NAME

xo_format - content of format descriptors for xo_emit

DESCRIPTION

libxo uses format strings to control the rendering of data into various output styles, including *text*, *XML*, *JSON*, and *HTML*. Each format string contains a set of zero or more "field descriptions", which describe independent data fields. Each field description contains a set of "modifiers", a "content string", and zero, one, or two "format descriptors". The modifiers tell **libxo** what the field is and how to treat it, while the format descriptors are formatting instructions using printf(3)-style format strings, telling **libxo** how to format the field. The field description is placed inside a set of braces, with a colon (‘:’) after the modifiers and a slash (‘/’) before each format descriptors. Text may be intermixed with field descriptions within the format string.

The field description is given as follows:

```
’{’ [ role | modifier ]* [’,’ long-names ]* ’:’ [ content ]
    [ ’/’ field-format [ ’/’ encoding-format ]] ’}’
```

The role describes the function of the field, while the modifiers enable optional behaviors. The contents, field-format, and encoding-format are used in varying ways, based on the role. These are described in the following sections.

Braces can be escaped by using double braces, similar to "%%" in printf(3). The format string "{{braces}}" would emit "{braces}".

In the following example, three field descriptors appear. The first is a padding field containing three spaces of padding, the second is a label ("In stock"), and the third is a value field ("in-stock"). The in-stock field has a "%u" format that will parse the next argument passed to the xo_emit(3), function as an unsigned integer.

```
xo_emit("{P:   }{Lwc:In stock}{:in-stock/%u}\n", 65);
```

This single line of code can generate text ("In stock: 65\n"), XML ("<in-stock>65</in-stock>"), JSON (’"in-stock": 65’), or HTML (too lengthy to be listed here).

While roles and modifiers typically use single character for brevity, there are alternative names for each which allow more verbose formatting strings. These names must be preceded by a comma, and may follow any single-character values:

```
xo_emit("{L,white,colon:In stock}{,key:in-stock/%u}\n", 65);
```

Field Roles

Field roles are optional, and indicate the role and formatting of the content. The roles are listed below; only one role is permitted:

M	Name	Description
C	color	Field is a color or effect
D	decoration	Field is non-text (e.g. colon, comma)
E	error	Field is an error message
L	label	Field is text that prefixes a value
N	note	Field is text that follows a value
P	padding	Field is spaces needed for vertical alignment
T	title	Field is a title value for headings
U	units	Field is the units for the previous value field
V	value	Field is the name of field (the default)
W	warning	Field is a warning message
[	start-anchor	Begin a section of anchored variable-width text
]	stop-anchor	End a section of anchored variable-width text

EXAMPLE:

```
xo_emit("{L:Free}{D::}{P:   }{:free/%u} {U:Blocks}\n",
        free_blocks);
```

When a role is not provided, the "value" role is used as the default.

Roles and modifiers can also use more verbose names, when preceded by a comma:

EXAMPLE:

```
xo_emit("{,label:Free}{,decoration::}{,padding:   }"
        "{,value:free/%u} {,units:Blocks}\n",
        free_blocks);
```

The Color Role ({C:})

Colors and effects control how text values are displayed; they are used for display styles (TEXT and HTML).

```
xo_emit("{C:bold}{:value}{C:no-bold}\n", value);
```

Colors and effects remain in effect until modified by other "C"-role fields.

```
xo_emit("{C:bold}{C:inverse}both{C:no-bold}only inverse\n");
```

If the content is empty, the "reset" action is performed.

```
xo_emit("{C:both,underline}{:value}{C:}\n", value);
```

The content should be a comma-separated list of zero or more colors or display effects.

```
xo_emit("{C:bold,underline,inverse}All three{C:no-bold,no-inverse}\n");
```

The color content can be either static, when placed directly within the field descriptor, or a printf-style format descriptor can be used, if preceded by a slash ("/"):

```
xo_emit("{C:/%s%s}{:value}{C:}", need_bold ? "bold" : "",
    need_underline ? "underline" : "", value);
```

Color names are prefixed with either "fg-" or "bg-" to change the foreground and background colors, respectively.

```
xo_emit("{C:/fg-%s,bg-%s}{Lwc:Cost}{:cost/%u}{C:reset}\n",
    fg_color, bg_color, cost);
```

The following table lists the supported effects:

Name	**Description**
bg-xxxxx	Change background color
bold	Start bold text effect
fg-xxxxx	Change foreground color
inverse	Start inverse (aka reverse) text effect
no-bold	Stop bold text effect
no-inverse	Stop inverse (aka reverse) text effect
no-underline	Stop underline text effect
normal	Reset effects (only)
reset	Reset colors and effects (restore defaults)
underline	Start underline text effect

The following color names are supported:

Name
black
blue
cyan

default
green
magenta
red
white
yellow

The Decoration Role ({D:})

Decorations are typically punctuation marks such as colons, semi-colons, and commas used to decorate the text and make it simpler for human readers. By marking these distinctly, HTML usage scenarios can use CSS to direct their display parameters.

```
xo_emit("{D:(({:name}{D:))}\n", name);
```

The Gettext Role ({G:})

libxo supports internationalization (i18n) through its use of gettext(3). Use the "{G:}" role to request that the remaining part of the format string, following the "{G:}" field, be handled using **gettext**(). Since **gettext**() uses the string as the key into the message catalog, **libxo** uses a simplified version of the format string that removes unimportant field formatting and modifiers, stopping minor formatting changes from impacting the expensive translation process. A developer change such as changing "/%06d" to "/%08d" should not force hand inspection of all .po files.

The simplified version can be generated for a single message using the "xopo -s <text>" command, or an entire .pot can be translated using the "xopo -f <input> -o <output>" command.

```
xo_emit("{G:}Invalid token\n");
```

The {G:} role allows a domain name to be set. **gettext**() calls will continue to use that domain name until the current format string processing is complete, enabling a library function to emit strings using it's own catalog. The domain name can be either static as the content of the field, or a format can be used to get the domain name from the arguments.

```
xo_emit("{G:libc}Service unavailable in restricted mode\n");
```

The Label Role ({L:})

Labels are text that appears before a value.

```
xo_emit("{Lwc:Cost}{:cost/%u}\n", cost);
```

If a label needs to include a slash, it must be escaped using two backslashes, one for the C compiler and

one for **libxo**.

```
xo_emit("{Lc:Low\\/warn level}{:level/%s}\n", level);
```

The Note Role ({N:})

Notes are text that appears after a value.

```
xo_emit("{:cost/%u} {N:per year}\n", cost);
```

The Padding Role ({P:})

Padding represents whitespace used before and between fields. The padding content can be either static, when placed directly within the field descriptor, or a printf-style format descriptor can be used, if preceded by a slash ("/"):

```
xo_emit("{P:       }{Lwc:Cost}{:cost/%u}\n", cost);
xo_emit("{P:/30s}{Lwc:Cost}{:cost/%u}\n", "", cost);
```

The Title Role ({T:})

Titles are heading or column headers that are meant to be displayed to the user. The title can be either static, when placed directly within the field descriptor, or a printf-style format descriptor can be used, if preceded by a slash ("/"):

```
xo_emit("{T:Interface Statistics}\n");
xo_emit("{T:/%20.20s}{T:/%6.6s}\n", "Item Name", "Cost");
```

The Units Role ({U:})

Units are the dimension by which values are measured, such as degrees, miles, bytes, and decibels. The units field carries this information for the previous value field.

```
xo_emit("{Lwc:Distance}{:distance/%u}{Uw:miles}\n", miles);
```

Note that the sense of the ’w’ modifier is reversed for units; a blank is added before the contents, rather than after it.

When the XOF_UNITS flag is set, units are rendered in XML as the "units" attribute:

```
<distance units="miles">50</distance>
```

Units can also be rendered in HTML as the "data-units" attribute:

```
<div class="data" data-tag="distance" data-units="miles"
    data-xpath="/top/data/distance">50</div>
```

The Value Role ({V:} and {:})

The value role is used to represent the a data value that is interesting for the non-display output styles (XML and JSON). Value is the default role; if no other role designation is given, the field is a value. The field name must appear within the field descriptor, followed by one or two format descriptors. The first format descriptor is used for display styles (TEXT and HTML), while the second one is used for encoding styles (XML and JSON). If no second format is given, the encoding format defaults to the first format, with any minimum width removed. If no first format is given, both format descriptors default to "%s".

```
xo_emit("{:length/%02u}x{:width/%02u}x{:height/%02u}\n",
      length, width, height);
xo_emit("{:author} wrote
      author, poem, year);
```

The Anchor Roles ({[:} and {]:})

The anchor roles allow a set of strings by be padded as a group, but still be visible to xo_emit(3) as distinct fields. Either the start or stop anchor can give a field width and it can be either directly in the descriptor or passed as an argument. Any fields between the start and stop anchor are padded to meet the minimum width given.

To give a width directly, encode it as the content of the anchor tag:

```
xo_emit("({[:10}{:min/%d}/{:max/%d}{]:})\n", min, max);
```

To pass a width as an argument, use "%d" as the format, which must appear after the "/". Note that only "%d" is supported for widths. Using any other value could ruin your day.

```
xo_emit("({[:/%d}{:min/%d}/{:max/%d}{]:})\n", width, min, max);
```

If the width is negative, padding will be added on the right, suitable for left justification. Otherwise the padding will be added to the left of the fields between the start and stop anchors, suitable for right justification. If the width is zero, nothing happens. If the number of columns of output between the start and stop anchors is less than the absolute value of the given width, nothing happens.

Widths over 8k are considered probable errors and not supported. If XOF_WARN is set, a warning will be generated.

Field Modifiers

Field modifiers are flags which modify the way content emitted for particular output styles:

M	Name	Description
a	argument	The content appears as a "const char *" argument
c	colon	A colon (":") is appended after the label
d	display	Only emit field for display styles (text/HTML)
e	encoding	Only emit for encoding styles (XML/JSON)
h	humanize (hn)	Format large numbers in human-readable style
	hn-space	Humanize: Place space between numeric and unit
	hn-decimal	Humanize: Add a decimal digit, if number < 10
	hn-1000	Humanize: Use 1000 as divisor instead of 1024
k	key	Field is a key, suitable for XPath predicates
l	leaf-list	Field is a leaf-list, a list of leaf values
n	no-quotes	Do not quote the field when using JSON style
q	quotes	Quote the field when using JSON style
t	trim	Trim leading and trailing whitespace
w	white space	A blank (" ") is appended after the label

For example, the modifier string "Lwc" means the field has a label role (text that describes the next field) and should be followed by a colon (’c’) and a space (’w’). The modifier string "Vkq" means the field has a value role, that it is a key for the current instance, and that the value should be quoted when encoded for JSON.

Roles and modifiers can also use more verbose names, when preceded by a comma. For example, the modifier string "Lwc" (or "L,white,colon") means the field has a label role (text that describes the next field) and should be followed by a colon (’c’) and a space (’w’). The modifier string "Vkq" (or ":key,quote") means the field has a value role (the default role), that it is a key for the current instance, and that the value should be quoted when encoded for JSON.

The Argument Modifier ({a:})

The argument modifier indicates that the content of the field descriptor will be placed as a UTF-8 string (const char *) argument within the xo_emit parameters.

```
EXAMPLE:
  xo_emit("{La:} {a:}\n", "Label text", "label", "value");
TEXT:
  Label text value
JSON:
  "label": "value"
```

```
XML:
  <label>value</label>
```

The argument modifier allows field names for value fields to be passed on the stack, avoiding the need to build a field descriptor using snprintf(1). For many field roles, the argument modifier is not needed, since those roles have specific mechanisms for arguments, such as "{C:fg-%s}".

The Colon Modifier ({c:})

The colon modifier appends a single colon to the data value:

```
EXAMPLE:
  xo_emit("{Lc:Name}{:name}\n", "phil");
TEXT:
  Name:phil
```

The colon modifier is only used for the TEXT and HTML output styles. It is commonly combined with the space modifier ('{w:}'). It is purely a convenience feature.

The Display Modifier ({d:})

The display modifier indicated the field should only be generated for the display output styles, TEXT and HTML.

```
EXAMPLE:
  xo_emit("{Lcw:Name}{d:name} {:id/%d}\n", "phil", 1);
TEXT:
  Name: phil 1
XML:
  <id>1</id>
```

The display modifier is the opposite of the encoding modifier, and they are often used to give to distinct views of the underlying data.

The Encoding Modifier ({e:})

The encoding modifier indicated the field should only be generated for the encoding output styles, such as JSON and XML.

```
EXAMPLE:
  xo_emit("{Lcw:Name}{:name} {e:id/%d}\n", "phil", 1);
TEXT:
  Name: phil
```

```
XML:
  <name>phil</name><id>1</id>
```

The encoding modifier is the opposite of the display modifier, and they are often used to give to distinct views of the underlying data.

The Humanize Modifier ({h:})

The humanize modifier is used to render large numbers as in a human-readable format. While numbers like "44470272" are completely readable to computers and savants, humans will generally find "44M" more meaningful.

"hn" can be used as an alias for "humanize".

The humanize modifier only affects display styles (TEXT and HMTL). The "no-humanize" option will block the function of the humanize modifier.

There are a number of modifiers that affect details of humanization. These are only available in as full names, not single characters. The "hn-space" modifier places a space between the number and any multiplier symbol, such as "M" or "K" (ex: "44 K"). The "hn-decimal" modifier will add a decimal point and a single tenths digit when the number is less than 10 (ex: "4.4K"). The "hn-1000" modifier will use 1000 as divisor instead of 1024, following the JEDEC-standard instead of the more natural binary powers-of-two tradition.

```
EXAMPLE:
   xo_emit("{h:input/%u}, {h,hn-space:output/%u}, "
     "{h,hn-decimal:errors/%u}, {h,hn-1000:capacity/%u}, "
     "{h,hn-decimal:remaining/%u}\n",
      input, output, errors, capacity, remaining);
TEXT:
   21, 57 K, 96M, 44M, 1.2G
```

In the HTML style, the original numeric value is rendered in the "data-number" attribute on the element:

```
<div class="data" data-tag="errors"
    data-number="100663296">96M</div>
```

The Gettext Modifier ({g:})

The gettext modifier is used to translate individual fields using the gettext domain (typically set using the "{G:}" role) and current language settings. Once libxo renders the field value, it is passed to

gettext(3), where it is used as a key to find the native language translation.

In the following example, the strings "State" and "full" are passed to **gettext**() to find locale-based translated strings.

```
xo_emit("{Lgwc:State}{g:state}\n", "full");
```

The Key Modifier ({k:})

The key modifier is used to indicate that a particular field helps uniquely identify an instance of list data.

```
EXAMPLE:
   xo_open_list("user");
   for (i = 0; i < num_users; i++) {
            xo_open_instance("user");
      xo_emit("User {k:name} has {:count} tickets\n",
        user[i].u_name, user[i].u_tickets);
      xo_close_instance("user");
   }
   xo_close_list("user");
```

Currently the key modifier is only used when generating XPath values for the HTML output style when XOF_XPATH is set, but other uses are likely in the near future.

The Leaf-List Modifier ({l:})

The leaf-list modifier is used to distinguish lists where each instance consists of only a single value. In XML, these are rendered as single elements, where JSON renders them as arrays.

```
EXAMPLE:
   xo_open_list("user");
   for (i = 0; i < num_users; i++) {
      xo_emit("Member {l:name}\n", user[i].u_name);
   }
   xo_close_list("user");
XML:
   <user>phil</user>
   <user>pallavi</user>
JSON:
   "user": [ "phil", "pallavi" ]
```

The No-Quotes Modifier ({n:})

The no-quotes modifier (and its twin, the ’quotes’ modifier) affect the quoting of values in the JSON output style. JSON uses quotes for string values, but no quotes for numeric, boolean, and null data. xo_emit(3) applies a simple heuristic to determine whether quotes are needed, but often this needs to be controlled by the caller.

```
EXAMPLE:
  const char *bool = is_true ? "true" : "false";
  xo_emit("{n:fancy/%s}", bool);
JSON:
  "fancy": true
```

The Plural Modifier ({p:})

The plural modifier selects the appropriate plural form of an expression based on the most recent number emitted and the current language settings. The contents of the field should be the singular and plural English values, separated by a comma:

```
xo_emit("{:bytes} {Ngp:byte,bytes}\n", bytes);
```

The plural modifier is meant to work with the gettext modifier ({g:}) but can work independently.

When used without the gettext modifier or when the message does not appear in the message catalog, the first token is chosen when the last numeric value is equal to 1; otherwise the second value is used, mimicking the simple pluralization rules of English.

When used with the gettext modifier, the ngettext(3) function is called to handle the heavy lifting, using the message catalog to convert the singular and plural forms into the native language.

The Quotes Modifier ({q:})

The quotes modifier (and its twin, the ’no-quotes’ modifier) affect the quoting of values in the JSON output style. JSON uses quotes for string values, but no quotes for numeric, boolean, and null data. xo_emit(3) applies a simple heuristic to determine whether quotes are needed, but often this needs to be controlled by the caller.

```
EXAMPLE:
  xo_emit("{q:time/%d}", 2014);
JSON:
  "year": "2014"
```

The White Space Modifier ({w:})

The white space modifier appends a single space to the data value:

EXAMPLE:
xo_emit("{Lw:Name}{:name}\n", "phil");
TEXT:
Name phil

The white space modifier is only used for the TEXT and HTML output styles. It is commonly combined with the colon modifier ('{c:}'). It is purely a convenience feature.

Note that the sense of the 'w' modifier is reversed for the units role ({Uw:}); a blank is added before the contents, rather than after it.

Field Formatting

The field format is similar to the format string for printf(3). Its use varies based on the role of the field, but generally is used to format the field's contents.

If the format string is not provided for a value field, it defaults to "%s".

Note a field definition can contain zero or more printf-style "directives", which are sequences that start with a '%' and end with one of following characters: "diouxXDOUeEfFgGaAcCsSp". Each directive is matched by one of more arguments to the xo_emit(3) function.

The format string has the form:

'%' format-modifier * format-character

The format-modifier can be:

- a '#' character, indicating the output value should be prefixed with "0x", typically to indicate a base 16 (hex) value.
- a minus sign ('-'), indicating the output value should be padded on the right instead of the left.
- a leading zero ('0') indicating the output value should be padded on the left with zeroes instead of spaces (' ').
- one or more digits ('0' - '9') indicating the minimum width of the argument. If the width in columns of the output value is less than the minimum width, the value will be padded to reach the minimum.
- a period followed by one or more digits indicating the maximum number of bytes which will be examined for a string argument, or the maximum width for a non-string argument. When handling

ASCII strings this functions as the field width but for multi-byte characters, a single character may be composed of multiple bytes. xo_emit(3) will never dereference memory beyond the given number of bytes.

- a second period followed by one or more digits indicating the maximum width for a string argument. This modifier cannot be given for non-string arguments.

- one or more ’h’ characters, indicating shorter input data.

- one or more ’l’ characters, indicating longer input data.

- a ’z’ character, indicating a ’size_t’ argument.

- a ’t’ character, indicating a ’ptrdiff_t’ argument.

- a ’ ’ character, indicating a space should be emitted before positive numbers.

- a ’+’ character, indicating sign should emitted before any number.

Note that ’q’, ’D’, ’O’, and ’U’ are considered deprecated and will be removed eventually.

The format character is described in the following table:

C Argument Type	Format
d int	base 10 (decimal)
i int	base 10 (decimal)
o int	base 8 (octal)
u unsigned	base 10 (decimal)
x unsigned	base 16 (hex)
X unsigned long	base 16 (hex)
D long	base 10 (decimal)
O unsigned long	base 8 (octal)
U unsigned long	base 10 (decimal)
e double	[-]d.ddde+-dd
E double	[-]d.dddE+-dd
f double	[-]ddd.ddd
F double	[-]ddd.ddd
g double	as ’e’ or ’f’
G double	as ’E’ or ’F’
a double	[-]0xh.hhhp[+-]d

A double [-]0Xh.hhhp[+-]d
c unsigned char a character
C wint_t a character
s char * a UTF-8 string
S wchar_t * a unicode/WCS string
p void * ’%#lx’

The ’h’ and ’l’ modifiers affect the size and treatment of the argument:

Mod d, i o, u, x, X
hh signed char unsigned char
h short unsigned short
l long unsigned long
ll long long unsigned long long
j intmax_t uintmax_t
t ptrdiff_t ptrdiff_t
z size_t size_t
q quad_t u_quad_t

UTF-8 and Locale Strings

All strings for **libxo** must be UTF-8. **libxo** will handle turning them into locale-based strings for display to the user.

For strings, the ’h’ and ’l’ modifiers affect the interpretation of the bytes pointed to argument. The default ’%s’ string is a ’char *’ pointer to a string encoded as UTF-8. Since UTF-8 is compatible with *ASCII* data, a normal 7-bit *ASCII* string can be used. "%ls" expects a "wchar_t *" pointer to a wide-character string, encoded as 32-bit Unicode values. "%hs" expects a "char *" pointer to a multi-byte string encoded with the current locale, as given by the LC_CTYPE, LANG, or LC_ALL environment variables. The first of this list of variables is used and if none of the variables are set, the locale defaults to *UTF-8*.

libxo will convert these arguments as needed to either UTF-8 (for XML, JSON, and HTML styles) or locale-based strings for display in text style.

```
xo_emit("All strings are utf-8 content {:tag/%ls}",
    L"except for wide strings");
```

"%S" is equivalent to "%ls".

For example, a function is passed a locale-base name, a hat size, and a time value. The hat size is

formatted in a UTF-8 (ASCII) string, and the time value is formatted into a wchar_t string.

```
void print_order (const char *name, int size,
                  struct tm *timep) {
    char buf[32];
    const char *size_val = "unknown";

        if (size > 0)
        snprintf(buf, sizeof(buf), "%d", size);
        size_val = buf;
    }

    wchar_t when[32];
    wcsftime(when, sizeof(when), L"%d%b%y", timep);

    xo_emit("The hat for {:name/%hs} is {:size/%s}.\n",
          name, size_val);
    xo_emit("It was ordered on {:order-time/%ls}.\n",
          when);
}
```

It is important to note that xo_emit(3) will perform the conversion required to make appropriate output. Text style output uses the current locale (as described above), while XML, JSON, and HTML use UTF-8.

UTF-8 and locale-encoded strings can use multiple bytes to encode one column of data. The traditional "precision" (aka "max-width") value for "%s" printf formatting becomes overloaded since it specifies both the number of bytes that can be safely referenced and the maximum number of columns to emit. xo_emit(3) uses the precision as the former, and adds a third value for specifying the maximum number of columns.

In this example, the name field is printed with a minimum of 3 columns and a maximum of 6. Up to ten bytes are in used in filling those columns.

```
xo_emit("{:name/%3.10.6s}", name);
```

Characters Outside of Field Definitions

Characters in the format string that are not part of a field definition are copied to the output for the TEXT style, and are ignored for the JSON and XML styles. For HTML, these characters are placed in a with class "text".

EXAMPLE:
xo_emit("The hat is {:size/%s}.\n", size_val);
TEXT:
The hat is extra small.
XML:
<size>extra small</size>
JSON:
"size": "extra small"
HTML:
The hat is
extra small
.

’%n’ is Not Supported

libxo does not support the ’%n’ directive. It is a bad idea and we just do not do it.

The Encoding Format (eformat)

The "eformat" string is the format string used when encoding the field for JSON and XML. If not provided, it defaults to the primary format with any minimum width removed. If the primary is not given, both default to "%s".

EXAMPLE

In this example, the value for the number of items in stock is emitted:

```
xo_emit("{P:   }{Lwc:In stock}{:in-stock/%u}\n",
      instock);
```

This call will generate the following output:

TEXT:
In stock: 144
XML:
<in-stock>144</in-stock>
JSON:
"in-stock": 144,
HTML:

In stock
:

```
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock">144</div>
</div>
```

Clearly HTML wins the verbosity award, and this output does not include XOF_XPATH or XOF_INFO data, which would expand the penultimate line to:

```
<div class="data" data-tag="in-stock"
  data-xpath="/top/data/item/in-stock"
  data-type="number"
  data-help="Number of items in stock">144</div>
```

WHAT MAKES A GOOD FIELD NAME?

To make useful, consistent field names, follow these guidelines:

Use lower case, even for TLAs

Lower case is more civilized. Even TLAs should be lower case to avoid scenarios where the differences between "XPath" and "Xpath" drive your users crazy. Using "xpath" is simpler and better.

Use hyphens, not underscores

Use of hyphens is traditional in XML, and the XOF_UNDERSCORES flag can be used to generate underscores in JSON, if desired. But the raw field name should use hyphens.

Use full words

Do not abbreviate especially when the abbreviation is not obvious or not widely used. Use "data-size", not "dsz" or "dsize". Use "interface" instead of "ifname", "if-name", "iface", "if", or "intf".

Use <verb>-<units>

Using the form <verb>-<units> or <verb>-<classifier>-<units> helps in making consistent, useful names, avoiding the situation where one app uses "sent-packet" and another "packets-sent" and another "packets-we-have-sent". The <units> can be dropped when it is obvious, as can obvious words in the classification. Use "receive-after-window-packets" instead of "received-packets-of-data-after-window".

Reuse existing field names

Nothing is worse than writing expressions like:

```
if ($src1/process[pid == $pid]/name ==
    $src2/proc-table/proc/p[process-id == $pid]/proc-name) {
    ...
}
```

Find someone else who is expressing similar data and follow their fields and hierarchy. Remember the quote is not "Consistency is the hobgoblin of little minds" but "A foolish consistency is the hobgoblin of little minds".

Think about your users

Have empathy for your users, choosing clear and useful fields that contain clear and useful data. You may need to augment the display content with xo_attr(3) calls or "{e:}" fields to make the data useful.

Do not use an arbitrary number postfix

What does "errors2" mean? No one will know. "errors-after-restart" would be a better choice. Think of your users, and think of the future. If you make "errors2", the next guy will happily make "errors3" and before you know it, someone will be asking what is the difference between errors37 and errors63.

Be consistent, uniform, unsurprising, and predictable

Think of your field vocabulary as an API. You want it useful, expressive, meaningful, direct, and obvious. You want the client application's programmer to move between without the need to understand a variety of opinions on how fields are named. They should see the system as a single cohesive whole, not a sack of cats.

Field names constitute the means by which client programmers interact with our system. By choosing wise names now, you are making their lives better.

After using xolint(1) to find errors in your field descriptors, use "xolint -V" to spell check your field names and to detect different names for the same data. "dropped-short" and "dropped-too-short" are both reasonable names, but using them both will lead users to ask the difference between the two fields. If there is no difference, use only one of the field names. If there is a difference, change the names to make that difference more obvious.

SEE ALSO

libxo(3), xolint(1), xo_emit(3)

HISTORY

The **libxo** library first appeared in FreeBSD 11.0.

AUTHORS

libxo was written by Phil Shafer <*phil@freebsd.org*>.