the next element in the array. The **onceOnly** parameter, determines the behavior of subsequent calls after all elements have been returned.

isComplete(obj, prop) returns **true** if all elements in the array have been returned, but the next loop through the array has not started; otherwise **false**.

The first two parameters in all routines represent the object and property the loop array is contained in. Like the **orArray** utility object, these can be substitued with a single table array, reducing the number of parameters by one.

By default, both **get** routines reset the loop when they finish all entries. To change this behavior, specify the **onceOnly** parameter with one of three values:

false (default) - returns all values in the list, then resets its tracking mechanism to start over for subsequent calls.

true - returns all values in the list once, then **-1** on subsequent calls.

repeatLast - returns all values in the list once, then returns the last value in the list for all subsequent calls.

See orUtilArray onpage 93 for an example of this.

Example

A "loop array," as defined by this extension, is simply a normal array where the first element is reserved for tracking purposes. Usually, it is initially defined as zero. Here's an example of a property array, which is also a loop array, of fruit objects:

(tracking value)

```
object bowl with  listOfFruit 0 apple orange banana peach;
```

For property arrays, the two looping methods, **getNext()** and **getAny()**, both take the object and property as the first two parameters. Iterative calls to these return each element in the property list:

```
for(i=0:i<9:i++){  !-- print up to 9 elements in the array
    el = util.orLoopProperty.getNext(bowl, listOfFruit);
    print (name)el," ";
}
```

The above use of **getNext()** returns the elements in the array sequentially:

```
apple orange banana peach apple orange banana peach apple
```

Notice that the elements repeat after they've all been returned.

The **getAny()** routine behaves identically to **getNext()** except the returned values are selected at random. Substituting **getAny** in place of **getNext** in the above produces a different output:

```
peach orange apple banana banana orange peach apple peach
```

You will probably get these in a different order, since getAny makes random choices.

Despite the random sequence, we can see above that the elements still do not repeat until they have all been returned. More than simply tracking a position in an array, the orLoopProperty extension tracks which items have been previously returned and excludes them from future requests.

Limitations

For the Z-machine, the maximum number of array entries supported by the orLoopProperty object is 16 (17 if you count the tracking entry). For Glulx this maximum is 32 (plus the initial tracking element).

orUtilMap

The **util.orMap** utility object contains routines useful in navigating the world map. The following describes these:

reverseDirection(dir) - returns the opposite direction of **dir**, regardless of whether it is a compass direction or a direction property. All 12 directions (including cardinal, inter-cardinal, up, down, in, and out) are supported.

pathFinder - a placeholder property used to host the **orMapPathFinder** searching algorithm if it is included.

The following demonstrates:

Direction property.

```
if(util.orMap.reverseDirection(nw_to)==se_to) print "true ";
if(util.orMap.reverseDirection(u_obj)==d_obj) print "true";

true true
```

Direction object.

orUtilMapPathFinder

The orUtilMapPathFinder extension provides a search algorithm used to determine a path between two objects or locations. Unlike most other **util** extensions, orUtilMapPathFinder doesn't extend **util**, instead it extends the **orMap** object, which resides on **util**, by adding the **pathFinder** object. Calling a routine looks something like this:

```
util.orMap.pathFinder.determinePath(kitchen, study);
```

The following are the primary routines exposed by this extension:

determinePath(start, end) calculates the shortest path between **start** and **end**. The **start** and **end** parameters can be locations or game objects, in which case the location of the objects is inferred.

The **determinePath** routine will also resolve paths through doors.

There is a limited amount of scratch workspace used to determine a correct path. If there is not enough space, the path will not be findable. The constant **PATHDEPTH** is used to allocate this scratch space. By default it is set to 30, however it can be defined at a greater value in the main program's source.

When complete, the **determinePath** routines populates the **path** property array with a list of rooms.

One of the following is returned:

-1 : There is no connecting path between the two given locations.

-2 : Ran out of workspace while trying to calculate path. Resolve this by increasing the value of **PATHDEPTH**.

-3 : Although there ***was*** a path correctly determined, there was ***not*** enough space in the path property to contain it.

room obj : The first connected room which must be traveled to in order to finally reach the destination.

convertPathToDirections() converts the resulting room list (the **path** property) into directions which can be followed to get to the destination.

orUtilNum

The orUtilNum extension provides routines focused on using numbers. The following lists these routines which are exposed off of **util.orNum**:

getBit(val, bitNum) returns the value of the bit specified by **bitNum** in **val**; either **true** or **false**. The first bit is number **0**.

isBetween(v, val1, val2) returns **true** or **false** if **v** is in the range specified by **val1** and **val2**. The **val1** and **val2** values are considered signed. A value equal to **val1** or **val2** is considered in the range.

max(val1, val2, val3, val4, val5) returns the highest of the supplied parameters. This routine considers these signed values and will interpret numbers with the highest bit set as negative numbers. Calls with fewer than five parameters should provide **orMaxEnd** as the last.

min(val1, val2, val3 val4 val5) returns the lowest of the supplied parameters. This routine considers parameters as signed values and will interpret numbers with the highest bit set as negative numbers. Calls with fewer than five parameters should provide **orMinEnd** as the last parameter.

pow(num, exp) returns the exponent result of **num**, raised to the power of **exp**. For example:

```
print util.orNum.pow(3,4);
81
```

setBitOn(val, bitNum) accepts the value **val** and flips the bit referenced by **bitNum** on, returning the result. If the specified bit is already on, the return value matches the original passed in value for **val**, unchanged.

setBitOff(val, bitNum) accepts the value **val** and flips the bit referenced by **bitNum** off, returning the result.If the specified bit is already on, the return value matches the original passed in value for **val**, unchanged.

Specifying Bits
Routines which manage bits (getBit, setBitOn, and setBitOff) usually require the bit number to be specified.

Since the Z-Machine works with 16 bit words, bits are numbered from 0 to 15; for Glulx, they range from 0 to 31. 0 always represents the least significant bit.

uIsBetween(v1, v2) an variant of **isBetween**, this routine returns **true** or **false** if **v** is in the range specified by **val1** and **val2**. The **val1** and **val2** values are considered unsigned. A value equal to **val1** or **val2** is considered in the range.

uIsGreaterThan(v1, v2) performs an unsigned comparison of the two parameters, returning **true** if **v1** is greater than **v2**, otherwise **false**.

uIsGreaterThanOrEqualTo(v1, v2) performs an unsigned comparison of the two parameters, returning **true** if **v1** is greater than or equal to **v2**, otherwise **false**.

uIsLessThan(v1, v2) performs an unsigned comparison of the two parameters, returning **true** if **v1** is less than **v2**, otherwise **false**.

uIsLessThanOrEqualTo(v1, v2) performs an unsigned comparison of the two parameters, returning **true** if **v1** is less than or equal to **v2**, otherwise **false**.

These four routines do not have signed counterparts, since the native comparison operators (<,>,<=,>=) are more concise and part of the language.

umax(val1, val2, val3, val4, val5) returns the highest of the supplied parameters. This routine considers these unsigned values. The **val3**, **val4**, and **val5** parameters are optional.

umin(val1, val2, val3, val4, val5) returns the lowest of the supplied unsigned parameters. Because missing parameters are passed in as zero, calls with fewer than five parameters should provide **orMinEnd** (see the **min** routine for examples).

These two routines behave consistently with their **min** and **max** counterparts, except they consider parameters to be unsigned values.

Additionally there are two property values, which act as constants:

maxWord is the highest signed value which Inform can handle.

uMaxWord is the highest unsigned value which Inform can handle.

These differ between the Z-Machine and Glulx platforms.

orUtilParser

The **util.orParser** utility object is a place holder for routines which aid in parsing. At the moment, there is just one; as more parser routines are created, they will be added

At the moment, there is only one routine attached to the util.orParser object, and its not terribly sophisticated, but as more routines are created, they will go here.

excludeFromDisambiguation(obj) is a helper for parsing hooks like **ext_chooseObjects**. It prevents an object from consideration when constructing a list of items to disambiguate (e.g., "Which do you mean..."). More than just making it less likely to be selected, it removes **obj** from consideration entirely.

orUtilRef

The **util.orRef** object provides routines for dealing with references.

isRoutine(val) returns **true** if **val** is a routine; otherwise **false**.

isString(val) returns **true** if **val** is a string; otherwise **false**.

isObject(val) returns **true** if **val** is a routine; otherwise **false**.

isDictionaryWord(val) returns **true** if the passed in value is a dictionary word; otherwise **false**.

Note: isDictionaryWord() only works reliably for Glulx!

Distinguishing dictionary words isn't actually possible on the Z-Machine. I include it anyway because there is precedence: the standard library attempts to do so in a DEBUG section of **parser.h** and the approach used here is marginally *less* unreliable.

resolveValue(obj, prop, param1, param2, param3, param4) an enhanced version of the standard library's flawed, and now unused, **valueOrRun()** routine. This resolve a property regardless of whether it is a value or a routine.

orUtilStr

The **util.orStr** object is a collection of routines which simplify managing **orString**s, buffers, and literals.

I talk about orStrings on page 61.

ambiguousFree(buf, state) this routine will free the passed in buffer if **state** is **true**. This is semantically the same as:

```
if(state) free(buf);
```

I speak of "dynamic memory" allocation for orStrings on page 60. Buffers may also be managed in this way, with their own flavor of **lock()** and **free()**.

ambiguousLock(buf) passed an ambiguous buffer, this routine will detect if it is a "free" buffer. If so, it will lock it and return **true**; otherwise it will return **false**. Calling processes should remember this return value and pass it to the **ambiguousFree** routine. The intent for this pair of routines is to ensure all ambiguous buffers can be made safe, temporarily, and subsequently returned to their original state.

Usually, the lock state of a buffer return value is known (i.e., the routine's result is either always locked, or always free), but there are exceptions.

Return values for which the lock state is unknown are called **"ambiguous buffers"** which may have been locked previously, or not. Special care must be made to lock and unlock these without introducing memory errors.

areEqual(buf1, buf2, caseInsensitive) for comparing **orString**s/buffers/literals, regardless of their type. This routine returns **true** if they match; otherwise **false**. Comparison will be made regardless of case if **caseInsenesitive** is **true**.

format(pattern, param1, param2, param3) returns an ephemeral **orString** instance, formatted according to the rules defined on the **format** routine an **orString** instance. (see page 63 for details)

new(unkStr) allocates a new orString object and sets its value to that of **unkStr**, regardless of its type (string literal, character array, or another **orString** object).

print(buf) will print the value of **buf** regardless of if it is a string literal, an **orString** object, or a buffer character table array.

I almost named "ambiguous buffers" "Schrodinger's buffers" but it just doesn't quite roll off the tongue.

toAmbiguousBuffer(unkStr) returns the passed in value as a buffer array which can be used by buffer routines (such as those contained in **util.orBuf**). This routine does NOT lock any buffers itself, but depending on the type of string object passed in, the returned value may or may not be free. As such, we refer to the returned value as an "ambiguous buffer". Calling processes therefore should NOT try to lock or free the return value using normal techniques but use **ambiguousLock** and **ambiguousFree** instead.

trim(buf) trims whitespace from the front and end of the passed in string, and returns an ephemeral **orString** object populated with the result.

I discuss "ephemeral" return values on page 61.

orUtilUi

The `util.orUi` object contains a collection of routines for managing the user interface, including input and output.

`activateMain()` sets focus to the main window. Subsequent output is directed there.

`activateStatus()` sets focus to the status window. Subsequent output is directed there.

`eraseMain()` erases the main window.

`eraseScreen()` erases the whole screen, including the main and status windows.

`eraseStatus()` erases the status window.

`getChar()` pauses for keyboard input and returns the key pressed by the user.

`getScreenHeight()` returns the height of the screen measured in characters rows.

`getScreenWidth()` returns the width of the screen measured in fixed-width characters.

`getStatusHeight()` returns the height of the status bar in character rows.

`hideCursor()` hides the cursor by moving it off screen.

`pauseForInput()` hides the cursor, pauses for input, then restores focus to the main screen once input is detected.

`position(x,y)` moves the cursor to the position specified by the coordinates. For Z-Machine, this is only effective when the status bar has focus.

`setStatusHeight(h)` sets the height of the status bar.

17 A Few Additional Utilities

Select Utilities

Here are a handful of additional utilities, distinguished from those utilities not covered in this guide only by the likelihood they will be used by game authors.

orDeque

While the topic of data structures, including the "double-ended queue," is a fundamental in computer science, its a bit obscure for Inform discussions. Therefore, I don't dig very deep into potential uses of this extension, and instead just lay out what it does in bare-bones fashion.

The orDeque extension is a base class for adding queue-based and stack-based operations to inherited objects. Inheriting from this class requires space to be reserved for data elements in a property array and, for easiest use, placing the name of that property array in the **arrayPropName** property, like so:

```
orDeque heroStk
    with heroes 0 0 0 0 0 0 0 0 0 0
    ,   arrayPropName heroes;
```

With this, **heroStk** can be used as a stack (or a queue):

```
heroStk.push("Aquaman");
heroStk.push("Batman");
heroStk.push("Flash");
heroStk.push("Superman");
heroStk.push("Wonder Woman");
heroStk.push("Zatara");

while(heroStk.getLength()>0) print(string)heroStack.pop()," ";

Zatara Wonder Woman Superman Flash Batman Aquaman
```

I separate the property name from the base implementation to allow for more advanced structures which track multiple queues in the same object.

Specifying the **arrayPropName** value isn't strictly necessary. Instead, you can specify the name of your property array explicitly on each call, such as:

```
heroStk.push("Aquaman", heroes);
heroStk.pop(heroes);
```

The following is a list of all routines exposed by the **orDeque** class:

clear(prop) removes all elements in the stack/queue.

dequeue(prop) removes and returns an element from the front of the queue.

enqueue(newOption, prop) adds a new element to the end of the queue.

getLength(prop) returns the number of elements in the stack/queue.

Remember, the **prop** parameter is not needed if you defined arrayPropName appropriately.

peek(prop) returns the front/top-most element from the stack/queue, but does not remove it.

peekEnd(prop) returns the end/bottom-most element from the stack/queue, but does not remove it.

pop(prop) removes an element from the top of the stack and returns it.

popEnd(prop) removes an element from the bottom of the stack and returns it.

push(newOption, prop) adds a new element to the top of the stack.

orHookBanner

It isn't necessary to include orHookBanner manually since the orLibrary does so automatically.

The orHookBanner extension provides an additive way to supplement the game's banner.

Normal Behavior

The standard library's game banner prints the **story** and **headline** constants along with versioning information about the compiler and standard library:

```
My Game
Copyright (c) 2025 : Author Itarian
Release 200 / Serial number 250929 / Inform v6.42 Library v6.12.6
```

Modifying the banner beyond this is done through a **replace Banner** directive and by substituting your own version of the **Banner** routine, or by defining a **LanguageBanner** routine.

Revised Behavior

The orHookBanner extension will print **orBannerText**, if defined by the game developer as a constant or routine. Here it is, defined as a constant...

```
constant orBannerText "New players should type ~help~.^";
```

...the text is displayed with the game banner, alongside the orLibrary version:

```
My Game
Copyright (c) 2025 : Author Itarian
Release 200 / Serial number 250929 / Inform v6.42 Library v6.12.6
orLibrary Release 2.0 (2025.04.01)
New players should type "help".
```

If the orPrint extension is included, **orBannerText** may also include print rules:

```
constant orBannerText "New $b;players$n should type $i;~help~.
    $n;^";
```

```
New players should type "help".
```

Additionally, this extension makes the **ext_bannerText** hook available for extensions to define additional banner content without overriding the **Banner** routine:

```
object myBanner LibraryExtentions
   with ext_bannerText[;
      print "~How to play~ text sourced from IFTheory.org";
   ];
```

Producing:

```
My Game
Copyright (c) 2025 : Author Itarian
Release 200 / Serial number 250929 / Inform v6.42 Library v6.12.6
orLibrary Release 2.0 (2025.04.01)
"How to play" text sourced from IFTheory.org
New players should type "help".
```

Although this extension was once used throughout the original ORLibrary, most of its functionality was rolled into subsequent releases of the standard library, specifically in the **LibraryExtensions** object.

There are still cases where it makes sense to use this extension though.

orRoutineList

The orRoutineList class exposes methods for adding and removing routines from a list and calling them in different ways. The following lists the routines exposed by this class:

`add(prop, r)` adds a adds a routine, **`r`**, to the list of routines stored in an array housed in the **`prop`** property.

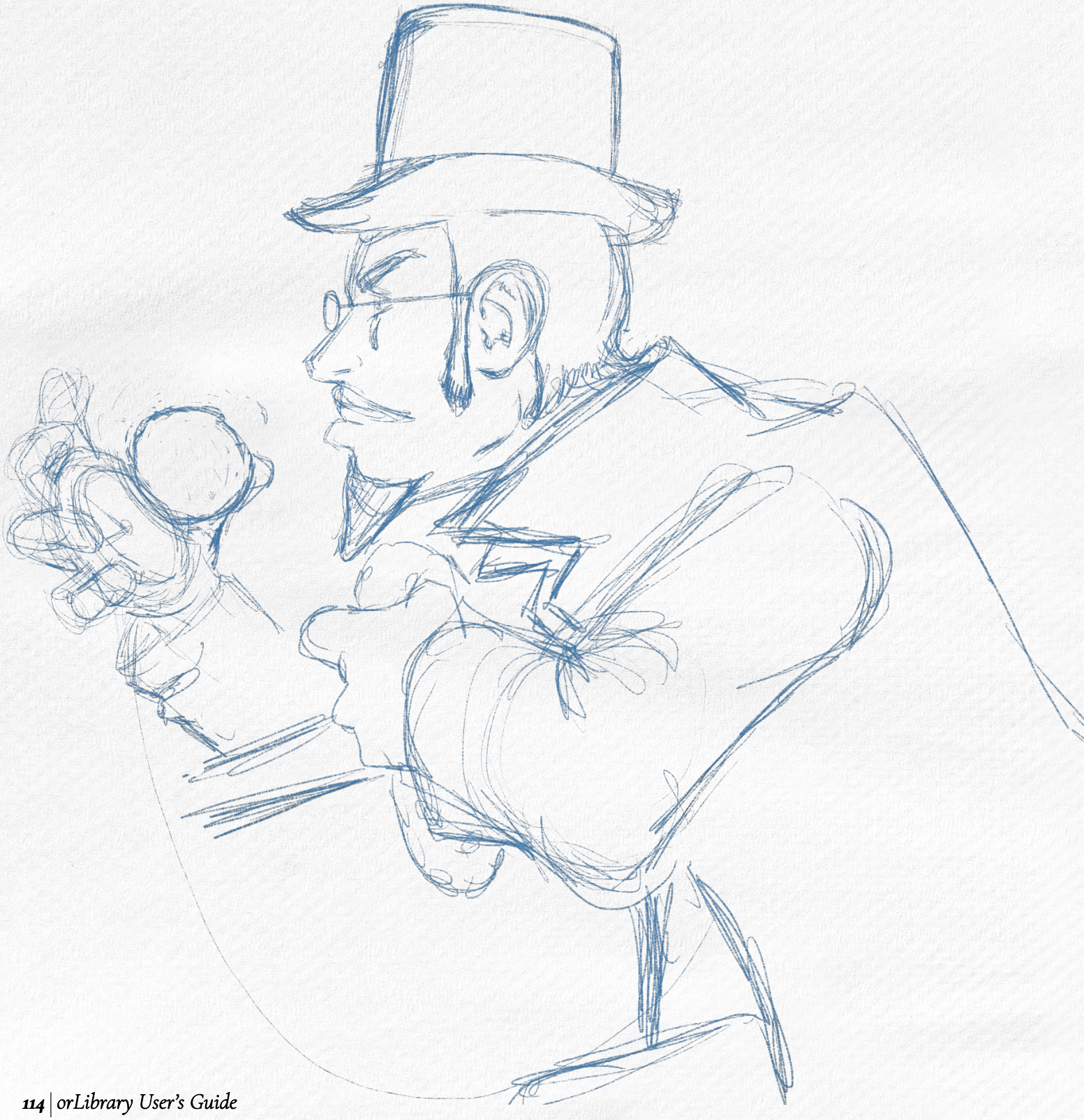
`remove(prop, r)` searches the list of routines stored in the **`prop`** property array for all occurrences of **`r`** and removes it.

`runAccumulate(prop, p1, p2, p2)` runs each of the routines in the routine list stored in the **`prop`** property array, passing in the parameters **`p1`**, **`p2`**, and **`p3`**. Adds the results together, and returns the total.

`runUntil(prop, untilVal, p1, p2, p3)` runs each of the routines in the routine list, passing in the parameters **`p1`**, **`p2`**, and **`p3`** to each, until one of them returns a result equal to the **`untilVal`** parameter.

`runWhile(prop, whileVal, p1, p2, p3)` runs each of the routines in the routine list, passing in the parameters **`p1`**, **`p2`**, and **`p3`** to each, until one of them returns a result not equal to the **`whileVal`** parameter

Additionally, an instance of **`orRoutineList`** can define the property routine **`betweenCalls`**, which is called between the calls to each of the routines. This is useful in cases where the routines change game state which must be reset each time.



Part Seven:
Technical Topics

18 Technical Topics

Sized Buffers

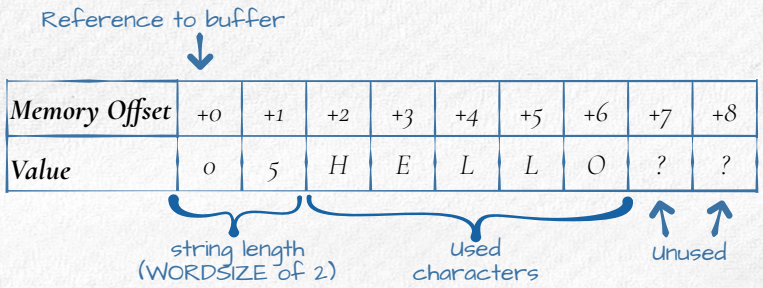
The orLibrary implements a concept called "Sized Buffers" which can be used interchangeably with the typical character buffers used by Inform.

A Quick Backgrounder on Normal Buffers

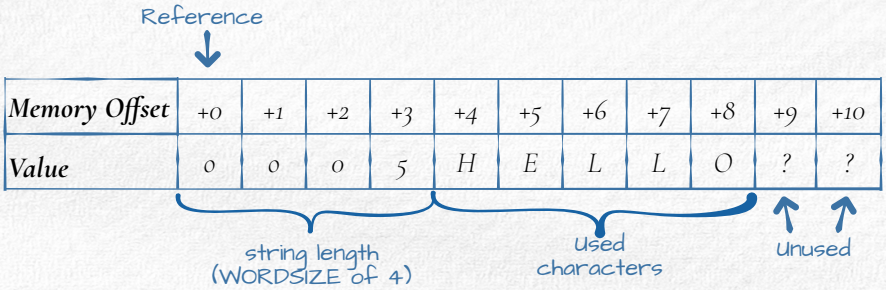
"A single byte each..." except in special cases, such as foreign languages, where accented characters are recorded in double byte format.

In Inform, character arrays/buffers are most commonly used to represent strings for manipulation purposes. The structure of these consists of a WORD, containing the number of characters stored in the array, followed by a string of characters, generally taking up a single byte each.

The following diagram depicts this model as it is expressed on the Z-machine platform:



Glulx, having a 32 bit WORD, deviates from this only slightly:



If you need a technical deep dive on traditional Inform arrays, take a look at Zac McKracken's excellent article, "Arrays, strings, and other Inform goodies" persisted forever at:

https://web.archive.org/web/20040223043422/http://www.onyxring.com/InformGuide.aspx?article=41

We won't cover standard Inform buffers more deeply than this, they've been documented in other places, but a few of points are worth mentioning:

- References to buffer arrays point to the start of the buffer (i.e., memory offset **+0** in the above chart).
- The length of the string in the buffer can be obtained with the (**word**) *array operator*:

```
addr-->0
```

- The first character of the buffer string can be obtained with the **byte** *array operator*, adjusted past the space used to store the length:

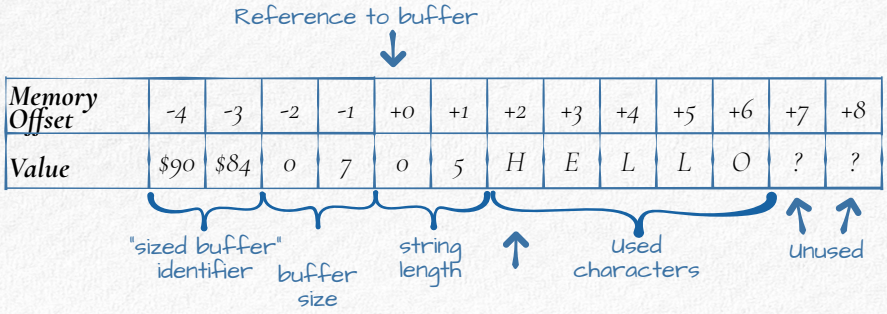
```
addr->WORD_SIZE
```

Although buffer arrays store the *length* of the string they contain, there is not a way to retrieve the *allocated size* of the buffer itself. As such, code writing strings to buffers can easily overrun them.

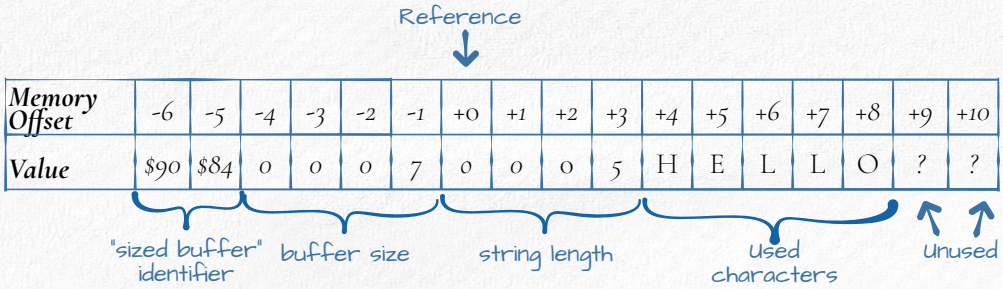
See the "Size vs. Length" side bar on page 101 for the difference between size and length.

Sized Buffers

Sized Buffers expand on the traditional model by also storing the actual buffer size, in characters, thereby providing a limit for string operations to honor. Here's a Z-machine version of the same string depicted previously, but in Sized Buffer format:



In Glulx, the memory footprint looks like this:



Notice two features:

- The structure of Sized Buffers is identical to that of normal buffers from memory offset +0, forward.

- References to sized buffers are offset ***past*** the start of allocated memory, positioned four bytes in for the Z-machine, and six bytes in for Glulx, at the `+0` offset depicted above.

  This preserves the normal approaches for determining the string length (`addr-->0`) and accessing characters (`addr->WORDSIZE`). In nearly all cases, sized buffers are indistinguishable by code written for normal string buffers.

The memory preceding buffer reference (i.e., left of the +0 offset) is where information for size-aware code (such as that used by the `util.orBuf` utility) is stored.

I cover the orBuf utility object on page 101.

Accessing the size of the buffer can be done explicitly with...

```
addr-->-1
```

...or with the more readable:

```
util.orBuf.getSize(addr)
```

The first two bytes of a Sized Buffer (the allocated memory, not the adjusted reference) always equal $90 and $84. This two-byte length serves to distinguish Sized Buffers from standard buffers, allowing code to test for this with the obscure...

```
if(addr->(-WORDSIZE-2)==$90 && addr->(-WORDSIZE-1)==$84) ...
```

...or more concisely with:

```
if(util.orBuf.isSized(addr)) ...
```

The **orBuf** utility object also provides the **convertToSizedBuffer()** routine to convert a standard buffer into a Sized Buffer. The following demonstrates:

```
array addr buffer 18;
```

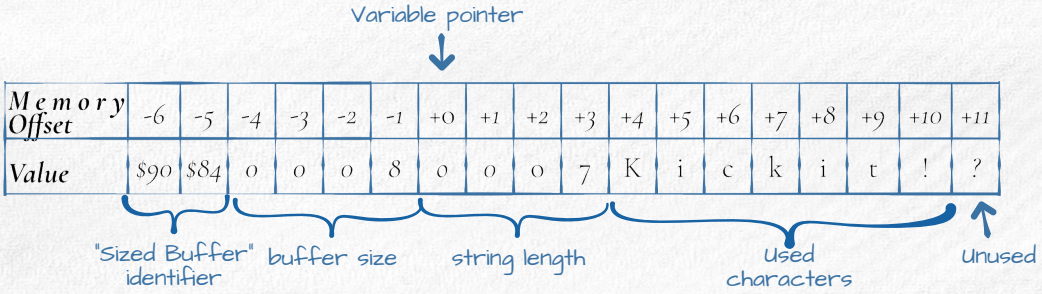
•••

```
addr=util.orBuf.convertToSizedBuffer(addr, 18);
```

Now we can treat **addr** the same way we would any other buffer:

```
PrintToArray(addr,"Kickit!");
```

And the result will look like this for Glulx:



Just to reiterate the behavior: notice the buffer size is 8, even though the memory allocated was 18. This reflects the space available for characters. The formula to produce this is straightforward:

Actual allocated array:	18	
	Glulx	***Z-Machine***
*Subtract WORDSIZE used to store **current** string length*	-4	-2
*Subtract WORDSIZE used to store **max** string length*	-4	-2
Subtract 2 bytes for the Sized Buffer Identifier	-2	-2
Resulting Buffer Size	**8**	**12**

Note the $9084 indicator is always two bytes. It is not a WORD so remains two bytes regardless of the Glulx or Z-code target platform.

Running the Unit Tests

19

Technical Topics

20 The orExtension Framework

Technical Topics

Historically, Inform 6 authors have had limited options for factoring their work into reusable packages. To a large degree, this is due to the standard library's nuanced sequencing requirements which stipulate different bits of code be included at different places in relation to the three standard include files. Its not uncommon for extensions to include multiple "chunks" of code, each requiring inclusion at different points in a game's source code.

The orExtensionFramework was created to address this seeming lack of elegance and to simplify the use of extensions for game authors. It brings the following characteristics to Inform extensions:

- Extensions need be **#included** only once, near the top of a game's source, regardless of the kinds of code they contain or where the code ultimately needs to be injected.
- Extensions may include other extensions, automatically resolving dependencies while ensuring dependent extensions are included only the appropriate number of times, even if other extensions include them as well.

The four places which code can be injected into an Inform 6 program are:

1) The **REPLACE** section, which appears before the inclusion of Parser.h. REPLACE directives go here.

2) The **MESSAGE** section, appearing between Parser.h and Verblib.h, is where things like the SACK_OBJECT, and the task_scores array go.

3) The **CODE** section, between the Verblib.h and Grammar.h, is the primary place for custom code.

4) The **GRAMMAR** section, following inclusion of the Grammar.h file, includes all new Verb and Extend directives.

The orExtensionsFramework is not limited to orLibrary extensions, assuming they follow the structural requirements. You can use it even if you choose not to include any of the orLibrary extensions at all.

Installing the Base orExtensionFramework

To install the empty framework, without any extensions, download it from the following URL:

```
https://github.com/onyxring/orLibrary-for-I6/blob/main/
    orExtensionFramework.zip
```

Unzip the framework into its own folder. There are six files:

```
#parser.h
#verblib.h
#grammar.h
#orExtensionFramework.h
#orExtensionRegistry.h
orExtensionTemplate.h
```

No surprise. These steps are the same ones for setting up the orLibrary, covered on page 5.

Add the path to these files to your project's ICL settings, just as you do the standard library. Then substitute **include "#parser"** for the traditional **#include "parser"**, **include "#verblib"** instead of **include "verblib"**, and **include "#grammar"** rather than **include "grammar"**.

Don't let these alternative filenames put you off. These variant include files do nothing more than wrap the standard library's files. You can confirm this by opening them up in your text editor. Here's the #parser.h file, for example:

```
#include "#orExtensionFramework"; #include "parser"; #include "#orExtensionFramework";
```

The empty framework is now setup correctly. Compiling it, even without adding any extensions, should reward you with the following:

```
Inform 6.42 (10th February 2024)
orExtensionFramework: pre-PARSER section.
----
orExtensionFramework: AFTER PARSER section.
----
orExtensionFramework: AFTER VERBLIB section.
----
orExtensionFramework: AFTER GRAMMAR section.
----
```

Date and version will vary.

The #orExtensionFramework.h File

Despite the uninteresting output, there's a fair amount happing. The core component of the framework, the #orExtensionFramework.h file, does several things. It traces the various stages of inclusion for the standard library; it sets up the necessary constants used to manage extensions; it checks to make sure there are no obvious issues in the way it was included.

If it detects something is amiss, it will either throw a compile time warning, letting the developer know they have a non-catastrophic issue which needs attending to, or it will throw a hard error if the problem is something which can't be ignored. For example:

```
ERROR: Missing inclusion detected. The orExtensionFramework must
be included four times...
```

Assuming things are all good, the extension framework passes control to the #orExtensionRegistry.h file, giving all registered extensions an opportunity to do something.

The orExtensionTemplate.h File

The orExtensionTemplate.h file provides a great starting point for creating new extensions which adhere with the framework's required structure. Just copy the template to a new file, with your extension's name, open it in your favorite editor, and replace all occurrences of **<REPLACEWITHNAME>** to the *same name* you specified for your extension (not including the .h extension).

Same name isn't a strict requirement. For example, I use slightly different file names to distinguish primary extensions from supporting extensions, but this pattern keeps things straightforward.

Let's step through the structure of an extension. The following example assumes the **<REPLACEWITHNAME>** text has been replaced with **myExtension**.

The Block 1: Dependencies Declaration

The first few lines, excluding header comments will look something like this (we used **myExtension** as the name for this example, but you should use your own):

```
#ifndef         orExtensionFramework_STAGE;
default         myExtension_STAGE  0;
!-------------------------------------------------------------
! INCLUDE DEPENDENCIES

!-------------------------------------------------------------
```

We aren't including dependent extensions in this example, but if we did it would just look like another include:

```
#Include "myDependency";
```

This piece of the template code checks to see if the extension has been included *before* the orExtensionFramework.h file. If so, it defines a "stage constant" to remember this and includes the declared dependencies. Note that, dependencies are also expected to be registered and must comply to the same structure expected by the framework. As such, each dependent extension included in this phase, performs the same above steps, even adding their own dependencies.

The net effect of this is extensions do not contribute code to the compilation process unless they are first included explicitly (before **#parser)**.

Block 2: Usage Error Safeguards

The next block of code performs some general housekeeping:

```
#ifnot;
#ifndef        myExtension_STAGE; message fatalerror orXFErrorInclude; #endif;
#iftrue(       myExtension_STAGE  < LIBRARY_STAGE);
   #undef      myExtension_STAGE  ;
   Constant    myExtension_STAGE  LIBRARY_STAGE;
   #ifdef      myExtension_STAGE  ; #endif;
   message "   myExtension...";
```

It guards against the extension being included in error and ensures it isn't included more times that it should be.

Block 3: Extension Code

The next four parts of the template are all similar to each other:

```
#iftrue (LIBRARY_STAGE == BEFORE_PARSER);

#endif;
#iftrue (LIBRARY_STAGE == AFTER_PARSER);

#endif;
#iftrue (LIBRARY_STAGE == AFTER_VERBLIB);

#endif;
#iftrue (LIBRARY_STAGE == AFTER_GRAMMAR);

#endif;
```

Most extensions put the majority of their code between verblib and grammar.

The content of these four segments is injected into the game at their corresponding locations. As you add code to a new extension, place it in the appropriate segment. You are free to delete the conditionals you don't supply code for.

Block 4: Closing the Extension

The last two **`#endif`**s close out conditional blocks started in Block 1:

```
!-------------------------------------------------------
#endif;
#endif;
```

The #orExtensionRegistry.h File

The #orExtensionRegistry.h is just a list of **`#include`** directives for all extensions, wrapped in conditionals. The one we just downloaded will be empty, other than with comments. As you add extensions to your library, you'll register them with lines like this:

```
#ifdef myExtension_STAGE; #include "myExtension"; #endif;
```

The key point here is that inclusion of **`myExtension.h`** isn't even attempted unless its corresponding "stage constant" has been previously defined, which extensions do themselves when first included.

21 The LibraryExtensions Object

Technical Topics

Inform's standard library defines the LibraryExtensions object to support specifically named properties, or "hooks," which extensions may use to override default behavior. Because the LibraryExtensions object was introduced late in the standard library's development, it's full potential was never fully realized. It isn't covered in the DM4 and its documentation in the wild is sparse at best. As such, in this section we'll briefly cover the "hook" properties defined in vanilla Inform and how to use them, plus new hooks introduced by the orLibrary.

In this guide I generally avoid covering intrinsic features of the standard library, but for the LibraryExtensions object I make an exception. From what I can find, it isn't covered adequately elsewhere and I use this feature quite a lot in the orLibrary, even building on the foundation it provides.

Inform's LibraryExtensions Object

The following is an example of extension code (or user code) leveraging the **ext_newRoom** hook to conditionally announce the player character's arrival in a new location:

```
object  _ballroomAnnouncer LibraryExtensions
    with ext_newRoom [;
        if(royalBallInProgress == true && location has inPalace)
            "~Her Majesty the Queen!~ declares the herald.";
    ];
```

Notice the objects hook properties are defined on *must* be children of the **LibraryExtensions** object to be in effect.

If the **ext_newRoom** property seems familiar, it may be because its analogous to the standard library's **NewRoom** entry point. In fact, all of the entry points covered in the DM4 (§A5) have hook counterparts, plus a few new ones introduced to the standard library after the DM4 was released.

Prior to the advent of **LibraryExtensions**, unmodified Inform required developers to handle all conditions in a single **NewRoom** routine, leaving no room for extensions to weigh in. This approach continues to work fine; however, defining **ext_newRoom** hook instances, as we do above, allows us to arrange multiple blocks of logic, spread across different files, which each handle different scenarios. The above example, and the one below, can legally coexist:

```
object  _inTown LibraryExtensions
    with ext_newRoom [;
            if(location has inTown)
                    "~Look!~ Whisper the townsfolk. ~It's HER!~";
    ];
```

Hook Return Values and Variations

Most entry points have an override ability. That is, a hook only runs if its associated entry point chooses to let it by returning an appropriate value. This has the side-effect of invalidating some of the DM4. In particular, its declaration that some entry points have "No return value" is no longer true (**NewRoom**, for example).

Each property hook has its own implementation and therefore its own functional nuances. For instance, the **ext_newRoom** hook is called by the standard library *only* if the **NewRoom** entry point returns **false** (or isn't defined at all). If so, all instances of **ext_newRoom** are called, regardless of their return value.

Other property hooks behave differently. The **ext_parseNoun** hook, for example, is activated *only* if the **ParseNoun** entry point returns **-1** (or is not defined). If so, individual instances of **ext_parseNoun** are called, one at a time, until one returns something other than **-1**. This return value is then accepted by the parser and any remaining hook instances are ignored.

In nearly all cases, acceptable return values from entry points mirror those defined for their corresponding entry point.

Standard Library Hooks

The following are all `LibraryExtensions` hooks defined in the Inform 6 standard library. Except where identified, more detail on the function of these hooks can be found in the DM4's coverage of the associated entry points, or in the "Game Author's Guide to Glulx Inform" (which I'm abbreviating as GAGGI here) for Glulx-specific entry points.

I identify these exceptions by prefixing them with ★.

`ext_afterLife` Called when the player character has died.
Runs following the `AfterLife` entry point, if it returns `false`.
Runs until all instances have run.

`ext_afterPrompt` Called after the game prompt has been printed.
Runs following the `AfterPrompt` entry point, if it returns `0`.
Runs until all instances have run.

Technically, `false` and `0` are the same in Inform; as they are in many languages; however, semantically, they differ so I make this distinction based on the underlying code of the standard library.

★ `ext_afterRestore` Called after the interpreter has restored a saved game.
Runs following the `AfterRestore` entry point, if it returns `1`.
Runs until all instances have run.

★ `ext_afterSave` Called after the interpreter has saved the game state to a file.
Runs following the `AfterSave` entry point, if it returns `1`.
Runs until all instances have run.

These relate to relatively new entry points which are not covered in the DM4 or the GAGGI.

`ext_amusing` Called after the player has won the game, provided the `AMUSING_PROVIDED` constant has been defined.
Runs following the `Amusing` entry point, if it returns `false`.
Runs until all instances have run.

`ext_beforeParsing` Called after the parser has received player input, but before any parsing has actually taken place.
Runs following the `BeforeParsing` entry point, if it returns `false`.
Runs until one of the hook instances does not return `false`.

The standard library's `WN` variable is reset before each hook instance is called, effectively reseting the parsing state for each until a non-`FALSE` value is returned.

`ext_chooseObjects` Like the `ChooseObjects` entry point, this hook is called in two different contexts: First to validate objects for consideration when the parser is processing an all grammar token; second to resolve ambiguity between multiple objects.
Runs following the `ChooseObjects` entry point, if it returns `0`.
Runs until: (if resolving all) one of the instances does not return `0`.
(if resolving ambiguous candidates) all instances have run.

When resolving ambiguous candidates, the highest value returned from all hook instances is ultimately returned to the parser.

`ext_darkToDark` Called when the player moves from one dark room to another.
Runs following the `DarkToDark` entry point, if it returns `false`.
Runs until all instances have run.

`ext_deathMessage` Called when the player character has died, but with a non-standard `deadflag` value (i.e., `3` or more) to print a custom message.
Runs following the `DeathMessage` entry point, if it returns `false`.
Runs until all instances have run.

`ext_gamePostRoutine` Called after all game actions.
Runs following the `GamePostRoutine` entry point, if it returns `false`.
Runs until one of the instances does not return `false`.

`ext_gamePreRoutine` Called before every game action.
Runs following the `GamePreRoutine` entry point, if it returns `false`.
Runs until one of the instances does not return `0`.

Glulx-only entrypoints are documented in the GAGGI, found here:

https://eblong.com/zarf/glulx/inform-guide.txt

`ext_handleGlkEvent` (Glulx only) Called every time a Glk event occurs.
Runs following the `HandleGlkEvent` entry point, if it returns `0`.
Runs until one of the instances does not return `0`.

`ext_identifyGlkObject` (Glulx only) Called in three phases: one phase for setting Glk object references to `0`, another to set references for windows, streams, and files, and a last phase to set references to object types other than windows, streams, or files.
Runs following the `IdentifyGlkObject` entry point, if it returns `false`.
Runs until one of the hook instances does not return `false`.

`ext_initGlkWindow` (Glulx only) Called in five phases: 1) at the beginning of execution; 2) before creating the story window; 3) before the creation of the status window; 4) at the end of window setup; 5) before creating a quote box window.
Runs following the `InitGlkWindow` entry point, if it returns `false`.
Runs until one of the instances does not return `0`.

`ext_initialise` Called to perform pre-game setup.
Runs before the `Initialise` entry point.
Runs until all instances have run.

`ext_inScope` Called to provide an opportunity to change the library's definition of what is in scope.
Runs following the `InScope` entry point, if it returns `false`.
Runs until one of the instances does not return `false`.

`ext_lookRoutine` Called after every room description.
Runs following the `LookRoutine` entry point, if it returns `false`.
Runs until all instances have run.

`ext_messages` Called every time a library message is printed (via the `L_M` routine), before the message is actually output.
Runs following the `LibraryMessages` object's `before` property is processed.
Runs until one of the instances does not return `false`.

`ext_newRoom` Called when the player character's room changes, before anything is printed.
Runs following the `NewRoom` entry point, if it returns `false`.
Runs until all instances have run.

The ObjectDoesNotFit entry point is not in the DM4 or the GAGG1.

`ext_objectDoesNotFit` Called when taking game objects or inserting them into something else to determine if there is enough room to do so.
Runs following the `ObjectDoesNotFit` entry point, if it returns `false`.
Runs until one of the instances does not return `false`.

`ext_parseNoun` Called when attempting to match player input to a game object, assuming the object's parse_name property routine returned was unsuccessful (i.e., returned `-1`).
Runs following the `ParseNoun` entry point, if it returns `-1` or is not defined.

The standard library's WN variable is reset for each hook instance until -1 is not returned.

Runs until one of the instances does not return `-1`.

`ext_parseNumber` Called to match a number from player input.
Runs following the `ParseNumber` entry point, if it returns `0`.
Runs until one of the instances does not return `0`.

I make a small change to this in the orHookParser extension (see the sidebar annotation on page 129 for more).

`ext_parserError` Called to print an alternative parser error message than the default.
Runs following the `ParseError` entry point, if it returns `0`.
Runs until one of the instances does not return `0`.

`ext_printRank` Called to append text after the score is printed.
Runs following the `PrintRank` entry point, if it returns `false`.
Runs until one of the instances does not return `0`.

ext_printVerb Called to print the printed verb when the parser asks a question.
Runs following the **PrintVerb** entry point, if it returns **0**.
Runs until one of the instances does not return **0**.

ext_printTaskName Called to print the name of a numbered task.
Runs following the **PrintTaskName** entry point, if it returns **false**.
Runs until one of the instances does not return **0**.

ext_timePasses Called after each turn where time passes.
Runs following the **TimePasses** entry point, if it returns **0**.
Runs until all instances have run.

ext_unknownVerb Called when the parser encounters an unknown verb so you may identify it.
Runs following the **UnknownVerb** entry point, if it returns **false**.
Runs until one of the instances does not return **false**.

New Hooks Introduced by the orLibrary

Unlike the hooks introduced by the standard library to extend the pre-existing entry points, *most* orLibrary hooks have no equivalent entry points.

In the future, if it makes sense, I may choose to introduce equivalent entry points, but... meh.

The various orLibrary extensions, combined, add several dozen new **LibraryExtensions** hooks to the standard library, plus a handful of hooks specific to the extensions they support. The following is a list of these, grouped by extension.

orDialogue.h

The orDialogue hooks are used to influence the way conversation occurs.

ext_topicAsk Called when a character is about to ASK about an **orTopic**.
Runs before the ASK command is executed.
Runs until one of the instances returns **true**.
Return values: **true** aborts the ASK command; otherwise, it proceeds.

ext_topicAsked Called after a character has ASKed about an **orTopic**.
Runs before the target character has answered.
Runs until one of the hook instances returns **true**.
Return values: **true** causes the target character to not answer; otherwise, they proceed.

ext_topicTell Called when a character is about to TELL about an **orTopic**.
Runs before the TELL command is executed.
Runs until one of the instances returns **true**.
Return values: **true** aborts the TELL command; otherwise, it proceeds.

ext_topicTold Called after a character has told (via the TELL action) about an **orTopic**.
Runs before the target character has answered.
Runs until all instances have run.
Return values: none.

_orHookBanner.h

The orHookBanner extension provides a hook for extensions to add information to the game's banner.

ext_bannerText Called to supplement the game's banner text.
Runs before the **orBannerText** entry point/constant.
Runs until all instances have run.
Return values: none.

_orHookInformLibrary.h

The orHookInformLibrary extension adds several hooks to the standard library's `InformLibrary` object, providing places for extensions to alter default behavior of the primary game loop.

`ext_playLoopAssignObjectVariables` Called when assigning the `noun` and `second` variables.
- ***Runs after*** the library has parsed player input (via `InformParser` object's `parse_input` property routine.
- ***Runs until*** one of the instances does not return `false`.
- ***Return values:*** `false` - continue with default behavior; any other value will skip default behavior.

`ext_playLoopPerformMultiObjectAction` Called when performing a command against multiple objects.
- ***Runs after*** the parser has considered single object actions.
- ***Runs until*** one of the instances does not return `0`.
- ***Return values:*** The largest return value of all instances is considered. `0` - continue with default behavior; `1` - go back and reassign object variables; `2` - end the turn; `3` - go back and parse the input buffer again.

`ext_playLoopPerformSingleObjectAction` Called when performing a simple command, when the `noun` and `second` variables are correctly resolved.
- ***Runs after*** the parser has assigned object variables.
- ***Runs until*** one of the instances does not return `0`.
- ***Return values:*** The largest return value of all instances is considered. `0` - continue with default behavior; `1` - go back and reassign object variables; `2` - end the turn; `3` - go back and parse the input buffer again.

`ext_playLoopRewriteParsedCommands` Called to change the action variable and swap entries in the `inputobjs` array.
- ***Runs after*** the library has parsed player input (via `InformParser` object's `parse_input` property routine.
- ***Runs until*** one of the instances does not return `false`.
- ***Return values:*** `false` - continue with default behavior; any other value will skip default behavior.

_orHookKeyboard.h

The orHookKeyboard extension adds `LibraryExtensions` hooks to the standard library's player input logic.

`ext_beforePrompt` Called before the game prompt has been printed. This hook is the logical counterpart to the standard library's `ext_afterPrompt` hook.
- ***Runs before*** the standard library has printed the game prompt.
- ***Runs until*** one of the instances does not return `false`.
- ***Return values:*** `false` - continue with default behavior; any other value will skip default behavior.

`ext_keyboardHandleComment` Called to override comment handling in player input.
- ***Runs before*** the standard library detects and handles comments.
- ***Runs until*** one of the instances does not return `false`.
- ***Return values:*** `-1` - go back and get a new line of input from the player; `false` - continue with default behavior; any other value will skip default behavior.

ext_keyboardHandleOops Called detect and handle players correcting mistakes from previous input.
Runs before the standard library detects and handles player corrections.
Runs until one of the instances does not return **false**.
Return values: **-1** - go back and get a new line of input from the player; **false** - continue with default behavior; any other value will skip default behavior.

ext_keyboardHandleUndo Called detect and handle players calling UNDO.
Runs before the standard library detects and handles UNDO commands.
Runs until one of the instances does not return **false**.
Return values: **-1** - go back and get a new line of input from the player; **false** - continue with default behavior; any other value will skip default behavior.

ext_keyboardGetInput Called before getting player input from the interpreter.
Runs before the standard library has decided to accept player input.
Runs until one of the instances does not return **false**.
Return values: any value other than **false** will skip normal behavior.

Called when deciding whether or not to get player input, which is slightly different than...

ext_keyboardPrepInput Called to for preliminary checks against player input.
Runs before the standard library checks the player input for inconsistencies.
Runs until one of the instances does not return **false**.
Return values: **-1** - go back and get a new line of input from the player; **false** - continue with default behavior; any other value will skip default behavior.

...this one, which is called when actually starting to get input.

ext_keyboardPrimitive Called when getting player input from the interpreter.
Runs before the standard library has started to accept player input.
Runs until one of the instances does not return **false**.
Return values: any value other than **false** will skip normal behavior.

_orHookParser.h

The orHookKeyboard extension adds **LibraryExtensions** hooks to the standard library's input parsing logic.

ext_adjudicate Called to determine which, from a list of possible candidates, should be selected by the parser.
Runs before the standard library attempts to process normal adjudication logic.
Runs until one of the instances does not return **false**.
Return values: the selected object to ; **false** to use default logic.

In addition to these new hooks, orHookParser also adds the **results** parameter to the **ext_parserError** hook. This is a local variable from the calling routine, but is used in the code which the hook can override, so it makes sense.

ext_forceLookAhead Called to determine if look ahead logic in the parser should occur, even in cases where it would not normally happen.
Runs before the standard library attempts look ahead resolution of each token.
Runs until one of the instances does not return **false**.
Return values: **true** to force look ahead logic; anything else to use default logic.

ext_suppressParsingImplicitTake Called to determine if implicit TAKE actions generated during parsing (not those generated from verb handlers) should occur as normal.
Runs before the parser attempts to perform implicit take actions.
Runs until one of the instances does not return **false**.
Return values: **-1** - go back and get a new line of input from the player; **false** - continue with default behavior; any other value will skip default behavior.

_orHookStandardLibrary.h

The orHookStandardLibrary extension adds several **LibraryExtensions** hooks to the standard library.

ext_afterAction Called after an action has been performed.
Runs after the standard library has performed an action.
Runs until all instances have run.
Return values: none.

ext_afterRoutines Called before running **after** property routines following specific verbs.
Runs before the standard library runs **after** property routines via the **AfterRoutines** routine.
Runs until one of the instances does not return **0**.
Return values: **0** - continue with default behavior; any other value will skip processing of **after** routines.

ext_beforeAction Called before an action is performed.
Runs before the standard library performs an action via the **ActionPrimitive** routine.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior; any other value will skip the action.

ext_displayStatus Called before displaying the status bar.
Runs before the standard library displays the status bar via the **DisplayStatus** routine.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior; any other value will skip default behavior.

ext_identical Used to override the default logic for determining if two objects are identical.
Runs before the standard library determines if two objects are indistinguishable via the **Identical** routine.
Runs until one of the instances does not return **true**.
Return values: **true** - continue with default behavior; **false**, skips the default logic.

ext_nextword Used to override the default logic for advancing the parser's current word number pointer.
Runs before incrementing the **wn** variable.
Runs until one of the instances does not return **0**.
Return values: **0** - continue with default behavior; **comma_word** if the current word is a comma; **THEN1__WD** if the current word is a full stop; the dictionary address is it is a recognized word.

ext_nounword Used to override the default logic for advancing the parser's current word number pointer when processing noun.
Runs before incrementing the **wn** variable.
Runs until one of the instances does not return **0**.
Return values: **0** - continue with default behavior; **1** if the current word is a word meaning "ME"; other values will be interpreted as either the dictionary address is it is a recognized noun; or an entry in the pronoun table (**+2**) if the word is a pronoun.

_orHookVerbs.h

The orHookVerbs extension adds hooks to select standard library verbs.

ext_examineSub Used to override, or supplement, the default behavior of the EXAMINE verb.
Runs before the EXAMINE verb is processed.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior; any other value skips default processing.

ext_preDescription Used to prefix text to the front of descriptions.
Runs before the standard library's EXAMINE verb prints a description.
Runs until all instances have run.
Return values: none.

ext_postDescription Used to add text to the end of descriptions.
Runs after the standard library's EXAMINE verb prints a description.
Runs until all instances have run.
Return values: none.

ext_lookSub Used to override, or supplement, the default behavior of the LOOK verb.
Runs before the LOOK verb is processed.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior; any other value skips default processing.

ext_goSub Used to override, or supplement, the default behavior of the GO verb.
Runs before the GO verb is processed.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior; any other value skips default processing.

ext_goSubNoDir Used to override the default behavior when the player characters attempts to GO in an invalid direction.
Runs before the library prints a "you can't go that way" type of message.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior; any other value skips default processing and aborts the rest of the GO routine's logic.

_orHookWriteAfterEntry.h

The orHookWriteAfterEntry extension adds hooks to change the logic underpinning the standard library's list building routines, specifically with regard to how it handles settings in the current list style (**c_style**) variable.

ext_sdAlwaysBit Used to override the default handling of the "always recurse downward" setting.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior.

ext_sdConcealBit Used to override the default handling of the "omit concealed objects" setting.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior.

ext_sdDoList Used to override the default behavior for actually printing the substance of a list.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior.

ext_sdFullInvBit Used to override the default handling of the "full inventory information after entry" setting.
Runs until one of the instances does not return **false**.
Return values: - **false** continue with default behavior.

ext_sdIsOrAre Used to override the default handling of the "print is or are" setting.
Runs until one of the instances does not return **false**.
Return values: **false** - continue with default behavior.

`ext_sdNewLineBit` Used to override the default handling of the "new line after each entry" setting.
Runs until one of the instances does not return `false`.
Return values: `false` - continue with default behavior.

`ext_sdPartInvBit` Used to override the default handling of the "brief inventory information after entry" setting.
Runs until one of the instances does not return `false`.
Return values: `false` - continue with default behavior.

`ext_sdRecurseBit` Used to override the default handling of the "recurse downward with usual rules" setting.
Runs until one of the instances does not return `false`.
Return values: `false` - continue with default behavior.

_orHookWriteList.h The orHookWriteList extension adds a hook to change the logic underpinning the standard library's list building routines.

`ext_wlrPluralMany` Used to override the default method for printing the quantity of an entry in a list.
Runs before printing the number of identical items.
Runs until one of the instances does not return `false`.
Return values: `false` - continue with default behavior; any other value skips default processing skips the default logic.

---

orImplicitCommands.h
`ext_generateImplicitCommands` Used to detect conditions where an implicit command is relevant, and generate it.
Runs before performing any action.
Runs until one of the instances does not return `false`.
Return values: none.

---

orMenu.h
`ext_handleMenuResult` Used to handle player selection of a menu item.
Runs after the player has selected an item in the menu.
Runs until one of the instances does not return `false`.
Return values: `false` - continue with the default behavior, which is to close the menu; any other value causes the menu to stay active.

Print Pattern Syntax (ABNF)

The discussion of the orPrint extension's print pattern syntax, covered on page 66, is appropriate for most audiences, but it is inexact. Below is the syntax for print patterns in the precise notation of Augmented Backus-Naur Form (ABNF):

```
esc-dollar = "$$"
esc-colon = "$:"
esc-open-parentheses = "$("
esc-close-parentheses = "$)"

colon = ":"
param-char = %x20-23 / %x25-27 /  %x2A-39 / %x3B-7E

pattern-name = *1(ALPHA / DIGIT)
object = DIGIT / "actor" / "player" / "noun" / "second"

param-token = *param-char
param-string = "(" param-token *(colon param-token) ")"

name-object = object [colon pattern-name]
name-object =/ pattern-name [colon object]
name-object =/ colon pattern-name
name-object =/ colon object

print-pattern = "$" name-object [param-string] [";"]
```

This line covers all printable characters except for: $,), and :.

For more information about the Augmented Backus-Naur Form, see the Internet Standard 68 (RFC 5234 at the time of this writing) found at:

https://www.rfc-editor.org/info/std68

Index


58008
NAT
1

>EXAMINE GUIDE

The orLibrary User's Guide is a detailed walkthrough of the second version of the OnyxRing Library for Inform 6. Updated from its original 2001 form, this latest iteration of the library provides dozens of easy to drop-in extensions built to accompany the standard library. Some of these are simple, others sophisticated, but all streamline development of parser-based interactive fiction for Z-machine and Glulx interpreters.

>ASK ABOUT AUTHOR

Jim Fisher is the author of a handful of Interactive Fiction titles, including his debut release of "Wade Wars: Book III" in 1992. His demonstrative works, for which he released source code, include his entry in the Great IF Toaster Contest "Got Toast?" and the conversational experiment "Medusa." His title "Statue" took first place in the 2003 Intro Comp competition. Jim's career in software design and development spans thirty years. By day he serves in the role of Enterprise Architect for a Fortune 100 company. By night, he immerses himself in the enjoyable act of Making New Things.