PunyInform

PunyInform

An Inform library for writing small and fast text adventures.

Version 5.6, 25 August 2024

PunyInform was conceived, designed and coded by Johan Berntsson and Fredrik Ramsberg.

Additional coding by Pablo Martinez and Tomas Öberg. Includes code from the Inform 6 standard library, by Graham Nelson.

Thanks to Stefan Vogt, Jason Compton, John Wilson, Hugo Labrande, Richard Fairweather, Adam Sommerfield, auraes, Hannesss and Garry Francis for issue reporting, advice, testing, code contributions and promotion. Thanks to David Kinder and Andrew Plotkin for helping out with compiler issues and sharing their deep knowledge of the compiler. Huge thanks to Graham Nelson for creating the Inform 6 compiler and library in the first place.

Contents

Introduction

PunyInform is a library written in Inform 6 which allows authors to create text adventures/interactive fiction using the Z-machine virtual machine.

The main goal of PunyInform is to allow for games which are fast and have a small memory footprint. This should make the games run well on older architectures, such as the 8-bit computers of the 1980s. Our main target is to make it suitable for games on the Commodore 64 using Ozmoo (https://github.com/johanberntsson/ozmoo)

PunyInform is based on the Inform 6 standard library, developed by Graham Nelson. In this document DM4 refers to the *Inform Designer's Manual, 4th edition*, which is availble online at: http://www.inform-fiction.org/manual/html/index.html

A PunyInform game can be compiled to Z-code version 3, 5 or 8 (z3, z5 or z8), but not Glulx. To compile games using PunyInform, you need the official Inform compiler maintained by David Kinder, at https://github.com/DavidKinder/Inform6. Binaries can be found at if-archive. Please note that PunyInform uses features that were introduced in Inform v6.36 and using earlier versions of the compiler will cause errors. We recommend using the latest stable release of the compiler.

Apart from this manual, you need to know about two more documents, both available in the documentation folder of the PunyInform distribution:

- Game Author's Guide - the first chapters of this document (all but the chapter on Optimization) is required reading for anyone writing a PunyInform game. The optimization chapter is recommended if you want to provide the best user experience possible.
- PunyInform QuickRef document - A neat index to everything included in the core PunyInform library (i.e. not the extensions).

Comparison with the Inform 6 Standard Library

A game written in PunyInform is very similar to a game written with the Inform 6 standard library. However, there are some differences that are documented in this section.

Getting Started

To compile a game, unpack the files, place the Inform 6 compiler binary (Get the source or an executable at http://www.ifarchive.org/indexes/if-archiveXinfocomXcompilersXinform6.html) in the base directory, and type e.g. `inform6 +lib -v3 -s -e library_of_horror.inf` (type `inform6 -h2` for an explanation of all commandline switches). PunyInform requires at least version v6.36 of the compiler.

You can use the minimal.inf file, supplied with PunyInform, as a starting point for developing a new game.

The general pattern of a PunyInform game is:

```
Constant INITIAL_LOCATION_VALUE = ...;

! Change "score" to "time" if you want time on the statusline
Constant STATUSLINE_SCORE; Statusline score;

! define library constants here

Include "globals.h";

! define your own global variables here

! add extension routines and other library customizations here

Include "puny.h";
```

```
! add normal game code here

[Initialise;
  "Welcome to the game!";
];
```

All library constants, including `Story, Headline, MAX_SCORE, OBJECT_SCORE, ROOM_SCORE, NUMBER_TASKS, TASKS_PROVIDED, AMUSING_PROVIDED, MAX_CARRIED` and `SACK_OBJECT` should be defined before including globals.h, if needed. The roles of these constants are documented in DM4.

Library customization, such as supplying an entry point routine such as `PrintTask`, goes between the globals.h and puny.h inclusions.

After including puny.h, you add game code and an `Initialise` routine, as in other Inform games.

Actions

PunyInform has most of the actions that the standard library has, but they are divided into four sets + the undo action. The basic set of actions is part of the core library. Then there is a set of normal actions which can be enabled by defining the constant OPTIONAL_EXTENDED_VERBSET and a set of meta actions which can be enabled by defining OPTIONAL_EXTENDED_METAVERBS. OPTIONAL_PROVIDE_UNDO provides the undo verb (z5 and z8 only). Finally, just as in the standard library, there is a set of debug verbs, which can be enabled by defining the symbol DEBUG.

The basic actions

Normal actions: Answer, Ask, AskTo, AskFor, Attack, Close, Consult, Cut, Dig, Disrobe, Drink, Drop, Eat, Enter, Examine, Exit, Fill, GetOff, Give, Go, Inv, Insert, Jump, JumpOver, Listen, Lock, Look, Open, Pull, Push, PushDir, PutOn, Remove, Rub, Search, **Shout** *, **ShoutAt*** , Show, Smell, SwitchOff, SwitchOn, Take, Tie, Tell, ThrowAt, Touch, Transfer, Turn, Unlock, Wait, Wear.

Meta actions: Again, FullScore, LookModeNormal, LookModeLong, LookModeShort, NotifyOn, NotifyOff, Oops, OopsCorrection, Quit, Restart, Restore, Save, Score, Version.

[*] The Shout action is not present in the standard library. Shout is triggered if the player types “shout”, “scream” or “yell”, or “shout hello sailor” (with `consult_from` and `consult_words` pointing out the words the player wants to shout, or 0 if no words were given.)

[*] The ShoutAt action is not present in the standard library. ShoutAt is triggered if the player types “shout at postman”.

OPTIONAL_EXTENDED_VERBSET actions

Normal actions: Blow, Burn, Buy, Empty, EmptyT, GoIn, Kiss, Mild, No, Pray, Set, SetTo, Sing, Sleep, Sorry, Strong, Squeeze, Swim, Swing, Taste, Think, Wake, WakeOther, Wave, WaveHands, Yes.

OPTIONAL_EXTENDED_METAVERBS actions

Meta actions: CommandsOn, CommandsOff, CommandsRead, Places, Objects, ScriptOn, ScriptOff, Verify.

Note: Places and Objects can be disabled by defining the constant NO_PLACES.

DEBUG actions

Meta actions: ActionsOn, ActionsOff, **Debug** *, GoNear, Pronouns, Purloin, RandomSeed, RoutinesOn, RoutinesOff, Scope, TimersOn, TimersOff, Tree.

[*] The Debug action is not present in the standard library. It is used to check which objects may need the reactive attribute, when using `OPTIONAL_MANUAL_REACTIVE`.

UNDO

Defining `OPTIONAL_PROVIDE_UNDO` activates the 'undo' command, which can be used on interpreters that support undo. This will also make the library mention undo as an option if the player dies/loses. If DEATH_MENTION_UNDO is defined, the option to use undo is always mentioned when the game ends, even if the player has won. Note that undo is not supported in z3 games.

Implicit actions

PunyInform has two routines for performing implicit actions:

- `ImplicitGrabIfNotHeld(object)` - Take something if it isn't already held by the player.
- `ImplicitDisrobeIfWorn(object)` - Take off something if it is currently worn by the player.

An action routine which wants to make sure the player is holding an item, e.g. `noun` will typically do this somewhere near the start of the action routine:

```
if(ImplicitGrabIfNotHeld(noun)) rtrue;
```

If, at the end of the `ImplicitGrabIfNotHeld` routine, the item is held by the player, the routine returns false, otherwise true. If it returns true, a message has also been printed. It's also legal to call this routine with a parameter value of 0, in which case it always returns false.

To take off an item if it's worn, use the `ImplicitDisrobeIfWorn` routine, which functions in much the same way.

Just like the Inform 6 parser, the PunyInform parser will try to take an object that matches a `held` token but isn't currently in the player's possession. However, and this differs from the Inform 6 parser, it will not try to take an object which has the `static`, `scenery` or `animate` attribute, with the exception of `animate` objects which have been made takeable through the use of a `DisallowTakeAnimate` routine. This allows `before` routines to react to things like "EAT WHALE" and "ATTACK TROLL WITH STEAMROLLER". This feature of the parser requires a little caution, since this means that when `before` routines are run and even when the action routine is run, an object that was matched by a `held` token is not guaranteed to be in the player's possession. As a game author, you typically solve this be adding a line like this at the start of any action routine which uses a `held` token: `if(ImplicitGrabIfNotHeld(noun)) rtrue;`.

The game can turn off all implicit actions by setting `no_implicit_actions = true`.

Detecting implicit actions

The library can generate implicit actions, as described above. This can even happen in two different circumstances - when the parser encounters a HELD token and the matched object isn't held, or when an action has been issued and the action routine decides to try to pick up an object or take off a worn object. Additionally, user code can also generate actions, e.g. `<<Take Ball>>;`. PunyInform has a mechanism to help decide if the action currently being processed was issued by the parser (based on player input), if it's an implicit action issued by the parser, or an action issued by an action routine, a before rule etc.

This is done with five global variables:

```
action_to_be
input_action
input_noun
input_second
input_direction
```

- While the parser is working out what the player means to say, `input_action` is set to a value < 0.
- `action_to_be` is initially set to -1, when the parser starts its work.
- When the parser has come far enough to consider different grammar lines, `action_to_be` is set to the currently considered action (a value $>= 0$).
- When the parser is ready to issue the action the player seems to intend, `action_to_be` is once again set to a value < 0, and `input_action`, `input_noun`, `input_second` and `input_direction` are set to the values of `action`, `noun`, `second` and `selected_direction` respectively.

E.g. to create a before rule which allows an object to be picked up, but not if it's an implicit action:

```
before [;
  Take:
    if(input_action ~= action)
      "You're just not sure if it's safe to pick it up...";
],
```

Writing action routines

PunyInform uses a few tricks to make action routines more compact:

- You can have the library print a message after the action routine ends, by returning a message#. I.e. `PrintMsg(MSG_DROP_DROPPED); return;` is equivalent to `return MSG_DROP_DROPPED;`.
- You can have the library run the after routines after the action routine ends, by setting `run_after_routines_msg` to a message number or 1. If you set it to 1, it means "run after routines, don't print a default message." If you set it to a message number, it means "run after routines, print this message if they return false." If this message needs an argument, you give the argument in `run_after_routines_arg_1`.

Note that you can do both of the above in the same action routine. The message number you return is printed first, then the after routines are run. Also note that a message number can be a constant holding a string, or a message number > 1000 that you invented and which your own `LibraryMessages` routine will print.

When writing your own action routine, you are responsible for setting `scope_modified = true;` whenever you do something which may change what's in scope, regardless if you're using `OPTIONAL_MANUAL_SCOPE` or not.

Calling action routines

Due to the special handling described in Writing action routines, you are adviced never to call action routines directly, e.g. instead of doing `LookSub();`, you're better off doing `<Look>;`, as this will ensure before- and after-routines are run as per the normal rules, and any messages the routine means to print will be printed.

Moving Objects

If you move an object into the player's possession in your own code, like `move Screwdriver to player;`, you should always do `update_moved = true;`. This is necessary to have PunyInform update the `moved` attribute and, when applicable,

score points for the object being picked up. If you issue a ##Take action or some action that does an implicit take, you don't need to bother with this.

If you have defined `OPTIONAL_MANUAL_SCOPE` and you do something in code which may affect scope (Essentially what the player can see), you need to do `scope_modified = true;`. As a rule of thumb, do this whenever you move an object using `move` or `remove` or you give or remove any of the attributes `open`, `transparent` and `light` for an object (as this may make previously invisible objects visible to the player, and vice versa). Library routines like PlayerTo and all action routines do this whenever they do any of this.

Note that when writing your own action routines, you are responsible for setting `scope_modified = true;` whenever you do something which may change what's in scope, regardless if you're using `OPTIONAL_MANUAL_SCOPE` or not.

Floating Objects

Just like the Inform 6 library, PunyInform expects the found_in property to hold either a routine or an array of values. If it's an array of values, each value can be either:

- a room, meaning the floating object is present if `location` is this room
- a non-room object, meaning that the floating object is present if the parent of the non-room object is `location`
- a class, meaning that the floating object is present if `location` belongs to this class

The modern-day Inform 6 library supports all of these cases, but DM4 doesn't mention the option of using a class.

Also, just like the Inform 6 library, giving a floating object the attribute `absent` overrides all of this and makes the object non-present in all rooms.

Animate Objects

If you want to allow the player to take certain animate objects, you can add an entry point routine called DisallowTakeAnimate which accepts one parameter and have it return false if the parameter value is one of those objects. I.e:

```
[ DisallowTakeAnimate p_object;
    if (p_object ofclass Puppy) rfalse;
];
```

Articles

PunyInform, unlike the Inform standard library, will not figure out when an object should have the indefinite article "an". You need to specify it using the

`article` property every time it should be “an”. Example:

```
Object Umbrella "umbrella"
    with
        name 'umbrella',
        article "an";
```

Another difference is that PunyInform doesn't support the `articles` (note the s) property. This was only added to the Inform library because it's useful for some languages other than English.

Plural

PunyInform can handle objects which share names, as long as they don't share all names, e.g. a blue book and a red book are okay. PunyInform does not offer support for indistinguishable objects, but it's possible to customize it enough to provide limited support even for this - see the file howto/indistinguishable.inf in the distribution. The library supports `pluralname` and the plural marking on dictionary words with the `//p` suffix.

For example

```
Object -> RedBook "red book"
    with name 'red' 'book' 'books//p';

Object -> BlueBook "blue book"
    with name 'blue' 'book' 'books//p';
```

can be used like

```
> take book
Do you mean the red book, or the blue book? > red

Taken.

> drop book
Dropped.

> take books
red book: Taken.
blue book: Taken.
```

Inventory

PunyInform supports two ways of printing player inventory:

- Wide mode: all objects are printed as one long line of text
- Tall mode: each object is printed on a line of of its own, indented

Inventory is in wide mode by default. This is controlled with the global variable `inventory_style`. Set it to 0 for tall mode or 1 for wide mode. If you want to allow the player to control this with commands "inventory wide" and "inventory tall", define the constant `OPTIONAL_FLEXIBLE_INVENTORY`. Unlike the standard library, switching modes is handled by the `##Inv` action - there are no actions just for switching mode.

list_together and LanguageNumber

The library has optional support for listing similar objects together when listing objects in room descriptions, inventory etc. To use this, define `OPTIONAL_LIST_TOGETHER`. If you want the library to print numbers using words, i.e. "three books", you must also define `OPTIONAL_LANGUAGE_NUMBER`. This defines the library routine `LanguageNumber(n)`, which you can also use in your own code. If you have defined `DIALECT_US`, LanguageNumber will take it into account.

The library defines `list_together` as a common property. If you'd rather make it an individual property, put `Property individual list_together;` in your source, before including "globals.h". Using an individual property is slower.

Look at the file howto/list_together.inf for some examples on how `list_together` can be used. The library doesn't really support indistiguishable objects, but can be made to support it, with some limitations. See file howto/indistinguishable.inf for an example of how to do this.

Note: `OPTIONAL_LANGUAGE_NUMBER` and `OPTIONAL_ALLOW_WRITTEN_NUMBERS` have some overlap. If you define both, the cost is 32 bytes less than what is stated in the chapter Customizing the library.

Printing the Contents of an Object

The standard library provides the routine `WriteListFrom()`. PunyInform provides `PrintContents()` instead. While not quite as versatile as WriteListFrom, it's meant to be easy to use, easy to remember how to use, and powerful enough to cover the needs for most situations. This is how it works:

```
PrintContents(p_first_text, p_obj, p_style);
Print what's in/on p_obj recursively, OR decide if there are any
contents to be printed and if so, if they should be prefixed by
"is" or "are".

p_first_text:
     A string containing a message to be printed before the first item
     in/on p_obj. Can also be 0 to not print a text, or a routine,
     which will then be called with p_obj as an argument, or 1 to not
```

```
     print anything but investigate contents (see Return value).
p_obj:
     The container/supporter/person whose contents we want to list.
p_style:
     Add together 0 or more of the following values:
     ISARE_BIT: Print "is " or "are " before the first object.
     NEWLINE_BIT: Print a newline before each object and indent objects.
     WORKFLAG_BIT: On the top level, only print objects that have the
       workflag attribute set.

Return value:
     true if any items were printed, false if not.
     If p_first_text == 1, instead return the plural-factor for
     the printable contents (0 = No contents, 1 = Contains a
     single, non-pluralname object, 2 = Contains a pluralname
     object or multiple objects.)
```

Some examples of typical usage:

```
if(PrintContents("On the table you can see ", OakTable)) print ".";
else print "There's nothing on the table.";

if(PrintContents("On the table ", OakTable, ISARE_BIT)) print ".";
```

There is also a routine called `PrintContentsFromR`, which does the same thing as `PrintContents`, with two exceptions: You point it to the first child object of the container or supporter, and it doesn't take a style parameter - instead it retains the last used style and indentation. You typically want to use it when you're printing a list within a list, e.g. if you're writing an `invent` routine. Specification:

```
PrintContentsFromR(p_first_text, p_obj);
Print p_obj and its siblings, listing the contents of each object
recursively.

p_first_text:
     A string containing a message to be printed before the first item
     in/on p_obj. Can also be 0 to not print a text, or a routine,
     which will then be called with p_obj as an argument.
p_obj:
     The first object to print.

Return value:
     true if any items were printed, false if not.
```

Sample use:

```
Object Box "box"
    with
```

```
        invent [;
            if (inventory_stage == 2) {
                if (c_style & NEWLINE_BIT) {
                    new_line;
                    PrintContentsFromR(0, child(self));
                } else if (PrintContentsFromR(" (which contains ", child(self)))
                    print ")";
                rtrue;
            }
        ],
...
```

Giving Orders

When giving an order to an non-player character, for example "john, give me the hammer", then only single, sufficiently specified objects are allowed in the command. Commands such as "john, take all" or "john, take the ball" (when there is more than one ball available, which would normally prompt a disambiguation question) are not allowed and will cause the game to ask you to be more specific.

Changing the Player

A PunyInform game by default defines an object `selfobj` representing the player, and sets the `player` variable to this object, but the game author can define their own customized player object, or even switch player objects mid-game. By setting `CUSTOM_PLAYER_OBJECT` to an object, the game will set `player` to this object instead, and never create the default player object. For an example on how to switch `player` mid-game, see the change_player.inf file in the howto folder.

Capacity

The `capacity` property doesn't have a default value in PunyInform. To check the capacity of an object, call `ObjectCapacity(object)`. If the object has a value, it's returned (unless the value is a routine, in which case it is executed and the return value is returned). If the object doesn't have a value for capacity, the value `DEFAULT_CAPACITY` is returned. This value is 100, unless you have defined it to be something else.

Doors

Simple doors

PunyInform supports defining doors just the way it's described in DM4. In addition to this, PunyInform supports a more convenient way to define a door. To enable it, define the constant `OPTIONAL_SIMPLE_DOORS`. This means two new mechanims come into play:

- If the door object has an array value for `found_in` with exactly two locations, it can leave out the `door_to` property. Instead, the library will assume that if the player is in the first location in the `found_in` array the door leads to the second location in the array and vice versa.
- If the door object has an array value for `found_in` with exactly two locations, it can also have an array value for `door_dir`. The first entry in `door_dir` corresponds to the first entry in `found_in` and the second entry in `door_dir` correnspond to the second entry in `found_in`. Use parenthesis around the values to avoid compiler warnings.

In any door object, you can use either one of these mechanisms, both of them, or none.

Example of a regular door in PunyInform:

```
Object -> BlueDoor "blue door"
    with
        name 'blue' 'door',
        door_to [;
            if(self in Hallway) return Office;
            return Hallway;
        ],
        door_dir [;
            if(self in Hallway) return n_to;
            return s_to;
        ],
        found_in Hallway Office,
    has static door openable;
```

And this is how to define the same door using OPTIONAL_SIMPLE_DOORS:

```
Object -> BlueDoor "blue door"
    with
        name 'blue' 'door',
        door_dir (n_to) (s_to),
        found_in Hallway Office,
    has static door openable;
```

Note: `OPTIONAL_SIMPLE_DOORS` adds 86 bytes to the library size, but it saves 22 bytes per door which uses both of the features. So if you use these features

for at least four doors, they save space.

DoorTo and DoorDir Routines

With the inclusion of Simple Doors, it becomes harder for game authors to figure out where a certain door leads, or in which direction it lies. To help with this, the library provides two new routines:

- `DoorTo(door_object)`: returns the object number of the room on the other side of the door, or 1 if a door_to routine was encountered, and the routine returned 1, meaning the routine printed a reply saying why it's not possible to go through the door.
- `DoorDir(door_object)`: returns the direction in which the door lies.

These routines work for regular doors as well as simple doors.

The with_key Property

Just as in the standard library, you can use the `with_key` property to say which key fits the lock for a lockable object. As an alternative to specifying an object as a value, PunyInform allows you to specify a routine. The routine should return false or the object id of the key that fits the lock. When this routine is called, `second` holds the object currently being considered as a key. This can be used to allow multiple keys fit a lock. Example:

```
Object RedDoor "red door"
    with
        name 'red' 'door',
        with_key [;
            if(second == RedKey or RubyKey or SmallKey) return second;
        ],
        ...
    has static door lockable locked;
```

Note: Of course PunyInform also supports the standard way of using `with_key`, like: `with_key RedKey,`

Daemons and Timers

Property `daemon` is an alias for property `time_out`. This means you can't have a daemon and a timer on the same object. If you want both, put one of them in another object, possibly a dummy object whose only purpose is to hold the timer/daemon.

If you need your daemons/timers to execute in a certain order, you can define the constant `OPTIONAL_ORDERED_TIMERS` and then set the property timer_order to any number for some or all objects with daemons/timers. A lower number

means the daemon/timer will execute earlier. The default value is 100. Note that this number should not be changed while a daemon or timer is running.

The reactive Attribute

NOTE: This section deals with an optimization you may want to perform before releasing a game. You can safely skip it while learning PunyInform and return to it when you're close to releasing a game, or not at all.

Each turn, PunyInform needs to check if any of the objects in scope provide any of these properties:

- `react_before`
- `react_after`
- `each_turn`
- `add_to_scope`
- `parse_name` (special, read below!)

These checks are rather time consuming, and they make the game noticably slower when many objects are in scope, at least on 8-bit machines. To alleviate this problem, PunyInform has an attribute `reactive` which all objects providing any of these properties must have (except possibly `parse_name` - keep reading for details). This allows PunyInform to only check for these properties in objects which have the attribute (checking an attribute is a lot faster than checking a property), so gameplay becomes faster. By default, PunyInform figures out which objects need to have this attribute when the game starts. This process causes a delay of about 6 ms per object on a C64, so about 1.5 seconds for a full-size z3 game. If you want the game startup to be quicker and/or you want to save some bytes, you can choose to set the `reactive` attribute manually instead. To do this, define the constant `OPTIONAL_MANUAL_REACTIVE`, compile the game in debug mode and type "DEBUG REACTIVE", and you will get a list of all objects that should have the attribute. Add the `reactive` attribute to each of these objects in the code. If all objects of a certain class should have the attribute, you can of course add the attribute to the class instead.

The `parse_name` property is special. By default, objects should NOT have the `reactive` attribute just because they provide the `parse_name` property. However, if you use `parse_name` rather sparingly, you can speed up parsing by defining the constant `OPTIONAL_REACTIVE_PARSE_NAME`. Of course, if you define this constant *and* `OPTIONAL_MANUAL_REACTIVE`, you must manually give all objects providing `parse_name` the `reactive` attribute. You should only define `OPTIONAL_REACTIVE_PARSE_NAME` if about 10% of your objects or less provide `parse_name`. If you use `parse_name` for lots and lots of objects, the game will be faster without this constant.

Note: `react_before` is an alias of `u_to`, `react_after` is an alias of `d_to`, `add_to_scope` is an alias of `in_to` and `parse_name` is an alias of `out_to`. For

this reason, locations which provide `u_to`, `d_to`, `in_to` or `out_to` will be included in the DEBUG REACTIVE report as needing the `reactive` attribute. Unless these locations provide `each_turn`, they actually don't need the attribute. Giving them the attribute makes the game a tiny bit slower in these locations.

Note: The player object (AKA selfobj) provides `each_turn` and `add_to_scope`, both set to NULL or 0, to allow the game author to set it to one of their own routines. If you don't set them to anything (typically done in `Initialise`), there is no need to give the player object the `reactive` attribute. If you do, you will lose it from the debug report, but each turn of the game will also be (slightly) slower.

Note: If you define `OPTIONAL_MANUAL_REACTIVE` and you have an object which has a `react_before` routine but doesn't have the `reactive` attribute, its `react_before` routine will never be run. Same thing with `react_after`, `each_turn`, `add_to_scope` and `parse_name`.

Statusline

Unless you start replacing routines (and avoid the z3 format) a PunyInform game always shows a statusline. You can select between two different types of statusline:

- To show score and turns in the status line, put `Constant STATUSLINE_SCORE; Statusline score;` in the beginning of the source.
- To show time in the status line, put `Constant STATUSLINE_TIME; Statusline time;` in the beginning of the source, and add a call to `SetTime` in the `initialise` routine (See example below).

```
Constant STATUSLINE_TIME; Statusline time;
Include "globals.h";
Include "puny.h";
[Initialise;
    SetTime(1 * 60 + 5, 5); ! 1:05 am, each turn 5 minutes
];
```

For z3 games, the layout of the statusline is controlled by the interpreter and not the library. You can only choose between displaying time or score. For z5 and z8 games, PunyInform has its own implementation of the statusline, and it differs a bit from the standard library's implementation:

- The statusline is printed using a lot fewer instructions, making it render noticably faster on slow machines.
- When the statusline is updated (typically once per turn), the new contents are printed over the existing contents. The standard library prints spaces over the old contents first, making the statusline appear to blink on slow machines.

- The statusline adapts to different screen widths by selecting one of five different layouts to, in addition to the room name, show score and moves, only score or nothing.
- The game can remove the score from the statusline by defining `OPTIONAL_SL_NO_SCORE`. Also, this happens automatically if the game defines `NO_SCORE`.
- The game can remove the number of moves from the statusline by defining `OPTIONAL_SL_NO_MOVES`.

Scoring

Scoring works as in DM4, but it divided into three parts:

- Basic scoring using the `score` variable and the `MAX_SCORE` constant
- Scoring using the `scored` attribute and the `OBJECT_SCORE` and `ROOM_SCORE` constants, enabled by `OPTIONAL_SCORED`
- The fullscore verb, enabling the player to see a breakdown of the score, enabled by `OPTIONAL_FULL_SCORE`

If the game has added points in code, like `score = score + 10;`, The "full score" output will also show a line summarizing these points as "Performing noteworthy actions" (this doesn't happen in the standard library).

If `OPTIONAL_FULL_SCORE` is enabled, you can also choose to define `TASKS_PROVIDED` to enable support for tasks. Read DM4 for details on how to use this.

If constant `NO_SCORE` is defined, no scoring mechanism is included. If it's a z3 game and the statusline is of type "score", a score is displayed on the statusline. The interpreter is responsible for showing the statusline in z3, and it can't be made not to show a value for score, unless it is set to show the time on the statusline. If the constant `NO_SCORE` is defined, the value of this constant (default is 0) is shown as the score on the statusline. Also see chapter Statusline

Moves / Turns

The global variable `turns` has the value -1 until the first turn starts. This allows user-supplied code being invoked during initialization and the first room description to act differently based on this.

Library Messages and Customization

All system messages that can be replaced can be found in the file messages.h.

PunyInform uses two forms of library messages: static strings and complex messages. A typical static string is `"Taken."`. If a message has parts that vary,

if the same message should be shared by several different message identifiers, or a newline should NOT be printed after the message, the message needs to be a complex message. A complex message has its own piece of code to print it.

Each message is defined as either a static string or a complex message in messages.h. You replace a message by defining constants and possibly a `LibraryMessages` routine before the inclusion of puny.h.

NOTE: A static string message can be replaced by a static string or a complex message, but a complex message can only be replaced by a complex message.

To replace a message with a static string, define a constant with the same name as the message identifier and give it a string value, i.e:

```
Constant MSG_INSERT_NO_ROOM "It's kinda full already, I'm afraid.";
```

To replace a message with a complex message, define a constant with the same name as the message identifier, give it a value in the range 1000-1299 and provide a `LibraryMessages` routine to handle it, i.e:

```
Constant MSG_EXAMINE_NOTHING_SPECIAL 1000;

[LibraryMessages p_msg p_arg_1 p_arg_2;
    switch(p_msg) {
    MSG_EXAMINE_NOTHING_SPECIAL:
        print_ret (The) noun, " looks perfectly normal in every way.";
    }
];
```

The `LibraryMessages` routine takes three arguments - a message identifier (`p_msg`) and two optional arguments (`p_arg_1` and `p_arg_2`) which a few messages use. The return value of this routine is unimportant. To avoid getting compiler warnings for `p_arg_1` and `p_arg_2` not being used, you may want to do `p_arg_1 = p_arg_2;` at one place in your code where it doesn't affect anything, like just before the `print_ret` in the example above.

IMPORTANT: If you have defined a constant to replace a certain error message with a complex message, you *have to* print something for this message.

Direction Handling

The Compass and the twelve direction objects, as described in DM4, are not available in PunyInform. Instead, there is a single object called `Directions` and two global variables called `selected_direction` and `selected_direction_index`. Another difference is that PunyInform lacks support for the directions NW, SW, NE and SE by default. To add support for these directions, define the constnant `OPTIONAL_FULL_DIRECTIONS`.

Whenever the player has typed a direction, noun or second is `Directions` and

`selected_direction` contains the property number for the direction the player typed. If the player didn't type a direction, `selected_direction` is 0. The name of the `Direction` object is always the currently selected direction, or "direction" if no direction is selected. As an example, let's implement a robot which will stop the player from going north or east:

```
Object Robot "Floyd"
    with
        react_before [;
            Go:
                if(selected_direction == n_to or e_to)
                    "~My mother always told me to avoid going ",
                        (name) Directions, ".~, says Floyd.";
        ],
    has animate;
```

There is also another variable, `selected_direction_index`, which can be used to look up the property number and the name of the direction:

```
! prints the property number, like 8
print direction_properties_array->selected_direction_index;
! prints the direction name, like "north"
print (string) direction_name_array-->selected_direction_index;
```

Note that `direction_properties_array` is a byte array and `direction_name_array` is a word array. The number of defined directions (8 or 12) is held in the constant `DIRECTION_COUNT`. This is useful if you're writing a library extension and want to iterate over all directions in a safe manner. Please note that the directions are stored in element 1, 2, 3 .. `DIRECTION_COUNT` in these arrays.

Fake Direction Objects.

For each direction, there is also a fake direction object: `FAKE_N_OBJ`, `FAKE_SW_OBJ`, `FAKE_OUT_OBJ` etc. If you need to generate an action in code which has a direction in it, this requires using the corresponding fake direction object, like this:

```
<<Go FAKE_N_OBJ>>;
<<PushDir Stone FAKE_NW_OBJ>>;
```

If you want to go in a direction and you know the property number for that direction, you can find the corresponding fake direction object by calling `DirPropToFakeObj()`:

```
dir_prop = ne_to; ! Or any direction you like
fake_obj = DirPropToFakeObj(dir_prop);
<<Go fake_obj>>;
```

There is also an inverse of this function, called `FakeObjToDirProp()`.

Each fake direction object is just a constant. PunyInform recognizes these constants and sets selected_direction and selected_direction_index properly.

Note: The *only* use for the fake direction objects is when issuing actions in code as outlined above.

Note that `AllowPushDir` isn't supported. See the pushdir.inf file in the howto folder for an example on how to implement pushing in PunyInform.

Disabling Directions

If you (perhaps temporarily) don't want the game to recognize the standard directions, you can set the global variable `normal_directions_enabled` to false.

Ship Directions

If you define the constant `OPTIONAL_SHIP_DIRECTIONS`, the parser will recognize 'fore' and 'f' as synonyms for north, 'aft' and 'a' as synonyms for south, 'port' and 'p' as synonyms for west, and 'starboard' and 'sb' as synonyms for east.

If you (temporarily) don't want the game to recognize ship directions, you can set the global variable `ship_directions_enabled` to false.

Look

When performing a Look action, PunyInform, just like the standard library, prints the room name, a newline and then the description of the room, held in the `description` property.

The standard library has a deprecated feature, allowing you to write the text describing the location in the `describe` property of the location object rather than the standard `description` property. This is not supported by PunyInform.

When deciding how to show objects, these are the rules that apply in PunyInform:

- If the object provides `describe`, print or run it. If it's a string, or it's a routine and it returns true, the object will not be described any further. Note that this string or routine should start by printing a newline, unless it's a routine which decides not to print anything at all.
- We will now figure out which the current description property of the object is:
  - If the object is a container or a door, it's `when_open` or `when_closed`, depending on its state.
  - If the object is a switchable object, it's `when_on` or `when_off`, depending on its state.
  - Otherwise, it's `initial`.

- If the object provides this property AND the object hasnt moved or the property is `when_off` or `when_closed`, then print a newline and run or print the string or routine held in the property.
- If, according to the above rules, nothing has yet been printed, include the object in the list of objects printed at the end.
- If `OPTIONAL_PRINT_SCENERY_CONTENTS` has been defined, print what can be seen in/on containers and supporters which have the scenery attribute.

Note: Thanks to aliasing, PunyInform uses only 27 common properties, which is 21 less than the Inform 6 library. This is necessary to support compiling to z3. However, this also means the library can't tell if an object provides `initial`, `when_on` or `when_open` - these are in fact all aliases for the same property. For this reason, the printing rules described above must be a little restrictive. In fact, the Inform Designer's Manual, 4th ed. describes rules which are equally restrictive, since Inform 6 used aliasing as well when the DM4 was released, but newer versions of the Inform 6 library are actually smarter than the DM4 says and will look at which properties are provided and act accordingly. For PunyInform, whenever you have problems getting the results you want using `when_on`, `when_open` etc, write the logic you like in a `describe` routine instead. That way you can make it work exactly the way you want.

Box Statements and Menus

The box statement is not available in version 3 games, and the usual menu extensions will not work either since version 3 games lack cursor control commands. Instead PunyInform provides an extension that approximates this functionality. See the Extensions section for more detail.

Scope

Scope in a text adventure means which objects an actor, usually the player, can currently interact with. This includes physical objects like the player object, furniture etc, but also abstract objects such as the direction object, discussion topics etc.

The standard library works out what's in scope whenever library code or user code performs an operation that needs to know what's in scope. This is a perfectly logical, straightforward way of handling scope. Unfortunately, it's also quite slow.

PunyInform on the other hand, works out what's in scope when the game starts, stores the list of objects that are in scope in an array, and keeps using that array over and over again, until something happens that may affect what's in scope, e.g. the player enters a new room, turns on the light, opens a container etc. This mechanism is an essential part of making the PunyInform library performant on 8-bit computers. The downsides are that (a) the game author has to decide at

compile time how big to make the array, i.e. how many objects can be in scope at the same time, and (b) the game author must sometimes tell the library to recalculate what's in scope.

Deciding the number of objects that can be in scope: This can be set by setting MAX_SCOPE before including globals.h, e.g. `Constant MAX_SCOPE = 70;`. The default value is 50. Imagine if the player picks up all takeable objects and places them in the location which has the most static and scenery objects (all cheap scenery objects count as a single object though). Then add the player, any clothes, body parts etc that the player always has access to, add ten, and that should be a good value. You can try it out by setting MAX_SCOPE to a really high value and using the 'scope' debug verb to see how many objects are actually in scope at any one time.

If the PunyInform library does something that may affect scope, e.g. moves the player to a new location, it sets the global variable `scope_modified` to `true`. The library will also set `scope_modified` whenever it calls an entry point routine or a property routine, e.g. a `before` routine, as such a routine *may* change what's in scope. Setting `scope_modified` is a signal that the cached copy of what's in scope is no longer valid - the next time the library needs to know what's in scope, it will have to figure it out from scratch.

If you write an action routine of your own, or a property routine for your own use, which the library is unaware of, you too need to set `scope_modified` as appropriate, e.g. if you write the action routine PaintSub, which may cause a transparent container to lose its transparency, the action routine must set `scope_modified` when this happens.

Scope Routines

The routines ScopeCeiling, LoopOverScope, ScopeWithin, TestScope and ObjectIsUntouchable are implemented as described in DM4.

You can use the `InScope(actor)` entry point routine to affect what's in scope. In this routine, you typically want to use the `scope_reason` global variable to affect scope only as needed. It works as documented in DM4, except that `scope_reason` will never have the value `TALKING_REASON`. Instead, the value `PARSING_REASON` is used even when figuring out what actors/objects you can talk to. E.g. if you want to allow the player to talk to an object called Hal when the player is on the bridge, you might add this routine before including "puny.h":

```
[ InScope p_actor;
  if(location == Bridge && scope_reason == PARSING_REASON)
    PlaceInScope(Hal);
  p_actor = 0; ! To get rid of warning that variable isn't used
  rfalse;
];
```

Keep in mind that PunyInform tries not to update scope more often than absolutely necessary. Due to this, you should never use the `scope_reason` variable in an `add_to_scope` routine. Even if you test it, and it seems to work, there are probably scenarios that you haven't tested that will make it break. The `InScope` routine is a different story - it is always consulted before deciding whether to use a cached scope copy, so you can safely use `scope_reason` in the `InScope` routine.

If you make a change in the game world which doesn't directly affect what's in scope, but you know that a certain `add_to_scope` routine needs to be run again, you can set `scope_modified = true;` to force a scope update.

The `scope_stage` global variable is supported and is used when the `scope` grammar token is used, so constructs like the code fragment below work as described in DM4.

```
Object questions "questions";
Object -> "apollo"
    with
        name 'apollo',
        description "Apollo is a Greek god.";

[ QueryTopic;
    switch (scope_stage) {
        1: rfalse;
        2: ScopeWithin(questions); rtrue;
        3: "At the moment, even the simplest questions confuse you.";
    }
];

[ QuerySub; noun.description();];
Verb 'what' * 'is'/'was' scope=QueryTopic -> Query;
```

Manual Scope (updates)

As stated above, whenever the library calls user-supplied code, e.g. a `before` routine, it has to assume that the routine may have done something that affects what's in scope, even though most of the time this isn't true.

If you want your game to be faster, you can tell the library to don't assume anything - you'll let the library know if your code does anything which may affect scope. You do this by defining the constant `OPTIONAL_MANUAL_SCOPE` before including "globals.h". Then make sure you always set `scope_modified` to `true` whenever you do something in code which can affect what's in scope. A simple rule is to do this anytime you use `move` or `remove` or you alter any of the attributes `open`, `transparent`, `light` (as this may make previously invisible objects visible to the player, and vice versa). This is already in place in the

`PlayerTo` routine as well as in the action routines for ##Open, ##Close etc. Sample usage:

```
Object Teleporter "teleporter"
    with
        name 'teleporter',
        capacity 1,
        before [ c;
            SwitchOn:
                c = child(self);
                if(c ~= 0) {
                    move c to SecretChamber;
                    scope_modified = true;
                    print_ret (The) c, " disappears!";
                }
        ],
    has container openable transparent;
```

Manual Scope Boost

If you are using manual scope updates, there is another option you can benefit from, called Manual Scope Boost. You enable it by defining the constant `OPTIONAL_MANUAL_SCOPE_BOOST`. Enabling this is a good idea if you think the player *will often be in a situation* where they try many things without changes to what's in scope, and *at least one* of the following is true: * No objects in scope provide `react_before` * No objects in scope provide `react_after` * No objects in scope provide `each_turn`

What this feature does is remember if there were any objects providing each of these properties last turn, and if scope hasn't changed, can use this information to draw the conclusion that the library doesn't need to check all objects again. This can make entering a new room maybe 1% slower, but consecutive moves in a room can become 10% faster or more.

Let's take some examples:

Game 1: The player has unlimited inventory and will often run around with 20-30 objects, sometimes in rooms with another 10-20 static objects. There's a lot to see in each room, so the player is likely to examine and try a lot of things in each room. The player object (which is always in scope) has an `each_turn` routine, but only one or two static objects use `react_before` or `react_after`. Analysis: Yay, this is an great candidate! While the `each_turn` step won't benefit from this, the `react_before` and `react_after` steps will.

Game 2: In almost every room there is an object providing `react_before`. The player will almost certainly be holding a parrot which provides `react_after` and `each_turn`. Analysis: In this case it's pointless to use manual scope boost, as all three of these steps will need to be run anyway almost all of the time.

If you want to see how often the different steps can be skipped as you play your game, define both `OPTIONAL_MANUAL_SCOPE_BOOST` and `DEBUG_MANUAL_SCOPE_BOOST`, and you'll get messages saying when these stages are run and when they are skipped.

Parser

The parser is to a large extent compatible with the parser in the standard library, for example `wn`, `NextWord()` and `NextWordStopped()` are implemented, and `noun/second/inp1/inp2/special_number/parsed_number` work the same.

General parse routines are supported with the exception of `GRP_REPARSE` which isn't supported. The reason for this is that version 3 games cannot retokenise the input from the reconstructed string.

ChooseObjectsFinal

PunyInform supports an entry point routine called `ChooseObjectsFinal`. This allows the game author to have a say at the final stage of picking an object when the player is vague. Among other things, this can be used to create a game with indistinguishable objects, see the file howto/indistinguishable.inf.

This is the flow of events in the parser when the player is to specify an object:

1. `ChooseObjects`, if present, is called for each candidate object.
2. The parser calculates its own score for each object, to say how good a match it is when considering the action, where the object is located etc.
3. A composite score is calculated for each object, where the score returned by `ChooseObjects`, if greater than 0, always trumps the score calculated by the parser. Only objects with the highest score are considered from now on.
4. If there is still more than one possible option, `ChooseObjectsFinal` gets called.

`ChooseObjectsFinal` is called with two arguments:

- `p_arr` - the address of a word array holding the objects to choose from
- `p_len` - the length of the array (this is always two or more)

At your disposal, you have two routines to assist in picking an object:

- `ChooseObjectsFinal_Pick(p_n)` - Pick this object. After calling this, you should return from the `ChooseObjectsFinal` routine without manipulating the array further.
- `ChooseObjectsFinal_Discard(p_n)` - Remove this candidate. After this, all objects that came after this object in the array (if any) have been moved up one notch, and the length of the array has been decreased by one.

Additionally, `ChooseObjectsFinal` may rearrange the objects in the array as it sees fit, e.g. swap the position of two items.

Here's a (far from perfect) sample `ChooseObjectsFinal` routine which will make the player prefer objects giving off light:

```
[ ChooseObjectsFinal p_arr p_len _i _o;
    for(_i = 0: _i < p_len: _i++) {
        _o = p_arr-->_i;
        if(p_len > 1 && _o hasnt light) {
            ChooseObjectsFinal_Discard(_i);
            _i--;
            p_len--;
        }
    }
];
```

ParseToken

The ParseToken library routine may be of limited use, but DM4 gives two examples of ways to use it. Both these examples can be implemented in PunyInform as well, but the constants are named differently.

Parsing a number: `ParseToken(TT_OBJECT, NUMBER_OBJECT)`

Parsing an object that's in a special scope: `ParseToken(TT_SCOPE, MyScopeRoutine)`

Colours

When writing a game in z5 or z8 format, it's possible to change the colour of text and background. PunyInform provides some support for this. There is an example of how to do this in howto/colours.inf in the PunyInform distribution. This code also shows how you might go about writing a game which can be compiled to z3 or z5, with colour support in the z5 version.

In order to use colours, do the following in `Initialise`:

- Check if the interpreter supports colour: `if(HDR_TERPFLAGS->0 & 1)` . If not, skip the rest of this list.
- Ask the user if they want to use colour. If they say no, skip the rest of this list, except the call to `ClearScreen` at the end.
- Set `clr_on` to `true`.
- Set `clr_bg` and `clr_fg` to the background and foreground colours of your choice.
- If you want player input to be a different colour, set `clr_fginput` to this colour.

- If you want the status line to be a different colour, set `clr_fgstatus` to this colour.
- Call `ClearScreen(WIN_ALL);`. This tells the interpreter to start using the new colours, and clears the screen.

After having done this, if you want to change the text colour for a piece of text in the game, call `ChangeFgColour` with the colour you want to use. This sets `clr_fg` to the new colour and, if `clr_on` is `true`, tells the interpreter to change the colour. To return to the original text colour, call `ChangeFgColour` again, with the original foreground colour. This routine, along with the colour constants shown below, is defined for z3 as well, to aid in writing games that can be compiled to either format.

The colour constants that can be used are:

```
CLR_BLACK
CLR_RED
CLR_GREEN
CLR_YELLOW
CLR_BLUE
CLR_MAGENTA
CLR_CYAN
CLR_WHITE
```

You can also set all colours to `CLR_DEFAULT`, to revert to the interpreter's default colours.

If you intend to release your game for Commodore computers (of course you do!), you should be aware that many colour combinations create artifacts on a real Commodore computer connected to a TV set. These artifacts make the text ugly and hard to read. For this reason, it is recommended you stick to these colour combinations, which all work fine:

- Black background: Red, green, yellow, magenta, cyan, white foreground
- Red background: White foreground
- Green background: Black, white foreground
- Yellow background: Black foreground
- Blue background: White foreground
- Magenta background: Black, white foreground
- Cyan background: Black, white, green foreground
- White background: Black, red, green, blue, magenta, cyan foreground

Unsupported Properties and Attributes

PunyInform lacks support for a few properties and attributes which the standard library support:

- Properties `articles` and `short_name_indef`: These are intended for use

with languages other than English, something PunyInform doesn't support.

- Property `plural`: This is intended for use with identical objects, something PunyInform doesn't support.
- Propery `number`: This property was added to the Inform library before individual properties were supported. With individual properties available, there is little reason to define this generic property. If you use it in your code, it's automatically created as an individual property. If you use it heavily, you may want to make it a common property. To do this, just add `Property number;` right after including "globals.h".
- Attribute `male`: If you define an animate object and don't give it `female` or `neuter`, it will be considered male automatically. This is done to leave one more attribute free for the game author's use.

Programming Advice

Error messages

The Inform standard veneer routine for printing informative messages for all sorts of runtime errors that can occur is replaced with a simpler routine in PunyInform, saving about 1.5 KB. However, the original routine is used if the constant RUNTIME_ERRORS is set to 2.

Debugging

By defining the constant `DEBUG` (or adding -D to the inform 6 compiler commandline), the game is compiled in debug mode. This means a number of meta verbs are available for inspecting the game world and examining which routines and actions are executed. These are the debug verbs supplied:

TREE : Show the object tree for the current location. *TREE [object]* : Show the object tree for this object.

GONEAR [object] : Teleport to the location of the object.

SCOPE [actor] : List the objects which are currently in scope for the actor. Actor defaults to player.

PRONOUNS : List what he, she, it and them are currently referring to.

RANDOM [number] : Seed the pseudo-random number generator, to make randomization predictable. Number defaults to 100.

PURLOIN [object] : Teleport the object into your inventory, no matter where it is.

ROUTINES [on/off] : Show which routines are being executed.

ACTIONS [on/off] : Show which actions are being invoked.

TIMERS [on/off] : Show which timers and daemons are being executed.

DEBUG REACTIVE : Show which object have the reactive attribute but shouldn't and which don't have it but probably should. See The reactive attribute.

DebugParseNameObject

Some debug verbs take an object or an actor as an argument. The scope for these verbs is unlimited - they can refer to objects which are in a different location or even in no location. It can be hard or even impossible for the parser to decide if an object which doesn't have a parent is a room or a normal object. This causes problems when an object has a `parse_name` routine. If you have problems referring to an object which has a `parse_name` routine and may lack a parent in the object tree, you can create a routine called `DebugParseNameObject(obj)` to help out. It should return true for every such object and false for other objects. It's okay if it returns true for other normal objects, but it must never return true for a room. Example implementation which returns true for the three objects in the list and false for all other objects:

```
#Ifdef DEBUG;
[ DebugParseNameObject p_obj;
    if(p_obj == RecordPlayer or Record or Robot) rtrue;
    else rfalse;
];
#Endif;
```

Customizing the Library

PunyInform is designed to be as small as possible to run well on old computers, and some features that add to the size have been made optional. If you want to enable these features, add a line like `Constant OPTIONAL_GUESS_MISSING_NOUN;` before including globals.h, but keep in mind that it will make the game larger. You can also change some parameters in the library from their default values, which may also affect the library size. Finally you can use abbreviations to reduce the game size further.

These customizations are described in detail in the following sections.

Optionals

The optional parts of PunyInform can be enabled with these constants:

Option	Bytes	Comment
DEBUG	2304	Enable some debugging verbs for game development. These include 'scope', 'random', 'pronouns', 'tree', 'purloin', 'gonear', 'routines', 'actions' and 'timers'/'daemons'.
CUSTOM_PLAYER_ OBJECT	-	Set it to an object ID and that object will be used as the player object, and the default player object (`selfobj`) won't be created.
NO_SCORE	-254	Don't include any code for keeping track of score. The value of the constant is shown as the score on the statusline in z3.
OPTIONAL_SL_NO_ SCORE	-40	Don't show score on the statusline in z5+ games. Defining `NO_SCORE` will define `OPTIONAL_SL_NO_SCORE` automatically.
OPTIONAL_SL_NO_ MOVES	-36	Don't show moves on the statusline in z5+ games.
OPTIONAL_ALLOW_ WRITTEN_NUMBERS	260	Enable parsing of 'one', 'two' etc as numbers.
OPTIONAL_ LANGUAGE_NUMBER	168	Add the routine `LanguageNumber(n)` to print a number with words, taking `DIALECT_US` into account. Can be used with `OPTIONAL_LIST_TOGETHER`. See list_together and LanguageNumber for some notes on this.
OPTIONAL_ NON_FLASHING_ STATUSLINE	44	Use an alternate statutsline routine for z5, which doesn't fill the statusline with reverse spaces before printing the statusline information. This alternate version is slightly faster, but it's not compatible with some old interpreters (MaxZip on MacOS and Infocom's interpreter for Amiga, maybe others as well.)

Option	Bytes	Comment
OPTIONAL_ EXTENDED_ METAVERBS	794	Add a set of less important, but nice to have, meta verbs to the grammar.
OPTIONAL_ EXTENDED_ VERBSET	1532	Add a set of less important, but nice to have, verbs to the grammar.
OPTIONAL_FLEXIBLE_ INVENTORY	64	Allow the player to type "inventory tall/wide" to switch inventory mode.
OPTIONAL_FULL_ DIRECTIONS	130	Include directions NW, SW, NE and SE. Including them also makes the parsing process slightly slower in z3 mode.
OPTIONAL_FULL_ SCORE	244	Add the fullscore verb, and optional support for tasks as described in DM4. Size grows by another 78 bytes if `OPTIONAL_SCORED` is defined.
OPTIONAL_GUESS_ MISSING_NOUN	406	Add code to guess missing parts of an incomplete input, such as a door when typing only 'open', and accepting the input with a "(assuming the wooden door)" message.
OPTIONAL_LIST_ TOGETHER	598	Support the list_together property for grouping similar objects in lists. See list_together and LanguageNumber for some notes on usage.
OPTIONAL_MANUAL_ REACTIVE	-42	Leave it to the author to set the `reactive` attribute, for faster game start. See The reactive attribute for instructions on how to use it.
OPTIONAL_MANUAL_ SCOPE	-4	Let the game code say when scope needs to be updated, for better performance. See Manual Scope for instructions on how to use it.

Option	Bytes	Comment
OPTIONAL_MANUAL_ SCOPE_BOOST	30	Look for opportunities to skip checking `react_before`, `react_after` and `each_turn`. See Manual Scope Boost for instructions on how to use it.
OPTIONAL_NO_ DARKNESS	-322	Skip support for light and darkness - there is always light everywhere. Unlike most of the optionals, this one makes the game smaller. Also, it stops the library from defining the `light` attribute.
OPTIONAL_ORDERED_ TIMERS	100	Lets you assign an order number (property `timer_order`, default = 100) to each timer or daemon, defining the order of execution - low numbers execute early.
OPTIONAL_PRINT_ SCENERY_CONTENTS	82	Have 'look' show what is in/on containers and supporters which have the scenery attribute.
OPTIONAL_PROVIDE_ UNDO	240	Add undo functionality (z5 and z8 only).
OPTIONAL_REACTIVE_ PARSE_NAME	16	Let objects which provide `parse_name` have the `reactive` attribute. See The reactive attribute for instructions on how to use it.
OPTIONAL_SCORED	20	Add support for the scored attribute as described in DM4.
OPTIONAL_SIMPLE_ DOORS	86	Allow for a simpler way of defining doors. This also ends up saving space if you have more than three doors that use this mechanism. See Doors for instructions on how to use it.

Option	Bytes	Comment
OPTIONAL_SHIP_ DIRECTIONS	88	Add 'fore', 'aft', 'port' and 'starboard' as directions. See Ship Directions for instructions on how to use it.

Parameters

The parameters listed in the table below can be adjusted in a game by defining them before globals.h is included.

Parameter	Default	Comment
DEFAULT_CAPACITY	100	Default number of items that can be in a container, on a supporter or held by a creature.
MAX_CARRIED	32	Max. number of items the player can carry at once
MAX_WHICH_OBJECTS	10	Max. number to include in a "which X do you mean?" parser question
MAX_MULTIPLE_OBJECTS	32	Max. number of objects that match "all" in an input such as "get all"
MAX_INPUT_CHARS	78	Max. number of characters in one line of input from the player
MAX_INPUT_WORDS	20	Max. number of words in a parsed sentence
MAX_FLOATING_OBJECTS	32	Max. number of floating objects
MAX_TIMERS	32	Max. number of timers/daemons running at once
MAX_SCOPE	50	Max. number of objects to consider when calculating the scope of the player or another actor
RUNTIME_ERRORS	1 or 2	Runtime error reporting: 0 = minimum, 1 = report all errors using error codes, 2 = report all errors using error messages. Default is 2 in `DEBUG` mode, and 1 when not in `DEBUG` mode.

Abbreviations

PunyInform can use a set of standard abbreviations to make strings more compact. If you want to provide your own abbreviations, define the constant CUSTOM_ABBREVIATIONS in your game. Keep in mind that you need to compile with the "-e" flag to make the compiler use abbreviations.

Limitations for z3

If you want to compile a game to z3 format, this is what you need to keep in mind:

- A z3 Z-code file can be no bigger than 128 KB.
- A game can have no more than 255 objects (classes also count as objects). The language and the library normally defines seven objects/classes, so you can define 248 objects/classes.
- A game can use no more than 30 common properties. PunyInform normally defines 27 common properties.
- A game can use no more than 32 attributes. PunyInform defines 29 attributes (+1 if `OPTIONAL_SCORED` is defined, -1 if `OPTIONAL_NO_DARKNESS` is defined).
- Arrays in common properties can only hold four values. Arrays in individual properties however, can hold 32 values. If you need more than four names for an object in a z3 game, you need to give it a `parse_name` routine.
- Routine calls can have no more than three arguments.
- When using message passing (like `MyBox.AddWeight(5)`), no more than one argument may be passed. (In regular Inform, message passing doesn't work at all in z3.)
- Dynamic object creation and deletion can not be used.
- The room name printed on the statusline is always the object name string. It can't be overridden with `short_name`. Read below for a possible workaround.
- It is not possible to support the 'undo' verb.
- Player input can't be re-tokenized. This is an advanced technique that is often used with non-English languages in the standard library.
- There is no support for bold or italic text.
- The cursor position can't be read or set. Among other things, this makes it impossible to print a menu and let the player move up and down in it with certain keys.
- It is not possible to wait for the user to press a key, or read a single keypress - you can only read a whole line of input.
- The interpreter is responsible for displaying the statusline. It will show a score even if you define NO_SCORE. It's not possible to hide the statusline.

When the player is inside an object, in a z5 game, the library will print the name of the object on the statusline, in definite form ("The box"). In a z3 game, the

object name string will be printed as-is, typically like "box". This behavior in z3 games is part of the Z-machine specification so it's nothing that the game or the library can change. If you want a z3 game to print a different name for when the player is inside the object, you can set the object name string to the desired name, and override it with `short_name` for all other uses, like this:

```
Object box "The box"
   with short_name "box",
   has container openable enterable;
```

Properties

Properties are used to store values in objects.

A property can either be common or individual. Common properties are a little faster to access and use a little less memory than individual properties. A z5 or z8 game can use a maximum of 62 common properties, while a z3 game can use a maximum of 30 common properties. PunyInform uses 27 common properties, so if you're building a z3 game, you can only add three common properties. The value of a common property can always be read, but it can only be written if it has been included in the object declaration. If you don't include it, there is no memory allocated to store a value. If you read the value of such a property, you just get the default value (typically 0).

A common property is created by declaring it with

```
Property my_property_name;
```

Declaring individual properties is optional, but may result in slightly shorter code. It's done with

```
Property individual my_property_name;
```

To access a property, you write *object.propertyname*, like this:

```
Dog.description = "The dog looks sleepy.";
```

To check if an object has a value for a property (to see if it can be written if it is a common property or to see if it can be read or written if it is an individual property), use *provides*:

```
If(Dog provides description) ...
```

A property can be used to store a 16-bit value, or an array of values. In z5, a property array can hold up to 32 values. In z3, a property array can only hold 4 values if it's in a common property but 32 values if it's in an individual property.

If a property is declared as additive, the values for an object are concatenated with the values of its class(-es), if any, and put into an array.

Extensions

PunyInform keeps the library code size down by only providing the most fundamental functionality by default, but ships with several extensions which can easily be added to games.

cheap_scenery

(Can also be used with the standard library)

This library extension provides a way to implement simple scenery objects which can be examined, using just a single object for the entire game. This helps keep both the object count and the dynamic memory usage down. For z3 games, which can only hold a total of 255 objects, this is even more important. To use it, include ext_cheap_scenery.h after globals.h. Then add a property called `cheap_scenery` to the locations where you want to add cheap scenery objects. You can add up to ten cheap scenery objects to one location in this way, and even more with `CS_ADD_LIST` and `CS_MAYBE_ADD_LIST` (see below). All in all there are five different record types that can be used in cheap scenery arrays:

(1) For each scenery object specify three values, in this order: word1, word2, and a reaction string/routine. word1 and word2 are dictionary words. The reaction string/routine is described below.

(2) If you only need a single noun, or you need multiple nouns and/or adjectives, you can write a number between 1 and 99, calculated as (10*adjectives)+nouns, followed by first the adjectives, then the nouns, and finally the reaction string/routine. E.g. a yellow record player might be added as `21 'yellow' 'record' 'player' "It's yellow!"` This means there are two adjectives (yellow and record) and one noun (player). Any combination of adjectives and nouns will be recognized in player input, as long as the adjectives come before the nouns, and at least one of the nouns is used.

(3) For more complex names, give the value `CS_PARSE_NAME`, followed by a routine to work like a `parse_name` routine and then the reaction string/routine.

(4) You can add a list stored in a property in the same or another object, by giving the value `CS_ADD_LIST`, followed by an object ID and a property name (put the property name within parentheses to avoid compiler warnings). You typically want to use a list like this to give general descriptions for scenery objects in a group of locations, or because you've hit the limitation of 32 values in a property array. If you want an exception for a certain scenery object in one location, just give it a different description *before* linking to the general list (See "Example usage" below).

(5) You can add a conditional list by specifying CS_MAYBE_ADD_LIST, a routine, an object ID and a property name. If the routine returns true, the list stored in that property of that object is added.

For record types 1 and 2 above, you can also precede the record with an ID between 100 and 500, and/or the value CS_THEM, in this order. If you add an ID, this can be used for performing actions on this cheap scenery object using `CSPerformAction`, or to choose what to do in the `SceneryReply` routine.

If you add CS_THEM, the object is considered a "them"-object by the parser, i.e. the player can type "EXAMINE DOORS. OPEN THEM". If you don't add CS_THEM, the object will still be considered a "them"-object if the player input that was matched contained a plural word, i.e. a word like `'books//p'`. For record type 3, the parse_name routine can set the global `parser_action = ##PluralFound` to signal that the object should be considered a "them"-object.

If what the player typed matches several cheap scenery records, the record that matches the most words in player input is used. If it's a tie, the first of the tied matches is used.

The reaction string/routine can be either:

- a string to be used as the description of the object
- a routine which will act as a `before` routine for the object - this can be used to trap the ##Examine action and print a dynamic description of the object, but also to react to any other actions the player may try to perform on the object.

If you want to use the same reaction string/routine for scenery objects in several locations, declare a constant to hold that string or routine address, and refer to the constant in each location. Note that if this is a routine, you typically want to end it with `default: rfalse;` since a named routine returns true by default, which would lead it to block all actions which the routine doesn't handle.

Before including this extension, you can define a string or routine called `SceneryReply`. If you do, it will be used whenever the player does something to a scenery object which isn't handled by the object. If `SceneryReply` is a string, it is printed. If it is a routine it is called. If the routine prints something, it should return true, otherwise false. The routine is called with three parameters - `word1`, `word2` and `id_or_routine`.

- If a record which has an ID was matched, `id_or_routine` is set to this ID.
- If a record using `CS_PARSE_NAME` was matched, `word1` is `CS_PARSE_NAME`, `word2` is `0` and if the entry doesn't specify an ID, `id_or_routine` is the routine address.
- If the record matched starts with a number 1-99 (signaling that it has 0-9 adjectives and 1-9 nouns), `word1` is the first adjective specified in the record or 0, while `word2` is the first noun specified in the record.
- For a "normal" record, holding two dictionary words and a reaction string/routine, `word1` and `word2` hold the two dictionary words specified in the record. Note that this may not be exactly what the player typed, e.g. the player may have typed "examine water" but the words listed in the property are `'blue' 'water'`. In this case, `word1` will be 'blue' and `word2` will be 'water'.

Note: If you want to use this library extension in a Z-code version 3 game, you must NOT declare `cheap_scenery` as a common property, or it will only be able to hold one scenery object instead of ten. For z5 and z8, you can declare it as a common property if you like, or let it be an individual property. Using a common property makes for a smaller and faster game.

Before including this extension, you can also define a string to replace the default answer, called CS_DEFAULT_MSG. This isn't terribly useful for a game programmer, as `SceneryReply` gives you a lot more power, and can be used to replace the default answer as well. However, it may come in handy for someone creating a library extension. Here's how to do it:

```
Constant CS_DEFAULT_MSG "Focus on the mission, soldier!";
```

If constant `RUNTIME_ERRORS` is defined and has a value greater than 0, the extension will complain about programming mistakes it finds in the `cheap_scenery` data in rooms. Otherwise, it will keep silent.

Example usage:

```
! Cheap Scenery IDs. Use values 100-500.
Constant CSP_LIBRARY 100;

[ SceneryReply word1 word2 id_or_routine;
    ! We can check location, if we want different answers in different rooms
    ! We can also check action, and there's even an implicit switch on action,
    ! so we can do things like: Take: "You're crazy.";
    switch(id_or_routine) {
    ParseNameAir:
        "You need the air to breathe, that's all.";
    CSP_LIBRARY:
        "The library is super-important. Better not mess with it.";
    }
```

```
    if(location == Library && word1 == 'book' && word2 == 'books')
        "Leave the books to the people who care about them.";
    rfalse;
];

Include "ext_cheap_scenery.h";
Include "puny.h";

[ ParseNameAir;
    if(NextWord() == 'air') return 1;
    rfalse;
];

[ WallDesc;
    Examine:
        "The walls are ",
            (string) random("all white", "claustrophobia-inducing", "scary",
                "shiny"), " here.";
    default:
        rfalse; ! Necessary in a named routine
];

Constant BOOKDESC "You're not interested in reading.";

Object Library "The Library"
    with
        description "You are in a big lovely library. You can examine or try to
            take the books, the shelves, the library, the air, the walls and
            the ceiling.",
        cheap_scenery
            CS_ADD_LIST Library (inside_scenery)
            4 'book' 'books' 'volume' 'volumes' BOOKDESC
            CS_THEM 'shelf' 'shelves' "They're full of books."
            CS_PARSE_NAME ParseNameAir "The air is oh so thin here."
            CSP_LIBRARY CS_PARSE_NAME [ _i _w;;
                _w = NextWord();
                if(_w == 'big') { _i++; _w = NextWord();}
                if(_w == 'lovely') { _i++; _w = NextWord();}
                if(_w == 'library') { _i++; return _i;}
                return 0;
            ] "It's truly glorious.",
        inside_scenery
            'wall' 'walls//p' WallDesc
            1 'ceiling' "The ceiling is quite high up.",
    has light;
```

Performing actions involving cheap scenery objects

To perform an action involving a cheap scenery object in code, use `CSPerformAction`. The cheap scenery object you want to refer to must be present in the current location and have an ID. The object may be in a list added using CS_ADD_LIST or CS_MAYBE_ADD_LIST. You can call this procedure in three ways:

- noun mode: Specify the action and the cheap scenery noun ID
- noun + second mode: Specify the action, the cheap scenery noun ID and a regular object for `second`
- reverse noun + second mode: Specify the action, the cheap scenery ID for `second` and (minus) a regular object for `noun`

Examples:

```
CSPerformAction(##Take, 100);         ! Perform a Take action, with:
                                      ! noun = cheap scenery object with ID 100
CSPerformAction(##PutOn, 100, Shelf); ! Perform a PutOn action, with:
                                      ! noun = cheap scenery object with ID 100
                                      ! second = Shelf
CSPerformAction(##PutOn, 100, -Ball); ! Perform a PutOn action, with:
                                      ! noun = Ball
                                      ! second = cheap scenery object with ID 100
```

Reacting to actions involving cheap scenery objects

Cheap scenery has a global variable called `cs_match_id`. This can be used to react to actions involving cheap scenery objects in code, e.g. in the before routine of a location. Here's sample code to react when the player tries to throw a ball at a window which has been implemented as cheap scenery:

```
Constant CS_WINDOW 100;

Object DiningRoom "Dining Room"
    with
        description "A lovely room.",
        cheap_scenery
            'table' 'tables//p' "The tables are made of wood."
            CS_WINDOW 1 'window' "It's a huge window.",
    before [;
        ThrowAt:
            if(noun == Ball && second == CheapScenery && cs_match_id == CS_WINDOW)
                "That's too risky, you could break it!";
    ];
```

Debugging cheap scenery arrays

To check for incorrect cheap scenery arrays in a game, compile it in DEBUG mode and type the verb "cstest". All cheap scenery arrays in the game will be checked, and any detected problems are printed. Note that the first element in an array is element 0, i.e. if this command reports a problem with element 1, this means it's the second element in the array.

This command prints a hash (#) for every array that is checked. If an array is added in several other arrays using CS_ADD_LIST or CS_MAYBE_ADD_LIST, it is tested once for each occurence.

cheap_scenery example

There is an example game in the PunyInform distribution that uses the cheap_scenery extension, in the file howto/cheapscenerydemo.inf.

flags

(Can also be used with the standard library)

Flags is a mechanism for keeping track of story progression. If you choose to use flags, four procedures with a total size of about 165 bytes are added to the story file, and one byte is added to dynamic memory for every eight flags. All in all this is a very memory-efficient way of keeping track of progress.

If you want to use flags, after including globals.h, set the constant `FLAG_COUNT` to the number of flags you need, and then include ext_flags.h.

You then specify a constant for each flag, like this:

```
Constant F_FED_PARROT 1; ! Has the parrot been fed?
Constant F_TICKET_OK 2; ! Has Hildegard booked her plane tickets?
Constant F_SAVED_CAT 3; ! Has the player saved the cat in the tree?
```

You get the idea – you give each flag a symbolic name so it's somewhat obvious what it does. Note that the first flag is flag #1, not flag #0.

Setting a flag on or off means calling the routine `SetFlag` or `ClearFlag`

To indicate that the player has saved the cat, call `SetFlag(F_SAVED_CAT)`, and to turn off that flag, call `ClearFlag(F_SAVED_CAT)`.

Testing a flag is accomplished by calling `FlagIsSet` or `FlagIsClear`. So if you have a piece of code that should only be run if the parrot has been fed, you would enclose it in an `if(FlagIsSet(F_FED_PARROT)) { ... };` statement.

Naturally, you can test if a flag is clear by calling `FlagIsClear` instead.

For all four routines, you can add one or two flags, to do more with a single function call. E.g. `ClearFlag(F_SAVED_CAT, F_TICKET_OK)` clears

both flags, and `if(FlagIsClear(F_TALKED_TO_BOB, F_FOUND_TREASURE, F_BROKE_WINDOW))` returns true only if all three flags are clear.

There are two more functions, which can be called with two or three arguments each: `AnyFlagIsSet(F1, F2, F3)` returns true if any of the flags are set. `AnyFlagIsClear(F1, F2, F3)` returns true if any of the flags are clear.

talk_menu

(Can also be used with the standard library, when compiling to Z-code)

This extension lets you define a set of topics for each character which the player can talk to him or her about. The player can type "TALK TO (someone)" and get a menu of topics they can talk about. Each topic can unlock new topics, set flags (if you include ext_flags.h), and run routines.

Setup

To use this extension, define a word array for each NPC that the player should be able to talk to. In each NPC object, define a `talk_array` property, with the corresponding array name as its value. Add zero or more talk topics to the array. End the array with the value TM_END.

Talk topics

A talk topic has the following form:

`STATUS [ID]* TOPIC PLAYERSAYS NPCSAYS [FLAGREF|UNLOCKREF|ROUTINE|STRING]*`

- `[]` = Optional
- `*` = can be more than one
- *STATUS* is either 0 (= `TM_INACTIVE` = not active) or 30 (= `TM_ACTIVE` = active) or 31 (= `TM_STALE`, has been used).
- *ID* is a number (300-600) which can be used as a reference to activate the topic in code or using an *UNLOCKREF* in `talk_array`. Note that IDs are local to the NPC - two different NPCs can use the same ID for different topics without risk of confusion. One topic can have multiple IDs, and multiple topics can use the same ID.
- *TOPIC* is a string or routine for the topic name
- *PLAYERSAYS* can be on any of these forms (ROUTSTR means a routine or string):
  - ROUTSTR
  - TM_ADD_BEFORE ROUTSTR ROUTSTR
  - TM_ADD_AFTER ROUTSTR ROUTSTR

– TM_ADD_BEFORE_AND_AFTER ROUTSTR ROUTSTR ROUTSTR

I.e. you can add a routine or string to run/print BEFORE the player's line, AFTER the player's line, or both. To mute the player's line, give the value TM_NO_LINE for ROUTSTR for the player's line.

- *NPCSAYS* is a string or routine for what the NPC replies. To mute the NPC's line, give the value TM_NO_LINE for ROUTSTR for the NPC's line.
- *FLAGREF* is a number 32-299 for a flag to be set (In order to use this, you must include ext_flags.h before including ext_talk_menu.h)
- *UNLOCKREF* is either a topic ID (300-600) or a relative reference to a topic (1 to 29) that should be activated by this topic. 1 means the next topic, 2 the topic after that etc. The target topic has to have status `TM_INACTIVE` (= 0) or `TM_ACTIVE` (= 30) for this to work. When a topic is used, it is set to status `TM_STALE`, and the only way to change it from status `TM_STALE` is to call `ReActivateTopic` or `ReInactivateTopic`.
- *ROUTINE* is the name of a routine to be run. In this routine, the global variable `current_talker` refers to the NPC being talked to.
- *STRING* is a string to be printed.

Whenever a routine is used for PLAYERSAYS, NPCSAYS or ROUTINE, it can set the global `talk_menu_talking` to false to end the conversation after the current topic. When doing this, you may want to use ROUTINE to print a suitable message about why the conversation ended.

Multiple topics with the same ID

It's possible to use the same ID for several topics. This can be useful to activate or inactivate a group of topics at once. If you never do this, you can set the global `talk_menu_multi_mode = false;` in `Initialise`, to gain a bit of performance. You can also choose to enable it just before making a call and then disable it again, e.g. `talk_menu_multi_mode = true; ActivateTopic(Captain, T_SUBMARINE_GROUP); talk_menu_multi_mode = false;`. Also note that a topic can have multiple IDs, so you can assign it one ID to be used when it's activated or inactivated on its own and another ID for when the whole group is activated or inactivated.

A sample talk array

Example of an array giving Linda one active topic (Weather), which will activate the next topic (Heat) and the topic with ID 300 (Herself), and if you ask about Heat, she'll just end the conversation without answering.

```
[ EnoughTalk; talk_menu_talking = false; ];
```

```
Array talk_array_linda -->
  0 300 "Herself" "Tell me more about yourself!" "I'm just an average girl."
  30 "Weather" "How do you like the weather?" "It's too hot for me." 1 300
  0 "Heat" "Say, don't you like hot weather?" TM_NO_LINE EnoughTalk
           "Linda looks upset and turns away."
  TM_END;

Object Linda "Linda"
  with
    talk_array talk_array_linda,
    ...
```

Allowing for more flags and/or topic IDs

If you find that you need more topic IDs, or more flags, you can define which number should be the lowest one to be considered an ID (32-600, default is 300) by defining the constant `TM_FIRST_ID`, i.e. to get 100 more IDs and 100 less flags, do this before including ext_talk_menu.h:

```
Constant TM_FIRST_ID = 200; ! 32-199 are now flags, while 200-600 are IDs
```

Should you find that you need both a lot of flags and a lot of topic IDs, you can:

1. Make sure all routines you refer to in `talk_array` are defined *after* including puny.h.
2. Set the constant `TM_LAST_ID` to 2000. Instead of 300-600, you can now use 300-2000 for topic IDs.
3. In conjuction with this, you can also use `TM_FIRST_ID` to define where flags end and topic IDs begin.

talk_menu routines

Apart from activating topics using UNLOCKREFs in the talk_array, you can also use these routines:

- `ActivateTopic(NPC, topic)`, returns true for success, false for fail: Activates the topic if it's currently inactive or active (not stale!).
- `ReActivateTopic(NPC, topic)`, returns true for success, false for fail: Activates the topic, regardless of its status.
- `InactivateTopic(NPC, topic)`, returns true for success, false for fail: Inactivates the topic if it's currently inactive or active (not stale!).
- `ReInactivateTopic(NPC, topic)`, returns true for success, false for fail: Inactivates the topic, regardless of its status.
- `GetTopicStatus(NPC, topic)`, returns topic status (`TM_INACTIVE`, `TM_ACTIVE` or `TM_STALE`)

If you call these routines with `DEBUG` defined *and* `RUNTIME_ERRORS` > 0 you will

be notified whenever a problem is detected. As usual, use `RUNTIME_ERRORS = 2` to get the maximum amount of information (This is the default when compiling in `DEBUG` mode).

talk_menu example

There is an example game in the PunyInform distribution that uses the talk_menu extension, in the file howto/talk_menu.inf.

Customizing talk_menu

You can customize the following messages by defining them as strings or routines before including this extension:

- `TM_MSG_YOU` - The string "You"
- `TM_MSG_TALK_ABOUT_WHAT` - The prompt asking the player to pick a topic.
- `TM_MSG_TOPICS_DEPLETED` - The message printed when there are no active topics left.
- `TM_MSG_EXIT` - The message printed when the player chooses to end the conversation.
- `TM_MSG_NO_TOPICS` - The message to tell the player they can't talk to the person now, since there are no active topics.
- `TM_MSG_EXIT_OPTION` - The text for the option to leave the conversation.
- `TM_MSG_PAGE_OPTION` (z5 only) - The text to indicate that N can be used to see the next page of options.
- `TM_MSG_EXIT_OPTION_SHORT` (z5 only) - The text for the option to leave the conversation, for screens with less than 40 columns.
- `TM_MSG_PAGE_OPTION_SHORT` (z5 only) - The text to indicate that N can be used to see the next page of options, for screens with less than 40 columns.

You can also decide exactly how a line is printed, including the printing of name of the actor saying the line, by defining a routine called `TMPrintLine` before including the extension. To do this, it's easiest to copy this routine from ext_talk_menu.h and use that as a starting point. Note that you don't need to use the `Replace` directive.

menu

This is an extension to let games show a menu of text options (for instance, when producing instructions which have several topics, or when giving clues). This can be done with the `DoMenu` routine, which is very similar to the DoMenu in the standard Inform library. In version 3 mode it will create a simple text version instead because of technical limitations.

When the player exits a menu, a LOOK command will be executed. This is usually a good idea to show the player that they're back in the game, and where

they are in the game. If you prefer not to have this behaviour, you can define the constant `M_NO_LOOK` before including this extension. In particular, this may be useful if you want to show a menu in the initialise routine, or in a context where a LOOK will be performed when the player exits the menu anyway.

A common way of using `DoMenu` is from a "help" verb, which can be declared like so:

```
Include "ext_menu.h";

! add HelpItem, HelpMenu and HelpInfo here

[ HelpSub;
    DoMenu(HelpItems, HelpMenu, HelpInfo);
];

Verb 'help' * -> Help;
```

Below is how `DoMenu` was described in the *Inform Designer's Manual, 3rd edition.*

Extract from DM3

Here is a typical call to `DoMenu`:

```
DoMenu("There is information provided on the following:^
       ^     Instructions for playing
       ^     The history of this game
       ^     Credits^", HelpMenu, HelpInfo);
```

Note the layout, and especially the carriage returns.

The second and third arguments are themselves routines. (Actually the first argument can also be a routine to print a string instead of the string itself, which might be useful for adaptive hints.) The `HelpMenu` routine is supposed to look at the variable `menu_item`. In the case when this is zero, it should return the number of entries in the menu (3 in the example). In any case it should set `item_name` to the title for the page of information for that item; and `item_width` to its length(*) in characters (this is used to centre titles on the screen). In the case of item 0, the title should be that for the whole menu.

The second routine, `HelpInfo` above, should simply look at `menu_item` (1 to 3 above) and print the text for that selection. After this returns, normally the game prints "Press [Space] to return to menu" but if the value 2 is returned it doesn't wait, and if the value 3 is returned it automatically quits the menu as if Q had been pressed. This is useful for juggling submenus about. Menu items can safely launch whole new menus, and it is easy to make a tree of these (which will be needed when it comes to providing hints across any size of game).

(*) DM3 actually says to set `item_width` to half the length of the title. This looks like a bug and makes using menus unintuitive, so we decided to change

this for ext_menu. However, if you're making a game that should be compilable with either PunyInform and its menu extension or the standard library where DoMenu is included, you may want to use the standard library behaviour on `item_width`. In this case, define the constant `EXT_MENU_STDLIB_MODE` before including this extension.

quote_box

This is an extension to let games show a simple quote box. For z5+ games, the extension will try to center the quote box on the screen, by reading the screen width reported by the interpreter in the header.

For z3, this information is not available. Instead, it can do it in two ways: 1. The game programmer tells the extension to assume the screen has a certain width and the extension uses this information to center the quote box. 2. The game programmer tells the extension to just indent the quote box a fixed number of characters.

To use (1), set the constant `QUOTE_V3_SCREEN_WIDTH` to the desired width, which has to be > 6.

To use (2), set the constant `QUOTE_V3_SCREEN_WIDTH` to the desired number of characters to indent by, which must be in the range 0-6.

By default, method (2) will be used, with 2 characters of indentation.

To display a quote box, create a word array holding the number of lines, the number of characters in the longest line, and then a string per line, and call QuoteBox with the array name as the argument.

```
Include "ext_quote_box.h";

Array quote_1 --> 5 35
"When I die, I want to go peacefully"
"in my sleep like my grandfather."
"Not screaming in terror, like the"
"passengers in his car."
"               -- Jack Handey";

[AnyRoutine;
  QuoteBox(quote_1);
];
```

waittime

This extension gives players an extended Wait command, which can be used to wait a certain number of turns, minutes or hours, or to wait until a certain time

of day is reached.

In a game showing time on the statusline, the player can use commands such as:

```
>wait for 5 minutes
>wait 1 hour
>wait until 1:20
>wait till three o'clock
>wait till quarter to five
>wait till 5 am
>wait 3 turns/moves
```

A turn or a move may be the same as a minute, depending on the time scale (how many minutes the clock is advanced per turn, or how many moves it takes before the clock is advanced one minute).

Note: Using words for numbers requires `OPTIONAL_ALLOW_WRITTEN_NUMBERS`.

While the player is waiting, the global variable `waittime_waiting` has the value true. A daemon or `each_turn` routine may show an event which could make the player want to abort the waiting and spring to action. If this happens, set `waittime_waiting` to false.

If game time is suddenly changed, typically using a SetTime() call, it is a good idea to abort any ongoing waiting.

This extension also includes the parse routine `parsetime` to parse times of day, like "1:20", "quarter to five", "3:10 pm" etc, which can also be used for other verbs which need this, like setting a watch or clock. The parsed time (in minutes after midnight) comes in `noun` or `second`.

In a game showing score/turns on the statusline, the commands to wait for a certain number of turns, minutes or hours still work. A minute is considered the same as a turn. The command to wait until a certain time of day is not available, and neither is the `parsetime` routine.

This extension must be included after including "puny.h". Before including it, you may define the constants `MAX_WAIT_MINUTES` and `MAX_WAIT_MOVES` to say how long the player is allowed to wait for using a single command.

Appendix A: List of Routines

PunyInform defines both public and private routines. The private routines are prefixed with an underscore (for example, _ParsePattern) and using them is not recommended. The public routines do not have this prefix, and are for general use. Most of the public routines work the same, or in a very similar manner, to corresponding routines in DM4, but PunyInform also offers a few extra routines not available in the standard library. All public routines are listed below in this section.

Library Routines

These library routines are supported by PunyInform, as described in DM4.

Library Routine	**Comment**
Banner	
CommonAncestor	
DrawStatusLine	Not available in version 3 games
LanguageNumber	Optional. See list_together and LanguageNumber.
IndirectlyContains	
LoopOverScope	
MoveFloatingObjects	
NextWord	
NextWordStopped	
ObjectIsUntouchable	
PlayerTo	
ParseToken	See ParseToken
PlaceInScope	
PronounNotice	
SetTime	
ScopeWithin	
TestScope	

Library Routine	Comment
TryNumber	
WordAddress	
WordLength	
WordValue	
YesOrNo	

These library routines are not supported

Library Routine	Comment
AllowPushDir	See Direction Handling

Entry Point Routines

These entry point routines are supported by PunyInform, as described in the DM4.

Entry Point Routine	Comment
AfterLife	
AfterPrompt	
Amusing	
BeforeParsing	
ChooseObjects	
DarkToDark	
DeathMessage	
GamePostRoutine	
GamePreRoutine	
Initialise	Mandatory.
InScope	The `et_flag` isn't supported.
LookRoutine	
NewRoom	
ParseNoun	Unlike in the standard library, this is called *before* the `parse_name` property.
ParseNumber	
PrintRank	`OPTIONAL_FULL_SCORE`
PrintTaskName	`OPTIONAL_FULL_SCORE` + `TASKS_PROVIDED`
PrintVerb	
TimePasses	
UnknownVerb	

These entry point routines are not supported

Entry Point Routine	Comment
ParserError	The parser internals differ too much

PunyInform Entry Point Routines

These entry point routines are supported by PunyInform, but not by the standard library.

Entry Point Routine	Comment
ChooseObjectsFinal	See ChooseObjectsFinal
DebugParseNameObject	See DebugParseNameObject
DisallowTakeAnimate	See Animate Objects
LibraryMessages	See Library Messages and Customization

Additional Public Routines

These routines are supported by both the standard library and PunyInform, but are not documented in DM4.

Routine Name	Comment
ChangeFgColour	Works for z5. Exists but does nothing for z3.
ClearScreen	Exists for z5 only. Call with argument WIN_ALL, WIN_STATUS or WIN_MAIN
NumberWord	
NumberWords	
PrintOrRun	
RunRoutines	
CTheyreorThats	Printing-rule
IsOrAre	Printing-rule
ItorThem	Printing-rule
ThatOrThose	Printing-rule

PunyInform Public Routines

These public routines are provided by PunyInform, but not by the standard library.

Routine Name	Comment
CObjIs	Printing-rule
CTheyreorIts	Printing-rule
FastSpaces	Prints the specified number of spaces in an efficient manner.
ImplicitDisrobeIfWorn	Take off the object if worn by actor, which must be player
ImplicitGrabIfNotHeld	Take the object if not held by actor, which must be player
OnOff	Printing-rule
ObjectCapacity	See Capacity
PrintContents	See Printing the Contents of an Object
PrintContentsFromR	See Printing the Contents of an Object
PrintMsg	
RunTimeError	

Appendix B: List of Properties

These are the properties defined by the library:

Property	Read more below
add_to_scope	
after	
article	
before	
cant_go	
capacity	Y
d_to	
daemon	
describe	
description	
door_dir	
door_to	
each_turn	
e_to	
found_in	
in_to	
initial	
inside_description	
invent	
life	
list_together	Y
n_to	
name	
ne_to	
nw_to	
orders	
out_to	

Property	Read more below
parse_name	Y
react_after	
react_before	
s_to	
se_to	
short_name	
sw_to	
time_left	
time_out	
timer_order	Y
u_to	
w_to	
when_closed	
when_off	
when_on	
when_open	
with_key	Y

The properties articles, number, plural and short_name_indef, which are supported by the Inform 6 library, are not supported by PunyInform.

- The `capacity` property doesn't have a default value in PunyInform. To check the capacity of an object, call `ObjectCapacity(object)`. If the object has a value, it's returned (unless the value is a routine, in which case it is executed and the return value is returned). If the object doesn't have a value for capacity, the value `DEFAULT_CAPACITY` is returned. This value is 100, unless you have defined it to be something else.
- `list_together` is only used if `OPTIONAL_LIST_TOGETHER` is defined. It is a common property, unless you declare it as an individual property using `Property individual list_together;`. Read more about list_together in list_together.
- The `parse_name` property works as described in DM4 except that, since PunyInform doesn't support identical objects, it is never called to check whether or not two objects which share the same parse_name routine are identical.
- `timer_order` is only used if `OPTIONAL_ORDERED_TIMERS` is defined. It is an individual property, unless you declare it as a common property using `Property timer_order;`. Read more about ordered timers in Daemons and timers.
- The `with_key` property can also hold a routine. The routine should return false or the object id of the key that fits the lock. When this routine is

called, `second` holds the object currently being considered as a key. This can be used to allow multiple keys fit a lock.

Appendix C: List of Attributes

These attributes are the same as in DM4.

Attribute	Read more below
absent	
animate	
clothing	
concealed	
container	
door	
edible	
enterable	
female	
general	
light	Y
lockable	
locked	
moved	Y
neuter	
on	
open	
openable	
pluralname	
proper	
scored	Y
scenery	
static	
supporter	
switchable	
talkable	
transparent	
visited	

Attribute	Read more below
workflag	
worn	

- `light` is not defined if `OPTIONAL_NO_DARKNESS` is defined.
- `scored` is only defined if `OPTIONAL_SCORED` is defined.
- For `moved` to be updated and `scored` to be considered, you need to set `update_moved` to true whenever moving objects into the player's possession in code. See Moving objects.

These attributes are used in the Inform standard library and are listed in DM4, but are not used in PunyInform.

Attribute	Comment
male	not needed, assumed if an object is animate and it is not female or neuter

These attributes are used in PunyInform but not in the Inform standard library.

Attribute	Comment
reactive	See The reactive attribute for instructions

Appendix D: List of Variables

These variables are the same as in DM4.

Variable	Comment
action	
actor	
c_style	
clr_bg	
clr_fg	
clr_fgstatus	
clr_on	
clr_fginput	
consult_from	
consult_words	
deadflag	
herobj	
himobj	
inp1	
inp2	
inventory_stage	
inventory_style	
itobj	
keep_silent	
listing_together	
location	
lookmode	
no_implicit_actions	
notify_mode	
num_words	
parsed_number	
parser_action	
parser_one	

Variable	Comment
parser_two	
real_location	
receive_action	
scope_reason	TALKING_REASON is not used, see Scope
scope_stage	
score	
second	
special_number	
the_time	
themobj	
verb_word	
verb_wordnum	
wn	

These variables exist in PunyInform only.

Variable
clr_fginput
run_after_routines_msg
run_after_routines_arg_1
scope_modified
selected_direction
selected_direction_index

These variables are used in the Inform standard library and are listed in DM4, but are not used in PunyInform.

Variable
clr_bgstatus
et_flag
lm_n
lm_o
standard_interpreter
vague_object

Appendix E: List of Constants

These constants are the same as in the standard library and DM4.

Constant Name
AMUSING_PROVIDED
CLR_BLACK
CLR_BLUE
CLR_CURRENT
CLR_CYAN
CLR_DEFAULT
CLR_GREEN
CLR_MAGENTA
CLR_RED
CLR_WHITE
CLR_YELLOW
EACH_TURN_REASON
GPR_FAIL
GPR_MULTIPLE
GPR_NUMBER
GPR_PREPOSITION
GPR_REPARSE
Headline
LOOPOVERSCOPE_REASON
MAX_CARRIED
MAX_SCORE
MAX_TIMERS
NUMBER_TASKS
OBJECT_SCORE
PARSING_REASON
REACT_AFTER_REASON
REACT_BEFORE_REASON

Constant Name
ROOM_SCORE
SACK_OBJECT
Story
TASKS_PROVIDED
TESTSCOPE_REASON

These constants are used in the Inform standard library and are listed in DM4, but are not used in PunyInform. Most of them are parser specific for the standard lib, and the PunyInform parser works differently.

Constant Name
ANIMA_PE
ASKSCOPE_PE
CANTSEE_PE
CLR_AZURE
CLR_PURPLE
ELEMENTARY_TT
EXCEPT_PE
ITGONE_PE
JUNKAFTER_PE
MMULTI_PE
MULTI_PE
NOTHELD_PE
NOTHING_PE
NUMBER_PE
SCENERY_PE
SCOPE_TT
STUCK_PE
TALKING_REASON
TOOFEW_PE
TOOLIT_PE
UPTO_PE
USE_MODULES
VAGUE_PE
VERB_PE

These constants exist in PunyInform only.

Constant Name
MAX_FLOATING_OBJECTS
MAX_INPUT_CHARS

Constant Name
MAX_INPUT_WORDS
MAX_MULTIPLE_OBJECTS
MAX_SCOPE
MAX_WHICH_OBJECTS
WIN_ALL
WIN_MAIN
WIN_STATUS

Appendix F: Grammar

Here are the standard verbs defined in the library.

Verbs
answer say speak
ask
attack break crack destroy
climb scale
close cover shut
cut chop prune slice
dig
drink sip swallow
drop discard throw
eat
enter cross
examine x
exit out outside
fill
get
give feed offer pay
go run walk
insert
inventory inv i
jump hop skip
leave
listen hear
lock
look l
open uncover unwrap
pick
pull drag
push clear move press shift
put place
read

Verbs
remove
rub clean dust polish scrub
search
shed disrobe doff
shout scream yell
show display present
sit lie
smell sniff
stand
switch
take carry hold
tell
tie attach fasten fix
touch feel fondle grope
turn rotate screw twist unscrew
unlock
wait z
wear don

This set of extended verbs are not included by default, but can be added by defining OPTIONAL_EXTENDED_VERBSET.

Verbs	Comment
blow	OPTIONAL_EXTENDED_VERBSET
bother curses darn drat	OPTIONAL_EXTENDED_VERBSET
burn light	OPTIONAL_EXTENDED_VERBSET
buy purchase	OPTIONAL_EXTENDED_VERBSET
consult	OPTIONAL_EXTENDED_VERBSET
empty	OPTIONAL_EXTENDED_VERBSET
in inside	OPTIONAL_EXTENDED_VERBSET
kiss embrace hug	OPTIONAL_EXTENDED_VERBSET
no	OPTIONAL_EXTENDED_VERBSET
peel	OPTIONAL_EXTENDED_VERBSET
pray	OPTIONAL_EXTENDED_VERBSET
pry prise prize lever jemmy force	OPTIONAL_EXTENDED_VERBSET
set adjust	OPTIONAL_EXTENDED_VERBSET
shit damn fuck sod	OPTIONAL_EXTENDED_VERBSET
sing	OPTIONAL_EXTENDED_VERBSET
sleep nap	OPTIONAL_EXTENDED_VERBSET
sorry	OPTIONAL_EXTENDED_VERBSET
squeeze squash	OPTIONAL_EXTENDED_VERBSET
swim dive	OPTIONAL_EXTENDED_VERBSET

Verbs	Comment
swing	OPTIONAL_EXTENDED_VERBSET
taste	OPTIONAL_EXTENDED_VERBSET
think	OPTIONAL_EXTENDED_VERBSET
transfer	OPTIONAL_EXTENDED_VERBSET
wake awake awaken	OPTIONAL_EXTENDED_VERBSET
wave	OPTIONAL_EXTENDED_VERBSET
yes y	OPTIONAL_EXTENDED_VERBSET

This set of PunyInform debug verbs are not included by default, but can be added by defining DEBUG.

Verbs	Comment
actions	DEBUG
debug	DEBUG
gonear	DEBUG
pronouns nouns	DEBUG
purloin	DEBUG
random	DEBUG
routines messages	DEBUG
scope	DEBUG
timers daemons	DEBUG
tree	DEBUG

These debug verbs defined in the library are not supported by PunyInform.

Verbs	Comment
abstract	not in PunyInform
changes	not in PunyInform
goto	not in PunyInform
showobj	not in PunyInform
showverb	not in PunyInform
trace	not in PunyInform

These are the meta verbs. Some are only included when OPTIONAL_EXTENDED_METAVERBS is defined, and some are not defined if NO_PLACES is defined.

Verbs	Comment
brief normal	
fullscore full	

Verbs	Comment
noscript unscript	OPTIONAL_EXTENDED_METAVERBS
notify	
objects	OPTIONAL_EXTENDED_METAVERBS and not NO_PLACES
places	OPTIONAL_EXTENDED_METAVERBS and not NO_PLACES
quit q die	
recording	OPTIONAL_EXTENDED_METAVERBS
replay	OPTIONAL_EXTENDED_METAVERBS
restart	
restore	
save	
score	
script transcript	OPTIONAL_EXTENDED_METAVERBS
superbrief short	
verify	OPTIONAL_EXTENDED_METAVERBS
verbose long	
version	

Appendix G: Runtime Errors

PunyInform has code to check for runtime errors, to detect programming errors either in the library or the game code. To make sure you have the best chances of finding problems in your code during the development and test stages, make it a habit to compile your game with DEBUG defined and RUNTIME_ERRORS either set to 2 or unset (it defaults to 2 if DEBUG is set). This enables all runtime error checking and reporting. Once the game has gone through serious testing and is ready for release, you may want to consider setting RUNTIME_ERRORS to 0, to skip as much as possible of the runtime error checking, especially if you intend to release the game for 8-bit platforms, as this makes the game a little faster and smaller. Before doing so, you should read through the error messages below and decide if you've done enough to make sure that error won't occur. Note that there is some guidance in Game Author's Guide on how to make sure your game works as well as possible, including setting the library limit constants such as MAX_SCOPE.

The following runtime checks are always active, regardless of the settings of DEBUG and RUNTIME_ERRORS:

- Check for incorrect message number, generating PunyInform error #4 (see below).
- Check if scope array is full. If RUNTIME_ERRORS is > 0, this condition prints PunyInform error #3 (see below). If RUNTIME_ERRORS = 0, nothing is printed, but objects that don't fit in the scope array are silently ignored. This means the player may be unable to address some of the objects that should be in scope, and these objects won't be checked for `react_before`, `react_after` or `each_turn` routines.
- Check if timer/daemon array is full. If RUNTIME_ERRORS is > 0, this condition prints PunyInform error #1 (see below). If RUNTIME_ERRORS = 0, nothing is printed, but the timer/daemon doesn't start.

A few of the error conditions below are only checked in DEBUG mode. These are denoted with *DEBUG*.

<!– **PunyInform error #1: Too many timers/daemons**

Too many timers/daemons are running at once. Increase MAX_TIMERS.

<!– **PunyInform error #2: Object lacks required property**

Check that all objects with timers have the time_left property, all door objects have the door_dir property, and all non-simple doors have the door_to property.

<!– **PunyInform error #3: Scope full**

PunyInform has run out of room to keep track of objects in scope. Increase MAX_SCOPE. See Game Author's Guide for advice on setting MAX_SCOPE.

<!– **PunyInform error #4: Unknown message#**

_PrintMsg was called with an unknown message number. If this happens, it's either due to a library bug, or the game code makes an incorrect call to PrintMsg or _PrintMsg.

<!– **PunyInform error #5: GoSub called with invalid direction property**

If this happens, it's most likely due to a library bug.

<!– **PunyInform error #6: Too many floating objects**

PunyInform has run out of room to store the floating objects (objects that provide the `found_in` property) in the game. Increase MAX_FLOATING_OBJECTS. E.g. if your game has a total of 50 floating objects, define `Constant MAX_FLOATING_OBJECTS 55;` (leaving some room for floating objects defined by library extensions). If the game starts, with RUNTIME_ERRORS > 0, and no error message is printed, you have set the limit high enough.

<!– **PunyInform error #7: DirPropToFakeObj called with non-dirprop**

Check the argument in your calls to DirPropToFakeObj. A correct call looks something like: `my_fake_dir = DirPropToFakeObj(e_to);`

<!– **PunyInform error #8: FakeObjToDirProp called with non-fakeobj**

Check the argument in your calls to FakeObjToDirProp. A correct call looks something like: `my_dir_prop = FakeObjToDirProp(FAKE_E_OBJ);`

<!– **PunyInform error #9: ChooseObjectsFinal_(Pick or Discard) called with nonexistent array index**

Your code attempted to pick or discard an array index which doens't exist, e.g. the array holds objects 0-3 and you asked to discard index 4.

Cheap_scenery error #1: A cheap scenery property of [OBJECT] extends beyond property length - check entries with 3+ words

You've made a mistake in a cheap scenery property, which means the cheap_scenery extension is trying to read the last entry but ends up past the end of the list. Note that this could be either in the `cheap_scenery` property,

or a property refered to by a CS_ADD_LIST entry. Example of an incorrect cheap scenery list which would lead to this error being printed:

```
Object Room "Room"
  with
    description "There are walls.",
    cheap_scenery
      5 'walls' "Very nice walls.", ! We state that there are five words, but only give one
  has light;
```

Cheap_scenery error #2: Element [N] in a cheap scenery property of [OBJECT] is part of a CS_ADD_LIST entry and should be a valid object ID but isn't

Right after a CS_ADD_LIST (= 101) element, there must be a valid object ID and then a valid property ID. If this message is printed, no valid object ID was found. Note that this could be either in the `cheap_scenery` property, or a property referred to by a CS_ADD_LIST entry.

Cheap_scenery error #3: Element [N] in a cheap scenery property of [OBJECT] is not a string or routine

The value in this position must be a reaction string/routine, and it's not a string or routine. Note that this could be either in the `cheap_scenery` property, or a property referred to by a CS_ADD_LIST entry.

Cheap_scenery error #4: Element [N] in a cheap scenery property of [OBJECT] should be a parse_name routine but isn't

Right after a CS_PARSE_NAME (= 100) element, a routine is expected. If this message is printed, the element found isn't a routine. Note that this could be either in the `cheap_scenery` property, or a property referred to by a CS_ADD_LIST entry.

Cheap_scenery error #5: First element of entry, at position [N] in a cheap scenery property of [OBJECT] should be a value 1-99, or a vocabulary word, but is 0

The very first element of an entry (the part of a cheap scenery list that deals with one cheap scenery item) must always be 1-99 or a vocabulary word. If this is printed, it was found to be 0. Note that this could be either in the `cheap_scenery` property, or a property referred to by a CS_ADD_LIST entry.

Flags error #1: Tried to use flag [N], but the highest flag number is [N]

You tried to set, clear or read a flag with too high a number. Increase FLAG_COUNT.

Flags error #2: Warning: Use of flag 0 is deprecated, and will not work as second or third argument to flag functions! *DEBUG* You tried to set, clear or read flag 0. This used to work, but has been deprecated. Flag 1

is now the first flag. When this warning comes up, it means the operation has still gone through, but you need to stop using flag 0. If you happen to use flag 0 as the second or third argument to a flag function, e.g. `SetFlag(5,0,1);` it will just silently fail.

Quote_box error #1: Tried to print quote wider than [N] characters

This is a hard limit of the quote box extension. Reformat your quote to be less wide.

Talk_menu error #1: Object [N] ([NAME]) [, which has talk topics in talk_array,] doesn't provide talk_start

This object either has topics in talk_array, or a call has been made to RunTalk, to start a conversation with this object, and the object lacks the talk_start property. You need to add it to the object.

Talk_menu error #2: Action [N] was not understood for [NAME]

The topic you just chose has a list of actions to be performed when it's chosen. Each action must be a string, a routine, a flag to be set or a topic to be activated. One of these actions isn't recognized as a valid value of any of these kinds.

Talk_menu error #3: Could not (in-)activate topic [N] for NPC [NAME] *DEBUG*

Activating/inactivativing this topic failed, typically because no topic matching the ID was found. E.g. Linda has topics 300-320 and you've tried to activate topic 350.