libxo Documentation

Release

Phil Shafer

Jun 30, 2021

Documentation Contents:

The libxo library allows an application to generate text, XML, JSON, and HTML output, suitable for both command line use and for web applications. The application decides at run time which output style should be produced. By using libxo, a single source code path can emit multiple styles of output using command line options to select the style, along with optional behaviors. libxo includes support for multiple output streams, pluralization, color, syslog, *humanized(3)* output, internationalization, and UTF-8. The library aims to minimize the cost of migrating code to libxo.

libxo ships as part of FreeBSD.

CHAPTER 1

Introducing libxo

The libxo library allows an application to generate text, XML, JSON, and HTML output using a common set of function calls. The application decides at run time which output style should be produced. The application calls a function "xo_emit" to product output that is described in a format string. A "field descriptor" tells libxo what the field is and what it means. Each field descriptor is placed in braces with printf-like *Format Strings*:

```
xo_emit(" {:lines/%7ju} {:words/%7ju} "
        "{:characters/%7ju} {d:filename/%s}\n",
        linect, wordct, charct, file);
```

Each field can have a role, with the 'value' role being the default, and the role tells libxo how and when to render that field (see *Field Roles* for details). Modifiers change how the field is rendered in different output styles (see *Field Modifiers* for details. Output can then be generated in various style, using the "–libxo" option:

```
% wc /etc/motd
      25     165    1140 /etc/motd
% wc --libxo xml,pretty,warn /etc/motd
<wc>
  <file>
    <lines>25</lines>
    <words>165</words>
    <characters>1140</characters>
    <filename>/etc/motd</filename>
  </file>
</wc>
% wc --libxo json,pretty,warn /etc/motd
{
  "wc": {
    "file": [
      {
        "lines": 25,
        "words": 165,
        "characters": 1140,
        "filename": "/etc/motd"
      }
```

(continues on next page)

(continued from previous page)

```
    ]
  }
}
% wc --libxo html,pretty,warn /etc/motd
<div class="line">
  <div class="text"> </div>
  <div class="data" data-tag="lines">     25</div>
  <div class="text"> </div>
  <div class="data" data-tag="words">    165</div>
  <div class="text"> </div>
  <div class="data" data-tag="characters">   1140</div>
  <div class="text"> </div>
  <div class="data" data-tag="filename">/etc/motd</div>
</div>
```

Same code path, same format strings, same information, but it's rendered in distinct styles based on run-time flags.

Tale of Two Code Paths

You want to prepare for the future, but you need to live in the present. You'd love a flying car, but need to get work done today. You want to support features like XML, JSON, and HTML rendering to allow integration with NETCONF, REST, and web browsers, but you need to make text output for command line users.

And you don't want multiple code paths that can't help but get out of sync:

```
/* None of this "if (xml) {... } else {...}"  logic */
if (xml) {
    /* some code to make xml */
} else {
    /* other code to make text */
    /* oops! forgot to add something on both clauses! */
}

/* And ifdefs are right out. */
#ifdef MAKE_XML
    /* icky */
#else
    /* pooh */
#endif
```

But you'd really, really like all the fancy features that modern encoding formats can provide. libxo can help.

CHAPTER 2

Getting libxo

libxo now ships as part of the FreeBSD Operating System (as of Release 11).

libxo source code lives on github:

> https://github.com/Juniper/libxo

The latest release of libxo is available at:

> https://github.com/Juniper/libxo/releases

We're using Semantic Versioning to number our releases. libxo is open source, distributed under the BSD license. We follow the branching scheme from A Successful Git Branching Model: we do development under the "*develop*" branch, and release from the "*master*" branch. To clone a developer tree, run the following command:

```
git clone https://github.com/Juniper/libxo.git -b develop
```

Issues, problems, and bugs should be directly to the issues page on our github site.

2.1 Downloading libxo Source Code

You can retrieve the source for libxo in two ways:

A. Use a "distfile" for a specific release. We use github to maintain our releases. Visit the release page to see the list of releases. To download the latest, look for the release witeh the green "Latest release" button and the green "libxo-RELEASE.tar.gz" button under that section.

   After downloading that release's distfile, untar it as follows:

   ```
   tar -zxf libxo-RELEASE.tar.gz
   cd libxo-RELEASE
   ```

   Solaris Users

Note: for Solaris users, your "`tar`" command lacks the "-z" flag, so you'll need to substitute "`gzip -dc $file | tar xf -`" instead of "`tar -zxf $file`".

B. Use the current build from github. This gives you the most recent source code, which might be less stable than a specific release. To build libxo from the git repo:

```
git clone https://github.com/Juniper/libxo.git
cd libxo
```

Be Aware

The github repository does **not** contain the files generated by "*autoreconf*", with the notable exception of the "*m4*" directory. Since these files (depcomp, configure, missing, install-sh, etc) are generated files, we keep them out of the source code repository.

This means that if you download the a release distfile, these files will be ready and you'll just need to run "configure", but if you download the source code from svn, then you'll need to run "*autoreconf*" by hand. This step is done for you by the "*setup.sh*" script, described in the next section.

2.2 Building libxo

To build libxo, you'll need to set up the build, run the "*configure*" script, run the "*make*" command, and run the regression tests.

The following is a summary of the commands needed. These commands are explained in detail in the rest of this section:

```
sh bin/setup.sh
cd build
../configure
make
make test
sudo make install
```

The following sections will walk through each of these steps with additional details and options, but the above directions should be all that's needed.

2.2.1 Setting up the build

Run the "*setup.sh*" script to set up the build. This script runs the "*autoreconf*" command to generate the "*configure*" script and other generated files:

```
sh bin/setup.sh
```

Note: We're are currently using autoreconf version 2.69.

2.2.2 Running the "configure" Script

Configure (and autoconf in general) provides a means of building software in diverse environments. Our configure script supports a set of options that can be used to adjust to your operating environment. Use "`configure --help`" to view these options.

We use the "*build*" directory to keep object files and generated files away from the source tree.

To run the configure script, change into the "*build*" directory, and run the "*configure*" script. Add any required options to the "`../configure`" command line:

```
cd build
../configure
```

Expect to see the "*configure*" script generate the following error:

```
/usr/bin/rm: cannot remove `libtoolT': No such file or directory
```

This error is harmless and can be safely ignored.

By default, libxo installs architecture-independent files, including extension library files, in the /usr/local directories. To specify an installation prefix other than /usr/local for all installation files, include the –prefix=prefix option and specify an alternate location. To install just the extension library files in a different, user-defined location, include the "*–with-extensions-dir=dir*" option and specify the location where the extension libraries will live:

```
cd build
../configure [OPTION]... [VAR=VALUE]...
```

Running the "make" Command

Once the "*configure*" script is run, build the images using the "`make`" command:

```
make
```

Running the Regression Tests

libxo includes a set of regression tests that can be run to ensure the software is working properly. These test are optional, but will help determine if there are any issues running libxo on your machine. To run the regression tests:

```
make test
```

2.2.3 Installing libxo

Once the software is built, you'll need to install libxo using the "`make install`" command. If you are the root user, or the owner of the installation directory, simply issue the command:

```
make install
```

If you are not the "*root*" user and are using the "*sudo*" package, use:

```
sudo make install
```

Verify the installation by viewing the output of "`xo --version`":

```
% xo --version
libxo version 0.3.5-git-develop
xo version 0.3.5-git-develop
```

CHAPTER 3

Formatting with libxo

Most unix commands emit text output aimed at humans. It is designed to be parsed and understood by a user. Humans are gifted at extracting details and pattern matching in such output. Often programmers need to extract information from this human-oriented output. Programmers use tools like grep, awk, and regular expressions to ferret out the pieces of information they need. Such solutions are fragile and require maintenance when output contents change or evolve, along with testing and validation.

Modern tool developers favor encoding schemes like XML and JSON, which allow trivial parsing and extraction of data. Such formats are simple, well understood, hierarchical, easily parsed, and often integrate easier with common tools and environments. Changes to content can be done in ways that do not break existing users of the data, which can reduce maintenance costs and increase feature velocity.

In addition, modern reality means that more output ends up in web browsers than in terminals, making HTML output valuable.

libxo allows a single set of function calls in source code to generate traditional text output, as well as XML and JSON formatted data. HTML can also be generated; " " elements surround the traditional text output, with attributes that detail how to render the data.

A single libxo function call in source code is all that's required:

```
xo_emit("Connecting to {:host}.{:domain}...\n", host, domain);

TEXT:
  Connecting to my-box.example.com...
XML:
  <host>my-box</host>
  <domain>example.com</domain>
JSON:
  "host": "my-box",
  "domain": "example.com"
HTML:
   <div class="line">
     <div class="text">Connecting to </div>
     <div class="data" data-tag="host"
          data-xpath="/top/host">my-box</div>
```

(continues on next page)

(continued from previous page)

```
    <div class="text">.</div>
    <div class="data" data-tag="domain"
         data-xpath="/top/domain">example.com</div>
    <div class="text">...</div>
  </div>
```

3.1 Encoding Styles

There are four encoding styles supported by libxo:

- TEXT output can be display on a terminal session, allowing compatibility with traditional command line usage.
- XML output is suitable for tools like XPath and protocols like NETCONF.
- JSON output can be used for RESTful APIs and integration with languages like Javascript and Python.
- HTML can be matched with a small CSS file to permit rendering in any HTML5 browser.

In general, XML and JSON are suitable for encoding data, while TEXT is suited for terminal output and HTML is suited for display in a web browser (see *xohtml*).

3.1.1 Text Output

Most traditional programs generate text output on standard output, with contents like:

```
36      ./src
40      ./bin
90      .
```

In this example (taken from *du* source code), the code to generate this data might look like:

```
printf("%d\t%s\n", num_blocks, path);
```

Simple, direct, obvious. But it's only making text output. Imagine using a single code path to make TEXT, XML, JSON or HTML, deciding at run time which to generate.

libxo expands on the idea of printf format strings to make a single format containing instructions for creating multiple output styles:

```
xo_emit("{:blocks/%d}\t{:path/%s}\n", num_blocks, path);
```

This line will generate the same text output as the earlier printf call, but also has enough information to generate XML, JSON, and HTML.

The following sections introduce the other formats.

3.1.2 XML Output

XML output consists of a hierarchical set of elements, each encoded with a start tag and an end tag. The element should be named for data value that it is encoding:

```
<item>
  <blocks>36</blocks>
  <path>./src</path>
</item>
<item>
  <blocks>40</blocks>
  <path>./bin</path>
</item>
<item>
  <blocks>90</blocks>
  <path>.</path>
</item>
```

XML is the W3C standard for encoding data.

3.1.3 JSON Output

JSON output consists of a hierarchical set of objects and lists, each encoded with a quoted name, a colon, and a value. If the value is a string, it must be quoted, but numbers are not quoted. Objects are encoded using braces; lists are encoded using square brackets. Data inside objects and lists is separated using commas:

```
items: [
    { "blocks": 36, "path" : "./src" },
    { "blocks": 40, "path" : "./bin" },
    { "blocks": 90, "path" : "./" }
]
```

3.1.4 HTML Output

HTML output is designed to allow the output to be rendered in a web browser with minimal effort. Each piece of output data is rendered inside a element, with a class name related to the role of the data. By using a small set of class attribute values, a CSS stylesheet can render the HTML into rich text that mirrors the traditional text content.

Additional attributes can be enabled to provide more details about the data, including data type, description, and an XPath location:

```
<div class="line">
  <div class="data" data-tag="blocks">36</div>
  <div class="padding">      </div>
  <div class="data" data-tag="path">./src</div>
</div>
<div class="line">
  <div class="data" data-tag="blocks">40</div>
  <div class="padding">      </div>
  <div class="data" data-tag="path">./bin</div>
</div>
<div class="line">
  <div class="data" data-tag="blocks">90</div>
  <div class="padding">      </div>
  <div class="data" data-tag="path">./</div>
</div>
```

CHAPTER 4

Command-line Arguments

libxo uses command line options to trigger rendering behavior. There are multiple conventions for passing options, all using the "`--libxo`" option:

```
--libxo <options>
--libxo=<options>
--libxo:<brief-options>
```

The *brief-options* is a series of single letter abbrevations, where the *options* is a comma-separated list of words. Both provide access to identical functionality. The following invocations are all identical in outcome:

```
my-app --libxo warn,pretty arg1
my-app --libxo=warn,pretty arg1
my-app --libxo:WP arg1
```

Programs using libxo are expecting to call the xo_parse_args function to parse these arguments. See *Parsing Command-line Arguments (xo_parse_args)* for details.

4.1 Option Keywords

Options is a comma-separated list of tokens that correspond to output styles, flags, or features:

Token	Action
color	Enable colors/effects for display styles (TEXT, HTML)
colors=xxxx	Adjust color output values
dtrt	Enable "Do The Right Thing" mode
flush	Flush after every libxo function call
flush-line	Flush after every line (line-buffered)
html	Emit HTML output
indent=xx	Set the indentation level
info	Add info attributes (HTML)
json	Emit JSON output
keys	Emit the key attribute for keys (XML)
log-gettext	Log (via stderr) each gettext(3) string lookup
log-syslog	Log (via stderr) each syslog message (via xo_syslog)
no-humanize	Ignore the {h:} modifier (TEXT, HTML)
no-locale	Do not initialize the locale setting
no-retain	Prevent retaining formatting information
no-top	Do not emit a top set of braces (JSON)
not-first	Pretend the 1st output item was not 1st (JSON)
pretty	Emit pretty-printed output
retain	Force retaining formatting information
text	Emit TEXT output
underscores	Replace XML-friendly "-"s with JSON friendly "_"s
units	Add the 'units' (XML) or 'data-units (HTML) attribute
warn	Emit warnings when libxo detects bad calls
warn-xml	Emit warnings in XML
xml	Emit XML output
xpath	Add XPath expressions (HTML)

Most of these option are simple and direct, but some require additional details:

- "colors" is described in *Color Mapping*.
- "flush-line" performs line buffering, even when the output is not directed to a TTY device.
- "info" generates additional data for HTML, encoded in attributes using names that state with "data-".
- "keys" adds a "key" attribute for XML output to indicate that a leaf is an identifier for the list member.
- "no-humanize" avoids "humanizing" numeric output (see *The Humanize Modifier ({h:})* for details).
- "no-locale" instructs libxo to avoid translating output to the current locale.
- "no-retain" disables the ability of libxo to internally retain "compiled" information about formatting strings (see *Retaining Parsed Format Information* for details).
- "underscores" can be used with JSON output to change XML-friendly names with dashes into JSON-friendly name with underscores.
- "warn" allows libxo to emit warnings on stderr when application code make incorrect calls.
- "warn-xml" causes those warnings to be placed in XML inside the output.

4.2 Brief Options

The brief options are simple single-letter aliases to the normal keywords, as detailed below:

Option	Action
c	Enable color/effects for TEXT/HTML
F	Force line-buffered flushing
H	Enable HTML output (XO_STYLE_HTML)
I	Enable info output (XOF_INFO)
i<num>	Indent by <number>
J	Enable JSON output (XO_STYLE_JSON)
k	Add keys to XPATH expressions in HTML
n	Disable humanization (TEXT, HTML)
P	Enable pretty-printed output (XOF_PRETTY)
T	Enable text output (XO_STYLE_TEXT)
U	Add units to HTML output
u	Change "-"s to "_"s in element names (JSON)
W	Enable warnings (XOF_WARN)
X	Enable XML output (XO_STYLE_XML)
x	Enable XPath data (XOF_XPATH)

4.3 Color Mapping

The "colors" option takes a value that is a set of mappings from the pre-defined set of colors to new foreground and background colors. The value is a series of "fg/bg" values, separated by a "+". Each pair of "fg/bg" values gives the colors to which a basic color is mapped when used as a foreground or background color. The order is the mappings is:

- black
- red
- green
- yellow
- blue
- magenta
- cyan
- white

Pairs may be skipped, leaving them mapped as normal, as are missing pairs or single colors.

For example consider the following xo_emit call:

```
xo_emit("{C:fg-red,bg-green}Merry XMas!!{C:}\n");
```

To turn all colored output to red-on-blue, use eight pairs of "red/blue" mappings separated by plus signs ("+"):

```
--libxo colors=red/blue+red/blue+red/blue+red/blue+\
               red/blue+red/blue+red/blue+red/blue
```

To turn the red-on-green text to magenta-on-cyan, give a "magenta" foreground value for red (the second mapping) and a "cyan" background to green (the third mapping):

```
--libxo colors=+magenta+/cyan
```

Consider the common situation where blue output looks unreadable on a terminal session with a black background. To turn both "blue" foreground and background output to "yellow", give only the fifth mapping, skipping the first four mappings with bare plus signs ("+"):

```
--libxo colors=++++yellow/yellow
```

4.4 Encoders

In addition to the four "built-in" formats, libxo supports an extensible mechanism for adding encoders. These are activated using the "encoder" keyword:

```
--libxo encoder=cbor
```

The encoder can include encoder-specific options, separated by either colons (":") or plus signs ("+"):

> –libxo encoder=csv+path=filesystem+leaf=name+no-header –libxo encoder=csv:path=filesystem:leaf=name:no-header

For brevity, the string "@" can be used in place of the string "encoder=".

> df –libxo @csv:no-header

CHAPTER 5

Format Strings

libxo uses format strings to control the rendering of data into the various output styles. 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-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.

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 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: 65n"), XML ("<in-stock>65</in-stock>"), JSON (""in-stock": 6'), 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);
```

CHAPTER 6

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:

R	Name	Description
C	color	Field has color and effect controls
D	decoration	Field is non-text (e.g., colon, comma)
E	error	Field is an error message
G	gettext	Call gettext(3) on the format string
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);
```

6.1 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,inverse}Ugly{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	Description
black	
blue	
cyan	
default	Default color for foreground or background
green	
magenta	
red	
white	
yellow	

When using colors, the developer should remember that users will change the foreground and background colors of terminal session according to their own tastes, so assuming that "blue" looks nice is never safe, and is a constant annoyance to your dear author. In addition, a significant percentage of users (1 in 12) will be color blind. Depending on color to convey critical information is not a good idea. Color should enhance output, but should not be used as the sole means of encoding information.

6.2 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);
```

6.3 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 tokenn");

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 moden");

See *Howto: Internationalization (i18n)* for additional details.

6.4 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);
```

6.5 The Note Role ({N:})

Notes are text that appears after a value:

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

6.6 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);
```

6.7 The Title Role ({T:})

Title 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");
```

Title fields have an extra convenience feature; if both content and format are specified, instead of looking to the argument list for a value, the content is used, allowing a mixture of format and content within the field descriptor:

```
xo_emit("{T:Name/%20s}{T:Count/%6s}\n");
```

Since the incoming argument is a string, the format must be "%s" or something suitable.

6.8 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>
```

6.9 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 \"{:poem}\" in {:year/%4d}\n,
        author, poem, year);
```

6.10 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 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.

CHAPTER 7

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)
g	gettext	Call gettext on field’s render content
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
n	no-quotes	Do not quote the field when using JSON style
p	plural	Gettext: Use comma-separated plural form
q	quotes	Quote the field when using JSON style
t	trim	Trim leading and trailing whitespace
w	white	A blank (” “) is appended after the label

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.

7.1 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. For many field roles, the argument modifier is not needed, since those roles have specific mechanisms for arguments, such as "{C:fg-%s}".

7.2 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.

7.3 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.

7.4 The Encoding Modifier ({e:})

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

```
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.

7.5 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");
```

See *The Gettext Role ({G:})*, *The Plural Modifier ({p:})*, and *Howto: Internationalization (i18n)* for additional details.

7.6 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 (See *Command-line Arguments*) 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>
```

7.7 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 value for the HTML output style when XOF_XPATH is set, but other uses are likely in the near future.

7.8 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:
    for (i = 0; i < num_users; i++) {
        xo_emit("Member {l:user}\n", user[i].u_name);
    }
XML:
    <user>phil</user>
    <user>pallavi</user>
JSON:
    "user": [ "phil", "pallavi" ]
```

The name of the field must match the name of the leaf list.

7.9 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 value, but no quotes for numeric, boolean, and null data. xo_emit 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
```

7.10 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. See *The Gettext Modifier ({g:})*.

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.

7.11 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 value, but no quotes for numeric, boolean, and null data. xo_emit 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 heuristic is based on the format; if the format uses any of the following conversion specifiers, then no quotes are used:

```
d i o u x X D O U e E f F g G a A c C p
```

7.12 The Trim Modifier ({t:})

The trim modifier removes any leading or trailing whitespace from the value:

```
EXAMPLE:
    xo_emit("{t:description}", "   some  input   ");
JSON:
    "description": "some input"
```

7.13 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.

CHAPTER 8

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 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 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:

Ltr	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

8.1 UTF-8 and Locale Strings

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 a 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 varibles. 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");

======== ================= ================================
 Format   Argument Type      Argument Contents
======== ================= ================================
 %s       const char \*      UTF-8 string
 %S       const char \*      UTF-8 string (alias for '%ls')
 %ls      const wchar_t \*   Wide character UNICODE string
 %hs      const char *       locale-based string
======== ================= ================================
```

"Long", not "locale"

The "*l*" in "%ls" is for "*long*", following the convention of "%ld". It is not "*locale*", a common mis-mnemonic. "%S" is equivalent to "%ls".

For example, the following 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 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 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 of data at the location given by 'name' are in used in filling those columns:

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

8.2 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:
    <div class="text">The hat is </div>
    <div class="data" data-tag="size">extra small</div>
    <div class="text">.</div>
```

8.3 "%m" Is Supported

libxo supports the '%m' directive, which formats the error message associated with the current value of "errno". It is the equivalent of "%s" with the argument strerror(errno):

```
xo_emit("{:filename} cannot be opened: {:error/%m}", filename);
xo_emit("{:filename} cannot be opened: {:error/%s}",
        filename, strerror(errno));
```

8.4 "%n" Is Not Supported

libxo does not support the '%n' directive. It's a bad idea and we just don't do it.

8.5 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".

8.6 Content Strings

For padding and labels, the content string is considered the content, unless a format is given.

8.7 Argument Validation

Many compilers and tool chains support validation of printf-like arguments. When the format string fails to match the argument list, a warning is generated. This is a valuable feature and while the formatting strings for libxo differ considerably from printf, many of these checks can still provide build-time protection against bugs.

libxo provide variants of functions that provide this ability, if the "–enable-printflike" option is passed to the "configure" script. These functions use the "_p" suffix, like "xo_emit_p()", xo_emit_hp()", etc.

The following are features of libxo formatting strings that are incompatible with printf-like testing:

- implicit formats, where "{:tag}" has an implicit "%s";
- the "max" parameter for strings, where "{:tag/%4.10.6s}" means up to ten bytes of data can be inspected to fill a minimum of 4 columns and a maximum of 6;
- percent signs in strings, where "{:filled}%" makes a single, trailing percent sign;
- the "l" and "h" modifiers for strings, where "{:tag/%hs}" means locale-based string and "{:tag/%ls}" means a wide character string;
- distinct encoding formats, where "{:tag/#%s/%s}" means the display styles (text and HTML) will use "#%s" where other styles use "%s";

If none of these features are in use by your code, then using the "_p" variants might be wise:

Function	printf-like Equivalent
xo_emit_hv	xo_emit_hvp
xo_emit_h	xo_emit_hp
xo_emit	xo_emit_p
xo_emit_warn_hcv	xo_emit_warn_hcvp
xo_emit_warn_hc	xo_emit_warn_hcp
xo_emit_warn_c	xo_emit_warn_cp
xo_emit_warn	xo_emit_warn_p
xo_emit_warnx	xo_emit_warnx_p
xo_emit_err	xo_emit_err_p
xo_emit_errx	xo_emit_errx_p
xo_emit_errc	xo_emit_errc_p

8.8 Retaining Parsed Format Information

libxo can retain the parsed internal information related to the given format string, allowing subsequent xo_emit calls, the retained information is used, avoiding repetitive parsing of the format string:

```
SYNTAX:
  int xo_emit_f(xo_emit_flags_t flags, const char fmt, ...);
EXAMPLE:
  xo_emit_f(XOEF_RETAIN, "{:some/%02d}{:thing/%-6s}{:fancy}\n",
                 some, thing, fancy);
```

To retain parsed format information, use the XOEF_RETAIN flag to the xo_emit_f() function. A complete set of xo_emit_f functions exist to match all the xo_emit function signatures (with handles, varadic argument, and printf-like flags):

Function	Flags Equivalent
xo_emit_hv	xo_emit_hvf
xo_emit_h	xo_emit_hf
xo_emit	xo_emit_f
xo_emit_hvp	xo_emit_hvfp
xo_emit_hp	xo_emit_hfp
xo_emit_p	xo_emit_fp

The format string must be immutable across multiple calls to xo_emit_f(), since the library retains the string. Typically this is done by using static constant strings, such as string literals. If the string is not immutable, the XOEF_RETAIN flag must not be used.

The functions xo_retain_clear() and xo_retain_clear_all() release internal information on either a single format string or all format strings, respectively. Neither is required, but the library will retain this information until it is cleared or the process exits:

```
const char *fmt = "{:name}  {:count/%d}\n";
for (i = 0; i < 1000; i++) {
    xo_open_instance("item");
    xo_emit_f(XOEF_RETAIN, fmt, name[i], count[i]);
}
xo_retain_clear(fmt);
```

The retained information is kept as thread-specific data.

8.9 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:
    <div class="line">
      <div class="padding">   </div>
      <div class="label">In stock</div>
      <div class="decoration">:</div>
      <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>
```

CHAPTER 9

The libxo API

This section gives details about the functions in libxo, how to call them, and the actions they perform.

9.1 Handles

libxo uses "handles" to control its rendering functionality. The handle contains state and buffered data, as well as callback functions to process data.

Handles give an abstraction for libxo that encapsulates the state of a stream of output. Handles have the data type "`xo_handle_t`" and are opaque to the caller.

The library has a default handle that is automatically initialized. By default, this handle will send text style output (`XO_STYLE_TEXT`) to standard output. The xo_set_style and xo_set_flags functions can be used to change this behavior.

For the typical command that is generating output on standard output, there is no need to create an explicit handle, but they are available when needed, e.g., for daemons that generate multiple streams of output.

Many libxo functions take a handle as their first parameter; most that do not use the default handle. Any function taking a handle can be passed NULL to access the default handle. For the convenience of callers, the libxo library includes handle-less functions that implicitly use the default handle.

For example, the following are equivalent:

```
xo_emit("test");
xo_emit_h(NULL, "test");
```

Handles are created using `xo_create` and destroy using `xo_destroy`.

9.1.1 xo_create

xo_handle_t *__xo_create__(xo_style_t *style*, xo_xof_flags_t *flags*)

The `xo_create` function allocates a new handle which can be passed to further libxo function calls. The `xo_handle_t` structure is opaque.

Parameters

- **style** (*xo_style_t*) – Output style (XO_STYLE_*)
- **flags** (*xo_xof_flags_t*) – Flags for this handle (XOF_*)

Returns New libxo handle

Return type xo_handle_t *

```
EXAMPLE:
    xo_handle_t *xop = xo_create(XO_STYLE_JSON, XOF_WARN | XOF_PRETTY);
    ....
    xo_emit_h(xop, "testing\n");
```

See also *Output Styles (XO_STYLE_*)* and *Flags (XOF_*)*.

9.1.2 xo_create_to_file

xo_handle_t ***xo_create_to_file**(FILE **fp*, unsigned *style*, unsigned *flags*)

The `xo_create_to_file` function is aconvenience function is provided for situations when output should be written to a different file, rather than the default of standard output.

The `XOF_CLOSE_FP` flag can be set on the returned handle to trigger a call to fclose() for the FILE pointer when the handle is destroyed, avoiding the need for the caller to perform this task.

Parameters

- **fp** (*FILE **) – FILE to use as base for this handle
- **style** (*xo_style_t*) – Output style (XO_STYLE_*)
- **flags** (*xo_xof_flags_t*) – Flags for this handle (XOF_*)

Returns New libxo handle

Return type xo_handle_t *

9.1.3 xo_set_writer

void **xo_set_writer**(xo_handle_t **xop*, void **opaque*, xo_write_func_t *write_func*, xo_close_func_t *close_func*, xo_flush_func_t *flush_func*)

The `xo_set_writer` function allows custom functions which can tailor how libxo writes data. The `opaque` argument is recorded and passed back to the functions, allowing the function to acquire context information. The *write_func* function writes data to the output stream. The *close_func* function can release this opaque data and any other resources as needed. The *flush_func* function is called to flush buffered data associated with the opaque object.

Parameters

- **xop** (*xo_handle_t **) – Handle to modify (or NULL for default handle)
- **opaque** (*void **) – Pointer to opaque data passed to the given functions
- **write_func** (*xo_write_func_t*) – New write function
- **close_func** (*xo_close_func_t*) – New close function
- **flush_func** (*xo_flush_func_t*) – New flush function

Returns void

9.1.4 xo_get_style

xo_style_t **xo_get_style**(xo_handle_t *xop)

Use the `xo_get_style` function to find the current output style for a given handle. To use the default handle, pass a `NULL` handle.

Parameters

- **xop** (*xo_handle_t **) – Handle to interrogate (or NULL for default handle)

Returns Output style (XO_STYLE_*)

Return type xo_style_t

```
EXAMPLE::
    style = xo_get_style(NULL);
```

Output Styles (XO_STYLE_*)

The libxo functions accept a set of output styles:

Flag	Description
XO_STYLE_TEXT	Traditional text output
XO_STYLE_XML	XML encoded data
XO_STYLE_JSON	JSON encoded data
XO_STYLE_HTML	HTML encoded data

The "XML", "JSON", and "HTML" output styles all use the UTF-8 character encoding. "TEXT" using locale-based encoding.

9.1.5 xo_set_style

void **xo_set_style**(xo_handle_t *xop, xo_style_t *style*)

The `xo_set_style` function is used to change the output style setting for a handle. To use the default handle, pass a `NULL` handle.

Parameters

- **xop** (*xo_handle_t **) – Handle to modify
- **style** (*xo_style_t*) – Output style (XO_STYLE_*)

Returns void

```
EXAMPLE:
    xo_set_style(NULL, XO_STYLE_XML);
```

9.1.6 xo_set_style_name

int **xo_set_style_name**(xo_handle_t *xop, const char *style*)

The `xo_set_style_name` function can be used to set the style based on a name encoded as a string: The name can be any of the supported styles: "text", "xml", "json", or "html".

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)

- **style** (*const char **) – Text name of the style

Returns zero for success, non-zero for error

Return type int

```
EXAMPLE:
    xo_set_style_name(NULL, "html");
```

9.1.7 xo_set_flags

void **xo_set_flags** (xo_handle_t *xop, xo_xof_flags_t *flags*)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)
- **flags** (*xo_xof_flags_t*) – Flags to add for the handle

Returns void

Use the `xo_set_flags` function to turn on flags for a given libxo handle. To use the default handle, pass a `NULL` handle.

```
EXAMPLE:
    xo_set_flags(NULL, XOF_PRETTY | XOF_WARN);
```

Flags (XOF_*)

The set of valid flags include:

Flag	Description
XOF_CLOSE_FP	Close file pointer on `xo_destroy`
XOF_COLOR	Enable color and effects in output
XOF_COLOR_ALLOWED	Allow color/effect for terminal output
XOF_DTRT	Enable "do the right thing" mode
XOF_INFO	Display info data attributes (HTML)
XOF_KEYS	Emit the key attribute (XML)
XOF_NO_ENV	Do not use the *LIBXO_OPTIONS* env var
XOF_NO_HUMANIZE	Display humanization (TEXT, HTML)
XOF_PRETTY	Make "pretty printed" output
XOF_UNDERSCORES	Replaces hyphens with underscores
XOF_UNITS	Display units (XML, HMTL)
XOF_WARN	Generate warnings for broken calls
XOF_WARN_XML	Generate warnings in XML on stdout
XOF_XPATH	Emit XPath expressions (HTML)
XOF_COLUMNS	Force xo_emit to return columns used
XOF_FLUSH	Flush output after each `xo_emit` call

The `XOF_CLOSE_FP` flag will trigger the call of the *close_func* (provided via `xo_set_writer`) when the handle is destroyed.

The `XOF_COLOR` flag enables color and effects in output regardless of output device, while the `XOF_COLOR_ALLOWED` flag allows color and effects only if the output device is a terminal.

The XOF_PRETTY flag requests "pretty printing", which will trigger the addition of indentation and newlines to enhance the readability of XML, JSON, and HTML output. Text output is not affected.

The XOF_WARN flag requests that warnings will trigger diagnostic output (on standard error) when the library notices errors during operations, or with arguments to functions. Without warnings enabled, such conditions are ignored.

Warnings allow developers to debug their interaction with libxo. The function xo_failure can used as a breakpoint for a debugger, regardless of whether warnings are enabled.

If the style is XO_STYLE_HTML, the following additional flags can be used:

Flag	Description
XOF_XPATH	Emit "data-xpath" attributes
XOF_INFO	Emit additional info fields

The XOF_XPATH flag enables the emission of XPath expressions detailing the hierarchy of XML elements used to encode the data field, if the XPATH style of output were requested.

The XOF_INFO flag encodes additional informational fields for HTML output. See *Field Information (xo_info_t)* for details.

If the style is XO_STYLE_XML, the following additional flags can be used:

Flag	Description
XOF_KEYS	Flag "key" fields for XML

The XOF_KEYS flag adds "key" attribute to the XML encoding for field definitions that use the "k" modifier. The key attribute has the value "key":

```
  xo_emit("{k:name}", item);

XML:
    <name key="key">truck</name>
```

xo_clear_flags

void **xo_clear_flags** (xo_handle_t *xop, xo_xof_flags_t *flags*)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)
- **flags** (*xo_xof_flags_t*) – Flags to clear for the handle

Returns void

Use the xo_clear_flags function to turn off the given flags in a specific handle. To use the default handle, pass a NULL handle.

xo_set_options

int **xo_set_options** (xo_handle_t *xop, const char *input*)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)
- **input** (*const char **) – string containing options to set

Returns zero for success, non-zero for error

Return type int

The `xo_set_options` function accepts a comma-separated list of output styles and modifier flags and enables them for a specific handle. The options are identical to those listed in *Command-line Arguments*. To use the default handle, pass a `NULL` handle.

xo_destroy

void **xo_destroy** (xo_handle_t *xop*)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)

Returns void

The `xo_destroy` function releases a handle and any resources it is using. Calling `xo_destroy` with a `NULL` handle will release any resources associated with the default handle.

9.2 Emitting Content (xo_emit)

The functions in this section are used to emit output. They use a `format` string containing field descriptors as specified in *Format Strings*. The use of a handle is optional and `NULL` can be passed to access the internal "default" handle. See *Handles*.

The remaining arguments to `xo_emit` and `xo_emit_h` are a set of arguments corresponding to the fields in the format string. Care must be taken to ensure the argument types match the fields in the format string, since an inappropriate or missing argument can ruin your day. The `vap` argument to `xo_emit_hv` points to a variable argument list that can be used to retrieve arguments via `va_arg`.

xo_ssize_t **xo_emit** (const char **fmt*, ...)

Parameters

- **fmt** – The format string, followed by zero or more arguments

Returns If XOF_COLUMNS is set, the number of columns used; otherwise the number of bytes emitted

Return type xo_ssize_t

xo_ssize_t **xo_emit_h** (xo_handle_t **xop*, const char **fmt*, ...)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)
- **fmt** – The format string, followed by zero or more arguments

Returns If XOF_COLUMNS is set, the number of columns used; otherwise the number of bytes emitted

Return type xo_ssize_t

xo_ssize_t **xo_emit_hv** (xo_handle_t **xop*, const char **fmt*, va_list *vap*)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)
- **fmt** – The format string

- **vap** (*va_list*) – A set of variadic arguments

Returns If XOF_COLUMNS is set, the number of columns used; otherwise the number of bytes emitted

Return type xo_ssize_t

9.2.1 Single Field Emitting Functions (xo_emit_field)

The functions in this section emit formatted output similar to `xo_emit` but where `xo_emit` uses a single string argument containing the description for multiple fields, `xo_emit_field` emits a single field using multiple arguments to contain the field description. `xo_emit_field_h` adds an ex- plicit handle to use instead of the default handle, while `xo_emit_field_hv` accepts a va_list for additional flexibility.

The arguments `rolmod`, `content`, `fmt`, and `efmt` are detailed in *Field Formatting*. Using distinct arguments allows callers to pass the field description in pieces, rather than having to use something like `snprintf` to build the format string required by `xo_emit`. The arguments are each NUL-terminated strings. The `rolmod` argument contains the `role` and `modifier` portions of the field description, the `content` argument contains the `content` portion, and the `fmt` and `efmt` contain the `field-format` and `encoding-format` por- tions, respectively.

As with `xo_emit`, the `fmt` and `efmt` values are both optional, since the `field-format` string defaults to "%s", and the `encoding-format`'s default value is derived from the `field-format` per *Field Formatting*. However, care must be taken to avoid using a value directly as the format, since characters like '{', '%', and '}' will be interpreted as formatting directives, and may cause xo_emit_field to dereference arbitrary values off the stack, leading to bugs, core files, and gnashing of teeth.

xo_ssize_t **xo_emit_field** (const char **rolmod*, const char **content*, const char **fmt*, const char **efmt*, ...)

Parameters

- **rolmod** (*const char **) – A comma-separated list of field roles and field modifiers
- **content** (*const char **) – The "content" portion of the field description string
- **fmt** (*const char **) – Contents format string
- **efmt** (*const char **) – Encoding format string, followed by additional arguments

Returns If XOF_COLUMNS is set, the number of columns used; otherwise the number of bytes emitted

Return type xo_ssize_t

```
EXAMPLE::
    xo_emit_field("T", title, NULL, NULL, NULL);
    xo_emit_field("T", "Host name is ", NULL, NULL);
    xo_emit_field("V", "host-name", NULL, NULL, host-name);
    xo_emit_field(",leaf-list,quotes", "sku", "%s-%u", "%s-000-%u",
                  "gum", 1412);
```

xo_ssize_t **xo_emit_field_h** (xo_handle_t **xop*, const char **rolmod*, const char **contents*, const char **fmt*, const char **efmt*, ...)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)
- **rolmod** (*const char **) – A comma-separated list of field roles and field modifiers
- **contents** (*const char **) – The "contents" portion of the field description string
- **fmt** (*const char **) – Content format string

- **efmt** (*const char **) – Encoding format string, followed by additional arguments

Returns If XOF_COLUMNS is set, the number of columns used; otherwise the number of bytes emitted

Return type xo_ssize_t

xo_ssize_t **xo_emit_field_hv**(xo_handle_t *xop, const char *rolmod, const char *contents, const char *fmt, const char *efmt, va_list vap)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)
- **rolmod** (*const char **) – A comma-separated list of field roles and field modifiers
- **contents** (*const char **) – The "contents" portion of the field description string
- **fmt** (*const char **) – Content format string
- **efmt** (*const char **) – Encoding format string
- **vap** (*va_list*) – A set of variadic arguments

Returns If XOF_COLUMNS is set, the number of columns used; otherwise the number of bytes emitted

Return type xo_ssize_t

9.2.2 Attributes (xo_attr)

The functions in this section emit an XML attribute with the given name and value. This only affects the XML output style.

The `name` parameter give the name of the attribute to be encoded. The `fmt` parameter gives a printf-style format string used to format the value of the attribute using any remaining arguments, or the vap parameter passed to `xo_attr_hv`.

All attributes recorded via `xo_attr` are placed on the next container, instance, leaf, or leaf list that is emitted.

Since attributes are only emitted in XML, their use should be limited to meta-data and additional or redundant representations of data already emitted in other form.

xo_ssize_t **xo_attr**(const char *name, const char *fmt, ...)

Parameters

- **name** (*const char **) – Attribute name
- **fmt** (*const char **) – Attribute value, as variadic arguments

Returns -1 for error, or the number of bytes in the formatted attribute value

Return type xo_ssize_t

```
EXAMPLE:
    xo_attr("seconds", "%ld", (unsigned long) login_time);
    struct tm *tmp = localtime(login_time);
    strftime(buf, sizeof(buf), "%R", tmp);
    xo_emit("Logged in at {:login-time}\n", buf);
XML:
    <login-time seconds="1408336270">00:14</login-time>
```

xo_ssize_t **xo_attr_h**(xo_handle_t *xop, const char *name, const char *fmt, ...)

Parameters

- **xop** (*xo_handle_t **) – Handle for modify (or NULL for default handle)

The `xo_attr_h` function follows the conventions of `xo_attr` but adds an explicit libxo handle.

xo_ssize_t **xo_attr_hv**(xo_handle_t **xop*, const char **name*, const char **fmt*, va_list *vap*)
The `xo_attr_h` function follows the conventions of `xo_attr_h` but replaced the variadic list with a variadic pointer.

9.2.3 Flushing Output (xo_flush)

xo_ssize_t **xo_flush**(void)

Returns -1 for error, or the number of bytes generated

Return type xo_ssize_t

libxo buffers data, both for performance and consistency, but also to allow for the proper function of various advanced features. At various times, the caller may wish to flush any data buffered within the library. The `xo_flush` call is used for this.

Calling `xo_flush` also triggers the flush function associated with the handle. For the default handle, this is equivalent to "fflush(stdio);".

xo_ssize_t **xo_flush_h**(xo_handle_t **xop*)

Parameters

- **xop** (*xo_handle_t **) – Handle for flush (or NULL for default handle)

Returns -1 for error, or the number of bytes generated

Return type xo_ssize_t

The `xo_flush_h` function follows the conventions of `xo_flush`, but adds an explicit libxo handle.

9.2.4 Finishing Output (xo_finish)

When the program is ready to exit or close a handle, a call to `xo_finish` or `xo_finish_h` is required. This flushes any buffered data, closes open libxo constructs, and completes any pending operations.

Calling this function is vital to the proper operation of libxo, especially for the non-TEXT output styles.

xo_ssize_t **xo_finish**(void)

Returns -1 on error, or the number of bytes flushed

Return type xo_ssize_t

xo_ssize_t **xo_finish_h**(xo_handle_t **xop*)

Parameters

- **xop** (*xo_handle_t **) – Handle for finish (or NULL for default handle)

Returns -1 on error, or the number of bytes flushed

Return type xo_ssize_t

void **xo_finish_atexit**(void)
The `xo_finish_atexit` function is suitable for use with *atexit(3)* to ensure that `xo_finish` is called on the default handle when the application exits.

9.3 Emitting Hierarchy

libxo represents two types of hierarchy: containers and lists. A container appears once under a given parent where a list consists of instances that can appear multiple times. A container is used to hold related fields and to give the data organization and scope.

YANG Terminology

libxo uses terminology from YANG (**RFC 7950**), the data modeling language for NETCONF: container, list, leaf, and leaf-list.

For XML and JSON, individual fields appear inside hierarchies which provide context and meaning to the fields. Unfortunately, these encoding have a basic disconnect between how lists is similar objects are represented.

XML encodes lists as set of sequential elements:

```
<user>phil</user>
<user>pallavi</user>
<user>sjg</user>
```

JSON encodes lists using a single name and square brackets:

```
"user": [ "phil", "pallavi", "sjg" ]
```

This means libxo needs three distinct indications of hierarchy: one for containers of hierarchy appear only once for any specific parent, one for lists, and one for each item in a list.

9.3.1 Containers

A "*container*" is an element of a hierarchy that appears only once under any specific parent. The container has no value, but serves to contain and organize other nodes.

To open a container, call xo_open_container() or xo_open_container_h(). The former uses the default handle and the latter accepts a specific handle. To close a level, use the xo_close_container() or xo_close_container_h() functions.

Each open call must have a matching close call. If the XOF_WARN flag is set and the name given does not match the name of the currently open container, a warning will be generated.

xo_ssize_t **xo_open_container** (const char ***name*)

Parameters

- **name** (*const char **) – Name of the container

Returns -1 on error, or the number of bytes generated

Return type xo_ssize_t

The `name` parameter gives the name of the container, encoded in UTF-8. Since ASCII is a proper subset of UTF-8, traditional C strings can be used directly.

xo_ssize_t **xo_open_container_h** (xo_handle_t ***xop*, const char ***name*)

Parameters

- **xop** (*xo_handle_t **) – Handle to use (or NULL for default handle)

The `xo_open_container_h` function adds a `handle` parameter.

xo_ssize_t **xo_close_container** (const char ***name*)

Parameters

- **name** (*const char **) – Name of the container

Returns -1 on error, or the number of bytes generated

Return type xo_ssize_t

xo_ssize_t **xo_close_container_h**(xo_handle_t *xop, const char *name)

Parameters

- **xop** (*xo_handle_t **) – Handle to use (or NULL for default handle)

The `xo_close_container_h` function adds a `handle` parameter.

Use the XOF_WARN flag to generate a warning if the name given on the close does not match the current open container.

For TEXT and HTML output, containers are not rendered into output text, though for HTML they are used to record an XPath value when the XOF_XPATH flag is set.

```
EXAMPLE:
    xo_open_container("top");
    xo_open_container("system");
    xo_emit("{:host-name/%s%s%s}", hostname,
            domainname ? "." : "", domainname ?: "");
    xo_close_container("system");
    xo_close_container("top");
TEXT:
    my-host.example.org
XML:
    <top>
      <system>
          <host-name>my-host.example.org</host-name>
      </system>
    </top>
JSON:
    "top" : {
      "system" : {
          "host-name": "my-host.example.org"
      }
    }
HTML:
    <div class="data"
         data-tag="host-name">my-host.example.org</div>
```

9.3.2 Lists and Instances

A "*list*" is set of one or more instances that appear under the same parent. The instances contain details about a specific object. One can think of instances as objects or records. A call is needed to open and close the list, while a distinct call is needed to open and close each instance of the list.

The name given to all calls must be identical, and it is strongly suggested that the name be singular, not plural, as a matter of style and usage expectations:

```
EXAMPLE:
    xo_open_list("item");
```

(continues on next page)

(continued from previous page)

```
    for (ip = list; ip->i_title; ip++) {
        xo_open_instance("item");
        xo_emit("{L:Item} '{:name/%s}':\n", ip->i_title);
        xo_close_instance("item");
    }

    xo_close_list("item");
```

Getting the list and instance calls correct is critical to the proper generation of XML and JSON data.

Opening Lists

xo_ssize_t **xo_open_list** (const char **name*)

Parameters

- **name** (*const char **) – Name of the list

Returns -1 on error, or the number of bytes generated

Return type xo_ssize_t

The `xo_open_list` function open a list of instances.

xo_ssize_t **xo_open_list_h** (xo_handle_t **xop*, const char **name*)

Parameters

- **xop** (*xo_handle_t **) – Handle to use (or NULL for default handle)

Closing Lists

xo_ssize_t **xo_close_list** (const char **name*)

Parameters

- **name** (*const char **) – Name of the list

Returns -1 on error, or the number of bytes generated

Return type xo_ssize_t

The `xo_close_list` function closes a list of instances.

xo_ssize_t **xo_close_list_h** (xo_handle_t **xop*, const char **name*)

Parameters

- **xop** – Handle to use (or NULL for default handle)

Opening Instances

xo_ssize_t **xo_open_instance** (const char **name*)

Parameters

- **name** (*const char **) – Name of the instance (same as the list name)

Returns -1 on error, or the number of bytes generated

Return type xo_ssize_t

The `xo_open_instance` function open a single instance.

xo_ssize_t **xo_open_instance_h** (xo_handle_t *xop, const char *name)

Parameters

- **xop** – Handle to use (or NULL for default handle)

Closing Instances

xo_ssize_t **xo_close_instance** (const char *name)

Parameters

- **name** (*const char **) – Name of the instance

Returns -1 on error, or the number of bytes generated

Return type xo_ssize_t

The `xo_close_instance` function closes an open instance.

xo_ssize_t **xo_close_instance_h** (xo_handle_t *xop, const char *name)

Parameters

- **xop** (*xo_handle_t **) – Handle to use (or NULL for default handle)

The `xo_close_instance_h` function adds a `handle` parameter.

```
EXAMPLE:
    xo_open_list("user");
    for (i = 0; i < num_users; i++) {
        xo_open_instance("user");
        xo_emit("{k:name}:{:uid/%u}:{:gid/%u}:{:home}\n",
                pw[i].pw_name, pw[i].pw_uid,
                pw[i].pw_gid, pw[i].pw_dir);
        xo_close_instance("user");
    }
    xo_close_list("user");
TEXT:
    phil:1001:1001:/home/phil
    pallavi:1002:1002:/home/pallavi
XML:
    <user>
        <name>phil</name>
        <uid>1001</uid>
        <gid>1001</gid>
        <home>/home/phil</home>
    </user>
    <user>
        <name>pallavi</name>
        <uid>1002</uid>
        <gid>1002</gid>
        <home>/home/pallavi</home>
    </user>
JSON:
    user: [
        {
            "name": "phil",
            "uid": 1001,
```

(continues on next page)

(continued from previous page)

```
            "gid": 1001,
            "home": "/home/phil",
        },
        {
            "name": "pallavi",
            "uid": 1002,
            "gid": 1002,
            "home": "/home/pallavi",
        }
    ]
```

9.3.3 Markers

Markers are used to protect and restore the state of open hierarchy constructs (containers, lists, or instances). While a marker is open, no other open constructs can be closed. When a marker is closed, all constructs open since the marker was opened will be closed.

Markers use names which are not user-visible, allowing the caller to choose appropriate internal names.

In this example, the code whiffles through a list of fish, calling a function to emit details about each fish. The marker "fish-guts" is used to ensure that any constructs opened by the function are closed properly:

```
EXAMPLE:
    for (i = 0; fish[i]; i++) {
        xo_open_instance("fish");
        xo_open_marker("fish-guts");
        dump_fish_details(i);
        xo_close_marker("fish-guts");
    }
```

xo_ssize_t **xo_open_marker** (const char **name*)

Parameters

- **name** (*const char **) – Name of the instance

Returns -1 on error, or the number of bytes generated

Return type xo_ssize_t

The `xo_open_marker` function records the current state of open tags in order for `xo_close_marker` to close them at some later point.

xo_ssize_t **xo_open_marker_h** (const char **name*)

Parameters

- **xop** (*xo_handle_t **) – Handle to use (or NULL for default handle)

The `xo_open_marker_h` function adds a `handle` parameter.

xo_ssize_t **xo_close_marker** (const char **name*)

Parameters

- **name** (*const char **) – Name of the instance

Returns -1 on error, or the number of bytes generated

Return type xo_ssize_t

The `xo_close_marker` function closes any open containers, lists, or instances as needed to return to the state recorded when `xo_open_marker` was called with the matching name.

xo_ssize_t **`xo_close_marker`** (const char **name*)

Parameters

- **`xop`** (*`xo_handle_t *`*) – Handle to use (or NULL for default handle)

The `xo_close_marker_h` function adds a `handle` parameter.

9.3.4 DTRT Mode

Some users may find tracking the names of open containers, lists, and instances inconvenient. libxo offers a "Do The Right Thing" mode, where libxo will track the names of open containers, lists, and instances so the close function can be called without a name. To enable DTRT mode, turn on the XOF_DTRT flag prior to making any other libxo output:

```
xo_set_flags(NULL, XOF_DTRT);
```

Each open and close function has a version with the suffix "_d", which will close the open container, list, or instance:

```
xo_open_container_d("top");
...
xo_close_container_d();
```

This also works for lists and instances:

```
xo_open_list_d("item");
for (...) {
    xo_open_instance_d("item");
    xo_emit(...);
    xo_close_instance_d();
}
xo_close_list_d();
```

Note that the XOF_WARN flag will also cause libxo to track open containers, lists, and instances. A warning is generated when the name given to the close function and the name recorded do not match.

9.4 Support Functions

9.4.1 Parsing Command-line Arguments (xo_parse_args)

int **`xo_parse_args`** (int *argc*, char ***argv*)

Parameters

- **`argc`** (*`int`*) – Number of arguments
- **`argv`** – Array of argument strings

Returns -1 on error, or the number of remaining arguments

Return type int

The `xo_parse_args` function is used to process a program's arguments. libxo-specific options are processed and removed from the argument list so the calling application does not need to process them. If successful, a new value for argc is returned. On failure, a message is emitted and -1 is returned:

```
argc = xo_parse_args(argc, argv);
if (argc < 0)
    exit(EXIT_FAILURE);
```

Following the call to xo_parse_args, the application can process the remaining arguments in a normal manner. See *Command-line Arguments* for a description of valid arguments.

9.4.2 xo_set_program

void **xo_set_program** (const char **name*)

Parameters

- **name** (*const char **) – Name to use as the program name

Returns void

The `xo_set_program` function sets the name of the program as reported by functions like `xo_failure`, `xo_warn`, `xo_err`, etc. The program name is initialized by `xo_parse_args`, but subsequent calls to `xo_set_program` can override this value:

```
EXAMPLE:
    xo_set_program(argv[0]);
```

Note that the value is not copied, so the memory passed to `xo_set_program` (and `xo_parse_args`) must be maintained by the caller.

9.4.3 xo_set_version

void **xo_set_version** (const char **version*)

Parameters

- **name** (*const char **) – Value to use as the version string

Returns void

The `xo_set_version` function records a version number to be emitted as part of the data for encoding styles (XML and JSON). This version number is suitable for tracking changes in the content, allowing a user of the data to discern which version of the data model is in use.

void **xo_set_version_h** (xo_handle_t **xop*, const char **version*)

Parameters

- **xop** (*xo_handle_t **) – Handle to use (or NULL for default handle)

The `xo_set_version` function adds a `handle` parameter.

9.4.4 Field Information (xo_info_t)

HTML data can include additional information in attributes that begin with "data-". To enable this, three things must occur:

First the application must build an array of xo_info_t structures, one per tag. The array must be sorted by name, since libxo uses a binary search to find the entry that matches names from format instructions.

Second, the application must inform libxo about this information using the `xo_set_info` call:

```
typedef struct xo_info_s {
    const char *xi_name;    /* Name of the element */
    const char *xi_type;    /* Type of field */
    const char *xi_help;    /* Description of field */
} xo_info_t;

void xo_set_info (xo_handle_t *xop, xo_info_t *infop, int count);
```

Like other libxo calls, passing `NULL` for the handle tells libxo to use the default handle.

If the count is -1, libxo will count the elements of infop, but there must be an empty element at the end. More typically, the number is known to the application:

```
xo_info_t info[] = {
    { "in-stock", "number", "Number of items in stock" },
    { "name", "string", "Name of the item" },
    { "on-order", "number", "Number of items on order" },
    { "sku", "string", "Stock Keeping Unit" },
    { "sold", "number", "Number of items sold" },
};
int info_count = (sizeof(info) / sizeof(info[0]));
...
xo_set_info(NULL, info, info_count);
```

Third, the emission of info must be triggered with the `XOF_INFO` flag using either the `xo_set_flags` function or the "`--libxo=info`" command line argument.

The type and help values, if present, are emitted as the "data-type" and "data-help" attributes:

```
<div class="data" data-tag="sku" data-type="string"
     data-help="Stock Keeping Unit">GRO-000-533</div>
```

void **xo_set_info** (xo_handle_t **xop*, xo_info_t **infop*, int *count*)

Parameters

- **xop** (*xo_handle_t **) – Handle to use (or NULL for default handle)
- **infop** (*xo_info_t **) – Array of information structures

Returns void

9.4.5 Memory Allocation

The `xo_set_allocator` function allows libxo to be used in environments where the standard *`realloc(3)`* and *`free(3)`* functions are not appropriate.

void **xo_set_allocator** (xo_realloc_func_t *realloc_func*, xo_free_func_t *free_func*)

Parameters

- **realloc_func** (*xo_realloc_func_t*) – Allocation function
- **free_func** (*xo_free_func_t*) – Free function

realloc_func should expect the same arguments as *`realloc(3)`* and return a pointer to memory following the same convention. *free_func* will receive the same argument as *`free(3)`* and should release it, as appropriate for the environment.

By default, the standard *`realloc(3)`* and *`free(3)`* functions are used.

9.4.6 LIBXO_OPTIONS

The environment variable "LIBXO_OPTIONS" can be set to a subset of libxo options, including:

- color
- flush
- flush-line
- no-color
- no-humanize
- no-locale
- no-retain
- pretty
- retain
- underscores
- warn

For example, warnings can be enabled by:

```
% env LIBXO_OPTIONS=warn my-app
```

Since environment variables are inherited, child processes will have the same options, which may be undesirable, making the use of the "`--libxo`" command-line option preferable in most situations.

9.4.7 Errors, Warnings, and Messages

Many programs make use of the standard library functions *err(3)* and *warn(3)* to generate errors and warnings for the user. libxo wants to pass that information via the current output style, and provides compatible functions to allow this:

```
void xo_warn (const char *fmt, ...);
void xo_warnx (const char *fmt, ...);
void xo_warn_c (int code, const char *fmt, ...);
void xo_warn_hc (xo_handle_t *xop, int code,
                 const char *fmt, ...);
void xo_err (int eval, const char *fmt, ...);
void xo_errc (int eval, int code, const char *fmt, ...);
void xo_errx (int eval, const char *fmt, ...);
```

```
void xo_message (const char *fmt, ...);
void xo_message_c (int code, const char *fmt, ...);
void xo_message_hc (xo_handle_t *xop, int code,
                    const char *fmt, ...);
void xo_message_hcv (xo_handle_t *xop, int code,
                     const char *fmt, va_list vap);
```

These functions display the program name, a colon, a formatted message based on the arguments, and then optionally a colon and an error message associated with either *errno* or the *code* parameter:

```
EXAMPLE:
    if (open(filename, O_RDONLY) < 0)
        xo_err(1, "cannot open file '%s'", filename);
```

9.4.8 xo_error

void **xo_error** (const char **fmt*, ...)

Parameters

- **fmt** (*const char **) – Format string

Returns void

void **xo_error_h** (xo_handle_t **xop*, const char **fmt*, ...)

Parameters

- **xop** (*xo_handle_t **) – libxo handle pointer
- **fmt** (*const char **) – Format string

Returns void

void **xo_error_hv** (xo_handle_t **xop*, const char **fmt*, va_list *vap*)

Parameters

- **xop** (*va_list*) – libxo handle pointer
- **fmt** (*const char **) – Format string
- **vap** – variadic arguments

Returns void

void **xo_errorn** (const char **fmt*, ...)

Parameters

- **fmt** (*const char **) – Format string

Returns void

void **xo_errorn_h** (xo_handle_t **xop*, const char **fmt*, ...)

Parameters

- **xop** (*xo_handle_t **) – libxo handle pointer
- **fmt** (*const char **) – Format string

Returns void

void **xo_errorn_hv** (xo_handle_t **xop*, int *need_newline*, const char **fmt*, va_list *vap*)

Parameters

- **xop** (*va_list*) – libxo handle pointer
- **need_newline** (*int*) – boolean indicating need for trailing newline
- **fmt** (*const char **) – Format string
- **vap** – variadic arguments

Returns void

The `xo_error` function can be used for generic errors that should be reported over the handle, rather than to stderr. The `xo_error` function behaves like `xo_err` for TEXT and HTML output styles, but puts the error into XML or JSON elements:

```
EXAMPLE::
    xo_error("Does not %s", "compute");
XML::
    <error><message>Does not compute</message></error>
JSON::
    "error": { "message": "Does not compute" }
```

The `xo_error_h` and `xo_error_hv` add a handle object and a variadic-ized parameter to the signature, respectively.

The `xo_errorn` function supplies a newline at the end the error message if the format string does not include one. The `xo_errorn_h` and `xo_errorn_hv` functions add a handle object and a variadic-ized parameter to the signature, respectively. The `xo_errorn_hv` function also adds a boolean to indicate the need for a trailing newline.

9.4.9 xo_no_setlocale

void **xo_no_setlocale** (void)

libxo automatically initializes the locale based on setting of the environment variables LC_CTYPE, LANG, and LC_ALL. The first of this list of variables is used and if none of the variables, the locale defaults to "UTF-8". The caller may wish to avoid this behavior, and can do so by calling the `xo_no_setlocale` function.

9.5 Emitting syslog Messages

syslog is the system logging facility used throughout the unix world. Messages are sent from commands, applications, and daemons to a hierarchy of servers, where they are filtered, saved, and forwarded based on configuration behaviors.

syslog is an older protocol, originally documented only in source code. By the time **RFC 3164** published, variation and mutation left the leading "<pri>" string as only common content. **RFC 5424** defines a new version (version 1) of syslog and introduces structured data into the messages. Structured data is a set of name/value pairs transmitted distinctly alongside the traditional text message, allowing filtering on precise values instead of regular expressions.

These name/value pairs are scoped by a two-part identifier; an enterprise identifier names the party responsible for the message catalog and a name identifying that message. Enterprise IDs are defined by IANA, the Internet Assigned Numbers Authority.

Use the `xo_set_syslog_enterprise_id` function to set the Enterprise ID, as needed.

The message name should follow the conventions in *What makes a good field name?*, as should the fields within the message:

```
/* Both of these calls are optional */
xo_set_syslog_enterprise_id(32473);
xo_open_log("my-program", 0, LOG_DAEMON);

/* Generate a syslog message */
xo_syslog(LOG_ERR, "upload-failed",
          "error <%d> uploading file '{:filename}' "
          "as '{:target/%s:%s}'",
          code, filename, protocol, remote);

xo_syslog(LOG_INFO, "poofd-invalid-state",
          "state {:current/%u} is invalid {:connection/%u}",
          state, conn);
```

The developer should be aware that the message name may be used in the future to allow access to further information, including documentation. Care should be taken to choose quality, descriptive names.

9.5.1 Priority, Facility, and Flags

The `xo_syslog`, `xo_vsyslog`, and `xo_open_log` functions accept a set of flags which provide the priority of the message, the source facility, and some additional features. These values are OR'd together to create a single integer argument:

```
xo_syslog(LOG_ERR | LOG_AUTH, "login-failed",
          "Login failed; user '{:user}' from host '{:address}'",
          user, addr);
```

These values are defined in <syslog.h>.

The priority value indicates the importance and potential impact of each message:

Priority	Description
LOG_EMERG	A panic condition, normally broadcast to all users
LOG_ALERT	A condition that should be corrected immediately
LOG_CRIT	Critical conditions
LOG_ERR	Generic errors
LOG_WARNING	Warning messages
LOG_NOTICE	Non-error conditions that might need special handling
LOG_INFO	Informational messages
LOG_DEBUG	Developer-oriented messages

The facility value indicates the source of message, in fairly generic terms:

Facility	Description
LOG_AUTH	The authorization system (e.g. *login(1)*)
LOG_AUTHPRIV	As LOG_AUTH, but logged to a privileged file
LOG_CRON	The cron daemon: *cron(8)*
LOG_DAEMON	System daemons, not otherwise explicitly listed
LOG_FTP	The file transfer protocol daemons
LOG_KERN	Messages generated by the kernel
LOG_LPR	The line printer spooling system
LOG_MAIL	The mail system
LOG_NEWS	The network news system
LOG_SECURITY	Security subsystems, such as *ipfw(4)*
LOG_SYSLOG	Messages generated internally by *syslogd(8)*
LOG_USER	Messages generated by user processes (default)
LOG_UUCP	The uucp system
LOG_LOCAL0..7	Reserved for local use

In addition to the values listed above, xo_open_log accepts a set of addition flags requesting specific logging behaviors:

Flag	Description
LOG_CONS	If syslogd fails, attempt to write to /dev/console
LOG_NDELAY	Open the connection to *syslogd(8)* immediately
LOG_PERROR	Write the message also to standard error output
LOG_PID	Log the process id with each message

9.5.2 xo_syslog

void **xo_syslog** (int *pri*, const char **name*, const char **fmt*, ...)

Parameters

- **pri** (*int*) – syslog priority
- **name** (*const char **) – Name of the syslog event
- **fmt** (*const char **) – Format string, followed by arguments

Returns void

Use the `xo_syslog` function to generate syslog messages by calling it with a log priority and facility, a message name, a format string, and a set of arguments. The priority/facility argument are discussed above, as is the message name.

The format string follows the same conventions as `xo_emit`'s format string, with each field being rendered as an SD-PARAM pair:

```
xo_syslog(LOG_ERR, "poofd-missing-file",
          "'{:filename}' not found: {:error/%m}", filename);

... [poofd-missing-file@32473 filename="/etc/poofd.conf"
      error="Permission denied"] '/etc/poofd.conf' not
      found: Permission denied
```

9.5.3 Support functions

xo_vsyslog

void **xo_vsyslog** (int *pri*, const char **name*, const char **fmt*, va_list *vap*)

Parameters

- **pri** (*int*) – syslog priority
- **name** (*const char **) – Name of the syslog event
- **fmt** (*const char **) – Format string
- **vap** (*va_list*) – Variadic argument list

Returns void

xo_vsyslog is identical in function to xo_syslog, but takes the set of arguments using a va_list:

```
EXAMPLE:
    void
    my_log (const char *name, const char *fmt, ...)
    {
        va_list vap;
        va_start(vap, fmt);
        xo_vsyslog(LOG_ERR, name, fmt, vap);
        va_end(vap);
    }
```

xo_open_log

void **xo_open_log** (const char **ident*, int *logopt*, int *facility*)

Parameters

- **indent** (*const char **) –
- **logopt** (*int*) – Bit field containing logging options
- **facility** (*int*) –

Returns void

xo_open_log functions similar to *openlog(3)*, allowing customization of the program name, the log facility number, and the additional option flags described in *Priority, Facility, and Flags*.

xo_close_log

void **xo_close_log** (void)

The xo_close_log function is similar to *closelog(3)*, closing the log file and releasing any associated resources.

xo_set_logmask

int **xo_set_logmask** (int *maskpri*)

Parameters

- **maskpri** (*int*) – the log priority mask

Returns The previous log priority mask

The xo_set_logmask function is similar to *setlogmask(3)*, restricting the set of generated log event to those whose associated bit is set in maskpri. Use LOG_MASK(pri) to find the appropriate bit, or LOG_UPTO(toppri) to create a mask for all priorities up to and including toppri:

```
EXAMPLE:
    setlogmask(LOG_UPTO(LOG_WARN));
```

xo_set_syslog_enterprise_id

void **xo_set_syslog_enterprise_id** (unsigned short *eid*)

Use the xo_set_syslog_enterprise_id to supply a platform- or application-specific enterprise id. This value is used in any future syslog messages.

Ideally, the operating system should supply a default value via the "kern.syslog.enterprise_id" sysctl value. Lacking that, the application should provide a suitable value.

Enterprise IDs are administered by IANA, the Internet Assigned Number Authority. The complete list is EIDs on their web site:

```
https://www.iana.org/assignments/enterprise-numbers/enterprise-numbers
```

New EIDs can be requested from IANA using the following page:

```
http://pen.iana.org/pen/PenApplication.page
```

Each software development organization that defines a set of syslog messages should register their own EID and use that value in their software to ensure that messages can be uniquely identified by the combination of EID + message name.

9.6 Creating Custom Encoders

The number of encoding schemes in current use is staggering, with new and distinct schemes appearing daily. While libxo provide XML, JSON, HMTL, and text natively, there are requirements for other encodings.

Rather than bake support for all possible encoders into libxo, the API allows them to be defined externally. libxo can then interfaces with these encoding modules using a simplistic API. libxo processes all functions calls, handles state transitions, performs all formatting, and then passes the results as operations to a customized encoding function, which implements specific encoding logic as required. This means your encoder doesn't need to detect errors with unbalanced open/close operations but can rely on libxo to pass correct data.

By making a simple API, libxo internals are not exposed, insulating the encoder and the library from future or internal changes.

The three elements of the API are:

- loading
- initialization
- operations

The following sections provide details about these topics.

libxo source contains an encoder for Concise Binary Object Representation, aka CBOR (**RFC 7049**), which can be used as an example for the API for other encoders.

9.6.1 Loading Encoders

Encoders can be registered statically or discovered dynamically. Applications can choose to call the `xo_encoder_register` function to explicitly register encoders, but more typically they are built as shared libraries, placed in the libxo/extensions directory, and loaded based on name. libxo looks for a file with the name of the encoder and an extension of ".enc". This can be a file or a symlink to the shared library file that supports the encoder:

```
% ls -1 lib/libxo/extensions/*.enc
lib/libxo/extensions/cbor.enc
lib/libxo/extensions/test.enc
```

9.6.2 Encoder Initialization

Each encoder must export a symbol used to access the library, which must have the following signature:

```
int xo_encoder_library_init (XO_ENCODER_INIT_ARGS);
```

`XO_ENCODER_INIT_ARGS` is a macro defined in "xo_encoder.h" that defines an argument called "arg", a pointer of the type `xo_encoder_init_args_t`. This structure contains two fields:

- `xei_version` is the version number of the API as implemented within libxo. This version is currently as 1 using `XO_ENCODER_VERSION`. This number can be checked to ensure compatibility. The working assumption is that all versions should be backward compatible, but each side may need to accurately know the version supported by the other side. `xo_encoder_library_init` can optionally check this value, and must then

set it to the version number used by the encoder, allowing libxo to detect version differences and react accordingly. For example, if version 2 adds new operations, then libxo will know that an encoding library that set `xei_version` to 1 cannot be expected to handle those new operations.

- xei_handler must be set to a pointer to a function of type `xo_encoder_func_t`, as defined in "xo_encoder.h". This function takes a set of parameters: - xop is a pointer to the opaque `xo_handle_t` structure - op is an integer representing the current operation - name is a string whose meaning differs by operation - value is a string whose meaning differs by operation - private is an opaque structure provided by the encoder

Additional arguments may be added in the future, so handler functions should use the `XO_ENCODER_HANDLER_ARGS` macro. An appropriate "extern" declaration is provided to help catch errors.

Once the encoder initialization function has completed processing, it should return zero to indicate that no error has occurred. A non-zero return code will cause the handle initialization to fail.

9.6.3 Operations

The encoder API defines a set of operations representing the processing model of libxo. Content is formatted within libxo, and callbacks are made to the encoder's handler function when data is ready to be processed:

Operation	Meaning (Base function)
XO_OP_CREATE	Called when the handle is created
XO_OP_OPEN_CONTAINER	Container opened (xo_open_container)
XO_OP_CLOSE_CONTAINER	Container closed (xo_close_container)
XO_OP_OPEN_LIST	List opened (xo_open_list)
XO_OP_CLOSE_LIST	List closed (xo_close_list)
XO_OP_OPEN_LEAF_LIST	Leaf list opened (xo_open_leaf_list)
XO_OP_CLOSE_LEAF_LIST	Leaf list closed (xo_close_leaf_list)
XO_OP_OPEN_INSTANCE	Instance opened (xo_open_instance)
XO_OP_CLOSE_INSTANCE	Instance closed (xo_close_instance)
XO_OP_STRING	Field with Quoted UTF-8 string
XO_OP_CONTENT	Field with content
XO_OP_FINISH	Finish any pending output
XO_OP_FLUSH	Flush any buffered output
XO_OP_DESTROY	Clean up resources
XO_OP_ATTRIBUTE	An attribute name/value pair
XO_OP_VERSION	A version string

For all the open and close operations, the name parameter holds the name of the construct. For string, content, and attribute operations, the name parameter is the name of the field and the value parameter is the value. "string" are differentiated from "content" to allow differing treatment of true, false, null, and numbers from real strings, though content values are formatted as strings before the handler is called. For version operations, the value parameter contains the version.

All strings are encoded in UTF-8.

CHAPTER 10

Encoders

This section gives an overview of encoders, details on the encoders that ship with libxo, and documentation for developers of future encoders.

10.1 Overview

The libxo library contains software to generate four "built-in" formats: text, XML, JSON, and HTML. These formats are common and useful, but there are other common and useful formats that users will want, and including them all in the libxo software would be difficult and cumbersome.

To allow support for additional encodings, libxo includes a "pluggable" extension mechanism for dynamically loading new encoders. libxo-based applications can automatically use any installed encoder.

Use the "encoder=XXX" option to access encoders. The following example uses the "cbor" encoder, saving the output into a file:

```
df --libxo encoder=cbor > df-output.cbor
```

Encoders can support specific options that can be accessed by following the encoder name with a colon (':') or a plus sign ('+') and one of more options, separated by the same character:

```
df --libxo encoder=csv+path=filesystem+leaf=name+no-header
df --libxo encoder=csv:path=filesystem:leaf=name:no-header
```

These examples instructs libxo to load the "csv" encoder and pass the following options:

```
path=filesystem
leaf=name
no-header
```

Each of these option is interpreted by the encoder, and all such options names and semantics are specific to the particular encoder. Refer to the intended encoder for documentation on its options.

The string "@" can be used in place of the string "encoder=".

df –libxo @csv:no-header

10.2 CSV - Comma Separated Values

libxo ships with a custom encoder for "CSV" files, a common format for comma separated values. The output of the CSV encoder can be loaded directly into spreadsheets or similar applications.

A standard for CSV files is provided in **RFC 4180**, but since the format predates that standard by decades, there are many minor differences in CSV file consumers and their expectations. The CSV encoder has a number of options to tailor output to those expectations.

Consider the following XML:

```
% list-items --libxo xml,pretty
<top>
  <data test="value">
    <item test2="value2">
      <sku test3="value3" key="key">GRO-000-415</sku>
      <name key="key">gum</name>
      <sold>1412</sold>
      <in-stock>54</in-stock>
      <on-order>10</on-order>
    </item>
    <item>
      <sku test3="value3" key="key">HRD-000-212</sku>
      <name key="key">rope</name>
      <sold>85</sold>
      <in-stock>4</in-stock>
      <on-order>2</on-order>
    </item>
    <item>
      <sku test3="value3" key="key">HRD-000-517</sku>
      <name key="key">ladder</name>
      <sold>0</sold>
      <in-stock>2</in-stock>
      <on-order>1</on-order>
    </item>
  </data>
</top>
```

This output is a list of `instances` (named "item"), each containing a set of `leafs` ("sku", "name", etc).

The CSV encoder will emit the leaf values in this output as `fields` inside a CSV `record`, which is a line containing a set of comma-separated values:

```
% list-items --libxo encoder=csv
sku,name,sold,in-stock,on-order
GRO-000-415,gum,1412,54,10
HRD-000-212,rope,85,4,2
HRD-000-517,ladder,0,2,1
```

Be aware that since the CSV encoder looks for data instances, when used with *The "xo" Utility*, the `--instance` option will be needed:

```
% xo --libxo encoder=csv --instance foo 'The {:product} is {:status}\n' stereo "in␣
↪route"
product,status
stereo,in route
```

10.2.1 The `path` Option

By default, the CSV encoder will attempt to emit any list instance generated by the application. In some cases, this may be unacceptable, and a specific list may be desired.

Use the "path" option to limit the processing of output to a specific hierarchy. The path should be one or more names of containers or lists.

For example, if the "list-items" application generates other lists, the user can give "path=top/data/item" as a path:

```
% list-items --libxo encoder=csv:path=top/data/item
sku,name,sold,in-stock,on-order
GRO-000-415,gum,1412,54,10
HRD-000-212,rope,85,4,2
HRD-000-517,ladder,0,2,1
```

Paths are "relative", meaning they need not be a complete set of names to the list. This means that "path=item" may be sufficient for the above example.

10.2.2 The `leafs` Option

The CSV encoding requires that all lines of output have the same number of fields with the same order. In contrast, XML and JSON allow any order (though libxo forces key leafs to appear before other leafs).

To maintain a consistent set of fields inside the CSV file, the same set of leafs must be selected from each list item. By default, the CSV encoder records the set of leafs that appear in the first list instance it processes, and extract only those leafs from future instances. If the first instance is missing a leaf that is desired by the consumer, the "leaf" option can be used to ensure that an empty value is recorded for instances that lack a particular leaf.

The "leafs" option can also be used to exclude leafs, limiting the output to only those leafs provided.

In addition, the order of the output fields follows the order in which the leafs are listed. "leafs=one.two" and "leafs=two.one" give distinct output.

So the "leafs" option can be used to expand, limit, and order the set of leafs.

The value of the leafs option should be one or more leaf names, separated by a period (".").

```
% list-items --libxo encoder=csv:leafs=sku.on-order
sku,on-order
GRO-000-415,10
HRD-000-212,2
HRD-000-517,1
% list-items -libxo encoder=csv:leafs=on-order.sku
on-order,sku
10,GRO-000-415
2,HRD-000-212
1,HRD-000-517
```

Note that since libxo uses terminology from YANG (RFC 7950), the data modeling language for NETCONF (RFC 6241), which uses "leafs" as the plural form of "leaf". libxo follows that convention.

10.2.3 The `no-header` Option

CSV files typical begin with a line that defines the fields included in that file, in an attempt to make the contents self-defining:

```
sku,name,sold,in-stock,on-order
GRO-000-415,gum,1412,54,10
HRD-000-212,rope,85,4,2
HRD-000-517,ladder,0,2,1
```

There is no reliable mechanism for determining whether this header line is included, so the consumer must make an assumption.

The csv encoder defaults to producing the header line, but the "no-header" option can be included to avoid the header line.

10.2.4 The `no-quotes` Option

RFC 4180 specifies that fields containing spaces should be quoted, but many CSV consumers do not handle quotes. The "no-quotes" option instruct the CSV encoder to avoid the use of quotes.

10.2.5 The `dos` Option

RFC 4180 defines the end-of-line marker as a carriage return followed by a newline. This `CRLF` convention dates from the distant past, but its use was anchored in the 1980s by the `DOS` operating system.

The CSV encoder defaults to using the standard Unix end-of-line marker, a simple newline. Use the "dos" option to use the `CRLF` convention.

10.3 The Encoder API

The encoder API consists of three distinct phases:

- loading the encoder
- initializing the encoder
- feeding operations to the encoder

To load the encoder, libxo will open a shared library named:

    ${prefix}/lib/libxo/encoder/${name}.enc

This file is typically a symbolic link to a dynamic library, suitable for `dlopen`(). libxo looks for a symbol called `xo_encoder_library_init` inside that library and calls it with the arguments defined in the header file "xo_encoder.h". This function should look as follows:

```
int
xo_encoder_library_init (XO_ENCODER_INIT_ARGS)
{
    arg->xei_version = XO_ENCODER_VERSION;
    arg->xei_handler = test_handler;

    return 0;
}
```

Several features here allow for future compatibility: the macro XO_ENCODER_INIT_ARGS allows the arguments to this function change over time, and the XO_ENCODER_VERSION allows the library to tell libxo which version of the API it was compiled with.

The function places in xei_handler should be have the signature:

```
static int
test_handler (XO_ENCODER_HANDLER_ARGS)
{
     ...
```

This function will be called with the "op" codes defined in "xo_encoder.h". Each op code represents a distinct event in the libxo processing model. For example OP_OPEN_CONTAINER tells the encoder that a new container has been opened, and the encoder can behave in an appropriate manner.

CHAPTER 11

The “xo” Utility

The `xo` utility allows command line access to the functionality of the libxo library. Using `xo`, shell scripts can emit XML, JSON, and HTML using the same commands that emit text output.

The style of output can be selected using a specific option: “-X” for XML, “-J” for JSON, “-H” for HTML, or “-T” for TEXT, which is the default. The “–style <style>” option can also be used. The standard set of “–libxo” options are available (see *Command-line Arguments*), as well as the *LIBXO_OPTIONS* environment variable.

The `xo` utility accepts a format string suitable for `xo_emit` and a set of zero or more arguments used to supply data for that string:

```
  xo "The {k:name} weighs {:weight/%d} pounds.\n" fish 6

TEXT:
  The fish weighs 6 pounds.

XML:
  <name>fish</name>
  <weight>6</weight>

JSON:
  "name": "fish",
  "weight": 6

HTML:
  <div class="line">
    <div class="text">The </div>
    <div class="data" data-tag="name">fish</div>
    <div class="text"> weighs </div>
    <div class="data" data-tag="weight">6</div>
    <div class="text"> pounds.</div>
  </div>
```

The `--wrap $path` option can be used to wrap emitted content in a specific hierarchy. The path is a set of hierarchical names separated by the ‘/’ character:

```
  xo --wrap top/a/b/c '{:tag}' value

XML:
  <top>
    <a>
      <b>
        <c>
          <tag>value</tag>
        </c>
      </b>
    </a>
  </top>

JSON:
  "top": {
    "a": {
      "b": {
        "c": {
          "tag": "value"
        }
      }
    }
  }
```

The `--open $path` and `--close $path` can be used to emit hierarchical information without the matching close and open tag. This allows a shell script to emit open tags, data, and then close tags. The `--depth` option may be used to set the depth for indentation. The `--leading-xpath` may be used to prepend data to the XPath values used for HTML output style:

```
EXAMPLE:
  #!/bin/sh
  xo --open top/data
  xo --depth 2 '{:tag}' value
  xo --close top/data

XML:
  <top>
    <data>
      <tag>value</tag>
    </data>
  </top>

JSON:
  "top": {
    "data": {
      "tag": "value"
    }
  }
```

When making partial lines of output (where the format string does not include a newline), use the `--continuation` option to let secondary invocations know they are adding data to an existing line.

When emitting a series of objects, use the `--not-first` option to ensure that any details from the previous object (e.g. commas in JSON) are handled correctly.

Use the `--top-wrap` option to ensure any top-level object details are handled correctly, e.g. wrap the entire output in a top-level set of braces for JSON output.

```
EXAMPLE:
  #!/bin/sh
  xo --top-wrap --open top/data
  xo --depth 2 'First {:tag} ' value1
  xo --depth 2 --continuation 'and then {:tag}\n' value2
  xo --top-wrap --close top/data

TEXT:
  First value1 and then value2

HTML:
  <div class="line">
    <div class="text">First </div>
    <div class="data" data-tag="tag">value1</div>
    <div class="text"> </div>
    <div class="text">and then </div>
    <div class="data" data-tag="tag">value2</div>
  </div>

XML:
  <top>
    <data>
      <tag>value1</tag>
      <tag>value2</tag>
    </data>
  </top>

JSON:
  {
    "top": {
      "data": {
      "tag": "value1",
      "tag": "value2"
      }
    }
  }
```

11.1 Lists and Instances

A "*list*" is set of one or more instances that appear under the same parent. The instances contain details about a specific object. One can think of instances as objects or records. A call is needed to open and close the list, while a distinct call is needed to open and close each instance of the list.

Use the `--open-list` and `--open-instances` to open lists and instances. Use the `--close-list` and `--close-instances` to close them. Each of these options take a `name` parameter, providing the name of the list and instance.

In the following example, a list named "machine" is created with three instances:

```
opts="--json"
xo $opts --open-list machine
NF=
for name in red green blue; do
    xo $opts --depth 1 $NF --open-instance machine
    xo $opts --depth 2 "Machine {k:name} has {:memory}\n" $name 55
```

(continues on next page)

(continued from previous page)

```
    xo $opts --depth 1 --close-instance machine
    NF=--not-first
done
xo $opts $NF --close-list machine
```

The normal `libxo` functions use a state machine to help these transitions, but since each `xo` command is invoked independent of the previous calls, the state must be passed in explicitly via these command line options.

The `--instance` option can be used to treat a single `xo` invocation as an instance with the given set of fields:

```
% xo --libxo:XP --instance foo 'The {:product} is {:status}\n' stereo "in route"
<foo>
  <product>stereo</product>
  <status>in route</status>
</foo>
```

11.2 Command Line Options

```
Usage: xo [options] format [fields]
  --close <path>        Close tags for the given path
  --close-instance <name> Close an open instance name
  --close-list <name>   Close an open list name
  --continuation OR -C  Output belongs on same line as previous output
  --depth <num>         Set the depth for pretty printing
  --help                Display this help text
  --html OR -H          Generate HTML output
  --instance OR -I <name> Wrap in an instance of the given name
  --json OR -J          Generate JSON output
  --leading-xpath <path> Add a prefix to generated XPaths (HTML)
  --not-first           Indicate this object is not the first (JSON)
  --open <path>         Open tags for the given path
  --open-instance <name> Open an instance given by name
  --open-list <name>   Open a list given by name
  --option <opts> -or -O <opts>  Give formatting options
  --pretty OR -p        Make 'pretty' output (add indent, newlines)
  --style <style>       Generate given style (xml, json, text, html)
  --text OR -T          Generate text output (the default style)
  --top-wrap            Generate a top-level object wrapper (JSON)
  --version             Display version information
  --warn OR -W          Display warnings in text on stderr
  --warn-xml            Display warnings in xml on stdout
  --wrap <path>         Wrap output in a set of containers
  --xml OR -X           Generate XML output
  --xpath               Add XPath data to HTML output)
```

11.3 Example

```
% xo 'The {:product} is {:status}\n' stereo "in route"
The stereo is in route
% xo -p -X 'The {:product} is {:status}\n' stereo "in route"
<product>stereo</product>
```

(continues on next page)

(continued from previous page)

```
<status>in route</status>
% xo --libxo xml,pretty 'The {:product} is {:status}\n' stereo "in route"
<product>stereo</product>
<status>in route</status>
```

CHAPTER 12

xolint

`xolint` is a tool for reporting common mistakes in format strings in source code that invokes `xo_emit`. It allows these errors to be diagnosed at build time, rather than waiting until runtime.

`xolint` takes the one or more C files as arguments, and reports and errors, warning, or informational messages as needed:

Option	Meaning
-c	Invoke 'cpp' against the input file
-C <flags>	Flags that are passed to 'cpp
-d	Enable debug output
-D	Generate documentation for all xolint messages
-I	Generate info table code
-p	Print the offending lines after the message
-V	Print vocabulary of all field names
-X	Extract samples from xolint, suitable for testing

The output message will contain the source filename and line number, the class of the message, the message, and, if -p is given, the line that contains the error:

```
% xolint.pl -t xolint.c
xolint.c: 16: error: anchor format should be "%d"
16          xo_emit("{[:/%s}");
```

The "-I" option will generate a table of xo_info_t structures, suitable for inclusion in source code.

The "-V" option does not report errors, but prints a complete list of all field names, sorted alphabetically. The output can help spot inconsistencies and spelling errors.

CHAPTER 13

xohtml

`xohtml` is a tool for turning the output of libxo-enabled commands into html files suitable for display in modern HTML web browsers. It can be used to test and debug HTML output, as well as to make the user ache to escape the world of '70s terminal devices.

`xohtml` is given a command, either on the command line or via the "-c" option. If not command is given, standard input is used. The command's output is wrapped in HTML tags, with references to supporting CSS and Javascript files, and written to standard output or the file given in the "-f" option. The "-b" option can be used to provide an alternative base path for the support files:

Option	Meaning
-b <base>	Base path for finding css/javascript files
-c <command>	Command to execute
-f <file>	Output file name

The "-c" option takes a full command with arguments, including any libxo options needed to generate html (`--libxo=html`). This value must be quoted if it consists of multiple tokens.

CHAPTER 14

xopo

The xopo utility filters ".pot" files generated by the *xgettext(1)* utility to remove formatting information suitable for use with the "{G:}" modifier. This means that when the developer changes the formatting portion of the field definitions, or the fields modifiers, the string passed to *gettext(3)* is unchanged, avoiding the expense of updating any existing translation files (".po" files).

The syntax for the xopo command is one of two forms; it can be used as a filter for processing a .po or .pot file, rewriting the "*msgid*" strings with a simplified message string. In this mode, the input is either standard input or a file given by the "-f" option, and the output is either standard output or a file given by the "-o" option.

In the second mode, a simple message given using the "-s" option on the command, and the simplified version of that message is printed on stdout:

Option	Meaning
-o <file>	Output file name
-f <file>	Use the given .po file as input
-s <text>	Simplify a format string

```
EXAMPLE:
    % xopo -s "There are {:count/%u} {:event/%.6s} events\n"
    There are {:count} {:event} events\n

    % xgettext --default-domain=foo --no-wrap \
        --add-comments --keyword=xo_emit --keyword=xo_emit_h \
        --keyword=xo_emit_warn -C -E -n --foreign-user \
        -o foo.pot.raw foo.c
    % xopo -f foo.pot.raw -o foo.pot
```

Use of the --no-wrap option for xgettext is required to ensure that incoming msgid strings are not wrapped across multiple lines.

CHAPTER 15

FAQs

This section contains the set of questions that users typically ask, along with answers that might be helpful.

15.1 General

15.1.1 Can you share the history of libxo?

In 2001, we added an XML API to the JUNOS operating system, which is built on top of FreeBSD. Eventually this API became standardized as the NETCONF API (**RFC 6241**). As part of this effort, we modified many FreeBSD utilities to emit XML, typically via a "-X" switch. The results were mixed. The cost of maintaining this code, updating it, and carrying it were non-trivial, and contributed to our expense (and the associated delay) with upgrading the version of FreeBSD on which each release of JUNOS is based.

A recent (2014) effort within JUNOS aims at removing our modifications to the underlying FreeBSD code as a means of reducing the expense and delay in tracking HEAD. JUNOS is structured to have system components generate XML that is rendered by the CLI (think: login shell) into human-readable text. This allows the API to use the same plumbing as the CLI, and ensures that all components emit XML, and that it is emitted with knowledge of the consumer of that XML, yielding an API that have no incremental cost or feature delay.

libxo is an effort to mix the best aspects of the JUNOS strategy into FreeBSD in a seemless way, allowing commands to make printf-like output calls with a single code path.

15.1.2 Did the complex semantics of format strings evolve over time?

The history is both long and short: libxo's functionality is based on what JUNOS does in a data modeling language called ODL (output definition language). In JUNOS, all subcomponents generate XML, which is feed to the CLI, where data from the ODL files tell is how to render that XML into text. ODL might had a set of tags like:

```
tag docsis-state {
    help "State of the DOCSIS interface";
    type string;
```

(continues on next page)

(continued from previous page)

```
}

tag docsis-mode {
    help "DOCSIS mode (2.0/3.0) of the DOCSIS interface";
    type string;
}

tag docsis-upstream-speed {
    help "Operational upstream speed of the interface";
    type string;
}

tag downstream-scanning {
    help "Result of scanning in downstream direction";
    type string;
}

tag ranging {
    help "Result of ranging action";
    type string;
}

tag signal-to-noise-ratio {
    help "Signal to noise ratio for all channels";
    type string;
}

tag power {
    help "Operational power of the signal on all channels";
    type string;
}

format docsis-status-format {
    picture "
State   : @, Mode: @, Upstream speed: @
Downstream scanning: @, Ranging: @
Signal to noise ratio: @
Power: @
";
    line {
        field docsis-state;
        field docsis-mode;
        field docsis-upstream-speed;
        field downstream-scanning;
        field ranging;
        field signal-to-noise-ratio;
        field power;
    }
}
```

These tag definitions are compiled into field definitions that are triggered when matching XML elements are seen. ODL also supports other means of defining output.

The roles and modifiers describe these details.

In moving these ideas to bsd, two things had to happen: the formatting had to happen at the source since BSD won't have a JUNOS-like CLI to do the rendering, and we can't depend on external data models like ODL, which was seen as too hard a sell to the BSD community.

The results were that the xo_emit strings are used to encode the roles, modifiers, names, and formats. They are dense and a bit cryptic, but not so unlike printf format strings that developers will be lost.

libxo is a new implementation of these ideas and is distinct from the previous implementation in JUNOS.

15.1.3 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 Don't 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's worse than writing expressions like:

```
if ($src1/process[pid == $pid]/name ==
    $src2/proc-table/proc-list
                /prc-entry[prcss-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". Consistency rocks!

Use containment as scoping In the previous example, all the names are prefixed with "proc-", which is redundant given that they are nested under the process table.

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() calls (*Attributes (xo_attr)*) or "{e:}" fields (*The Encoding Modifier ({e:})*) to make the data useful.

Don't 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's 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` to find errors in your field descriptors, use "`xolint -V`" to spell check your field names and to help you 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.

15.1.4 What does this message mean?

'A percent sign appearing in text is a literal'

The message "A percent sign appearing in text is a literal" can be caused by code like:

```
xo_emit("cost: %d", cost);
```

This code should be replaced with code like:

```
xo_emit("{L:cost}: {:cost/%d}", cost);
```

This can be a bit surprising and could be a field that was not properly converted to a libxo-style format string.

'Unknown long name for role/modifier'

The message "Unknown long name for role/modifier" can be caused by code like:

```
xo_emit("{,humanization:value}", value);
```

This code should be replaced with code like:

```
xo_emit("{,humanize:value}", value);
```

The hn-* modifiers (hn-decimal, hn-space, hn-1000) are only valid for fields with the {h:} modifier.

'Last character before field definition is a field type'

The message "Last character before field definition is a field type" can be caused by code like: A common typo:

```
xo_emit("{T:Min} T{:Max}");
```

This code should be replaced with code like:

```
xo_emit("{T:Min} {T:Max}");
```

Twiddling the "{" and the field role is a common typo.

'Encoding format uses different number of arguments'

The message "Encoding format uses different number of arguments" can be caused by code like:

```
xo_emit("{:name/%6.6s %%04d/%s}", name, number);
```

This code should be replaced with code like:

```
xo_emit("{:name/%6.6s %04d/%s-%d}", name, number);
```

Both format should consume the same number of arguments off the stack

'Only one field role can be used'

The message "Only one field role can be used" can be caused by code like:

```
xo_emit("{LT:Max}");
```

This code should be replaced with code like:

```
xo_emit("{T:Max}");
```

'Potential missing slash after C, D, N, L, or T with format'

The message "Potential missing slash after C, D, N, L, or T with format" can be caused by code like:

```
xo_emit("{T:%6.6s}\n", "Max");
```

This code should be replaced with code like:

```
xo_emit("{T:/%6.6s}\n", "Max");
```

The "%6.6s" will be a literal, not a field format. While it's possibly valid, it's likely a missing "/".

'An encoding format cannot be given (roles: DNLT)'

The message "An encoding format cannot be given (roles: DNLT)" can be caused by code like:

```
xo_emit("{T:Max//%s}", "Max");
```

Fields with the C, D, N, L, and T roles are not emitted in the 'encoding' style (JSON, XML), so an encoding format would make no sense.

'Format cannot be given when content is present (roles: CDLN)'

The message "Format cannot be given when content is present (roles: CDLN)" can be caused by code like:

```
xo_emit("{N:Max/%6.6s}", "Max");
```

Fields with the C, D, L, or N roles can't have both static literal content ("{L:Label}") and a format ("{L:/%s}"). This error will also occur when the content has a backslash in it, like "{N:Type of I/O}"; backslashes should be escaped, like "{N:Type of I\/O}". Note the double backslash, one for handling 'C' strings, and one for libxo.

'Field has color without fg- or bg- (role: C)'

The message "Field has color without fg- or bg- (role: C)" can be caused by code like:

```
xo_emit("{C:green}{:foo}{C:}", x);
```

This code should be replaced with code like:

```
xo_emit("{C:fg-green}{:foo}{C:}", x);
```

Colors must be prefixed by either "fg-" or "bg-".

'Field has invalid color or effect (role: C)'

The message "Field has invalid color or effect (role: C)" can be caused by code like:

```
xo_emit("{C:fg-purple,bold}{:foo}{C:gween}", x);
```

This code should be replaced with code like:

```
xo_emit("{C:fg-red,bold}{:foo}{C:fg-green}", x);
```

The list of colors and effects are limited. The set of colors includes default, black, red, green, yellow, blue, magenta, cyan, and white, which must be prefixed by either "fg-" or "bg-". Effects are limited to bold, no-bold, underline, no-underline, inverse, no-inverse, normal, and reset. Values must be separated by commas.

'Field has humanize modifier but no format string'

The message "Field has humanize modifier but no format string" can be caused by code like:

```
xo_emit("{h:value}", value);
```

This code should be replaced with code like:

```
xo_emit("{h:value/%d}", value);
```

Humanization is only value for numbers, which are not likely to use the default format ("%s").

'Field has hn-* modifier but not 'h' modifier'

The message "Field has hn-* modifier but not 'h' modifier" can be caused by code like:

```
xo_emit("{,hn-1000:value}", value);
```

This code should be replaced with code like:

```
xo_emit("{h,hn-1000:value}", value);
```

The hn-* modifiers (hn-decimal, hn-space, hn-1000) are only valid for fields with the {h:} modifier.

'Value field must have a name (as content)")'

The message "Value field must have a name (as content)")" can be caused by code like:

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

This code should be replaced with code like:

```
xo_emit("{:tag-name/%s}", "value");
```

The field name is used for XML and JSON encodings. These tags names are static and must appear directly in the field descriptor.

'Use hyphens, not underscores, for value field name'

The message "Use hyphens, not underscores, for value field name" can be caused by code like:

```
xo_emit("{:no_under_scores}", "bad");
```

This code should be replaced with code like:

```
xo_emit("{:no-under-scores}", "bad");
```

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.

'Value field name cannot start with digit'

The message "Value field name cannot start with digit" can be caused by code like:

```
xo_emit("{:10-gig/}");
```

This code should be replaced with code like:

```
xo_emit("{:ten-gig/}");
```

XML element names cannot start with a digit.

'Value field name should be lower case'

The message "Value field name should be lower case" can be caused by code like:

```
xo_emit("{:WHY-ARE-YOU-SHOUTING}", "NO REASON");
```

This code should be replaced with code like:

```
xo_emit("{:why-are-you-shouting}", "no reason");
```

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. Lower case rules the seas.

'Value field name should be longer than two characters'

The message "Value field name should be longer than two characters" can be caused by code like:

```
xo_emit("{:x}", "mumble");
```

This code should be replaced with code like:

```
xo_emit("{:something-meaningful}", "mumble");
```

Field names should be descriptive, and it's hard to be descriptive in less than two characters. Consider your users and try to make something more useful. Note that this error often occurs when the field type is placed after the colon ("{:T/%20s}"), instead of before it ("{T:/20s}").

'Value field name contains invalid character'

The message "Value field name contains invalid character" can be caused by code like:

```
xo_emit("{:cost-in-$$/%u}", 15);
```

This code should be replaced with code like:

```
xo_emit("{:cost-in-dollars/%u}", 15);
```

An invalid character is often a sign of a typo, like "{:]}" instead of "{]:}". Field names are restricted to lower-case characters, digits, and hyphens.

'decoration field contains invalid character'

The message "decoration field contains invalid character" can be caused by code like:

```
xo_emit("{D:not good}");
```

This code should be replaced with code like:

```
xo_emit("{D:(){:good}{D:)}", "yes");
```

This is minor, but fields should use proper roles. Decoration fields are meant to hold punctuation and other characters used to decorate the content, typically to make it more readable to human readers.

'Anchor content should be decimal width'

The message "Anchor content should be decimal width" can be caused by code like:

```
xo_emit("{[:mumble}");
```

This code should be replaced with code like:

```
xo_emit("{[:32}");
```

Anchors need an integer value to specify the width of the set of anchored fields. The value can be positive (for left padding/right justification) or negative (for right padding/left justification) and can appear in either the start or stop anchor field descriptor.

'Anchor format should be "%d"'

The message "Anchor format should be "%d"" can be caused by code like:

```
xo_emit("{[:/%s}");
```

This code should be replaced with code like:

```
xo_emit("{[:/%d}");
```

Anchors only grok integer values, and if the value is not static, if must be in an 'int' argument, represented by the "%d" format. Anything else is an error.

'Anchor cannot have both format and encoding format")'

The message "Anchor cannot have both format and encoding format")" can be caused by code like:

```
xo_emit("{[:32/%d}");
```

This code should be replaced with code like:

```
xo_emit("{[:32}");
```

Anchors can have a static value or argument for the width, but cannot have both.

'Max width only valid for strings'

The message "Max width only valid for strings" can be caused by code like:

```
xo_emit("{:tag/%2.4.6d}", 55);
```

This code should be replaced with code like:

```
xo_emit("{:tag/%2.6d}", 55);
```

libxo allows a true 'max width' in addition to the traditional printf-style 'max number of bytes to use for input'. But this is supported only for string values, since it makes no sense for non-strings. This error may occur from a typo, like "{:tag/%6..6d}" where only one period should be used.

CHAPTER 16

Howtos: Focused Directions

This section provides task-oriented instructions for selected tasks. If you have a task that needs instructions, please open a request as an enhancement issue on github.

16.1 Howto: Report bugs

libxo uses github to track bugs or request enhancements. Please use the following URL:

https://github.com/Juniper/libxo/issues

16.2 Howto: Install libxo

libxo is open source, under a new BSD license. Source code is available on github, as are recent releases. To get the most current release, please visit:

https://github.com/Juniper/libxo/releases

After downloading and untarring the source code, building involves the following steps:

```
sh bin/setup.sh
cd build
../configure
make
make test
sudo make install
```

libxo uses a distinct "*build*" directory to keep generated files separated from source files.

Use "`../configure --help`" to display available configuration options, which include the following:

```
--enable-warnings      Turn on compiler warnings
--enable-debug         Turn on debugging
--enable-text-only     Turn on text-only rendering
--enable-printflike    Enable use of GCC __printflike attribute
--disable-libxo-options  Turn off support for LIBXO_OPTIONS
--with-gettext=PFX     Specify location of gettext installation
--with-libslax-prefix=PFX  Specify location of libslax config
```

Compiler warnings are a very good thing, but recent compiler version have added some very pedantic checks. While every attempt is made to keep libxo code warning-free, warnings are now optional. If you are doing development work on libxo, it is required that you use –enable-warnings to keep the code warning free, but most users need not use this option.

libxo provides the `--enable-text-only` option to reduce the footprint of the library for smaller installations. XML, JSON, and HTML rendering logic is removed.

The gettext library does not provide a simple means of learning its location, but libxo will look for it in /usr and /opt/local. If installed elsewhere, the installer will need to provide this information using the "`--with-gettext=/dir/path`" option.

libslax is not required by libxo; it contains the "oxtradoc" program used to format documentation.

For additional information, see *Building libxo*.

16.3 Howto: Convert command line applications

Common question: How do I convert an existing command line application?

There are four basic steps for converting command line application to use libxo:

```
- Setting up the context
- Converting printf calls
- Creating hierarchy
- Converting error functions
```

16.3.1 Setting up the context

To use libxo, you'll need to include the "xo.h" header file in your source code files:

```
#include <libxo/xo.h>
```

In your main() function, you'll need to call xo_parse_args to handling argument parsing (*Parsing Command-line Arguments (xo_parse_args)*). This function removes libxo-specific arguments the program's argv and returns either the number of remaining arguments or -1 to indicate an error:

```
int
main (int argc, char **argv)
{
    argc = xo_parse_args(argc, argv);
    if (argc < 0)
        return argc;
    ....
}
```

At the bottom of your main(), you'll need to call xo_finish() to complete output processing for the default handle (*Handles*). This is required to flush internal information buffers. libxo provides the xo_finish_atexit function that is suitable for use with the *atexit(3)* function:

```
atexit(xo_finish_atexit);
```

16.3.2 Converting printf Calls

The second task is inspecting code for *printf(3)* calls and replacing them with xo_emit() calls. The format strings are similar in task, but libxo format strings wrap output fields in braces. The following two calls produce identical text output:

```
OLD::
  printf("There are %d %s events\n", count, etype);

NEW::
  xo_emit("There are {:count/%d} {:event} events\n", count, etype);
```

"count" and "event" are used as names for JSON and XML output. The "count" field uses the format "%d" and "event" uses the default "%s" format. Both are "value" roles, which is the default role.

Since text outside of output fields is passed verbatim, other roles are less important, but their proper use can help make output more useful. The "note" and "label" roles allow HTML output to recognize the relationship between text and the associated values, allowing appropriate "hover" and "onclick" behavior. Using the "units" role allows the presentation layer to perform conversions when needed. The "warning" and "error" roles allows use of color and font to draw attention to warnings. The "padding" role makes the use of vital whitespace more clear (*The Padding Role ({P:})*).

The "*title*" role indicates the headings of table and sections. This allows HTML output to use CSS to make this relationship more obvious:

```
OLD::
  printf("Statistics:\n");

NEW::
  xo_emit("{T:Statistics}:\n");
```

The "*color*" roles controls foreground and background colors, as well as effects like bold and underline (see *The Color Role ({C:})*):

```
NEW::
  xo_emit("{C:bold}required{C:}\n");
```

Finally, the start- and stop-anchor roles allow justification and padding over multiple fields (see *The Anchor Roles ({[:} and {]:})*):

```
OLD::
  snprintf(buf, sizeof(buf), "(%u/%u/%u)", min, ave, max);
  printf("%30s", buf);

NEW::
  xo_emit("{[:30}({:minimum/%u}/{:average/%u}/{:maximum/%u}{]:}",
          min, ave, max);
```

16.3.3 Creating Hierarchy

Text output doesn't have any sort of hierarchy, but XML and JSON require this. Typically applications use indentation to represent these relationship:

```
OLD::
  printf("table %d\n", tnum);
  for (i = 0; i < tmax; i++) {
      printf("    %s %d\n", table[i].name, table[i].size);
  }

NEW::
  xo_emit("{T:/table %d}\n", tnum);
  xo_open_list("table");
  for (i = 0; i < tmax; i++) {
      xo_open_instance("table");
      xo_emit("{P:    }{k:name} {:size/%d}\n",
              table[i].name, table[i].size);
      xo_close_instance("table");
  }
  xo_close_list("table");
```

The open and close list functions are used before and after the list, and the open and close instance functions are used before and after each instance with in the list.

Typically these developer looks for a "for" loop as an indication of where to put these calls.

In addition, the open and close container functions allow for organization levels of hierarchy:

```
OLD::
  printf("Paging information:\n");
  printf("    Free:      %lu\n", free);
  printf("    Active:    %lu\n", active);
  printf("    Inactive:  %lu\n", inactive);

NEW::
  xo_open_container("paging-information");
  xo_emit("{P:    }{L:Free:      }{:free/%lu}\n", free);
  xo_emit("{P:    }{L:Active:    }{:active/%lu}\n", active);
  xo_emit("{P:    }{L:Inactive:  }{:inactive/%lu}\n", inactive);
  xo_close_container("paging-information");
```

16.3.4 Converting Error Functions

libxo provides variants of the standard error and warning functions, *err(3)* and *warn(3)*. There are two variants, one for putting the errors on standard error, and the other writes the errors and warnings to the handle using the appropriate encoding style:

```
OLD::
  err(1, "cannot open output file: %s", file);

NEW::
  xo_err(1, "cannot open output file: %s", file);
  xo_emit_err(1, "cannot open output file: {:filename}", file);
```

16.3.5 Call xo_finish

One important item: call `xo_finish` at the end of your program so ensure that all buffered data is written out. You can call it explicitly call it, or use *atexit(3)* to have `xo_finish_atexit` called implicitly on exit:

```
OLD::
  exit(0);

NEW::
  xo_finish();
  exit(0);
```

16.4 Howto: Use "xo" in Shell Scripts

Needed

Documentation is needed for this area.

16.5 Howto: Internationalization (i18n)

How do I use libxo to support internationalization?

libxo allows format and field strings to be used a keys into message catalogs to enable translation into a user's native language by invoking the standard *gettext(3)* functions.

gettext setup is a bit complicated: text strings are extracted from source files into "*portable object template*" (.pot) files using the `xgettext` command. For each language, this template file is used as the source for a message catalog in the "*portable object*" (.po) format, which are translated by hand and compiled into "*machine object*" (.mo) files using the `msgfmt` command. The .mo files are then typically installed in the /usr/share/locale or /opt/local/share/locale directories. At run time, the user's language settings are used to select a .mo file which is searched for matching messages. Text strings in the source code are used as keys to look up the native language strings in the .mo file.

Since the xo_emit format string is used as the key into the message catalog, libxo removes unimportant field formatting and modifiers from the format string before use so that minor formatting changes will not impact the expensive translation process. We don't want a developer change such as changing "/%06d" to "/%08d" to 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:

```
EXAMPLE:
    % xopo -s "There are {:count/%u} {:event/%.6s} events\n"
    There are {:count} {:event} events\n

Recommended workflow:
    # Extract text messages
    xgettext --default-domain=foo --no-wrap \
        --add-comments --keyword=xo_emit --keyword=xo_emit_h \
        --keyword=xo_emit_warn -C -E -n --foreign-user \
        -o foo.pot.raw foo.c

    # Simplify format strings for libxo
    xopo -f foo.pot.raw -o foo.pot
```

(continues on next page)

(continued from previous page)

```
    # For a new language, just copy the file
    cp foo.pot po/LC/my_lang/foo.po

    # For an existing language:
    msgmerge --no-wrap po/LC/my_lang/foo.po \
            foo.pot -o po/LC/my_lang/foo.po.new

    # Now the hard part: translate foo.po using tools
    # like poedit or emacs' po-mode

    # Compile the finished file; Use of msgfmt's "-v" option is
    # strongly encouraged, so that "fuzzy" entries are reported.
    msgfmt -v -o po/my_lang/LC_MESSAGES/foo.mo po/my_lang/foo.po

    # Install the .mo file
    sudo cp po/my_lang/LC_MESSAGES/foo.mo \
            /opt/local/share/locale/my_lang/LC_MESSAGE/
```

Once these steps are complete, you can use the `gettext` command to test the message catalog:

```
gettext -d foo -e "some text"
```

16.5.1 i18n and xo_emit

There are three features used in libxo used to support i18n:

- The "{G:}" role looks for a translation of the format string.
- The "{g:}" modifier looks for a translation of the field.
- The "{p:}" modifier looks for a pluralized version of the field.

Together these three flags allows a single function call to give native language support, as well as libxo's normal XML, JSON, and HTML support:

```
printf(gettext("Received %zu %s from {g:server} server\n"),
       counter, ngettext("byte", "bytes", counter),
       gettext("web"));

xo_emit("{G:}Received {:received/%zu} {Ngp:byte,bytes} "
        "from {g:server} server\n", counter, "web");
```

libxo will see the "{G:}" role and will first simplify the format string, removing field formats and modifiers:

```
"Received {:received} {N:byte,bytes} from {:server} server\n"
```

libxo calls *gettext(3)* with that string to get a localized version. If your language were *Pig Latin*, the result might look like:

```
"Eceivedray {:received} {N:byte,bytes} omfray "
           "{:server} erversay\n"
```

Note the field names do not change and they should not be translated. The contents of the note ("byte,bytes") should also not be translated, since the "g" modifier will need the untranslated value as the key for the message catalog.

The field "{g:server}" requests the rendered value of the field be translated using *gettext(3)*. In this example, "web" would be used.

The field "{Ngp:byte,bytes}" shows an example of plural form using the "{p:}" modifier with the "{g:}" modifier. The base singular and plural forms appear inside the field, separated by a comma. At run time, libxo uses the previous field's numeric value to decide which form to use by calling *ngettext(3)*.

If a domain name is needed, it can be supplied as the content of the {G:} role. Domain names remain in use throughout the format string until cleared with another domain name:

```
printf(dgettext("dns", "Host %s not found: %d(%s)\n"),
    name, errno, dgettext("strerror", strerror(errno)));

xo_emit("{G:dns}Host {:hostname} not found: "
        "%d({G:strerror}{g:%m})\n", name, errno);
```

CHAPTER 17

Examples

17.1 Unit Test

Here is one of the unit tests as an example:

```
int
main (int argc, char **argv)
{
    static char base_grocery[] = "GRO";
    static char base_hardware[] = "HRD";
    struct item {
        const char *i_title;
        int i_sold;
        int i_instock;
        int i_onorder;
        const char *i_sku_base;
        int i_sku_num;
    };
    struct item list[] = {
        { "gum", 1412, 54, 10, base_grocery, 415 },
        { "rope", 85, 4, 2, base_hardware, 212 },
        { "ladder", 0, 2, 1, base_hardware, 517 },
        { "bolt", 4123, 144, 42, base_hardware, 632 },
        { "water", 17, 14, 2, base_grocery, 2331 },
        { NULL, 0, 0, 0, NULL, 0 }
    };
    struct item list2[] = {
        { "fish", 1321, 45, 1, base_grocery, 533 },
    };
    struct item *ip;
    xo_info_t info[] = {
        { "in-stock", "number", "Number of items in stock" },
        { "name", "string", "Name of the item" },
        { "on-order", "number", "Number of items on order" },
```

(continues on next page)

(continued from previous page)

```
        { "sku", "string", "Stock Keeping Unit" },
        { "sold", "number", "Number of items sold" },
        { NULL, NULL, NULL },
    };
    int info_count = (sizeof(info) / sizeof(info[0])) - 1;

    argc = xo_parse_args(argc, argv);
    if (argc < 0)
        exit(EXIT_FAILURE);

    xo_set_info(NULL, info, info_count);

    xo_open_container_h(NULL, "top");

    xo_open_container("data");
    xo_open_list("item");

    for (ip = list; ip->i_title; ip++) {
        xo_open_instance("item");

        xo_emit("{L:Item} '{k:name/%s}':\n", ip->i_title);
        xo_emit("{P:   }{L:Total sold}: {n:sold/%u%s}\n",
                ip->i_sold, ip->i_sold ? ".0" : "");
        xo_emit("{P:   }{Lwc:In stock}{:in-stock/%u}\n",
                ip->i_instock);
        xo_emit("{P:   }{Lwc:On order}{:on-order/%u}\n",
                ip->i_onorder);
        xo_emit("{P:   }{L:SKU}: {q:sku/%s-000-%u}\n",
                ip->i_sku_base, ip->i_sku_num);

        xo_close_instance("item");
    }

    xo_close_list("item");
    xo_close_container("data");

    xo_open_container("data");
    xo_open_list("item");

    for (ip = list2; ip->i_title; ip++) {
        xo_open_instance("item");

        xo_emit("{L:Item} '{:name/%s}':\n", ip->i_title);
        xo_emit("{P:   }{L:Total sold}: {n:sold/%u%s}\n",
                ip->i_sold, ip->i_sold ? ".0" : "");
        xo_emit("{P:   }{Lwc:In stock}{:in-stock/%u}\n",
                ip->i_instock);
        xo_emit("{P:   }{Lwc:On order}{:on-order/%u}\n",
                ip->i_onorder);
        xo_emit("{P:   }{L:SKU}: {q:sku/%s-000-%u}\n",
                ip->i_sku_base, ip->i_sku_num);

        xo_close_instance("item");
    }

    xo_close_list("item");
    xo_close_container("data");
```

(continues on next page)

(continued from previous page)

```
    xo_close_container_h(NULL, "top");

    return 0;
}
```

Text output:

```
% ./testxo --libxo text
Item 'gum':
   Total sold: 1412.0
   In stock: 54
   On order: 10
   SKU: GRO-000-415
Item 'rope':
   Total sold: 85.0
   In stock: 4
   On order: 2
   SKU: HRD-000-212
Item 'ladder':
   Total sold: 0
   In stock: 2
   On order: 1
   SKU: HRD-000-517
Item 'bolt':
   Total sold: 4123.0
   In stock: 144
   On order: 42
   SKU: HRD-000-632
Item 'water':
   Total sold: 17.0
   In stock: 14
   On order: 2
   SKU: GRO-000-2331
Item 'fish':
   Total sold: 1321.0
   In stock: 45
   On order: 1
   SKU: GRO-000-533
```

JSON output:

```
% ./testxo --libxo json,pretty
"top": {
  "data": {
    "item": [
      {
        "name": "gum",
        "sold": 1412.0,
        "in-stock": 54,
        "on-order": 10,
        "sku": "GRO-000-415"
      },
      {
        "name": "rope",
        "sold": 85.0,
        "in-stock": 4,
```

(continues on next page)

(continued from previous page)

```
        "on-order": 2,
        "sku": "HRD-000-212"
      },
      {
        "name": "ladder",
        "sold": 0,
        "in-stock": 2,
        "on-order": 1,
        "sku": "HRD-000-517"
      },
      {
        "name": "bolt",
        "sold": 4123.0,
        "in-stock": 144,
        "on-order": 42,
        "sku": "HRD-000-632"
      },
      {
        "name": "water",
        "sold": 17.0,
        "in-stock": 14,
        "on-order": 2,
        "sku": "GRO-000-2331"
      }
    ]
  },
  "data": {
    "item": [
      {
        "name": "fish",
        "sold": 1321.0,
        "in-stock": 45,
        "on-order": 1,
        "sku": "GRO-000-533"
      }
    ]
  }
}
```

XML output:

```
% ./testxo --libxo pretty,xml
<top>
  <data>
    <item>
      <name>gum</name>
      <sold>1412.0</sold>
      <in-stock>54</in-stock>
      <on-order>10</on-order>
      <sku>GRO-000-415</sku>
    </item>
    <item>
      <name>rope</name>
      <sold>85.0</sold>
      <in-stock>4</in-stock>
      <on-order>2</on-order>
      <sku>HRD-000-212</sku>
```

(continues on next page)

(continued from previous page)

```
    </item>
    <item>
      <name>ladder</name>
      <sold>0</sold>
      <in-stock>2</in-stock>
      <on-order>1</on-order>
      <sku>HRD-000-517</sku>
    </item>
    <item>
      <name>bolt</name>
      <sold>4123.0</sold>
      <in-stock>144</in-stock>
      <on-order>42</on-order>
      <sku>HRD-000-632</sku>
    </item>
    <item>
      <name>water</name>
      <sold>17.0</sold>
      <in-stock>14</in-stock>
      <on-order>2</on-order>
      <sku>GRO-000-2331</sku>
    </item>
  </data>
  <data>
    <item>
      <name>fish</name>
      <sold>1321.0</sold>
      <in-stock>45</in-stock>
      <on-order>1</on-order>
      <sku>GRO-000-533</sku>
    </item>
  </data>
</top>
```

HMTL output:

```
% ./testxo --libxo pretty,html
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name">gum</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold">1412.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock">54</div>
</div>
<div class="line">
```

(continues on next page)

(continued from previous page)

```
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order">10</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku">GRO-000-415</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name">rope</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold">85.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock">4</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order">2</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku">HRD-000-212</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name">ladder</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold">0</div>
</div>
<div class="line">
```

(continues on next page)

(continued from previous page)

```
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock">2</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order">1</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku">HRD-000-517</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name">bolt</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold">4123.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock">144</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order">42</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku">HRD-000-632</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name">water</div>
  <div class="text">':</div>
</div>
```

(continues on next page)

(continued from previous page)

```
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold">17.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock">14</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order">2</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku">GRO-000-2331</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name">fish</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold">1321.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock">45</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order">1</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku">GRO-000-533</div>
```

(continues on next page)

(continued from previous page)

```
</div>
```

HTML output with xpath and info flags:

```
% ./testxo --libxo pretty,html,xpath,info
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name"
       data-xpath="/top/data/item/name" data-type="string"
       data-help="Name of the item">gum</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold"
       data-xpath="/top/data/item/sold" data-type="number"
       data-help="Number of items sold">1412.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock"
       data-xpath="/top/data/item/in-stock" data-type="number"
       data-help="Number of items in stock">54</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order"
       data-xpath="/top/data/item/on-order" data-type="number"
       data-help="Number of items on order">10</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku"
       data-xpath="/top/data/item/sku" data-type="string"
       data-help="Stock Keeping Unit">GRO-000-415</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name"
       data-xpath="/top/data/item/name" data-type="string"
       data-help="Name of the item">rope</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
```

(continues on next page)

(continued from previous page)

```
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold"
       data-xpath="/top/data/item/sold" data-type="number"
       data-help="Number of items sold">85.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock"
       data-xpath="/top/data/item/in-stock" data-type="number"
       data-help="Number of items in stock">4</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order"
       data-xpath="/top/data/item/on-order" data-type="number"
       data-help="Number of items on order">2</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku"
       data-xpath="/top/data/item/sku" data-type="string"
       data-help="Stock Keeping Unit">HRD-000-212</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name"
       data-xpath="/top/data/item/name" data-type="string"
       data-help="Name of the item">ladder</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold"
       data-xpath="/top/data/item/sold" data-type="number"
       data-help="Number of items sold">0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock"
       data-xpath="/top/data/item/in-stock" data-type="number"
       data-help="Number of items in stock">2</div>
</div>
```

(continues on next page)

(continued from previous page)

```
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order"
       data-xpath="/top/data/item/on-order" data-type="number"
       data-help="Number of items on order">1</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku"
       data-xpath="/top/data/item/sku" data-type="string"
       data-help="Stock Keeping Unit">HRD-000-517</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name"
       data-xpath="/top/data/item/name" data-type="string"
       data-help="Name of the item">bolt</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold"
       data-xpath="/top/data/item/sold" data-type="number"
       data-help="Number of items sold">4123.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <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>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order"
       data-xpath="/top/data/item/on-order" data-type="number"
       data-help="Number of items on order">42</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku"
       data-xpath="/top/data/item/sku" data-type="string"
```

(continues on next page)

(continued from previous page)

```
       data-help="Stock Keeping Unit">HRD-000-632</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name"
       data-xpath="/top/data/item/name" data-type="string"
       data-help="Name of the item">water</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold"
       data-xpath="/top/data/item/sold" data-type="number"
       data-help="Number of items sold">17.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock"
       data-xpath="/top/data/item/in-stock" data-type="number"
       data-help="Number of items in stock">14</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order"
       data-xpath="/top/data/item/on-order" data-type="number"
       data-help="Number of items on order">2</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku"
       data-xpath="/top/data/item/sku" data-type="string"
       data-help="Stock Keeping Unit">GRO-000-2331</div>
</div>
<div class="line">
  <div class="label">Item</div>
  <div class="text"> '</div>
  <div class="data" data-tag="name"
       data-xpath="/top/data/item/name" data-type="string"
       data-help="Name of the item">fish</div>
  <div class="text">':</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">Total sold</div>
  <div class="text">: </div>
  <div class="data" data-tag="sold"
```

(continues on next page)

(continued from previous page)

```
       data-xpath="/top/data/item/sold" data-type="number"
       data-help="Number of items sold">1321.0</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">In stock</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="in-stock"
       data-xpath="/top/data/item/in-stock" data-type="number"
       data-help="Number of items in stock">45</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">On order</div>
  <div class="decoration">:</div>
  <div class="padding"> </div>
  <div class="data" data-tag="on-order"
       data-xpath="/top/data/item/on-order" data-type="number"
       data-help="Number of items on order">1</div>
</div>
<div class="line">
  <div class="padding">   </div>
  <div class="label">SKU</div>
  <div class="text">: </div>
  <div class="data" data-tag="sku"
       data-xpath="/top/data/item/sku" data-type="string"
       data-help="Stock Keeping Unit">GRO-000-533</div>
</div>
```

CHAPTER 18

Indices and tables

- genindex
- search

Index

Y