

Printing Template

B/print

Purpose

To manage the line skips which space paragraphs out, and to handle the printing of names of objects, pieces of text and numbers.

B/print. §1 Paragraph Control; §2 State; §3 Say Number; §4 Prompt; §5 Boxed Quotations; §6 Score Notification; §7 Status Line; §8 Status Line Utilities; §9 Banner; §10 Print Decimal Number; §11 Print English Number; §12 Print Text; §13 Print Or Run; §14 Short Name Storage; §15 Object Names I; §16 Standard Name Printing Rule; §17 Object Names II; §18 Object Names III; §19 Say One Of

§1. Paragraph Control. Ah, yes: the paragraph breaking algorithm. In *T_EX: The Program*, Donald Knuth writes at §768: “It’s sort of a miracle whenever `\halign` and `\valign` work, because they cut across so many of the control structures of T_EX.” It’s sort of a miracle whenever Inform 7’s paragraph breaking system works, too. Most users probably imagine that it’s implemented by having I7 look at where the cursor currently is (at the start of a line or not) and whether a line has just been skipped. In fact, the virtual machines simply do not offer facilities like that, and so we have to use our own book-keeping. Given the huge number of ways in which text can be printed, this is a delicate business. For some years now, “spacing bugs” – those where a spurious extra skipped line appears in a paragraph break, or where, conversely, no line is skipped at all – have been the least welcome in the Inform bugs database.

The basic method is to set `say__p`, the paragraph flag, when we print any matter; every so often we reach a “divide paragraph” point – for instance when one rule has finished and before another is about to start – and at those positions we look for `say__p`, and print a skipped line (and clear `say__p` again) if we find it. Thus:

```
> WAIT
The clock ticks ominously. ...first rule
    ...skipped line printed at a Divide Paragraph point
Mme Tourmalet rises from her chair and slips out. second rule
    ...skipped line printed at a Divide Paragraph point
>
```

A divide paragraph point occurs between any two rules in an action rulebook, but not an activity rulebook: many activities exist to print text, such as the names of objects, and there would be wild spacing accidents if paragraphs were divided there. Inform places DPPs elsewhere, too: and the text substitution “[conditional paragraph break]” allows the user to place one anywhere.

A traditional layout convention handed down from Infocom makes an exception of the first paragraph to appear after the prompt, but only in one situation. Ordinarily, the first paragraph of any turn appears straight after the prompt:

```
> EXAMINE DOG
Mme Tourmalet’s borzoi looks as if it means fashion, not business.
```

The command is echoed on screen as the player types, but this doesn’t set the paragraph flag, which is still clear when the text “Mme Tourmalet’s...” begins to be printed. The single exception occurs when the command calls for the player to go to a new location, when a skipped line is printed before the room description for the new room. Thus:

```
> SOUTH
    ...the “going look break”
Rocky Beach
```

(Note that this is not inherent in the looking action:

```
> LOOK
Rocky Beach
```

...which obeys the standard paragraphing conventions.)

So much for automatic paragraph breaks. However, we need a variety of different ways explicitly to control paragraphs, in order to accommodate traditional layout conventions handed down from Infocom.

The simplest exceptional kind of paragraph break is a “command clarification break”, in which a single new-line is printed but there is no skipped line: as the name implies, it’s traditionally used when a command such as OPEN DOOR is clarified. For example:

```
(first unlocking the oak door) ...now a command clarification break
You open the oak door.
```

This is not quite the same thing as a “run paragraph on” break, in which we also deliberately suppress the skipped line, but make an exception for the skipped line which ought to appear last before the prompt: the idea is to merge two or more paragraphs together.

```
> TAKE ALL
marmot: Taken. ...we run paragraph on here
weasel: Taken. ...and also here
      ...despite which the final skip does occur
> ...before the next prompt
```

A more complicated case is “special look spacing”, used for the break which occurs after the (boldface) short name of a room description is printed. This is tricky because it is sometimes followed directly by a long description, and we don’t want a skipped line:

```
Villa Christiane ...a special look spacing break
The walled garden of a villa in Cap d’Agde.
      ...a Divide Paragraph break
Mme Tourmalet’s borzoi lazes in the long grass.
```

But sometimes it is followed directly by a subsequent paragraph, and again we want no skip:

```
Villa Christiane ...a special look spacing break
Mme Tourmalet’s borzoi lazes in the long grass.
```

And sometimes it is the only content of the room description and is followed only by the prompt:

```
Villa Christiane ...a special look spacing break
      ...a break inserted before the prompt
>
```

To recap, we have five kinds of paragraph break:

- (a) Standard breaks at “divide paragraph points”, used between rules.
- (b) The “going look break”, used before the room description after going to a new room.
- (c) A “command clarification break”, used after text clarifying a command.
- (d) A “run paragraph on” break, used to merge multiple paragraphs into a single block of text.
- (e) The “special look spacing” break, used after the boldface headline of a room description.

We now have to implement all of these behaviours. The code, while very simple, is highly prone to behaving unexpectedly if changes are made, simply because of the huge number of circumstances in which paragraphs are printed: so change nothing without very careful testing.

§2. **State.** The current state is stored in a combination of two global variables:

- (1) `say__p`, the “say paragraph” flag, which is set if a paragraph break needs to be printed before the next text can begin;
- (2) `say__pc`, originally named as the “paragraph completed” flag, but which is now a bitmap:
 - (2a) `PARA_COMPLETED` is set if a standard paragraph break has been made since the last time the flag was cleared;
 - (2b) `PARA_PROMPTSKIP` is set to indicate that the current printing position does not follow a skipped line, and that further material is expected which will run on from the previous paragraph, but that if no further material turns up then a skipped line would be needed before the next prompt;
 - (2c) `PARA_SUPPRESSPROMPTSKIP` is set to indicate that, despite `PARA_PROMPTSKIP` being set, no skipped line is needed before the prompt after all;
 - (2d) `PARA_NORULEBOOKBREAKS` suppresses divide paragraph points in between rules in rulebooks; it treats all rulebooks, and in particular action rulebooks, the way activity rulebooks are treated. (The flag is used for short periods only and never across turn boundaries, prompts and so on.)
 - (2e) `PARA_CONTENTEXPECTED` is set after a paragraph division as a signal that if any contents looks likely to be printed soon then `say__p` needs to be set, because a successor paragraph will then have started. This is checked by calling `ParaContent()` – while it’s slow to have to call this routine so often, that’s better than compiling inline code with the same effect, because minimising compiled code size is more important, and speed is never a big deal when printing.

Not all printing is to the screen: sometimes the output is to a file, or to memory, and in that case we want to start the switched output at a clear paragraphing state and then go back to the screen afterwards without any sign of change. The correct way to do this is to push the `say__p` and `say__pc` variables onto the VM stack and call `ClearParagraphing()` before starting to print to the new stream, and then pull the variables back again before resuming printing to the old stream.

In no other case should any code alter `say__pc` except via the routines below.

```
!Constant TRACE_I7_SPACING;
[ ClearParagraphing;
  say__p = 0; say__pc = 0;
];
[ DivideParagraphPoint;
  #ifdef TRACE_I7_SPACING; print "[DPP", say__p, say__pc, "]; #endif;
  if (say__p) {
    new_line; say__p = 0; say__pc = say__pc | PARA_COMPLETED;
    if (say__pc & PARA_PROMPTSKIP) say__pc = say__pc - PARA_PROMPTSKIP;
    if (say__pc & PARA_SUPPRESSPROMPTSKIP) say__pc = say__pc - PARA_SUPPRESSPROMPTSKIP;
  }
  #ifdef TRACE_I7_SPACING; print "[-->", say__p, say__pc, "]; #endif;
  say__pc = say__pc | PARA_CONTENTEXPECTED;
];
[ ParaContent;
  if (say__pc & PARA_CONTENTEXPECTED) {
    say__pc = say__pc - PARA_CONTENTEXPECTED;
    say__p = 1;
  }
];
[ GoingLookBreak;
  if (say__pc & PARA_COMPLETED == 0) new_line;
  ClearParagraphing();
];
[ CommandClarificationBreak;
```

```

    new_line;
    ClearParagraphing();
];
[ RunParagraphOn;
    #ifdef TRACE_I7_SPACING; print "[RPO", say__p, say__pc, "]; #endif;
    say__p = 0;
    say__pc = say__pc | PARA_PROMPTSKIP;
    say__pc = say__pc | PARA_SUPPRESSPROMPTSKIP;
];
[ SpecialLookSpacingBreak;
    #ifdef TRACE_I7_SPACING; print "[SLS", say__p, say__pc, "]; #endif;
    say__p = 0;
    say__pc = say__pc | PARA_PROMPTSKIP;
];
[ EnsureBreakBeforePrompt;
    if ((say__p) ||
        ((say__pc & PARA_PROMPTSKIP) && ((say__pc & PARA_SUPPRESSPROMPTSKIP)==0)))
        new_line;
    ClearParagraphing();
];
[ PrintSingleParagraph matter;
    say__p = 1;
    say__pc = say__pc | PARA_NORULEBOOKBREAKS;
    PrintText(matter);
    DivideParagraphPoint();
    say__pc = 0;
];

```

§3. Say Number. The global variable `say__n` is set to the numerical value of any quantity printed, and this is used for the text substitution “[s]”, so that “You have been awake for [turn count] turn[s].” will expand correctly.

```

[ STextSubstitution;
    if (say__n ~= 1) print "s";
];

```

§4. Prompt. This is the text printed just before we wait for the player’s command: it prompts him to type.

```

[ PrintPrompt i;
    style roman;
    EnsureBreakBeforePrompt();
    PrintText( (+ command prompt +) );
    ClearBoxedText();
    ClearParagraphing();
    enable_rte = true;
];

```

§5. Boxed Quotations. These appear once only, and happen outside of the paragraphing scheme: they are normally overlaid as windows on top of the regular text. We can request one at any time, but it will appear only at prompt time, when the screen is fairly well guaranteed not to be scrolling. (Only fairly well since it's just possible that *Border Zone*-like tricks with real-time play might be going on, but whatever happens, there is at least a human-appreciable pause in which the quotation can be read before being taken away again.)

```
Global pending_boxed_quotation; ! a routine to overlay the quotation on screen
[ DisplayBoxedQuotation Q;
    pending_boxed_quotation = Q;
];
[ ClearBoxedText i;
    if (pending_boxed_quotation) {
        for (i=0: Runtime_Quotations_Displayed-->i: i++)
            if (Runtime_Quotations_Displayed-->i == pending_boxed_quotation) {
                pending_boxed_quotation = 0;
                return;
            }
        Runtime_Quotations_Displayed-->i = pending_boxed_quotation;
        ClearParagraphing();
        pending_boxed_quotation();
        ClearParagraphing();
        pending_boxed_quotation = 0;
    }
];
```

§6. Score Notification. This doesn't really deserve to be at I6 level at all, but since for traditional reasons we need to use conditional compilation on NO_SCORING, and since we want a fancy text style for Glulx, ...

```
[ NotifyTheScore;
#ifdef NO_SCORING;
    if (notify_mode == 1) {
        DivideParagraphPoint();
        VM_Style(NOTE_VMSTY);
        print "["; L_M(##Miscellany, 50, score-last_score); print ".]~";
        VM_Style(NORMAL_VMSTY);
    }
#endif;
];
```

§7. **Status Line.** Status line printing happens on the upper screen window, and outside of the paragraph control system.

Support for version 6 of the Z-machine is best described as grudging. It requires a heavily rewritten DrawStatusLine equivalent, to be found in “ZMachine.i6t”.

```
#Ifdef TARGET_ZCODE;
#Iftrue (#version_number == 6);
[ DrawStatusLine; Z6_DrawStatusLine(); ];
#Endif;
#Endif;

#Ifndef DrawStatusLine;
[ DrawStatusLine width posb;
  @push say__p; @push say__pc;
  BeginActivity(CONSTRUCTING_STATUS_LINE_ACT);
  VM_StatusLineHeight(1); VM_MoveCursorInStatusLine(1, 1);
  if (statuswin_current) {
    width = VM_ScreenWidth(); posb = width-15;
    spaces width;
    ClearParagraphing();
    if (ForActivity(CONSTRUCTING_STATUS_LINE_ACT) == false) {
      VM_MoveCursorInStatusLine(1, 2);
      switch(metaclass(left_hand_status_line)) {
        String: print (string) left_hand_status_line;
        Routine: left_hand_status_line();
      }
      VM_MoveCursorInStatusLine(1, posb);
      switch(metaclass(right_hand_status_line)) {
        String: print (string) right_hand_status_line;
        Routine: right_hand_status_line();
      }
    }
    VM_MoveCursorInStatusLine(1, 1); VM_MainWindow();
  }
  ClearParagraphing();
  EndActivity(CONSTRUCTING_STATUS_LINE_ACT);
  @pull say__pc; @pull say__p;
];
#Endif;
```

§8. **Status Line Utilities.** Two convenient routines for the default values of `right_hand_status_line` and `left_hand_status_line` respectively. `SL_Location` also implements the text substitution “[player’s surroundings]”.

```
[ SL_Score_Moves;
  if (not_yet_in_play) return;
  #ifdef NO_SCORING; print sline2; #ifndef; print sline1, "/", sline2; #endif;
];

[ SL_Location;
  if (not_yet_in_play) return;
  if (location == thedark) {
    BeginActivity(PRINTING_NAME_OF_DARK_ROOM_ACT);
    if (ForActivity(PRINTING_NAME_OF_DARK_ROOM_ACT) == false)
      L_M(##Miscellany, 71);
    EndActivity(PRINTING_NAME_OF_DARK_ROOM_ACT);
  } else {
    FindVisibilityLevels();
    if (visibility_ceiling == location) print (name) location;
    else print (The) visibility_ceiling;
  }
];
```

§9. **Banner.** Note that NI always compiles `Story` and `Headline` texts, but does not always compile a `Story_Author`.

```
[ Banner;
BeginActivity(PRINTING_BANNER_TEXT_ACT);
if (ForActivity(PRINTING_BANNER_TEXT_ACT) == false) {
  VM_Style(HEADER_VMSTY);
  print (string) Story;
  VM_Style(NORMAL_VMSTY);
  new_line;
  print (string) Headline;
  #ifdef Story_Author;
  print " by ", (string) Story_Author;
  #endif; ! Story_Author
  new_line;
  VM_Describe_Release();
  print " / Inform 7 build ", (string) NI_BUILD_COUNT, " ";
  print "(I6/v"; inversion;
  print " lib ", (string) LibRelease, ") ";
  #ifdef STRICT_MODE;
  print "S";
  #endif; ! STRICT_MODE
  #ifdef DEBUG;
  print "D";
  #endif; ! DEBUG
  new_line;
}
EndActivity(PRINTING_BANNER_TEXT_ACT);
];
```

§10. Print Decimal Number. `DecimalNumber` is a trivial function which just prints a number, in decimal digits. It is left over from the I6 library's support routines for Glulx, where it was intended as a stub to pass to the Glulx `Glulx_PrintAnything` routine (which I7 does not use). In I7, however, it's also used as the default printing routine for new kinds of value.

```
[ DecimalNumber num; print num; ];
```

§11. Print English Number. Another traditional name, this: in fact it prints the number as text in whatever is the current language of play.

```
[ EnglishNumber n; LanguageNumber(n); ];
```

§12. Print Text. The routine for printing an I7 "text" value, which might text with or without substitutions.

```
[ PrintText x;
  if (x ofclass String) print (string) x;
  if (x ofclass Routine) (x)();
];
[ I7_String x; PrintText(x); ]; ! An alternative name now used only by extensions
```

§13. Print Or Run. This utility remains from the old I6 library: it essentially treats a property as textual and prints it where possible. Where the `no_break` flag is set, we expect the text to form only a small part of a paragraph, and it's inappropriate to break here: for instance, for printing the "printed name" of an object. Where the flag is clear, however, the text is expected to form its own paragraph.

Where `PrintOrRun` is used in breaking mode, which is only for a very few properties in I7 (indeed at present only `initial` and `description`), the routine called is given the chance to decide whether to print or not. It should return `true` or `false` according to whether it did so; this allows us to divide the paragraph or not accordingly.

```
[ PrintOrRun obj prop no_break routine_return_value;
  !print "( ", obj, ".", prop, ";", say__p, say__pc, " )";
  if (prop == 0) {
    print (name) prop; routine_return_value = true;
  } else {
    switch (metaclass(obj.prop)) {
      nothing:
        routine_return_value = false;
      String:
        if (obj.prop == EMPTY_TEXT_VALUE) break;
        print (string) obj.prop; !if (no_break == false) new_line;
        routine_return_value = true;
      Routine:
        routine_return_value = RunRoutines(obj, prop);
        !print "[ ", routine_return_value, " ]";
    }
  }
  if (routine_return_value) {
    say__p = 1;
    if (no_break == false) {
      new_line;
    }
  }
];
```

```

        !print "(DP->", say__p, say__pc, ")";
        DivideParagraphPoint();
        !print "(to", say__p, say__pc, ")";
    }
}
!print "(-->", say__p, say__pc, ")";
return routine_return_value;
];

```

§14. **Short Name Storage.** None of the following functions should be called for the Z-machine if the short name exceeds the size of the following buffer: whereas the Glulx implementation of `VM_PrintToBuffer` will safely truncate overlong text, that's impossible for the Z-machine, and horrible results will follow.

`CPrintOrRun` is a variation on `PrintOrRun`, simplified by not needing to handle entire paragraphs (so, no fuss about dividing) but complicated by having to capitalise the first letter. We do this by writing to the buffer and then altering the first character.

```

Array StorageForShortName buffer 250;
[ CPrintOrRun obj prop v length i;
    if ((obj ofclass String or Routine) || (prop == 0))
        VM_PrintToBuffer (StorageForShortName, 160, obj);
    else {
        if (obj.prop == NULL) rfalse;
        if (metaclass(obj.prop) == Routine or String)
            VM_PrintToBuffer(StorageForShortName, 160, obj, prop);
        else return RunTimeError(2, obj, prop);
    }
    length = StorageForShortName-->0;
    StorageForShortName->WORDSIZE = VM_LowerToUpperCase(StorageForShortName->WORDSIZE);
    for (i=WORDSIZE: i<length+WORDSIZE: i++) print (char) StorageForShortName->i;
    if (i>WORDSIZE) say__p = 1;
    return;
];
[ Cap str nocaps;
    if (nocaps) print (string) str;
    else CPrintOrRun(str, 0);
];

```

§15. **Object Names I.** We now begin the work of printing object names. In the lowest level of this process we print just the name itself (without articles attached), and we do it by carrying out an activity.

```

[ PSN__ o;
    if (o == 0) { print (string) NOTHING__TX; rtrue; }
    switch (metaclass(o)) {
        Routine: print "<routine ", o, ">"; rtrue;
        String:  print "<string ~", (string) o, "~>"; rtrue;
        nothing: print "<illegal object number ", o, ">"; rtrue;
    }
    CarryOutActivity(PRINTING_THE_NAME_ACT, o);
];

```

§16. **Standard Name Printing Rule.** In its initial state, the “printing the name of” activity has just one rule: the following “for” rule.

```
Global caps_mode = false;
[ STANDARD_NAME_PRINTING_R obj;
  obj = parameter_object;
  if (obj == 0) {
    print (string) NOTHING__TX; return;
  }
  switch (metaclass(obj)) {
    Routine: print "<routine ", obj, ">"; return;
    String:  print "<string ~", (string) obj, "~>"; return;
    nothing: print "<illegal object number ", obj, ">"; return;
  }
  if (obj == player) {
    if (indef_mode == NULL && caps_mode) print (string) YOU__TX;
    else print (string) YOURSELF__TX;
    return;
  }
  #Ifdef LanguagePrintShortName;
  if (LanguagePrintShortName(obj)) return;
  #Endif; ! LanguagePrintShortName
  if (indef_mode && obj.&short_name_indef ~= 0 &&
    PrintOrRun(obj, short_name_indef, true) ~= 0) return;
  if (caps_mode &&
    obj.&cap_short_name ~= 0 && PrintOrRun(obj, cap_short_name, true) ~= 0) {
    caps_mode = false;
    return;
  }
  if (obj.&short_name ~= 0 && PrintOrRun(obj, short_name, true) ~= 0) return;
  print (object) obj;
];
```

§17. **Object Names II.** The second level of the system for printing object names handles the placing of articles in front of them: *the* red herring, *an* elephant, *Some* bread. The following routine allows us to choose:

- (a) *obj*, the object whose name is to be printed;
- (b) *acode*, the kind of article needed: capitalised definite (0), lower case uncapitalised definite (1), or uncapitalised indefinite (2);
- (c) *pluralise*, a flag forcing to a plural form (e.g., “some” being the pluralised form of an indefinite article in English);
- (d) *capitalise*, a flag forcing us to capitalise the article – it’s by setting this that we can achieve the fourth option missing from (b), viz., capitalised indefinite. (All of this is a legacy design from a time when the I6 library did not support capitalised indefinite articles.)

The routine then looks after issues such as which contraction form to use: for instance, in English, whether to use “a” or “an” for the indefinite singular depends on the text of the object’s name.

```
Global short_name_case;
[ PrefaceByArticle obj acode pluralise capitalise i artform findout artval;
  if (obj provides articles) {
    artval=(obj.&articles)-->(acode+short_name_case*LanguageCases);
    if (capitalise)
```

```

        print (Cap) artval, " ";
    else
        print (string) artval, " ";
    if (pluralise) return;
    print (PSN__) obj; return;
}
i = GetGNAOfObject(obj);
if (pluralise) {
    if (i < 3 || (i >= 6 && i < 9)) i = i + 3;
}
i = LanguageGNAsToArticles-->i;
artform = LanguageArticles
    + 3*WORDSIZE*LanguageContractionForms*(short_name_case + i*LanguageCases);
#Iftrue (LanguageContractionForms == 2);
if (artform-->acode ~= artform-->(acode+3)) findout = true;
#Endif; ! LanguageContractionForms
#Iftrue (LanguageContractionForms == 3);
if (artform-->acode ~= artform-->(acode+3)) findout = true;
if (artform-->(acode+3) ~= artform-->(acode+6)) findout = true;
#Endif; ! LanguageContractionForms
#Iftrue (LanguageContractionForms == 4);
if (artform-->acode ~= artform-->(acode+3)) findout = true;
if (artform-->(acode+3) ~= artform-->(acode+6)) findout = true;
if (artform-->(acode+6) ~= artform-->(acode+9)) findout = true;
#Endif; ! LanguageContractionForms
#Iftrue (LanguageContractionForms > 4);
findout = true;
#Endif; ! LanguageContractionForms
#Ifdef TARGET_ZCODE;
if (standard_interpreter ~= 0 && findout) {
    StorageForShortName-->0 = 160;
    @output_stream 3 StorageForShortName;
    if (pluralise) print (number) pluralise; else print (PSN__) obj;
    @output_stream -3;
    acode = acode + 3*LanguageContraction(StorageForShortName + 2);
}
#Ifnot; ! TARGET_GLULX
if (findout) {
    if (pluralise)
        Glulx_PrintAnyToArray(StorageForShortName, 160, EnglishNumber, pluralise);
    else
        Glulx_PrintAnyToArray(StorageForShortName, 160, PSN__, obj);
    acode = acode + 3*LanguageContraction(StorageForShortName);
}
#Endif; ! TARGET_
Cap (artform-->acode, ~~capitalise); ! print article
if (pluralise) return;
print (PSN__) obj;
];

```

§18. **Object Names III.** The routines accessible from outside this segment.

```
[ IndefArt obj i;
  if (obj == 0) { print (string) NOTHING__TX; rtrue; }
  i = indef_mode; indef_mode = true;
  if (obj has proper) { indef_mode = NULL; print (PSN__) obj; indef_mode = i; return; }
  if ((obj provides article) && (obj.article ~= EMPTY_TEXT_VALUE)) {
    PrintOrRun(obj, article, true); print " ", (PSN__) obj; indef_mode = i;
    return;
  }
  PrefaceByArticle(obj, 2); indef_mode = i;
];

[ CIndefArt obj i;
  if (obj == 0) { CPrintOrRun(NOTHING__TX, 0); rtrue; }
  i = indef_mode; indef_mode = true;
  if (obj has proper) {
    indef_mode = NULL;
    caps_mode = true;
    print (PSN__) obj;
    indef_mode = i;
    caps_mode = false;
    return;
  }
  if ((obj provides article) && (obj.article ~= EMPTY_TEXT_VALUE)) {
    CPrintOrRun(obj, article); print " ", (PSN__) obj; indef_mode = i;
    return;
  }
  PrefaceByArticle(obj, 2, 0, 1); indef_mode = i;
];

[ DefArt obj i;
  i = indef_mode; indef_mode = false;
  if ((~~obj ofclass Object) || obj has proper) {
    indef_mode = NULL; print (PSN__) obj; indef_mode = i;
    return;
  }
  PrefaceByArticle(obj, 1); indef_mode = i;
];

[ CDefArt obj i;
  i = indef_mode; indef_mode = false;
  if ((obj ofclass Object) && (obj has proper || obj == player)) {
    indef_mode = NULL;
    caps_mode = true;
    print (PSN__) obj;
    indef_mode = i;
    caps_mode = false;
    return;
  }
  if ((~~obj ofclass Object) || obj has proper) {
    indef_mode = NULL; print (PSN__) obj; indef_mode = i;
    return;
  }
  PrefaceByArticle(obj, 0); indef_mode = i;
];
```

```
[ PrintShortName obj i;
  i = indef_mode; indef_mode = NULL;
  PSN__(obj); indef_mode = i;
];
```

§19. **Say One Of.** These routines are described in the Extensions chapter of the Inform documentation.

```
[ I7_S00_PAR oldval count; if (count <= 1) return count; return random(count); ];
[ I7_S00_RAN oldval count v; if (count <= 1) return count;
  v = oldval; while (v == oldval) v = random(count); return v; ];
[ I7_S00_STI oldval count v; if (oldval) return oldval; return I7_S00_PAR(oldval, count); ];
[ I7_S00_CYC oldval count; oldval++; if (oldval > count) oldval = 1; return oldval; ];
[ I7_S00_STOP oldval count; oldval++; if (oldval > count) oldval = count; return oldval; ];
[ I7_S00_TAP oldval count tn rn c; if (count <= 1) return count; tn = count*(count+1)/2;
  rn = random(tn); for (c=1;c<=count;c++) { rn = rn - c; if (rn<=0) return (count-c+1); } ];
[ I7_S00_TRAN oldval count; if (oldval<count) return oldval+1;
  return count + 1 + I7_S00_RAN(oldval%(count+1), count); ];
[ I7_S00_TPAR oldval count; if (oldval<count) return oldval+1;
  return count + 1 + I7_S00_PAR(oldval%(count+1), count); ];
Array I7_S00_SHUF->32;
[ I7_S00_SHU oldval count sd ct v i j s ssd scope cc base;
  base = count+1;
  v = oldval%base; oldval = oldval/base; ct = oldval%base; sd = oldval/base;
  if (count > 32) return I7_S00_PAR(oldval, count);
  if (count <= 1) v = count;
  else {
    !print "^In v=", v, " ct=", ct, " sd=", sd, "^";
    cc = base*base;
    scope = MAX_POSITIVE_NUMBER/cc - cc - base;
    if (scope%2==0) scope--;
    if (scope<0) scope = -scope;
    !print "Scope = ", scope, "^";
    if (sd == 0) { sd = random(scope); ct=0; }
    for (i=0:i<count:i++) I7_S00_SHUF->i = i;
    ssd = sd;
    for (i=0:i<count-1:i++) {
      j = (sd)%(count-i)+i; sd = (sd*31973)+17; if (sd<0) sd=-sd;
      s = I7_S00_SHUF->j; I7_S00_SHUF->j = I7_S00_SHUF->i; I7_S00_SHUF->i = s;
    }
    !for (i=0:i<count:i++) print I7_S00_SHUF->i, " "; print "^";
    v = (I7_S00_SHUF->ct)+1;
    ct++; if (ct >= count) { ct = 0; ssd = 0; }
  }
  !print "Out v=", v, " ct=", ct, " ssd=", sd, "^";
  !print "Return ", v + ct*base + ssd*base*base, "^";
  return v + ct*base + ssd*base*base;
];
```