

Roger Firth's IF pages

InfLight -- Inform debugging



On this site, you'll find an overview of tools and techniques to assist in understanding and debugging your Inform games (I've done similar things about object definitions at [InFancy](#) and about NPCs at [InfAct](#)). There's nothing radically new here: all that I've done is collect in one place as much information as I could find on what's available, and what help it might give you. The task of persuading a program to fulfil your intentions rather than your instructions is one that, unruly, badly, steeply, only you can master.

The material is loosely organised as follows:

[The compilation process](#)

Setting the scene -- a gentle warmup.

[Compiling](#)

How to compile a game (on a PC) **and see what went wrong**, and how to control the compiler using switches.

[Switches](#)

A roundup of the behaviour of the more useful switches.

[Tools](#)

Utilities, other than interpreters, which process a game file.

[Static debug](#)

Looking inside objects, and making small changes to the game's state.

[Dynamic debug](#)

Smell the smoke, see the flames, hear the crash.

[Infix](#)

Getting real tricky with Graham's debug module.

[Nitfol](#)

A 'terp with attitude (and an inbuilt debugger).

[Numbers](#)

The Z-machine's use of untyped numbers for all purposes, and an introduction to the memory map.

[Hints and tips](#)

Time for some motherhood 'n' apple pie.

Conventions

To clarify where the various example displays come from, a little colour-coding is used:

```
This is a sample of text in an Inform source file.
```

```
This is from a Z-machine interpreter at run-time.
```

```
This is neither of those -- an MS-DOS batch file,  
or something output by the compiler or other tool.
```

Acknowledgements

Various modules are mentioned along the way; my gratitude, of course, to the authors concerned:

- Start at the [Inform home page](#) for details of the **compiler** and the **library**, various **interpreters** (including **nitfol**), **Ztools** and **Disinform**.
- Pick up [Marnie Parker's ObjLstr.h](#) and [John Cater's DebugLib.h](#) from the archives at [if-archive/infocom/compilers/inform6/library/contributions](http://if-archive.infocom.compilers/inform6/library/contributions).
- My **CheckOut.h** and **Dump.h** are in the archives, or you can get them (and the Cloak of Darkness example game) from [this site](#).

We'll start with some fairly familiar background material.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

InfLight -- Inform debugging

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

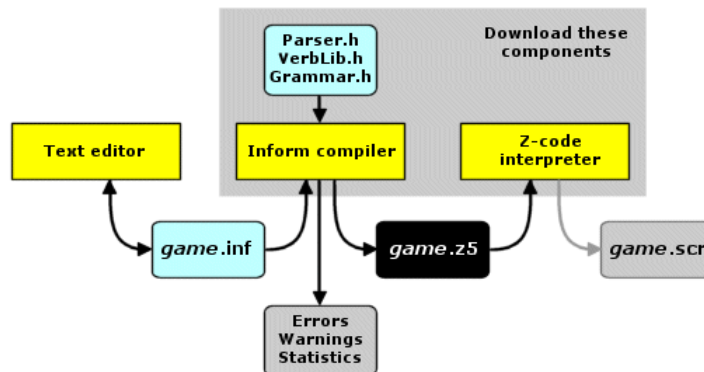
Before we get too heavily involved in the technical detail, we'll summarise the steps you go through to build and play a game. There are three major stages:

1. creating a (human-readable) text file which contains the game's source,
2. compiling that source file into (binary) Z-code held in a second file,
3. executing the Z-code within a Z-machine interpreter in order to play the game.

Text editor

You need a text editing program (rather than a word processor) to create and modify your game file. Any editor which doesn't mess around with the source will do; on Windows you can use the standard NotePad, though you'll find it much more efficient to get hold of a proper programmers' editor like [TextPad](#). On a Macintosh, [BBEdit Lite](#) is a good bet. Most people find that **syntax colouring** -- where the editor displays Inform keywords in one colour, strings in a second colour, comments in a third... -- is helpful in minimising syntax errors. [Here](#) is a comprehensive list of editors.

There's an ongoing debate about Inform IDEs (Integrated Development Environments) which -- theoretically -- ease the problems of building and debugging games by combining editor, compiler and interpreter into a single intelligent toolset. There are currently a few IDEs under construction (see for example [Inform Explorer](#) and [IF-IDE](#)), but they have yet to prove themselves in the heat of battle.



Inform compiler and Library files

The compiler transforms Inform source games into Z-code, or (more often) explains that it is unable to do so because of the errors in your program. It may also issue warnings: for example when you've created a variable but not used it, or when part of your program can never be executed. Although warnings don't prevent the Z-code being generated, they suggest something isn't quite as you intended; it's good policy to resolve the situation rather than let the warning recur. And it can reports loads of statistical and other data about your game.

The compiler is maintained by Graham Nelson. It's written in C, and Graham has taken great pains to make it **portable**. Portability means that the compiler can be made to run -- virtually unchanged -- on PCs, Apple Macs, UNIX boxes and a wide variety of other computing devices, and therefore that a given game will compile into identical Z-code on all of those platforms. The fact that a game's Z-code is always the same, irrespective of the computer on which it originated, is one of Inform's strengths.

Not that you have to worry much about the compiler. A host of public-spirited souls have already got it working on almost every conceivable platform, so all you need do is download the appropriate ready-to-run version. See Graham's [Inform Downloads](#) page or [Jonadab's list](#). (You can also find the compiler's C source program via Graham's page, but you're most unlikely to need it in that form.)

You also need to download the Library: **PARSER.H**, **VERBLIB.H**, **GRAMMAR.H** and a few other Inform source files which provide a basic framework for your game. Thanks to the Library, you get a bare-bones environment in which your players can receive commands, move around, and interact with things. The Inform program that you write then builds on this foundation, fleshing it out with your own rooms, objects and people.

Z-code interpreter

The Z-machine is an imaginary computer which exists not as physical hardware, but rather as a software interpreter able to read and execute Z-code instructions; it's an example of what are often known as **Virtual Machines**. As with the Inform compiler, there are versions of the Z-machine interpreter readily available for PCs, Macs and most other things besides. *Unlike* the compiler, the various interpreters aren't all based on a single program of Graham's (though they do all conform to a [well-defined standard](#) in whose authorship he played a major part). Standardization means that any piece of Z-code, irrespective of where or how it was compiled, will run on any Z-machine. (And, as a nice bonus, those same Z-machines will also run any of the old Infocom games: This Is Not A Coincidence.)

The Z-code interpreter is primarily intended for playing Inform text adventure games (though as a general computing engine it can occasionally be seen employed for some alternative purpose, commonly referred to as an 'abuse of the Z-machine'). So, it reads a Z-code file created by the compiler, and then displays the game's descriptions and command prompts in a scrolling text window. The only time that the interpreter normally writes to an output file is when creating a **transcript** -- a copy of your commands and the game's responses.

Controlling the compiler

The operation of the compiler can be controlled in certain respects, using mechanisms which are rather sparsely explained in the *Designer's Manual*. More detail is available from the compiler's help, but people often forget to look there. Here's what the [Windows compiler](#) says; compilers on other platforms may have differing rules for the handling of file names.

Game sizes

The compiler is able to generate several flavours of Z-code, broadly similar but matching different versions of Infocom's Z-machine as it evolved over time in the 1980s. Only two of these are used nowadays -- Version 5 (the default) and Version 8 -- and these are identical other than in the maximum supported game size. In a Version 5 game, the Z-code is limited to 256K bytes (large enough for most games), while the Version 8 limit is 512K bytes. This trick is accomplished by using **packed addressing** -- see the later [Numbers](#) segment for more detail.

You use the compiler's **v8** switch to generate Version 8 Z-code rather than the default Version 5 Z-code. Don't confuse this with the compiler's **\$small**, **\$large** and **\$huge** commands; these control how much memory is needed to *compile* the game, and are largely unrelated to the size of the game itself once compiled. (A game with lots of little objects may consume more compiler memory than a game with a few large objects, even though the two games generate a similar amount of Z-code.)

Glulx

The reason that Z-code sets an absolute limit of 512K bytes on a compiled game is that the Z-machine uses a 16-bit architecture. A word of 16 bits can address 64K memory locations; packed addressing stretches this to 256K or 512K locations, but that's as good as it gets. The inability to create larger games, together with some more subtle memory limitations plus the lack of sensible graphical and audio capabilities, have contributed towards the development of a 32-bit Virtual Machine, called **Glulx** (don't ask). You can read about Glulx, and also download copies of the new compiler, Library and **Glulxe** interpreter, from Andrew Plotkin's [comprehensive pages](#). The nice thing is: the new compiler reads your existing Inform source files and generates either Z-code or Glulx code. Hardly any changes to the games are needed -- see Zarf's [Inform guide](#) for details. There's lots more good stuff at Adam Cadre's [Gull introduction](#) (whose first few pages provide some excellent background to Glulx *and* Inform), and at Marnie Parker's [Glulx for Dunces](#).

Footnote: A short evolutionary history

I think it happened roughly like this:

1. In 1975, Will Crowther and Don Woods wrote the first text adventure in FORTRAN; it ran on mainframes and mini-computers.
2. At the MIT AI-Lab, Mark Blank, Tim Anderson et al played Adventure; they were sure that if an adventure game could be written in FORTRAN, a *better* one could be done in MDL (a Lisp-like language). The result, around 1978, was Dungeon, (from which Bob Supnik at DEC created a FORTRAN version); the MDL original, however, was soon renamed Zork.
3. Since Zork was an obvious success, its authors formed Infocom to address the home computer market. MDL being unavailable there, they devised the similar but simpler ZORK Implementation Language (ZIL) in which to create their games, and the (virtual) Z-machine on which to run them. Zork was divided into the Zork I, II and III we know today.
4. Eventually, Infocom went out of business. Activision is believed to possess the ZIL source for Zork, Enchanter and all the other 30-some Infocom games, but has never made it available. In any case, the ZIL compiler and details of Infocom's Z-machine have been lost.
5. By reverse-engineering the Z-code of the published Infocom games, a group of dedicated hobbyists known as the Infocom Task Force (ITF) reconstructed the specification of the original Z-machine. This made it possible to write additional Z-machines (for example in Java and Perl), and therefore to play the old Infocom games on many more platforms.
6. There was, however, no way to write *new* Z-code until Graham Nelson devised the Inform compiler. Inform programs probably bear little or no relationship to ZIL; that doesn't matter. What's significant is that Inform games compile into Z-code, and therefore run on all of those Z-machines.
7. And now, Andrew Plotkin's Glulxe is a new VM, bringing many benefits to today's game writers, but incompatible with the Z-machine. Its Glulx compiler reads Inform programs, and produces code for either the Z-machine or for Glulxe. At this point, therefore, the 20-year-old link back to Infocom's ZORK is finally broken.

Enough of all this talk; let's make something happen.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

InfLight -- Inform debugging




[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

You're going to need four things:


- an Inform **game** -- we'll be using my simple Cloak of Darkness example **CLOAK.INF** in this discussion;
- the Inform **compiler** -- **INFRMW32.EXE** on a PC;
- the Inform **library** -- the nine files **ENGLISH.H**, **GRAMMAR.H**, **INFIX.H**, **LINKLPA.H**, **LINKLV.H**, **PARSER.H**, **PARSERM.H**, **VERBLIB.H** and **VERBLIBM.H** (if you discover that the files you've downloaded are called just **ENGLISH**, **GRAMMAR** and so on, don't worry: simply rename them to have a **.H** extension);
- an Inform **interpreter** like **Frotz**, **Nitfol** or **Zip**.

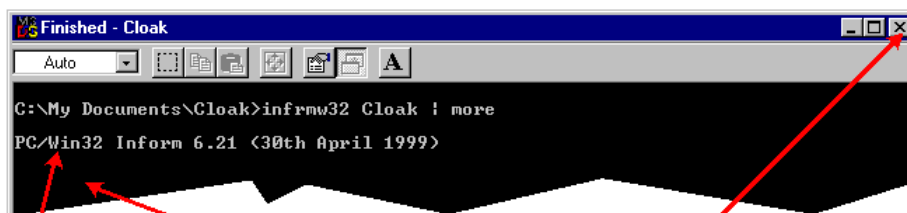
To start with, put all of these files in the same directory or folder. See Graham's [Inform Downloads](#) page or [Jonadab's list](#) if you've not yet got the library plus a compiler and interpreter for your machine.

On a PC

(Sorry -- this bit is PC-specific; I'm sure it's much easier on a Mac.) The PC compiler isn't a Windows application, which means that double-clicking its icon  in Windows Explorer doesn't do anything useful. So you can either use **MS-DOS Prompt** to open an MS-DOS window and type the compilation commands there, or (my recommendation) use a batch file with an icon which you *can* double-click. Using a text editor, create a file named after the game you're compiling -- in this example **CLOAK.BAT** -- which contains a single line:

```
infrmw32 Cloak | more
```

Now, simply double-click on the **CLOAK.BAT** icon  to compile the game and display any compiler messages in a window. (Apparently, this works only under Windows 9*; on NT and beyond you don't see the messages unless you invoke the batch file from the command prompt -- bummer.)



This is the compiler's version number and date

Any compilation errors or warnings will appear here - no news is good news

Read any messages, then click here to close the window

The word "Finished" in the window's title bar tells you that the compilation is complete; fix any errors or warnings that have been reported, then just close the window and retry the compilation. Remember that some trivial syntax errors -- a missing quote, an extra brace -- can trigger a rash of errors as the compiler loses context; if you find such an error, correct it and recompile to see if all the others go away.

An alternative technique is to redirect compiler messages to a file, by changing **CLOAK.BAT** to look like this (but I generally find it more convenient to display in the window):

```
infrmw32 Cloak >Cloak.txt
```

Yet another possibility: if you're using an Integrated Development Environment (IDE), chances are you'll be able to edit, compile and view the results through a single interface. Even if you're not, some editors provide much the same capabilities. For example, [TextPad](#) is easily configured to run the **INFRMW32.EXE** compiler against the game file you're currently editing, displaying the compiler's messages in a second editor window. Double-click on an error message... and the editor takes you straight to that line in the game file. And you can [download a configuration file](#) to apply syntax colouring to Inform games: I highly recommend this editor.

Library location

Your game will **Include** a number of library files -- **PARSER.H**, **VERBLIB.H** and **GRAMMAR.H** as a minimum, plus any optional packages like **OBJLSTR.H** which you may be using. By default, the compiler expects to find these in the same directory as the source file; you can reduce clutter and duplication by holding them in one or more standard locations, but you then need to tell the compiler where to look. The following syntax instructs the compiler to look for **Included** files (a) in the game's directory, then (b) in a **Library** sub-directory below that, and finally (c) in a **Library** directory alongside the game's directory. You can easily modify this search pattern to suit your own needs:

```
infrmw32 +.\,..\Library,..\Library Cloak | more
```

Compiler switches

You can control various aspects of the way that your game is compiled by using compiler **switches** (which are nothing to do with the Inform

language's `switch` statement or the library's `switchable` attribute). For example, the `s` switch -- they're all one or two characters long -- causes the compiler to display a host of statistics about the game, occasionally useful for checking that you're not about to exceed a Z-machine limit. There are three ways to set a switch; on the compiler command line:

```
infrmw32 -s Cloak | more
```

or secondly in an Inform Command Language file, for example `CLOAK.ICL`:

```
-s
```

which you mention in parentheses on the command line:

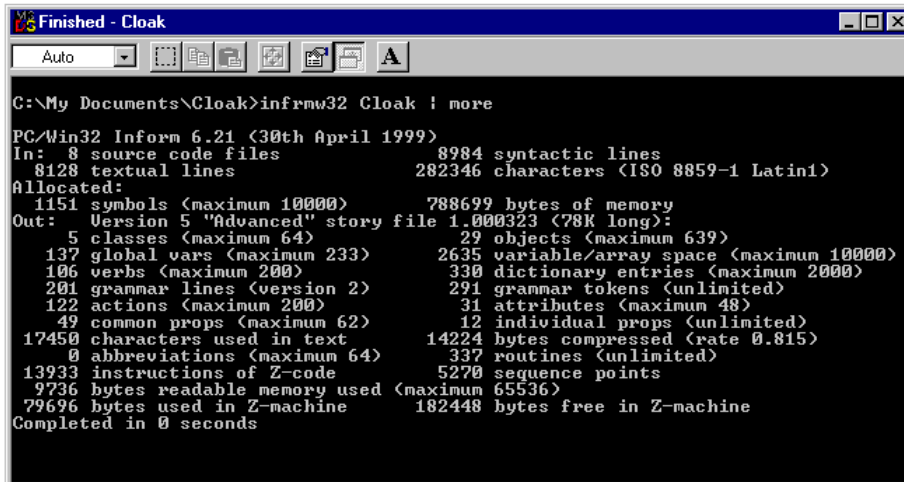
```
infrmw32 (CLOAK.ICL) Cloak | more
```

or finally at the very head of the game itself:

```
! ===== !
! Cloak of Darkness - a simple demonstration of Interactive Fiction
! This version for INFORM written by Roger Firth on 12Sep99
! ===== !

Switches s;
Constant Story "Cloak of Darkness";
Constant Headline "^A basic IF demonstration.^";
:
:
:
```

In this instance, all methods produce the same effect:



```
Finished - Cloak
Auto
C:\My Documents\Cloak>infrmw32 Cloak ! more
PC/Win32 Inform 6.21 (30th April 1999)
In: 8 source code files          8984 syntactic lines
   8128 textual lines          282346 characters (ISO 8859-1 Latin1)
Allocated:
  1151 symbols (maximum 10000)   788699 bytes of memory
Out:  Version 5 "Advanced" story file 1.000323 (78K long):
     5 classes (maximum 64)      29 objects (maximum 639)
    137 global vars (maximum 233) 2635 variable/array space (maximum 10000)
    106 verbs (maximum 200)       330 dictionary entries (maximum 2000)
    201 grammar lines (version 2)  291 grammar tokens (unlimited)
    122 actions (maximum 200)     31 attributes (maximum 48)
    49 common props (maximum 62)  12 individual props (unlimited)
  17450 characters used in text   14224 bytes compressed (rate 0.815)
     0 abbreviations (maximum 64)  337 routines (unlimited)
  13933 instructions of Z-code    5270 sequence points
   9736 bytes readable memory used (maximum 65536)
  79696 bytes used in Z-machine  182448 bytes free in Z-machine
Completed in 0 seconds
```

Note that on the command-line or in an ICL file, you use a minus sign to introduce the switch; if you have several you can give them separately `-s -z -v8` or run them together `-szv8`. At the head of the game, you *don't* use a minus sign, and if you have several switches you *must* run them together `Switches szv8`;. However, since several of the switches seem *not* to work properly at the head of the game, it's probably better to avoid this method.

Now that we know how to set them, the next segment describes some of the more useful switches.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

Roger Firth's IF pages

InfLight -- Inform debugging

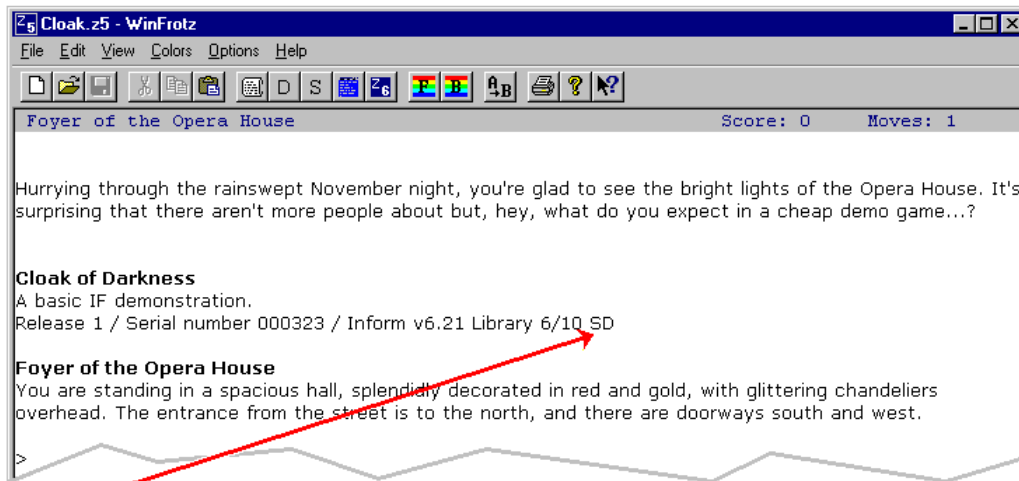


[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ **Switches** ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

In this segment, we'll be looking at switches which control how your game is compiled, and which tell you more about it.

Checking and debug -- S and D

Firstly, the **S** and **D** switches, which control the inclusion of code into your game to provide respectively Strict error checking and some Debug capabilities. Even if you don't use the Debug commands, you'll sure as hell be grateful that Strict checking discovers many of your coding snafus before your players do. Remember the message you see at the start of every game?



These letters remind you that this game has been compiled with the Strict and Debug switches set

Almost uniquely among all the switches, **S** and **D** are highlighted in the game's introductory banner (the other one so favoured is X, showing that the [Infix debugger](#) is present). Seeing them shown like that here is a bit odd, since we haven't so far said anything about setting them. And indeed, Inform's behaviour here is -- controversial, perhaps -- since it turns **S** on (and with it **D**) by default. That's what you get out of the box; anything else, you need to ask for. This means that if you *don't* want Strict checking, you need explicitly to switch it off; this uses the `~S` construct (think of removing an attribute from an object), which is one of the instances which doesn't work with **Switches** ... ;.

If you want this combination	Use this on the command line
Debug + Strict (the default)	-DS (but why bother?)
Debug alone	-D~S
Strict alone	You can't (Strict always implies Debug)
neither	~~S

So, the moral here seems to be: while you're developing and testing your game, you'll want both Strict error checking and Debug active, so accept the default setting. When you're ready to release the game, you *don't* want Debug active, which means turning off Strict also, which means **-~S** on the command line.

Strict checking includes things like ensuring that you don't try to address array entries which are before the start or after the end of the array. It also prevents you from performing operations on object number 0 -- more easily done than you'd imagine, for example after evaluating `parent(myObject)` when `myObject` currently *has* no parent. Before the advent of Strict checking, errors like this were such a common cause of games mysteriously crashing that they become somewhat fancifully known as [Vile Zero Errors from Hell](#). Nowadays, thankfully, they're a lot less prevalent.

How memory is used -- o and z

While both **S** and **D** affected the code being compiled into your game, the next set of switches just control what the compiler tells you. The **o** switch provides a summary of offsets (in hexadecimal) to important divisions within the Z-machine:

```
Offsets in story file:
00042 Synonyms      0010a Defaults      00188 Objects      0031e Properties
00823 Variables     0126f Parse table   01971 Actions      01a65 Preactions
01a67 Adjectives    01a67 Dictionary    02608 Code         11d78 Strings
```

The **z** switch provides the same sort of information, but in more detail, and more readably:

```
Dynamic |-----+ 00000
memory  |   header   |
+-----+ 00040
| abbreviations |
+-----+ 00042
| abbreviations table |
```

	header extension	00102
	property defaults	0010a
	objects	00188
	object short names, common prop values	0031e
	class numbers table	005fd
	symbol names table	00609
	indiv prop values	0080b
	global variables	00823
	arrays	00a03
	grammar table	0126f
Readable memory	actions	01971
	parsing routines	01a65
	adjectives	01a67
	dictionary	01a67
	Z-code	02608
Above readable memory	strings	11d78
		13750

The **s** switch, mentioned in the previous segment, also fits loosely into this category. While most of this stuff is fairly obscure, you need to be aware of the Z-machine's limits, especially if you're writing a large game. Remember that the top of readable memory (where the dictionary ends) can't be higher than \$0FFFF, and that if the top of virtual memory (where the strings end) goes over \$3FFFF, you'll need to switch from the default Version 5 to Version 8 with the **vs8** switch.

There's an additional compiler limitation: the section labeled "Z-code" cannot be larger than \$40000 (262,144) bytes. If the Z-code region becomes too large, your game will no longer compile due to compiler back-patch errors. (The example game shown here has a very small Z-code section of 63,355 bytes: \$11D78 - \$2608 = \$F770). Authors who note that their game is getting close to this limit are in for a difficult balancing act.

How things are numbered -- j and n

The **j** switch shows the number allocated to each object as the game is being compiled:

```

6 "compass"
7 "north wall"
8 "south wall"
9 "east wall"
10 "west wall"
11 "northeast wall"
12 "northwest wall"
13 "southeast wall"
14 "southwest wall"
15 "ceiling"
16 "floor"
17 "outside"
18 "inside"
19 "(darkness object)"
20 "(self object)"
21 "(Inform Parser)"
22 "(Inform Library)"
23 "(with no short name)"
24 "Foyer of the Opera House"
25 "Cloakroom"
26 "small brass hook"
27 "Foyer bar"
28 "velvet cloak"
29 "scrawled message"

```

The **n** switch performs the same function, at much greater length, for attributes, properties and actions:

```

Attr 00    animate
Attr 01    absent
Attr 02    clothing
⋮
Attr 28    female
Attr 29    neuter
Attr 30    pluralname
Prop 04    LA before
Prop 05    LA after
Prop 06    LA life

```

```

o
o
o
Prop 48 L parse_name
Prop 49 L articles
Prop 50 L inside_description
Action 'LetGo' is numbered 118
Action 'Receive' is numbered 120
Action 'ThrownAt' is numbered 122
Action 'Order' is numbered 124
Action 'TheSame' is numbered 126
Action 'PluralFound' is numbered 128
Action 'ListMiscellany' is numbered 130
Action 'Miscellany' is numbered 132
Action 'Prompt' is numbered 134
Action 'NotUnderstood' is numbered 136
Action 'Pronouns' is numbered 0
Action 'Quit' is numbered 1
Action 'Restart' is numbered 2
Action 'Restore' is numbered 3
o
o
o
Action 'Empty' is numbered 118
Action 'InvTall' is numbered 119
Action 'InvWide' is numbered 120
Action 'GoIn' is numbered 121

```

The first block of actions -- **LetGo** through **NotUnderstood** --- are the Fake actions; then come the real actions (but I'm not sure I understand why the numbers overlap).

The code that's generated -- a and t

If you should want to see a listing of the Z-code program itself, use the **a** switch (very long) or the **t** switch (very very long):

```

0 +00000 [ Main__
0 +00001 call_vs long_0 -> TEMP1
0 +00006 quit

33 +00008 [ Main

33 +00009 <*>; call_vn long_10 long_139 long_140
33 +00011 <*>; rtrue

146 +00014 [ LanguageToInformese

147 +00015 <*>; rtrue

160 +00018 [ LanguageContraction text

161 +00019 <*>; call_vs long_33 text short_0 -> sp
161 +00020 pull TEMP1
161 +00023 je TEMP1 short_97 short_101 short_105 to L1 if TRUE
161 +0002b je TEMP1 short_111 short_117 short_65 to L1 if TRUE
161 +00033 je TEMP1 short_69 short_73 short_79 to L1 if TRUE
161 +0003b je TEMP1 short_85 to L0 if FALSE
161 +00040 .L1
162 +00040 <*>; rtrue
162 +00041 .L0
163 +00041 <*>; rfalse
o
o
o
381 +0e5a0 [ PrintRank

381 +0e5a1 <*>; print_ret "."

387 +0e5a4 [ ParseNoun obj

387 +0e5a5 <*>; store obj obj
387 +0e5a8 <*>; ret long_65535

0 +00000 [ Main__
0 +00001 call_vs long_0 -> TEMP1
0 +00006 quit
ba

33 +00008 [ Main

33 +00009 <*>; call_vn long_10 long_139 long_140
33 +00011 <*>; rtrue
b0

146 +00014 [ LanguageToInformese

```



```

147 +00015 <*>;gt; rtrue          b0
160 +00018 [ LanguageContraction text
161 +00019 <*>;gt; call_vs      long_33 text short_0 -> sp
                                e0 27 00 21 01 00 00
161 +00020 pull                TEMP1
                                e9 7f ff
161 +00023 je                  TEMP1 short_97 short_101 short_105 to L1 if TRUE
                                c1 95 ff 61 65 69 80 01
161 +0002b je                  TEMP1 short_111 short_117 short_65 to L1 if TRUE
                                c1 95 ff 6f 75 41 80 01
161 +00033 je                  TEMP1 short_69 short_73 short_79 to L1 if TRUE
                                c1 95 ff 45 49 4f 80 01
161 +0003b je                  TEMP1 short_85 to L0 if FALSE
                                41 ff 55 00 00
161 +00040 .L1
162 +00040 <*>;gt; rtrue          b0
162 +00041 .L0
163 +00041 <*>;gt; rfalse        b1
o
o
o
381 +0e5a0 [ PrintRank
381 +0e5a1 <*>;gt; print_ret     "."
                                b3 96 45
387 +0e5a4 [ ParseNoun obj
387 +0e5a5 <*>;gt; store         obj obj
                                2d 01 01
387 +0e5a8 <*>;gt; ret           long_65535
                                8b ff ff

```

How things are spelt -- r

Finally in this segment, don't forget the **r** switch which dumps all of the text in the game to a file **gametext.txt**. Skip over the first 1000 or so lines containing stuff from the library, and you'll come to your strings:

```

You are standing in a spacious hall, splendidly decorated in red and gold,...
You've only just arrived, and besides, the weather outside seems to be...
Foyer of the Opera House
The walls of this small room were clearly once lined with hooks, though now...
Cloakroom
It's just a small brass hook,
with a cloak hanging on it.
screwed to the wall.
small brass hook
The bar, much rougher than you'd have guessed after the opulence of the foyer...
Blundering around in the dark isn't a good idea!
In the dark? You could easily disturb something!
Foyer bar
A handsome cloak, of velvet trimmed with satin, and slightly spattered with...
This isn't the best place to leave a smart cloak lying around.
velvet cloak
The message, neatly marked in the sawdust, reads...
The message has been carelessly trampled, making it difficult to read. You...
scrawled message
^^Hurrying through the rainswept November night, you're glad to see the bright..
You have lost

```

Before you release a game, copy this stuff and paste it into an editor or word processor with a spell-checker: yule bee sir prized watt ewe fined.

Next, we glance at some other ways of inspecting a game file from the outside.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

InfLight -- Inform debugging




[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ **Tools** ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

Before we look at what we can discover while running a game, we should mention other techniques for examining Inform files from the outside.

infodump

Part of the Ztools package, **infodump** reads a compiled Inform program file and reports on the contents of its **low memory** -- the various data tables such as the header, the globals and the dictionary. Like the compiler, it's not a Windows application, so you can either type in an MS-DOS window, or use a batch file (say **CLOAKINFO.BAT**):

```
infodump -f Cloak.z5 >CloakInfo.txt
```

Double-click on the **CLOAKINFO.BAT** icon  to dump the information to a file (it's really too long to display in the window). Various command-line switches control the output: **-i** for the header (that's the default anyhow), **-m** for the memory map, **-a** for any abbreviations, **-o** for the objects, **-t** for the object tree, **-g** for grammar information, **-d** for the dictionary, or **-f** for all of them.

```
Story file is Cloak.z5

**** Story file header ****

Z-code version:          5
Interpreter flags:      None
Release number:         1
Size of resident memory: 2608
Start PC:                2609
Dictionary address:     1a67
Object table address:   010a
Global variables address: 0823
Size of dynamic memory: 126f
Game flags:             Supports undo
Serial number:          000326
Abbreviations address:  0042
File size:              13750
Checksum:               37a7
Terminating keys address: 126e
  Keys used:
Header extension address: 0102
Inform Version:         6.21
Header extension length: 0003
Unicode table address:  0000

**** Story file map ****

Base  End  Size  Description
 0    3f    40  Story file header
40    41     2  Abbreviation data
42    101   c0  Abbreviation pointer table
102   109     8  Header extension table
10a   31d   214 Object table
31e   5fc   2df  Property data
5fd   822   226
823   a02   1e0  Global variables
a03   126e  86c
126f  1342   d4  Grammar pointer table
1343  1970   62e  Grammar data
1971  1a64    f4  Action routine table
1a65  1a66     2
1a67  2607   ba1  Dictionary
2608  1374f  11148 Paged memory

**** Abbreviations ****

[ 0] " "
o
o
o
[95] " "

**** Objects ****

Object count = 29

1. Attributes: None
   Parent object: 0 Sibling object: 0 Child object: 5
   Property address: 031e
   Description: "Class"
   Properties:
o
o
o
29. Attributes: 17
   Parent object: 27 Sibling object: 0 Child object: 0
```

```

Property address: 05e1
Description: "scrawled message"
Properties:
  [41] 00 00
  [35] 42 a7
  [ 1] 1f 87 21 e2 1d bc

**** Object tree ****

[ 1] "Class"
. [ 5] "CompassDirection"
[ 2] "Object"
o
o
o
[ 27] "Foyer bar"
. [ 29] "scrawled message"
[ 28] "velvet cloak"

**** Parse tables ****

Verb entries = 106

255. 1 entry, verb = "score"
    [00 0c 0f] "score"

254. 2 entries, verb = "full", synonyms = "fullscore"
    [00 0d 0f] "full"
    [00 0d 42 22 06 0f] "full score"
o
o
o
151. 2 entries, verb = "dig"
    [00 38 01 00 00 0f] "dig noun"
    [00 38 01 00 00 42 25 d2 01 00 01 0f] "dig noun with held"

150. 1 entry, verb = "hang"
    [00 12 01 00 01 42 20 68 01 00 00 0f] "hang held on noun"

**** Verb action routines ****

Action table entries = 122

action# action-routine "verb..."

0. 9714 "nouns"
1. d66c "die"
o
o
o
120. da78 "i wide"
121. ed08 "cross"
    "in"

**** Prepositions ****

Table entries = 0

**** Dictionary ****

Word separators = ". , ""
Word count = 330, word size = 9


[ 1] . [ 2] , [ 3] a [ 4] about
[ 5] abstract [ 6] actions [ 7] adjust [ 8] again
o
o
o
[ 325] with [ 326] wreck [ 327] x [ 328] y
[ 329] yes [ 330] z

```

txd

txd is the other major component in the Ztools package. It too reads a compiled Inform program file and reports on the contents of its **high memory** -- the routines and strings. Again, it's not a Windows application, so you can either type in an MS-DOS window, or use a batch file (say **CLOAKTXD.BAT**):

```
txd -agn Cloak.z5 >CloakTxd.txt
```

Double-click on the **CLOAKTXD.BAT** icon  to dump the information to a file (it's far far too long to display in the window):

```

[Start of code at 2608]

Main routine 2608, 0 locals

2609: call_vs      2610 -> gef
260e: quit

Routine 2610, 0 locals

```

```

2611: call_vn      10c88 #0016 #0049
2619: rtrue

Routine 261c, 0 locals

261d: rtrue

Routine 2620, 1 local

2621: call_vs      11bd8 local0 #00 -> sp
2628: pull        gef
262b: je          gef #61 #65 #69 2644
2632: je          gef #6f #75 #41 2644
2639: je          gef #45 #49 #4f 2644
2640: je          gef #55 ~2645
2644: rtrue
2645: rfalse
◊
◊
◊
Routine 11d5c, 1 local

11d5d: call_2s     11794 local0 -> sp
11d63: je          sp #03 11d6e
11d67: call_2s     11310 #23 -> sp
11d6d: ret_popped
11d6e: print_paddr local0
11d70: rtrue

Routine 11d74, 2 locals

11d75: rfalse

[End of code at 11d76]

[Start of text at 11d78]

11d78: S001 "Cloak of Darkness"
11d88: S002 "^A basic IF demonstration.^^"
11da0: S003 "991113"
◊
◊
◊
13730: S410 "StorageForShortName"
13740: S411 "task_scores"
13748: S412 "task_done"

[End of text at 13750]

[End of file]

```

Definitely not an easy or friendly read, but this (or the compiler's **a** and **t** switches) are just about the only way of discovering the structure of the Z-machine's high memory -- as we said before, you've no direct access to this area from a running program.

disinform

Just for completeness, we should mention **disinform** (or **uninform**, or **deinform**; the nomenclature seems confused), which maps the output from **infodump** and **txd** into something closer to Inform syntax. Development of this seems to have stopped some way short of completeness; what you get looks like this:

```

![[Main routine]
![[_____]];

[rn_2608 ;
gef=rn_2610();
@quit      ;
];

![[_____]];

[rn_2610 ;
rn_10c88(16,49);
rtrue;
];

![[_____]];

[rn_261c ;
rtrue;
];

![[_____]];

[rn_2620 local0 ;
gef=rn_11bd8(local0,00);
if (gef==61 or 65 or 69)&&(gef==6f or 75 or 41)&&(gef==45 or 49 or 4f) ||(gef==55)
{
rtrue;
}
}
rfalse;

```

```
];
;
;
;
[rn_11d5c local0 ;
if (rn_11794(local0)~=03)
{
return (rn_11310(23));
}
print (string) local0;
rtrue;
];

![_];

[rn_11d74 local0 local1 ;
rfalse;
];
```

Realistically, this isn't a useful tool as it currently stands.

Having inspected a game from the outside, the obvious next step to to take it for a test drive.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ **[Tools](#)** ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

InfLight -- Inform debugging



[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

If you've compiled with the **D** switch -- which is the default -- then you can use the debugging commands built into the Inform library. In this segment, we'll talk about commands which present information on the static state of your game, and allow some small modifications.

SHOWOBJ

SHOWOBJ displays the properties and attributes of an object (whether or not that object is currently in scope). So you could say:

```
Foyer of the Opera House

>SHOWOBJ HOOK
Object "small brass hook" (26) in "Cloakroom"
  has scenery supporter
  with name 'small' 'brass' 'hook' 'peg',
  description [...] (16978),
```

If you don't specify an object, **SHOWOBJ** displays the properties and attributes of your current location. This is useful because rooms often don't have a **name** property, thus preventing you from referring directly to them:

```
>SHOWOBJ
Object "Foyer of the Opera House" (24)
  has light visited
  with n_to "You've only just arrived, and besides, the weather outside seems
  to be getting worse." (20391),
  s_to Foyer bar (27),
  w_to Cloakroom (25),
  description "You are standing in a spacious hall, splendidly decorated in
  red and gold, with glittering chandeliers overhead. The entrance from the
  street is to the north, and there are doorways south and west." (20357),
```

Even so, you'll sometimes find it useful to be able to refer to rooms by name when fighting tricky problems; adding a **name** property to a room is easy enough:

```
Object foyer "Foyer of the Opera House"
  with name 'foyer',
  description
    "You are standing in a spacious hall..."
```

TREE and SCOPE

The **TREE** command displays the names and numbers of all objects (generally a very long list), using indentation to reflect ownership:

```
>TREE
compass (6)
  the north wall
  the south wall
  the east wall
  the west wall
  the northeast wall
  the northwest wall
  the southeast wall
  the southwest wall
  the ceiling
  the floor
  the outside
  the inside
Darkness (19)
(Inform Parser) (21)
(Inform Library) (22)
(LibraryMessages) (23)
Foyer of the Opera House (24)
  yourself
    a velvet cloak (being worn)
Cloakroom (25)
  a small brass hook
Foyer bar (27)
  a scrawled message
```

More often, you're interested in only one part of the tree:

```
>TREE ME
yourself (20) (in Foyer of the Opera House 24)
```

```
    a velvet cloak (being worn)
>TREE CLOAK
a velvet cloak (28) (in yourself 20)
```

SCOPE lists those objects currently in scope:

```
>SCOPE
1: yourself (20)
2: a velvet cloak (28)
```

GOTO and GONEAR

To move the player directly from place to place, you can **GOTO** a room number (given by **TREE** or **LIST ROOMS**), or you can **GONEAR** an object -- something in that room, or the room itself if you've given it a **name** property:

```
>GOTO 27

Darkness
It is pitch dark, and you can't see a thing.

>GONEAR HOOK

Cloakroom
The walls of this small room were clearly once lined with hooks, though now only
one remains. The exit is a door to the east.
```

ABSTRACT and PURLOIN

To move objects around, use **ABSTRACT** or **PURLOIN**. These ignore scope and any attributes which might affect (or be affected by) a normal **TAKE** command, so you can re-arrange the scenery if you wish:

```
>ABSTRACT CLOAK TO HOOK
[Abstracted.]

>PURLOIN HOOK
[Purloined.]

>INV
You are carrying:
  a small brass hook
  a velvet cloak (being worn)

>SHOWOBJ HOOK
Object "small brass hook" (26) in "yourself"
  has moved scenery supporter
  with name 'small' 'brass' 'hook' 'peg',
  description [...] (18222),
```

LIST

SHOWOBJ, which works only on one object at once, is significantly enhanced by the **LIST** command provided by the **ObjLstr.h** package. Using it, you can execute **SHOWOBJ** on a whole range of objects. For example:

```
>LIST DIRS
Object "Foyer of the Opera House" (24)
  has light visited
  with n_to "You've only just arrived, and besides, the weather outside seems to
  be getting worse." (20391),
  s_to Foyer bar (27),
  w_to Cloakroom (25),
  -----
Object "Cloakroom" (25)
  has light
  with e_to Foyer of the Opera House (24),
  -----
Object "Foyer bar" (27)
  with n_to Foyer of the Opera House (24),
  -----
```

Especially flexibly, you can select objects according to their properties, attributes, or both:

```
>LIST HAS SCENERY
Object "small brass hook" (26) in "Cloakroom"
  has scenery supporter
  with name 'small' 'brass' 'hook' 'peg',
  description [...] (18133),
```

```

-----
Object "scrawled message" (29) in "Foyer bar"
  has scenery
  with name 'message' 'sawdust' 'floor',
        description [...] (18221),
        number 0,
-----

>LIST WITH BEFORE AFTER
Object "yourself" (20) in "Foyer of the Opera House"
  has animate concealed proper transparent
  with before NULL,
        after NULL,
        life NULL,
        orders 0,
        description [...] (4955),
        describe NULL,
        number 0,
        time_out NULL,
        each_turn NULL,
        capacity 100,
        short_name [...] (4952),
        parse_name 0,
-----

Object "velvet cloak" (28) in "yourself"
  has clothing general moved worn
  with name 'handsome' 'dark' 'black' 'velvet' 'satin' 'cloak',
        before [...] (18202),
        after [...] (18218),
        description "A handsome cloak, of velvet trimmed with satin, and slightly
                    spattered with raindrops. Its blackness is so deep that it almost seems
                    to suck light from the room." (20479),
-----

```

CHECKOUT

While **LIST** tells you about all of your rooms, it doesn't perform any verification. For that, try my **CheckOut.h** package, which implements a single debugging command **CHECKOUT**. This looks at the paths between rooms -- well, those that are easy to analyse anyhow -- and reports on one-way and asymmetric paths (those where turning by 180 degrees doesn't bring you back to your starting point). Sometimes those are intentional, sometimes the result of errors in creating or editing the game's map. There's nothing to demonstrate here -- the example game is entirely symmetric -- but running it within the standard example **Advent.inf** produces a sizeable crop of reports:

```

At End Of Road
You are standing at the end of a road before a small brick building. Around you
is a forest. A small stream flows out of the building and down a gully.

>CHECKOUT
At End Of Road->North->In Forest: no apparent return
At Hill In Road->South->In Forest: no apparent return
In Forest->North->itself
In Forest->South->itself
In Forest->West->itself
In Forest->North->At End Of Road: no apparent return
In Forest->East->In A Valley: no apparent return
In Forest->South->In Forest: no apparent return
In Forest->West->In A Valley: no apparent return
In Forest->Down->In A Valley: no apparent return
At Slit In Streambed->East->In Forest: no apparent return
At Slit In Streambed->West->In Forest: no apparent return
Outside Gate->East->In Forest: no apparent return
Outside Gate->South->In Forest: no apparent return
Outside Gate->West->In Forest: no apparent return
At West End of Hall of Mists->Up->Maze: one asymmetric return
At West End of Hall of Mists->South->Maze: one asymmetric return
oo
oo
At Fork in Path->NorthEast->At Junction With Warm Walls: one asymmetric return
At Junction With Warm Walls->South->At Fork in Path: one asymmetric return
SW End of Repository->Down->Outside Gate: no apparent return

```

DUMP

An alternative way of finding out the current state of your game is the **DUMP** command provided by my **Dump.h** package. Working at a somewhat lower level than the other commands in this segment, **DUMP** focuses on the Z-machine's storage allocation. You can inspect the game's memory map (including where the values can be found):

```

>DUMP
----- DYNAMIC (read/write)
00000: Header:          $0000
00040: String pool:       $0040
00042: 32 Low strings:   $0018-->0
00082: 64 Abbreviations: ($0018-->0)+64
00102: Header extension: $0036-->0
00000: Alphabet:         $0034-->0
00000: UniCode:         (( $0036-->0)+6)-->0

```



```

0010A: Property defaults:      $000A-->0
00188: Object tree:          ($000A-->0)+126
0031E: Common properties:    #cpv__start
005F4: Class numbers:       #classes_tables
00600: Identifier names:     #identifiers_table
0081C: Array names:         #array_names_offset
0081E: Individual properties: #ipv__start
00836: Global variables:    $000C-->0
00A16: Arrays:             #array__start
013DF: Terminating chars:  $002E-->0
----- STATIC (read only)
013E0: Grammars:           $000E-->0
01BAB: Action pointers:     #actions_table
01CBB: Preactions (not used): #preactions_table
01CBD: Adjectives (not used): #adjectives_table
01CBD: Dictionary:         $0008-->0
----- HIGH (no read/write) $0004-->0
02978: First unreadable byte: #readable_memory_offset
02979: Initial PC:         $0006-->0
02978: Zcode:             #code_offset (packed)
14D9C: Static strings:     #strings_offset (packed)
17268: Top of memory       $001A-->0 (packed)

```

a range of addresses:

```

>DUMP 0 $3F
00000: 059D0001 29782979 1CBD010A 083613E0
00010: 00123030 30333239 00425C9A 76620646
00020: 1E500050 001E0101 00000000 0F0E13DF
00030: 00000100 00000102 00000000 362E3231

```

or any of the Z-machine tables:

```

>DUMP HEADER
----- Header
00000: 05 Z-machine version
00001: 9D Interpreter flags: timed_keys/fixed_pitch/italic/bold/colour
00002: 0001 Game release
00004: 02978 ==>High memory
00006: 02979 ==>Initial PC
00008: 01CBD ==>Dictionary
0000A: 0010A ==>Objects
0000C: 00836 ==>Global variables
0000E: 013E0 ==>Static memory
00010: 0012 Game flags: undo/fixed_pitch/
00012: "000329" Game serial
00018: 00042 ==>Abbreviations
0001A: 17268 Length
0001C: 7662 Checksum
0001E: 06 Interpreter: IBM PC
0001F: "F" Interpreter version
00020: 1E Screen height (lines)
00021: 50 Screen width (chars)
00022: 0050 Screen height (units)
00024: 001E Screen width (units)
00026: 01 Font width (units)
00027: 01 Font height (units)
00028: 0000 Routines offset / 8
0002A: 0000 Strings offset / 8
0002C: 0F Background colour
0002D: 0E Foreground colour
0002E: 013DF ==>Terminating chars
00030: 0000 Pixels to stream 3
00032: 01 00 Interpreter conformance
00034: 00000 ==>Alphabet
00036: 00102 ==>Header extension
00038: 0000 -
0003A: 0000 -
0003C: "6.21" Inform version

```

To be honest, you rarely need to be this concerned with actual storage locations, but just occasionally it can be the only way to discover what's actually happening.

Moving on along, we next look at dynamic debugging -- watching it all go wrong.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ **Static** ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

Roger Firth's IF pages

InfLight -- Inform debugging



[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ **Dynamic** ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

Although inspecting things from the game's input line is informative, you frequently need to observe how values change during execution. The general term for this process is **tracing**.

Tracing calls to a routine

One useful technique built into Inform is to trace calls to a routine. As it happens, there are almost no standalone routines in our example game, so to illustrate the point, let's transform an embedded routine into a standalone one:

```
Object hook "small brass hook" cloakroom
  with name 'small' 'brass' 'hook' 'peg',
  description [; DescribeHook(self); ],
  has scenery supporter;

[ DescribeHook obj;
  print "It's just a small brass hook, ";
  if (cloak in obj) "with a cloak hanging on it.";
  "screwed to the wall.";
  ];
```

So far, this change has no effect on the game's behaviour. But if we modify the declaration of **DescribeHook** to have an asterisk * as its first parameter, then at run-time we see:

```
[ DescribeHook * obj;
  print "It's just a small brass hook, ";
  o
  o
  o
```

```
>X HOOK
[ DescribeHook(obj = 26) ]
It's just a small brass hook, screwed to the wall.
```

Alternatively, you can turn tracing on for *all* routines -- without having to edit in any asterisks -- by compiling with the **g** switch. For this simple example game it works reasonably well, but in most cases you get far too much output; you're better selectively tracing where you need to:

```
>X HOOK
[ hook.description() ]
[ DescribeHook(obj = 26) ]
It's just a small brass hook, screwed to the wall.

>HANG CLOAK ON HOOK
[ cloak.before() ]
(first taking it off)

[ cloak.before() ]
[ cloak.after() ]
You take off the velvet cloak.
[ cloak.after() ]
You put the velvet cloak on the small brass hook.

[Your score has just gone up by one point.]
```

Temporary PRINT statements

The asterisk trick works only on calls to standalone routines. A more general technique, albeit with more typing involved, is simply to add a temporary **PRINT** statement; this can both clarify which path the program is taking, and show you appropriate values:

```
[ DescribeHook obj;
  print "**** ", obj, " ****^";
  print "It's just a small brass hook, ";
  o
  o
  o
```

```
>X HOOK
**** 26 ****
It's just a small brass hook, screwed to the wall.
```

The `Dump.h` package includes a tool `IS` which does this more elegantly:

```
[ DescribeHook obj;
  is(obj);
  print "It's just a small brass hook, ";
  o
  o
  o

>X HOOK
*****
** This number: 26 $001A $$000000000011010
** Might be an object:    small brass hook
** Might be an attribute: workflag
** Might be a property:  react_before
** Might be an action:   VagueGo
** Might be a read/write memory address containing: -28778 $8F96 $$1000111110010110
*****
It's just a small brass hook, screwed to the wall.
```

You can also use `IS` as a command:

```
>IS $2AF3
** This number: 10995 $2AF3 $$0010101011110011
** Might be a read-only memory address containing: 13972 $3694 $$0011011010010100
** which is dictionary word: 'hook'.
** Might be a packed routine address $0ABCC.
```

If you're a bit unsure what exactly a given number represents at some point, `IS` can be very convenient.

You might also be interested in the `DebugLib.h` package which does a similar job of printing object information. It gives less detail, but does provide three routines -- `Print_Crit()`, `Print_Warn()` and `Print_Trace()` -- enabling you to select a level of debug output.

```
[ DescribeHook obj;
  Print_Trace(obj, "in DescribeHook()");
  print "It's just a small brass hook, ";
  o
  o
  o

>X HOOK
small brass hook trace: In DescribeHook()
It's just a small brass hook, screwed to the wall.
```

ACTIONS, ROUTINES and TIMERS

An informative commentary is provided by the `ACTIONS` command:

```
>ACTIONS ON
[Action listing on.]

>WEST
[ Action Go with noun 10 (west wall) ]
[Moving yourself to Cloakroom]

Cloakroom
The walls of this small room were clearly once lined with hooks, though now only
one remains. The exit is a door to the east.
[Giving Cloakroom visited]

>X HOOK
[ Action Examine with noun 26 (small brass hook) ]
It's just a small brass hook, screwed to the wall.

>HANG CLOAK ON HOOK
[ Action PutOn with noun 28 (velvet cloak) and second 26 (small brass hook) ]
[Giving Foyer bar light]
[Giving velvet cloak -general]
(first taking it off)

[ Action Disrobe with noun 28 (velvet cloak) (from < > statement) ]
[Giving velvet cloak ~worn]
You take off the velvet cloak.
[Moving velvet cloak to small brass hook]
You put the velvet cloak on the small brass hook.

[Your score has just gone up by one point.]
```

The similar trace from `ROUTINES` is lengthier, with more evidence of the Library itself:

```

>ROUTINES ON
[Message listing on.]
[ "(Inform Parser)".parse_input(4275) ]

>WEST
[ "(Inform Library)".begin_action(27,10,0,0) ]
[ "yourself".orders() ]
[Moving yourself to Cloakroom]

Cloakroom
The walls of this small room were clearly once lined with hooks, though now only
one remains. The exit is a door to the east.
[Giving Cloakroom visited]
[ "(Inform Library)".end_turn_sequence() ]
[ "(Inform Parser)".parse_input(4275) ]

>X HOOK
[ "(Inform Library)".begin_action(32,26,0,0) ]
[ "yourself".orders() ]
[ "small brass hook".description() ]
It's just a small brass hook, screwed to the wall.
[ "(Inform Library)".end_turn_sequence() ]
[ "(Inform Parser)".parse_input(4275) ]

>HANG CLOAK ON HOOK
[ "(Inform Library)".begin_action(18,28,26,0) ]
[ "yourself".orders() ]
[ "velvet cloak".before() ]
[Giving Foyer bar light]
[Giving velvet cloak ~general]
(first taking it off)

[ "(Inform Library)".begin_action(41,28,0,1) ]
[ "yourself".orders() ]
[ "velvet cloak".before() ]
[Giving velvet cloak ~worn]
[ "velvet cloak".after() ]
You take off the velvet cloak.
[Moving velvet cloak to small brass hook]
[ "velvet cloak".after() ]
You put the velvet cloak on the small brass hook.
[ "(Inform Library)".end_turn_sequence() ]

[Your score has just gone up by one point.]
[ "(Inform Parser)".parse_input(4275) ]

```

TIMERS does a similar job for daemons and timers, but there aren't any of those in the example game.

TRACE

Finally, you can wheel out the heavyweight **TRACE** command, which offers five levels of detail about the Parser's operation. Here's what you see with the first three levels:

```

>TRACE 1
[Parser tracing set to level 1.]

>X HOOK
[Parsing for the verb 'x' (1 lines)]
[line 0 * noun -> Examine]
[line successfully parsed]
It's just a small brass hook, screwed to the wall.

>HANG CLOAK ON HOOK
[Parsing for the verb 'hang' (1 lines)]
[line 0 * held 'on' noun -> PutOn]
[line successfully parsed]
(first taking it off)

You take off the velvet cloak.
You put the velvet cloak on the small brass hook.

[Your score has just gone up by one point.]

>TRACE 2
[Parser tracing set to level 2.]

>X HOOK
[ "x" x / "hook" hook ]
[Parsing for the verb 'x' (1 lines)]

[line 0 * noun -> Examine]
[line 0 token 1 word 2 : noun]
[line 0 token 2 word 3 : END]
[line successfully parsed]
It's just a small brass hook, screwed to the wall.

```

```
>HANG CLOAK ON HOOK
[ "hang" hang / "cloak" cloak / "on" on / "hook" hook ]
[Parsing for the verb 'hang' (1 lines)]

[line 0 * held 'on' noun -> PutOn]
[line 0 token 1 word 2 : held]
[line 0 token 2 word 3 : 'on']
[line 0 token 3 word 4 : noun]
[line 0 token 4 word 5 : END]
[Line successfully parsed]
(first taking it off)

You take off the velvet cloak.
You put the velvet cloak on the small brass hook.

[Your score has just gone up by one point.]
```

```
>TRACE 3
[Parser tracing set to level 3.]
```

```
>X HOOK
[ "x" x / "hook" hook ]
[Parsing for the verb 'x' (1 lines)]

[line 0 * noun -> Examine]
[line 0 token 1 word 2 : noun]
[Object list from word 2]
[Calling NounDomain on location and actor]
[ND returned the small brass hook]
[token resulted in success]
[line 0 token 2 word 3 : END]
[Line successfully parsed]
It's just a small brass hook, screwed to the wall.
```

```
>HANG CLOAK ON HOOK
[ "hang" hang / "cloak" cloak / "on" on / "hook" hook ]
[Parsing for the verb 'hang' (1 lines)]

[line 0 * held 'on' noun -> PutOn]
[line 0 token 1 word 2 : held]
[Object list from word 2]
[token resulted in success]
[line 0 token 2 word 3 : 'on']
[token resulted in success]
[line 0 token 3 word 4 : noun]
[Object list from word 4]
[Calling NounDomain on location and actor]
[ND returned the small brass hook]
[token resulted in success]
[line 0 token 4 word 5 : END]
[Line successfully parsed]
(first taking it off)

You take off the velvet cloak.
You put the velvet cloak on the small brass hook.

[Your score has just gone up by one point.]
```

As you can see, **TRACE** comes into its own if you've invented a new verb grammar which obstinately fails to work properly. Most of the time, you don't need to know this stuff.

Introduced in the most recent Inform upgrade, the Infix debugger is our next topic.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ **Dynamic** ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

Roger Firth's IF pages

InfLight -- Inform debugging



[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

While the **D** switch provides a basic set of debug commands, the **X** switch gives you **Infix**, a much more powerful debugging environment. With this switch set, you'll see **SX** displayed in the game's banner, rather than the usual **SD**.

All Infix commands are introduced by a semicolon ; as their first character (just ; on its own outputs some instructions). If the next character is a space, Infix treats what follows as an expression to be evaluated (we'll come back to that in a minute). Otherwise, Infix recognises these commands: **;EXAMINE**, **;GIVE**, **;MOVE**, **;REMOVE**, **;WATCH** and **<**.

;EXAMINE or **;X**

You can **;EXAMINE** many aspects of a game; the nice bit is, you've now got access to the internal names from the program:

```
>;X STORY
; Constant Story == 31179

>;X 31179
; The number 31179 == $79cb

>;X FOYER
Object "Foyer of the Opera House" (24)
  has light visited
  with n_to "You've only just arrived, and besides, the weather outside seems to
  be getting worse." (32105),
  s_to Foyer bar (27),
  w_to Cloakroom (25),
  description "You are standing in a spacious hall, splendidly decorated in
  red and gold, with glittering chandeliers overhead. The entrance from the
  street is to the north, and there are doorways south and west." (32071),

>;X N_TO
; Property n_to (numbered 7)
Provided by: Foyer of the Opera House (24), Foyer bar (27)

>;X LIGHT
; Attribute light (numbered 10)
Each of these "has light": Foyer of the Opera House (24), Cloakroom (25)
```

Here's some information on verb grammars:

```
>;X 'HANG'
; Dictionary word 'hang' (address 10823): verb
Verb 'hang'
  * held 'on' noun -> PutOn

>;X PUTONSUB
; Routine PutOnSub (number 163, packed address 18710)

>;X ##PUTON
; Action PutOn (numbered 18)
'discard' * multiexcept 'on' / 'onto' noun -> PutOn
'drop' * multiexcept 'on' / 'onto' noun -> PutOn
'hang' * held 'on' noun -> PutOn
'put' * multiexcept 'on' / 'onto' noun -> PutOn
'throw' * multiexcept 'on' / 'onto' noun -> PutOn
```

;GIVE

Use **;GIVE** to set and clear object attributes.

```
>;GIVE BAR LIGHT
; give (the Foyer bar) light

>SOUTH

Foyer bar
The bar, much rougher than you'd have guessed after the opulence of the foyer to
the north, is completely empty. There seems to be some sort of message scrawled
in the sawdust on the floor.
```

;MOVE and **;REMOVE**

;MOVE and **;REMOVE** also work just like the equivalent program commands; as with **ABSTRACT**, you can bend your own rules:

```

>DROP CLOAK
This isn't the best place to leave a smart cloak lying around.

>;MOVE CLOAK TO LOCATION
; move (the velvet cloak) to (the Foyer of the Opera House)

>;GIVE CLOAK ~WORN
; give (the velvet cloak) ~worn

>LOOK

Foyer of the Opera House
You are standing in a spacious hall, splendidly decorated in red and gold, with
glittering chandeliers overhead. The entrance from the street is to the north,
and there are doorways south and west.

You can see a velvet cloak here.

>REMOVE CLOAK
; remove (the velvet cloak)

```

;WATCH or ;W

The `;WATCH` command notifies you when something happens to a watched object; you can also watch routines:

```

>;W CLOAK
; Watching object "velvet cloak" (28).

>REMOVE CLOAK
[ "velvet cloak".before() ]
[ cloak.before() ]
[Giving velvet cloak ~worn]
[ "velvet cloak".after() ]
[ cloak.after() ]
You take off the velvet cloak.

>HANG IT ON HOOK
[ "velvet cloak".before() ]
[ cloak.before() ]
[Giving velvet cloak ~general]
[Moving velvet cloak to small brass hook]
[ "velvet cloak".after() ]
[ cloak.after() ]
You put the velvet cloak on the small brass hook.

[Your score has just gone up by one point.]

```

;<

The last command that we'll mention -- `;<` -- invokes an action, even one that would otherwise be intercepted:

```

>DROP CLOAK
This isn't the best place to leave a smart cloak lying around.

>;< DROP CLOAK
; <Drop (the velvet cloak)>
(first taking the velvet cloak off)
You take off the velvet cloak.
Dropped.

```

Note that in the first line of input, "DROP" is a verb which normally calls the Drop action, but the cloak's **before** routine prevents this happening. On the second input line, "DROP" is the actual action, which then happens unimpeded.

Expressions

Finally, Infix can evaluate expressions and assign values to certain things.

```

>; INITIALISE
; == 22478

>; INITIALISE()

Hurrying through the rainswept November night, you're glad to see the bright
lights of the Opera House. It's surprising that there aren't more people about
but, hey, what do you expect in a cheap demo game...?

; == 1

>; 'CLOAK'
; == 10193

```

```
>; SCORE = SCORE + 1  
; == 1
```

[Your score has just gone up by one point.]

Supposedly, you can also send messages to objects, but so far the syntax for that escapes me.

Next, a quick look at an interpreter which places special emphasis on the debugging process.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ **Infix** ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

Roger Firth's IF pages

InfLight -- Inform debugging



[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

Nitfol is a Z-machine interpreter with built-in debugging support. In addition to the standard compiled game file, it uses a debugging information file **GAMEINFO.DBG**; use the compiler's **k** switch to request that this file be created:

```
infrmw32 -k-D-S Cloak | more
```

Because nitfol is a cross-platform interpreter, it doesn't take much advantage of any presentational bells and whistles which your environment may offer; all you get on a PC is a pretty plain window:

```
G nitfol
Cloakroom                               Score: 0   Moves: 3

Hurrying through the rainswept November night, you're glad to see the bright lights of the Opera House. It's surprising that there aren't more people about but, hey, what do you expect in a cheap demo game...?

Cloak of Darkness
A basic IF demonstration.
Release 1 / Serial number 000402 / Inform v6.21 Library 6/10

Foyer of the Opera House
You are standing in a spacious hall, splendidly decorated in red and gold, with glittering chandeliers overhead. The entrance from the street is to the north, and there are doorways south and west.

>WEST

Cloakroom
The walls of this small room were clearly once lined with hooks, though now only one remains. The exit is a door to the east.

>x hook
It's just a small brass hook, screwed to the wall.

>/symbol-file gameinfo.dbg
/help
Help is available on the following commands:
'info breakpoints' 'quit' 'show language' 'condition' 'restore' 'break' 'stepi' 'restart' 'object-tree' 'disable display' 'select-frame' 'alias'
'down-silently' 'frame' 'give' 'set' 'print' 'up' '#' 'continue' 'dumpmem' 'display' 'undo' 'move' 'up-silently' 'show copying' 'recording off' 'jump'
'recording on' 'alias' 'globals' 'backtrace' 'finish' 'find' 'down' 'ignore' 'nexti' 'replay off' 'help' 'redo' 'enable' 'until' 'replay' 'unalias'
'remove' 'info sources' 'delete' 'symbol-file' 'automap' 'show warranty' 'disable' 'undisplay' 'step' 'enable display' 'info source' 'next'
```

The forward slash **/** introduces a debugging command; here it's being used to direct nitfol to the compiler's information file, and then to list all of the available debugging commands.

Sadly, nitfol assumes a familiarity with the UNIX **gdb** debugger which I don't possess; also, it seems to crash rather too readily. I hope to revisit this segment armed with strengthened knowledge and/or software; in the short term, I feel this is a tool to be treated with caution.

Better late than never, here's a quick tutorial on numbers and addresses.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

InfLight -- Inform debugging



[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ **Numbers** ▶ [Hints](#)

In the Z-machine, everything works by numbers. By that, I mean that the multitude of internal cross-references within even the smallest game are all stored as numeric values, either as an index (object number 6, action number 32, property number 99, ...) or as an address (a routine, a string, a variable, etc). Index values tend to be small, and are often held in a single byte (or even in just part of a byte). An address, on the other hand, is always a two-byte number pointing to the memory location where the item is stored (possibly an unwise generalisation, but I think it's true enough for our purposes).

Sixteens bits -- two bytes -- give a range of addresses from 0 to 65535 (hexadecimal \$0000 to \$FFFF), far too small to encompass all of the items in a typical game. There needs to be a way of extending the address space (while keeping the address values themselves within the two-byte limit), and so the Z-machine uses two modes: **byte addressing** and **packed addressing**.

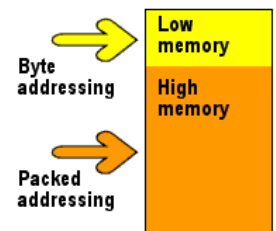
Byte and packed addresses

A byte address is a 16-bit number which points directly to any byte in the 0-64K address space. In a Version 5 game, a packed address is a 16-bit number which if multiplied by four (giving an 18-bit number with zero in the lowest two bits) then points to every fourth byte in a 0-256K address space. In a Version 8 game, a packed address is a 16-bit number which if multiplied by eight (giving an 19-bit number with zero in the lowest three bits) then points to every eighth byte in a 0-512K address space. For example, a byte address of \$3210 explicitly points to the byte at address \$3210, whereas a V5 packed address of \$3210 implicitly points to the byte at address \$0C840, and a V8 packed address of \$3210 implicitly points to the byte at address \$19080.

The packed address multiplier of x4 or x8 is the only significant difference between Version 5 and Version 8 games. In the rest of this discussion, we'll keep things simple by illustrating only Version 5 addressing (noting in passing that Version 6 and 7 games use a complex variant on Version 5 addressing that is definitely best forgotten).

Things would get pretty chaotic if the two address modes became totally mixed up (though they do become fairly confusing, as we'll see at the end). Therefore, Inform divides the Z-machine address space cleanly into **low memory** (which uses just byte addressing) and **high memory** (which uses just packed addressing).

Furthermore, high memory is used for just two things: all the routines and then all the strings; everything else is stored in low memory. This low/high distinction is quite important; because a packed address can point only to every fourth byte, lots of space would be wasted if small items were held in high memory. However, since a typical string consumes rather more than four bytes, and a typical routine occupies a lot more than four bytes, the fact that up to three bytes are unused at the end of each one becomes fairly unimportant.



Divisions of memory

Low memory begins at address 0, and extends upwards until all items apart from the routines and the strings have been allocated an address. High memory begins immediately after low memory ends, and extends upwards until the routines and then the strings have also been accommodated. Everything in low memory must be reachable using a byte address up to \$FFFF, so that low memory can't be larger than \$FFFF (64K) bytes. Items in high memory must be reachable using a packed address up to \$FFFF, so that the combination of low memory plus high memory can't be larger than \$3FFFF (256K) bytes.

There's a further sub-division, of much smaller significance. Low memory is split into **dynamic memory** (where a program can both read and write values) and **static memory** (where reading is permitted but writing isn't). Dynamic memory lies in the bottom portion of the low memory address space, and contains objects, variables and arrays -- things which commonly get updated during the course of a game. Static memory lies in the top portion of the low memory address space, and contains stuff like the verb grammars and the dictionary, which are continually read but never changed as a game progresses. The distinction, dating back to an era when machines had little physical memory, is nowadays of minimal importance.

Access to memory

We've said that a program can read and write values in low memory. This happens all the time, generally without you being aware of the details (for example, the Inform statement `score=score+1`; fetches the `score` global variable from dynamic memory, increments it and stores the result), though you can occasionally glimpse the mechanics (as in the statement `width=0->33`; for fetching the current screen width). So you might well ask: what can a program do with a high memory address? The answer is short: run it or print it. That's it.

If `my_var` is a variable of some sort containing an address in high memory, then your effective choices are `my_var()`; (which runs it, if a routine) or `print (string) my_var`; (which outputs it, if a string). The point is, your program can access high memory only on the Z-machine's terms. Routine and string creation is the sole prerogative of the compiler, and your program can't even *read* the contents of high memory, let alone update it. So, no fancy tricks with self-modifying code, and no string manipulation features like concatenation or subset extraction or..., well, anything but print.

What's in a number?

One final issue. Those who've been following closely may have spotted a dilemma; if everything is held as a number, how can the Z-machine distinguish between byte addresses, packed addresses and things that aren't addresses at all? Another short answer: it can't, other than from the context in which the number's being used. For sure, the compiler knows the difference, but by the time the Z-machine gets to see it, the value \$3210 might be the byte address of a variable, the packed address of a string, or the number of islands in the Pacific. So, if we take this somewhat artificial room:

```

Object test_room "Test room"
has light
with name 'test' 'room',
      n_to central_lobby,
      description [;
        print "The room is full of odd devices.^";
        print "Self=", (name) self, "^";
        print "Prop1=", (object) self.prop1, "^";
        print "Prop2=", (address) self.prop2, "^";
        print "Prop3="; self.prop3();
        print "Prop4=", (string) self.prop4, "^";
      ],
      prop1 ticket,           ! an object
      prop2 'ticket',        ! a dictionary word
      prop3 [; "ticket of bright yellow."; ], ! a routine
      prop4 "bright yellow ticket."; ! a string

```

and change the four properties to read as follows (children, don't try this at home):

```

      prop1 26,               ! an object
      prop2 10228,          ! a dictionary word
      prop3 18247,         ! a routine
      prop4 22346;         ! a string

```

then at run-time the two rooms behave identically:

>LOOK

```

Test room
The room is full of odd devices.
Self=Test room
Prop1=yellow ticket
Prop2=ticket
Prop3=ticket of bright yellow.
Prop4=bright yellow ticket.

```

There's no reason to do this, of course, but it illustrates the point about knowing what the numbers signify.

Finally, a few fragments of the bleedin' obvious.

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ **Numbers** ▶ [Hints](#)

InfLight -- Inform debugging



[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)

To end with, here are a few nuggets of advice on how to solve (or better, avoid) problems. In no particular order...

- Without becoming paranoid, program defensively. Assume things may go wrong, probably on account of your mistakes and lapses. So, test for sensible replies, check a property is supplied before using it, always have a **default** clause in your **switch** statements...
- Evolve a programming style that you're comfortable with -- naming, layout, indentation, etc -- and then stick with it. You need to be able to interpret your own code fluidly: Write Once, Read Many applies.
- So do it; read the damn stuff. Work through your own logic, sceptically, trying to convince yourself that you've covered all the angles.
- Indentation's a wonderful device, but it means zilch to the compiler. Use braces to enforce your statement grouping, and be wary of single-statement blocks (especially when adding to existing code). Similarly, take care to match **else** statements to the appropriate **if**. For example, you may have written the program on the left, but the compiler sees the logic on the right:

```
if (conditionA)
  if (conditionB)
    statement1;
else
  statement2; statement3;

if (conditionA)
  if (conditionB)
    statement1;
  else
    statement2;
statement3;
```

- Remember that, although they're all numbers, the object **cloak**, the dictionary word '**cloak**' and the string "**cloak**" are quite distinct. Check *carefully* what syntax is required, and don't try comparing apples with oranges.
- Inform often uses routine return values of true or false to control what happens next. Are you concentrating? you'll get a return value of **true** from: **print_ret...**, "**string**", **rtrue**, **return**, **return non_zero_value**, and **]** at the end of a standalone routine. You'll get a return value of **false** from: **rfalse**, **return zero_value**, and **]** at the end of an embedded (property) routine.
- Proceed in small steps -- write something, test it, write a bit more, test a bit more. This is true in spades if you're changing some existing code whose precise details are no longer fresh in your mind. It's a common and embarrassing trap to assume things all went wrong with your latest change; it could just as easily have been that trivial one-liner half an hour ago...
- Be very grateful for Strict checking in the compiler, and leave it on throughout the development time. Then, when you turn it off prior to releasing your game, test everything again. Different code is generated, and you shouldn't assume the compiler is totally glitch-free.
- Inform's good, but it ain't perfect. Before you get desperate, check the [Inform patch site](#), just in case you're victim of a Well Known Problem.
- The newsgroup **rec.arts.int-fiction** is populated by kindly helpful souls who like nothing better than to crack a clearly-defined and well-bounded problem. So, if you plan to ask for help there, be sure to give full details; a tiny cut-down example which demonstrates your difficulty is especially helpful.
- But, before you post your message, pause awhile. It's amazing how many folks seem to solve their own problem within an hour or so of posting to the newsgroup. Compose your message -- it always helps to clarify your thoughts -- then hang onto it briefly before you send. After all, there's only so much goodwill to go round; don't cry "wolf" too often.

And that's all we've got time for...

[Intro](#) ▶ [Process](#) ▶ [Compiling](#) ▶ [Switches](#) ▶ [Tools](#) ▶ [Static](#) ▶ [Dynamic](#) ▶ [Infix](#) ▶ [Nitfol](#) ▶ [Numbers](#) ▶ [Hints](#)