




The LXR Developer's Manual

for version 2.0

This manual is released under 

Revision history

Author	Date	Rev	Comment
P. Gerlier	2013-11-25	1.0	Initial version (for release 2.0.0)

Licence statement

This manual is released under GNU FDL (*GNU Free Documentation License*) v1.3. It is available at <http://www.gnu.org/licenses/fdl-1.3.txt>.

LXR itself is distributed under GNU GPLv2 (or higher) license (<http://www.gnu.org/licenses/old-licenses/gpl-2.0.txt>). The code examples in this manual are also released under GNU GPL v3 (or higher) to permit their free reuse.

Copyright © 2013-2013 P. Gerlier

- Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
- A copy of the license is included in the section entitled "GNU Free Documentation License".

This manual is distributed in the hope that it will be useful, but **WITHOUT ANY WARRANTY**; without even the implied warranty of **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**.

The LXR logo on the cover page is © 2012 A. Littoz and released under Creative Commons Attribution-Share Alike 3.0 licence (CC-BY SA).

Document name

The file name for this document is structured as T-SR-L-DR.f where:

- T is a short title (like LXRUserManual),
- SR is the software release number associated with this document (like 0.10), may be omitted if the document is not related to a specific release,
- L is the ISO 639 alpha 2 language code with optional country variant (like en_UK),
- DR is the document revision number (like 1.0),
- f is the file format or file name extension (like odt for Open Document Format or pdf).

I would like to express my gratitude to Patrick Gerlier who spent a lot of time writing the LXR manuals. One of the major failures of Open Source Software is lack of providing adequate documentation. He bridged that gap first with the User's Manual and now with the Developer's Manual. I wish I could qualify it as "wonderful" but since neither he nor I are native English speakers, we cannot objectively assess the quality of this work. He also had the courage to thoroughly read LXR code and put in writing his forensic analysis. When I remember my hard time understanding LXR internals when I took over maintenance responsibility from Malcom, I thank him for that invaluable help for those wishing to put their hands under the hood. May their learning curve be very steep!

André J. Littoz

I appreciate greatly André's kindness and, in my turn, I would like to point out his patience and art of explanation. When I wanted to dig into LXR, I contacted him to get information. Despite my numerous e-mails and my somewhat frequent dumb remarks, he always answered, trying to explain in simple words why I was wrong or what I should do to get what I intended. I believe that, in fact, he likes being in that position. I am convinced that rewording a user's problem helped him to discover lurking bugs and also to improve LXR usefulness. I hope my contribution will appeal to LXR users, but as we learn from SourceForge statistics, less than 40% of LXR users download the User's Manual, though the download ratio improves with the release number.

Patrick Gerlier

Table of Contents

1 LXR Components	1
1.1. Global outline	1
1.2. Directory organisation	2
1.3. Internal information	4
2 LXR Engine	7
2.1. Principle of operation	7
2.2. Preserving state between invocations	8
2.3. URL parsing and HTTP management	8
2.3.a. Initialisation	8
2.3.b. Support routines	10
2.3.c. Internal routines	11
2.4. Configuration file management	12
2.4.a. Initialisation	12
2.4.b. API	13
2.4.c. unmappath algorithm	15
2.4.c.1. Replacement transformation	15
2.4.c.2. Pattern transformation	15
2.4.c.3. Inverting step	16
2.5. HTML stream generation management	17
2.5.a. Basic routines	17
2.5.b. Page structure utilities	18
2.5.c. Template editing functions	18
2.5.c.1. Functions for headers and footers	19
2.5.c.2. Functions for title area	19
2.5.c.3. Functions for developers	23
2.5.c.4. Functions for content area	23
2.6. Markup management	23
2.6.a. Driver routines	23
2.6.b. Support routines	24
2.7. File parsing	25
2.7.a. Support routines	25
2.7.b. Parsing algorithm	26
2.7.c. Algorithm limitations	31
2.8. Language parsing	32
2.8.a. Initialisation	32
2.8.b. Public methods	32
2.8.c. Support routines	33
2.9. File access management	34
2.9.a. Public methods	34
2.9.b. Support methods	36
2.10. Database management	37
2.10.a. Support methods	38
2.11. Local customisation	41
2.12. Derived language parsers	43
2.12.a. Generic parser	43
2.12.b. C parser	44
2.12.c. COBOL parser	44
2.12.d. HTML parser	45
2.12.e. Java parser	45
2.12.f. Make parser	45

2.12.g. Pascal parser.....	45
2.12.h. Perl parser.....	45
2.12.i. Python parser.....	45
2.12.j. Ruby parser.....	45
2.13. Specialised file access managers.....	46
2.13.a. BitKeeper manager.....	46
2.13.b. CVS manager.....	46
2.13.c. GIT manager.....	49
2.13.d. Mercurial manager.....	50
2.13.e. Plain files manager.....	52
2.13.f. Subversion manager.....	52
2.14. Specialised database managers.....	54
2.14.a. MySQL.....	54
2.14.b. Oracle.....	54
2.14.c. PostgreSQL.....	54
2.14.d. SQLite.....	55
3 Index Generator.....	57
3.1. Process outline.....	57
3.2. Internal support routines.....	58
3.3. External support routines.....	60
3.3.a. VTescape.pm.....	60
3.3.b. Tagger.pm.....	61
3.3.c. Multi-threaded attempt.....	61
4 Database Architecture.....	63
4.1. Tables.....	63
4.1.a. files and status tables.....	64
4.1.b. releases table.....	64
4.1.c. langtypes table.....	65
4.1.d. symbols table.....	65
4.1.e. definitions table.....	65
4.1.f. usages table.....	66
4.1.g. Unique numbering tables.....	66
4.2. Queries.....	66
4.3. Database engine specifics.....	67
4.3.a. MySQL.....	68
4.3.b. PostgreSQL.....	68
4.3.c. SQLite.....	68
4.3.d. Oracle.....	69
5 LXR Main Scripts.....	71
5.1. source script.....	71
5.2. ident script.....	73
5.3. diff script.....	74
5.4. search script.....	75
5.5. showconfig script.....	76
6 Configuration Wizard.....	79
6.1. Process Outline.....	79
6.2. Support library.....	80
6.2.a. ContextMgr.pm.....	80
6.2.b. LCLInterpreter.pm.....	81
6.2.c. QuestionAnswer.pm.....	87
6.3. LXR Configuration Language (LCL).....	89

6.3.a. Syntax.....	89
6.3.b. LCL commands.....	92
6.3.b.1. Error suppression.....	92
6.3.b.2. Shell command insertion.....	92
6.3.b.3. Message display.....	92
6.3.b.4. User interaction.....	93
6.3.b.5. Conditional interpretation.....	97
6.3.b.6. Array content insertion.....	98
6.3.b.7. Variable assignment.....	99
6.3.b.8. File inclusion.....	100
6.3.b.9. Pass 2 interpretation.....	101
6.3.c. Standard symbol dictionary.....	101
6.4. Standard templates.....	103
7 Auxiliary Scripts.....	105
7.1. Linux kernel exploration script.....	105
7.1.a. Process outline.....	105
7.1.b. Support routines.....	105
7.1.c. Interaction with lxrkernel.conf.....	106
7.2. Database reconstruction script.....	108
7.2.a. Process outline.....	108
7.2.b. Maintenance issue.....	108
8 Release Tool.....	109
8.1. Command line.....	109
8.2. Process outline.....	109
8.3. Support routines.....	110

Project LXR	The LXR Developer's Manual I LXR Components	Language en_UK
Software release 2.0		Document revision 1.0

LXR Components

The goal of **LXR** is to display a source file with symbol highlighting in an *HTML* browser. Highlighting encompasses both visual appearance and hyperlink creation to benefit from the browser ability to navigate from one page to another through a single click.

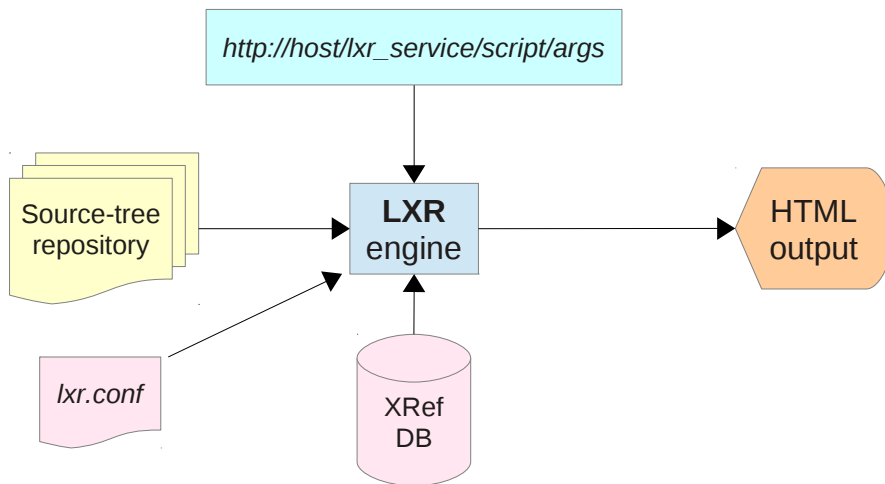
Features and operation are described in the *LXR User's Manual* which should be read before this manual.

1.1. Global outline

LXR is presently composed of a set of specialised *Perl* scripts (*diff* for difference display, *ident* for identifier search, *search* for free-text search, *showconfig* for monitoring configuration and *source* for display) and a support library also written in *Perl*.

These scripts are driven from a URL (in the browser address bar) describing the intended action. **LXR** eventually retrieves a source file from the source-tree repository and merges cross-reference data from its internal database into an *HTML* stream representing the edited file. Identification of the data sources come from the configuration file *lxr.conf*.

This is summarised in the following figure:



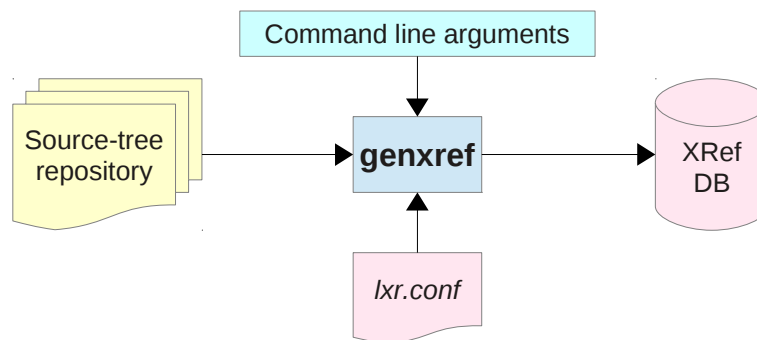
Drawing 1: Data sources in LXR

The cross-reference database has been previously created by script *genxref* in two internal passes over the source-tree:

- the first pass collects symbol definitions with help of *ctags*;

- the second pass enters all occurrences to the previous symbols.

This process can be slightly adapted through command line arguments. See the following flow diagram:



Drawing 2: Database creation process

The **LXR** engine receives control in one of the externally visible scripts, namely *diff*, *ident*, *search*, *showconfig* or *source*, which dispatches calls to services:

- URL decoding to understand what is expected and configuration file parsing to retrieve the parameters implied by the request (always done);
- access to a source file or source directory through an abstraction layer hiding the differences between real filesystem or version control system (*diff* and *source*);
Usually, this means also parsing the source file. Although functionally related, this service is independent from source access. It is also an abstraction layer hiding the differences between the source languages.
- access to the cross-reference database in order either to highlight a symbol (*source* or *diff*) or to dump the references (*ident*);
- eventually, if the feature is enabled in configuration, access to auxiliary database for free-text search (*search*);
- *HTML* stream generation (always done).

1.2. Directory organisation

Source code for **LXR** is contained in a directory called *LXR root directory* in the *User's Manual*. Its organisation is fit both for run time and development. This comes from the fact that **LXR** is written in an interpreted language which does not need compilation and linking. This may not remain true in the future.

This root directory contains the externally visible scripts, *i.e.* commands, and sub-directories for the support library:

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	I LXR Components	Document revision 1.0

- *diff*, *ident*, *search*, *showconfig*, *source*: the user commands issued through a browser
- *genxref*: database content initialisation script

Note:

Since this is not a browser “command”, this script would be better located in the *scripts/* directory but it uses the common support library which expects the “master” script being launched from the *LXR root directory*. The support library retrieves this *LXR root directory* path from the OS-absolute path of the “master” script. If the script is launched from somewhere else, computing the library location fails.

- *robots.txt*: web crawling security file to prevent spider robots from indexing the source-tree

Note:

This file is effective only if it is located at the root of the web site.

- *LXRimages/*: graphics stuff for insertion into *HTML* pages
- *doc/*: traditional summary information for installation (changes, licence, installation notes, ...)
- *lib/*: support library, containing
 - *Local.pm*: custom description extracting functions to comment directory listings; intended to be adapted by every tree manager to suit his needs
Not in *LXR/* because it is not strictly part of the standard support library: it is supposed to be written by the end-user though it has seldom been.
 - *magic.mime*: a 2004-hacked version of magic numbers for binary file detection, to be used by *Perl* module *File::MMagic*

Note:

Could this file be deleted since it is rather old? Change log does not tell what was added or modified to improve the test. A more recent and comprehensive file ships with the *file* package in every distribution (located at */usr/share/misc/magic* in **Fedora**).

Location of this file can be given in 'magicmime' configuration parameter.

- *LXR/*: the support library, strictly speaking; its content is detailed below

Note:

The support library is isolated in a directory because it used to be copied into *Perl* library in the **Apache 1** era. Being contained in a directory, it did not mess up the system library and it was easy to remove it. This is no longer necessary with the newer version (and has never been with other web servers). We could thus spare one directory level.

- *scripts/*: directory containing various utility scripts (configuration wizards, maintenance functions, ...)

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	I LXR Components	Document revision 1.0

- *templates/*: templates for creating configuration files and *HTML* pages; some are used only during initial configuration, others (like those in *html/*) are routinely used by the *LXR* engine

**CAUTION:**

The structure of this directory and the names of the contained files are known to the configuration wizard. Consequently, any change must be forwarded to the wizard.

The support library *LXR/* provides the services needed by the browser “commands”. The files related to the abstraction layers are “linked” to directories containing the implementations. Currently, three layers are defined:

- *Files.pm* and *Files/*: access to source repositories;
- *Index.pm* and *Index/*: access to the cross-reference database;
- *Lang.pm* and *Lang/*: language parsing.

The other files provide the following services:

- *Common.pm*: URL parsing and HTTP management

Note:

Used to be the only non-specialised file in the library, but grew so much it was decided to split it into smaller more manageable units; however it kept its original name which could cause now confusion.

- *Config.pm*: access and management of *lxr.conf* configuration data
- *Markup.pm*: highlighting of file, string, ...; needs language parsing service
- *SimpleParse.pm*: rudimentary context free parser based on pattern matching
- *Template.pm*: *HTML* stream generator expanding page templates

All these services are detailed in the following chapters.

1.3. Internal information

Many *Perl* source files contain *POD* (plain old documentation) blocks describing the package and its routines. This documentation can be extracted to be displayed by a browser. A typical command (set on several lines for readability but it is a single logical line) is:

```
$ pod2html --htmlroot=hr
--infile=LXR/root/directory/documented_file
--outfile=html_file
--title="File extracted documentation"
```

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	I LXR Components	Document revision 1.0

where:

hr

base URL for the resulting pages (used to cross-link the pages),

documented_file

an **LXR** source file, such as *lib/LXR/Common.pm*,

html_file

the output *HTML* documentation page,

title

some fancy title for the *HTML* page.

For instance, to collect the documentation from *lib/LXR/Common.pm* into subdirectory *LXRdoc/* of your personal home directory, with this subdirectory being the *HTML* document root, launch the following command:

```
$ # As usual, current working directory is supposed to be LXR root
$ cd LXR_root_directory

$ pod2html --infile=lib/LXR/Common.pm --outfile=~/.LXRdoc/common.html
--title="Common package"
```

The last command is split by the word processor but it is written as a single line.

Open the web browser and point it to the appropriate URL. In case the *LXRdoc/* directory is not integrated into the web server document root, it may however be displayed with an address of the form:

```
file:///home/myself/LXRdoc
```

and follow the links for the different files.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

2 LXR Engine

2.1. Principle of operation

LXR operation is based on interpreting the request URL.

Schematically, an *LXR* URL has the following structure:

```
http://host_name/web_path/script/tree/source_tree_path?arguments
----- web server realm ----- === LXR control parameters ===
```

Note:

Of course, https can be used instead of http.

The first part `http://host_name/web_path/script` is used by the web server to route the request to the appropriate script. `host_name` is associated with the computer through DNS (the alternative to directly reference the computer makes use of a numeric IP address). The association between `web_path/script` and the LXR root directory and a script is defined in the web server configuration file.

The second part `tree/source_tree_path?arguments` drives script processing. The role of subparts is:

- `tree`: identifies the source-tree to manage (may be omitted if a single tree is handled by *LXR*);
- `source_tree_path`: designates the source file to process (relevant only for scripts *diff* and *source*);
- `arguments`: optional key-value pairs modifying a script default behaviour (such as choice of version to display)

Many variations are possible. Notably, the tree designation has not always been positioned after the script name.

1. *Tree-specific host name: alias to host_name but not to be considered for DNS translation if many trees are served since it involves too many manual steps;*
2. *Tree prefix for host name: easy with external DNS but tedious for localhost;*
3. *Tree name embedded in web_path: needs a tweak in web server configuration, but this restricts the choice of web servers because the tweak cannot always be ported from one to another.*

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

2.2. Preserving state between invocations

HTTP is a stateless protocol; a script invocation must therefore completely describe in arguments what is indented. This is where *LXR* stores its internal state when an action is split into a series of script invocations or simply to remember some user choices. The state consists of:

- 'variables' values (if different from default values) as

```
var_name=value
```

- “Remembered” 'variables' current values (this is necessary for *diff* because it uses a “hop” to select the second file to compare to)

```
~var_name=value
```

- Overriding 'variables' values (this argument category defers variable update from value selection in a menu to next script invocation; it simplifies change of value by overriding a variable value after the “standard” argument has been taken into account)

```
!var_name=value
```

- Internal parameters (they may or may not have an equivalent configuration parameter)

```
_parm_name=value
```

2.3. URL parsing and HTTP management

This service resides in file *Common.pm*. It aims at extracting the semantic components from the URL and identify the relevant tree description in configuration file *lxr.conf*.

2.3.a. Initialisation

The very first function called is `httpinit`. Using the environment variables, it builds a safe canonical representation of the URL.

`hostname` is reconstructed from variables `SERVER_NAME` and `SERVER_PORT` and stored in an LXR-normalised form in `$HTTP->{'host_access'}`. `web_path/script` is supposed safe (*i.e.* URL %-encoded) and copied from `SCRIPT_NAME`. The target virtual root is computed by removing the script name and stored in `$HTTP->{'script_path'}`. This will later allow to define the HTML `<base>` element.

`PATH_INFO` (containing `web_path`) is checked for possibly offending characters (which could

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

cause cross-site attack XSS) and truncated at the first unauthorised character¹. Eventually, path segments `./` are replaced by `/`.

The reconstructed host name and the virtual root are passed to configuration service to identify the target source-tree.

Parameters and their values are extracted from `QUERY_STRING`. If they correspond to “variables”, the current value of those “variables” is updated and the parameter deleted.

The remaining parameters will be blindly copied when an URL needs to be generated, thus preserving the internal state or user environment from one request to the other.

Since this service is always called first and only once, it launches initialisation for other services: access to repositories and cross-references database.

Global variables are set to their value for general access by other services:

<code>\$HTTP</code>	“hash” containing the decoded URL 'host_access' is <code>http://hostname:port</code> . 'script_path' is <code>web_path</code> without script name. 'path_info' is the path following the script name. 'path_root' is the first segment of 'path_info', <i>i.e.</i> possibly the tree name. 'this_url' is the full reconstructed URL. 'param' is a “hash” of <i>parameter/value</i> extracted from the query string. Note that some <i>parameters</i> are removed from this “hash” (namely those corresponding to “variables” and argument <code>_i</code>) to avoid later duplications in the generated link URLs.
<code>\$releaseid</code>	internal version identifier (may be different from the URL <code>_v</code>)
<code>\$pathname</code>	path to the required file (relative to <code>sourceroot</code>)
<code>\$identifier</code>	key for identifier search (cleaned URL <code>_i</code>)
<code>\$config</code>	“hash” containing the relevant configuration parameters from <i>lxr.conf</i> merged from the global and tree-specific parameters
<code>\$files</code>	“object” for source repository access
<code>\$index</code>	“object” for cross-reference database access
<code>\$HTMLheadOK</code>	set to 1 when HTML headers have been successfully generated

Finally, HTTP headers (presently `Last-Modified:` and `Content-Type:`) are emitted by `printhttp` and mode is switched from headers to content by emitting an empty line.

Package-private variables are:

<code>\$wwwdebug</code>	if set to 1, messages from <code>fatal</code> and <code>warning</code> are also emitted as HTML (they are always entered into the error log).
<code>\$HTTP_inited</code>	set to 1 when HTTP headers have been sent
<code>\$tmpcounter</code>	unique counter for temporary files

¹ This has not yet caused any malfunction. At least, no user complained.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

TODO:

Remove limitations on file names (notably special URL characters); improve URL %-encoded strings (may need updates to other functions); check possible XSS gaps.

Unreserved characters: A-Z a-z 0-9 - _ . ~

General delimiters: : / ? # [] @

Sub delimiters: ! * ' () ; & = + \$, %

2.3.b. Support routines

Routine name	Arguments	Description
http_wash	URL fragment	Returns its argument HTTP %-decoded
http_encode	String	Returns its argument HTTP %-encoded
fixpaths	File path	Prefixes its argument with / and removes all <i>directory/./</i> or <i>./</i> segments; if repository service tells it is a directory, suffixes argument with /
httpminimal		Emits a minimal set of HTTP headers, sufficient to display error information
printhttp		Emits HTTP headers for the current file
httpinit		Basic initialisation (see above)
clean_release	Release id	Returns its argument if that release (version) exists in the repository or the default one otherwise Note: this protects against maliciously crafted version in the URL
clean_identifier	Search name from URL	Returns its argument with “stray” characters removed (outside the alphanumeric set plus <code>_ : . , - `</code> and space) from the identifier name passed through the URL Note: the set of allowed characters must be consistent with language lexical definition. Note: this protects against maliciously crafted identifier in the URL
clean_path	File path	Returns its argument with “stray” and following characters (outside the alphanumeric set plus spacers <code>_ + . , - % ^ !</code>) and all <i>./</i> segments removed
httpclean		Disposes of allocated data structures <code>\$config</code> , <code>\$files</code> and calls <code>final_cleanup</code> for <code>\$files</code> (protection against memory leak, important under <i>FastCGI</i> , not fundamental under <i>CGI</i> but it is good programming practice)
tmpcounter		Returns a unique counter (useful for creating temporary files) Counter is kept in package-private global variable <code>\$tmpcounter</code>
nonvarargs		Returns an array of <i>variable=value</i> strings for URL arguments which are not configuration variables (arguments are taken from

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
		\$HTTP hash)
urlargs	Extra argument array (format of each element is var=value)	Returns a string suitable as a query string from configuration variables and optional argument array
fileref	Description CSS class File path Line number Extra argument array	Returns an <code><a></code> <i>HTML</i> element to invoke display of a directory or file (may scroll to a line within a file); line number and argument array optional Path name is URL %-encoded <i>HTML</i> delimiters in description are replace by their entity references (CAUTION! This means <i>HTML</i> elements cannot be embedded in the description unless special precaution is taken)
diffref	Description CSS class File path Extra argument array	Returns an <code><a></code> <i>HTML</i> element to invoke difference markup of a file; argument array optional See remarks in <code>fileref</code>
idref	Description CSS class Identifier name Extra argument array	Returns an <code><a></code> <i>HTML</i> element to invoke identifier lookup; argument array optional See remarks in <code>fileref</code>
incref	Name to display CSS class File path Extra argument array	Returns <code>fileref</code> (arguments) if <code>incfindfile</code> succeeds, <code>undef</code> otherwise

2.3.c. Internal routines

These routines are not exported. They are invoked from the hooks for `warn` and `die` statements. They intercept warnings and errors to log them into the web server journal file and display them on screen if requested by variable `$wwwdebug`. They return no value.

They are protected against *HTML* attack by transforming all `<` `>` characters in the message by their equivalent entity references. If multi-line messages are needed, lines are separated with `\n` which will be transformed into `
` elements.

Routine name	Arguments	Description
warning	Message	Inserts its argument into the error log and optionally prints it on screen as an <code><h4></code> <i>HTML</i> element if debugging mode is enabled
fatal	Message	Inserts the internal state and argument into the error log, optionally prints it on screen as an <code><h4></code> <i>HTML</i> element if debugging mode is enabled and aborts processing Note: can be used before initialisation is complete because it cares for HTTP headers

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

2.4. Configuration file management

This service resides in file *Config.pm*. It aims at collecting the set of configuration parameters relevant for a given source-tree (union of global tree-specific parameters). It offers a simple API for dealing with 'variables'.

CAUTION!

This service is also used by traditional command line scripts for which the web server environment variables are not defined. Consequently, you must not rely on implicit values for optional arguments to functions.

Name of configuration file *lxr.conf* is defined by global variable `$confname`. This variable can be modified to point to a different default location, such as */etc*.

2.4.a. Initialisation

The very first function called is `new`. It creates the “object” accessing configuration data. Its first two arguments represent the actual URL (host name and script path) used to activate a script. A third optional argument is a potential candidate for the tree name. A fourth optional argument contains the name of an alternate configuration file to replace the default name from variable `$confname`. It branches to internal function `_initialize` and returns its value (configuration object).

Arguments to `_initialize` are optional. Default values are computed if they are missing. Missing host name and script path are replaced based on server variables `SERVER_NAME`, `SERVER_PORT` and `SCRIPT_NAME`. No surrogate value is computed for a missing tree name. A missing configuration filename is replaced by the OS-absolute path of the executing script with the last segment substituted with the value of `$confname` (if `$confname` starts with a `/`, it is considered an OS-absolute path and used as is²).

The full configuration file is read and `eval`d. The first element of the resulting array is transferred into the created object as it applies to all trees.

If global parameter 'routing' exists (new in release 2.0), extra checks are made. 'single' routing must be applied only on a configuration file describing a single tree. In 'argument' routing, not requested from *genxref*, a manual selection is offered if no tree name was defined.

The remaining configuration elements are examined in a loop to find a match on host name (without port³) in 'host_names' and virtual root in 'virtroot' against the script path (final script name element removed). For backward compatibility, if there is no 'host_names', match is checked between 'baseurl' or 'baseurl_aliases' and 'virtroot'. On match, the element is

² This makes provision for installing *LXR* through a package with the scripts in some “system” directory and configuration file in */etc*.

³ This choice has been made by the maintainer to allow to serve simultaneously the same source-tree by different web-servers connected on different ports. To use port in the comparison, just remove a single line.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0		2 LXR Engine
		Document revision 1.0

transferred into the created object with duplicate key/value automatically overriding the global ones. If no match, the process is aborted; recovery is dependent on catching this exception in an outer handler.

After a match, various tests are made to insure critical parameters are present (only presence, not semantics or sensible test).

To avoid subsequent problems with dedicated *LXR* servers (where 'virtroot' reduces to '/'), the virtual root kept in the created object is enclosed⁴ in *HTML* path separators /. The resulting virtual root can then be used as a raw prefix, without the need to add a / separator when composing URL.

2.4.b. API

The other functions implement the API.

The following methods are for special access to the configuration file.

Routine name	Arguments	Description
emergency	Configuration file	Similar to <code>new</code> but never returns <code>undef</code> ; instead, returns whatever can be grabbed from configuration file (at least the global parameters) <i>To be used as an a fallback initialisation (if <code>new</code> failed) to allow to emit HTML code with the "standard" templates</i>
readconfig		Returns the content of the configuration file in a array; this is the same configuration file as the one scanned by <code>new</code> (name store in the configuration object) <i>To be used when there is a need to access other elements than the active one</i>
treeurl	Tree-specific parameters Global parameters	Tries to return a URL for the tree described by the first argument. Both arguments are usually obtained from <code>readconfig</code> . If <code>undef</code> is returned, this means HTML-relative references may be used; otherwise, the returned URL is the base for an absolute reference. <i>The algorithm may fail to give the correct answer. Read the caveats in the code.</i>
readfile	File path	Returns in an array the words (delimited by spaces) in the file <i>This is not a method but a regular sub for use in custom functions in the configuration file. File path may be relative to LXR root directory or absolute.</i>

The 'variables' can be manipulated through the following methods.

Routine name	Arguments	Description
allvariables		Returns an array containing the names of all variables

⁴ A '/lxr' virtual root becomes '/lxr/'. When *LXR* service is at the server root, care is taken to obtain '/' and not '//' which would be understood as the beginning of an host name, giving an erroneous link.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
variable	Name Value	If second argument present, set the variable to its value; return the current value of the variable
vardefault	Name	Returns default value for the variable
vardescription	Name Value	If second argument present, set the variable 'name' element to its value; returns variable "description" (value for key 'name')
varrange	Name	Returns an array of the explicit allowed values (value for key 'range') or a reference to the function computing this array
varexpend	String	All occurrences of \$name are replaced by the variable current value <i>No test is made to see if the variable exists</i>
value	Parameter name	Returns the value of the configuration parameter where all occurrences of \$name are replaced by the variable current value <i>No test is made to see if the variable exists</i>

Other miscellaneous services are:

Routine name	Arguments	Description
AUTOLOAD	At least one	Magic <i>Perl</i> method to instantiate barewords (useful mainly for dynamic subs) It provides a shorthand notation for <code>value(arg)</code> when <i>arg</i> is a configuration parameter. If this <i>arg</i> is a sub, it is applied to its arguments. <i>When writing new code for LXR, it should be better to avoid it so that intent of the word is immediately apprehended.</i>
mappath	File path Extra argument array	Applies the 'maps' rules to the file path; a local 'variables' environment may be created with the optional argument array
unmappath	File path Extra argument array	Tries to reverse the effect of the 'maps' rules on the file path Proceeds by dynamically computing a pseudo-inverse of the replacements CAUTION! <i>It might not be possible to invert the rules if they destroy information; consequently, the result must be considered unreliable.</i>
_ensuredirexists	Directory path	Checks that the final and all intermediate directories exist and eventually creates them (roughly equivalent to <code>mkdir -p</code>) Created directories have write-access enabled for everybody. <i>This is not a method by a regular sub.</i>

Note about AUTOLOAD

Presently, the 'variables' expansion feature is used for parameters 'sourcerootname' (custom path root in banner) and 'incprefix' (list of "include" directories).

If needed, it could also be activated for 'ignoredirs' (list of directories to ignore).

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

Is it meaningful to allow variable substitution in 'ignoredirs'? What kind of application would need it?

2.4.c. unmappath algorithm

The basic idea is to try to “invert” a rule *pattern* => *replacement* by creating a new regular expression based on *replacement* which leads to a substitution based on *pattern*. Since every rule destroys some information (*i.e.* the replaced part of the original string), this “inverse” formally does not exist, but any element of an appropriate equivalence class will do. **LXR** only needs to revert a file name to some generic form before applying again the rules.

Note:

unmappath is only used in script diff to find the common “stem” of both file names.

The rules are scanned in the reverse order of their application (last used, first reversed or stack order).

2.4.c.1. Replacement transformation

First, all occurrences of *\$variable* are expanded to the current variable value by `varexpand`. Thus, *replacement* looks like its final result and can match the file name.

Second, *replacement* may contain “capture substitution” *\$number* (*e.g.* \$1, \$2, ...). The exact original content does not matter. Every occurrence is replaced by `.+?`, meaning any MINIMAL run of characters. The minimal attribute (?), *non-greedy* in *Perl* parlance, is needed to avoid merging possible “interesting” sequences.

Last, if the original *pattern* is anchored either to the start or end of the string, the corresponding anchor `^` and/or `$` is set into *replacement*.

2.4.c.2. Pattern transformation

1. *Pattern* is scanned for optional sequences which are removed. Since they are optional, the *pattern* will match on a string not containing these optional sequences. Such a string is a “minimal” element of the equivalence class of all possible matching strings. The form of the optional elements is x^* or $x?$ possibly followed by `?` or `+`.
 - To cope with possible nested sub-patterns with `()` or `[]` delimiters, a coarse-grained regular expression captures everything from the current position up to quantifiers. If no quantifier is found, this step stops.
 - If the character preceding the quantifiers is a right parenthesis `)`, a group must be removed. Innermost nested groups (not containing another parenthesised group) as well as the rightmost unnested group are repeatedly erased. A last erasure is made on the group matching the ending parenthesis.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0



CAUTION!

The algorithm does not manage extended patterns (?...). Regular expressions for paths are supposed to be rather simple.

Notes:

Special care is taken for escaped characters (\x); the pair is processed as a whole. So, when a left parenthesis is found, it is a real grouping one (not an escaped one, equivalent to an arbitrary character).

There is no need to pay attention to quantifiers for nested groups or literal content since everything inside the external group will be erased. The goal of this sub-step is only to match the ending right parenthesis with the correct left parenthesis.

- If the character preceding the quantifiers is a right square bracket], it is associated with the nearest unescaped left square bracket and the group is erased.



CAUTION!

The algorithm does not manage POSIX character classes [[:...:]] nor does it handle a legal unescaped [.

- Otherwise, the preceding escaped or single character is erased.
2. Empty capture groups () are removed.
 3. + quantifiers (and their optional sub-quantifiers) are removed since a single occurrence is a “minimal” element of the equivalence class.
 4. The remaining parenthesised groups are replaced by their first alternative.
 - Scanning from left to right, if no left parenthesis (is found, this sub-step stops.
 - Proceeding from innermost unnested parenthesised group, only the first alternative is kept. This leaves a single group with the outermost parentheses. A last iteration keeps the (expanded) first alternative.
 5. Sets of characters [...] are replaced by the first character of the set. In case this is a “negation” set [^...], a % is tentatively used⁵.
 6. Finally, character classes \x are replaced by an arbitrary character of the class.

2.4.c.3. Inverting step

After these transformations, a new rule is applied to the file path with the roles of *pattern* and *replacement* exchanged: transformed *replacement* => transformed *pattern*.

After all applications, it is hoped that the path is reverted to a reasonable “template” path on which the original rules can be applied again with a different set of 'variables' values resulting in a realistic path.

⁵ This is not guaranteed to work in all circumstances, especially if % was excluded, but this is better than nothing.

2.5. HTML stream generation management

This service resides in file *Template.pm*. It retrieves *HTML* templates and manages their expansion. Special sequences in the templates are considered *macro invocations*:

```
$macro_name{...argument_to_macro ...}
```

macro_name is composed of alphanumeric characters only. The symbol is associated to a service function.

argument_to_macro is a sequence of characters passed to *macro_name*. For instance, this allows conditional insertion of the sequence or provides a sub-template for every element of an array.

Notes:

The argument may contain properly nested macro invocations. But argument expansion is under macro control, *i.e.* the macro must request expansion.

There are no spaces between *macro_name* and the left curly bracket.

No escape mechanism is provided to allow a right curly bracket within the argument.

In case argument is empty, the curly brackets may be omitted; both forms are equivalent:

```
$macro_name{}
$macro_name
```

2.5.a. Basic routines

Two basic routines are provided.

Routine name	Arguments	Description
gettemplate	File name Default prefix Default suffix	Returns the template from the designated file. If not found, returns the concatenation of prefix and suffix. An eventual warning message is issued with a warn statement.
expandtemplate	Template Function directory	Scans the template for <i>\$macro_name</i> and executes the associated function

`expandtemplate` first removes “non sticky” comments⁶ and multiple newlines from the template string. This is done in order to minimise the the amount of transmitted characters by not sending information meaningless to the browser.

⁶ A *sticky* comment is an *HTML* comment `<! - - ... - ->` whose opening delimiter is not followed by a spacer. The closing delimiter must also be preceded by a spacer due to the nature of the detecting regular expression.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

An inefficient algorithm, based on pattern matching (that's *Perl!*), is used to match opening curly brackets and define the extent of the arguments.

A finite state automaton would definitely do better but implementation is tedious in Perl.

The function directory is a *hash* with keys equal to *macro_name* and values equal to a sub reference. If *macro_name* exists, the associated function is called with the extracted argument. The returned string replaces the whole sequence. If it does not exist, the sequence is left unmodified.

2.5.b. Page structure utilities

They use the previously described basic routines.

Routine name	Arguments	Description
makeheader	Originator	Will try to find template <i>originatorhead</i> , otherwise <i>htmlhead</i> ; then expands the template and sends the result This routines builds the header area of all pages.
makefooter	Originator	Will try to find template <i>originortail</i> , otherwise <i>htmltail</i> ; then expands the template and sends the result This routines builds the footer area of all pages.
makeerrorpage	Originator	Retrieves the designated templates and sends the expanded result as a full page (no calls to <i>makeheader</i> nor <i>makefooter</i>). The function directory for <i>expandtemplate</i> is minimal since <i>LXR</i> initialisation could not be completed.

Originator is an identification transmitted by the caller. It is usually equal to the *LXR* script name (*diff*, *ident*, *search* or *source*); the exceptions are *config* for *showconfig*, *sourcedir* for *source* on a directory and *htmlfatal* when calling *makeerrorpage*. This allows the called functions to behave differently according to context.

Note:

Template.pm source text is different in *CVS/Git* from the release version. In the three preceding routines, substitution value for *\$LXRversion* is *%LXRVERSIONNUMBER%* in the source management tool (universal form), whereas it is set to the specific version number when it is publicly released. Customisation is made by script *set-lxr-version.sh* during the release process.

2.5.c. Template editing functions

All the functions listed in the following table are associated with a *macro_name*. Unless otherwise noted, they have two arguments: the first is the argument sequence of characters extracted from the template; the second is *originator* (see above).

Macros marked † in the following tables request expansion of their arguments through a call to *expandtemplate*, which means their argument may contain further macros.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

2.5.c.1. Functions for headers and footers

Information for <HEAD> section:

Routine name	Macro name	Description
titleexpand	title	Returns a string suitable for use in a <TITLE> element, describing the current operation.
baseurl <i>Does not use arguments</i>	baseurl	Returns the URL %-encoded base URL, suitable for use in a <BASE> element.
<i>Anonymous, no arguments</i>	encoding	Returns the value of parameter 'encoding'
stylesheet <i>Does not use arguments</i>	stylesheet	Returns the URL %-encoded file name in parameter 'stylesheet' suitable for use in a <LINK href=... rel="stylesheet"> element.
altstyleexpand	alternatestyle	Returns repeated expansions of its argument for every file listed in parameter 'alternate_stylesheet'

2.5.c.2. Functions for title area

Data for composing page title:

Routine name	Macro name	Description
targetexpand	target	Extracts the intended tree name from the URL; parameter 'routing' tells where to look for the tree name. If this parameter does not exist, tree name is supposed to precede script name (compatibility with previous versions) <i>This routine is only directly called by makeerrorpage; otherwise, it is indirectly called by captionexpand.</i>
captionexpand	caption	Returns the value of parameter 'caption' or an internal string if it does not exist.
bannerexpand	banner	For scripts involving a file operation (i.e. source and diff), returns a sequence of <A> links to every component of \$pathname file path, preceded by the expanded value of parameter 'sourcerootname'. The components are separated by <i>zero-width space</i> so that the browser knows where to break a long line without splitting a component. Zero-width avoids the annoying visual effect of visible spaces in a path. The whole sequence is class-tagged banner.
<i>Anonymous, no arguments</i>	pathname	Returns the current file path (global variable \$pathname) CAUTION! Special characters are not protected; to be used only in context where a file name is expected; do not use for display.
<i>Anonymous, no arguments</i>	path_escaped	Returns the current file path (global variable \$pathname) with "special" characters HTML-escaped. Can be safely used for display purpose.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Macro name	Description
<i>Anonymous, no arguments</i>	LXRversion	Returns <i>LXR</i> version

Note about LXRversion:

The version string is not protected against special *HTML* characters; it is supposed that the local *LXR* administrator knows what he is doing when altering the version string (with script *set-lxr-version.sh*) and does not compose foolish names.

Tree enumeration:

Routine name	Macro name	Description
forestexpand	forest [†]	Returns an empty string if less than 2 shareable trees are found; otherwise returns its expanded argument

Note:

A *shareable* tree is tree for which parameter 'shortcaption' has been defined.

CAUTION!

Implementation is not correct when a single shareable tree has been found because it does not test if this shareable tree is the current one (where it makes sense not to build a switching link to itself). If the current tree is not a shareable tree, the switching link should be displayed.



Within the `$forest` argument, the following macro can be used:

Routine name	Macro name	Description
treeexpand	trees [†]	Applies its argument to every shareable tree and returns the concatenation of all expansions

Fixed text may precede and follow `$trees{...}`. The argument is expanded for every shareable tree; the intent is to make a link or button to jump to this tree, or whatever the designer wants. Every expanded instance is wrapped inside a `` element with `class` attribute equal to `tree-sel` or `treelink` depending whether it is applied to the currently displayed tree or not.

The `$trees` argument may contain the following macros:

Routine name	Macro name	Description
<i>Anonymous, no arguments</i>	caption	Returns the value of parameter 'shortcaption' Do not use! Semantics is not yet defined.
<i>Anonymous, no arguments</i>	link	Do not use! Semantics is not yet defined.
<i>Anonymous,</i>	treelink	Returns an <code><A></code> element to jump to the designated tree.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Macro name	Description
<i>no arguments</i>		

Mode buttons or links:

Routine name	Macro name	Description
modeexpand	modes [†]	Returns a set of buttons or links to switch between <i>LXR</i> operating modes (<i>source</i> , <i>diff</i> , <i>ident</i> and <i>search</i>)
atticlink	atticlink	Returns an <A> element <i>show/hide attic files</i> only for <i>source</i> mode and if files are stored in a CVS repository.

Within the \$modes argument, the following macros can be used:

Routine name	Macro name	Description
<i>Anonymous, no arguments</i>	modelink	Returns <A> element to switch to the mode.
<i>Anonymous, no arguments</i>	modecss	Returns a <code>class</code> attribute for the node (<code>modes-sel</code> or <code>modes</code> depending on whether this is the current mode or not).
<i>Anonymous, no arguments</i>	modeaction	Returns an URL for use in the action attribute of the <FORM> tag.
<i>Anonymous, no arguments</i>	modeoff	Returns an attribute for the <BUTTON> (<code>disabled</code> or empty depending on whether this is the current mode or not).
<i>Anonymous, no arguments</i>	modename	Returns a string for use as the button name in the <BUTTON> element.
urlexpand	urlargs [†]	Returns the internal <i>LXR</i> state as a sequence of <i>variable=value</i> definitions to be used in a URL query string.

Note:

If you prefer *links* interface, use only \$modelink. The other five macros provide the necessary building blocks for the <FORM> button elements in the *buttons-and-menus* interface.

'variables' buttons or links:

Routine name	Macro name	Description
varexpand	variables [†]	Returns a set of buttons or links to set the variables. Conditional variables are not returned if their expression evaluates to false.
varbtnaction	varbtnaction	Returns an URL for use in the action attribute of the <FORM> tag.
urlexpand	urlargs	Returns the internal <i>LXR</i> state as a sequence of <i>variable=value</i> definitions to be used in a URL query string.

Within the \$variables argument, the following macros can be used:

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Macro name	Description
varlinks <i>One extra argument</i>	varlinks [†]	Returns a sequence of <A> elements for all possible values of the variable passed as third argument.
<i>Anonymous, no arguments</i>	varid	Returns the machine name of the variable.
<i>Anonymous, no arguments</i>	varname	Returns the human-friendly description of the variable ('description').
<i>Anonymous, no arguments</i>	varvalue	Returns the current value of the variable.
varmenuexpand <i>One extra argument</i>	varmenu [†]	Returns a sequence applied to all possible values of the variable passed as third argument. <i>Usually, the template develops in content of a <SELECT> element.</i>

Note:

If you prefer *links* interface, use \$varlinks while \$varid and \$varmenu are specific to *buttons-and-menus* interface. The other macros are common to both interfaces.

Formatting the \$varlinks argument is done with the following macro:

Routine name	Macro name	Description
<i>Anonymous, no arguments</i>	varvalue	Returns an <A> element for a value of a variable.

Formatting the \$varmenu argument is done with the following macros:

Routine name	Macro name	Description
<i>Anonymous, no arguments</i>	itemclass	Returns a class attribute suitable for a <SELECT> tag (varlink or var - sel for any/selected value).
<i>Anonymous, no arguments</i>	itemsel	Returns selected for the current value, empty otherwise. This is a suitable attribute for a <SELECT> tag.
<i>Anonymous, no arguments</i>	varvalue	Returns the current value of the variable

Various URL (there is probably no use for them):

Routine name	Macro name	Description
thisurl <i>Does not use arguments</i>	thisurl	Returns the URL-%-encoded current URL
dordotsurl <i>Does not use arguments</i>	dotdoturl	Returns the URL-%-encoded base URL with the last segment removed. CAUTION! Implementation is faulty because the last segment is unconditionally removed without checking it really exists (this can

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Macro name	Description
		<i>erase the host name!)</i> <i>This will not be corrected since there is probably no use for it.</i>

2.5.c.3. Functions for developers

Debugging information display:

Routine name	Macro name	Description
<code>devinfo</code> <i>Uses only first argument</i>	<code>devinfo[†]</code>	Returns information for all <i>Perl</i> modules

Within the `$devinfo` argument, the following macros can be used:

Routine name	Macro name	Description
<i>Anonymous, no arguments</i>	<code>moduleid</code>	Returns the <code>\$Id</code> string set by the version control system.
<i>Anonymous, no arguments</i>	<code>modpath</code>	Returns the module file name.
<i>Anonymous, no arguments</i>	<code>modtime</code>	Returns the last modification time.

2.5.c.4. Functions for content area

These functions are defined by the primary scripts (*diff*, *ident*, *search*, *showconfig* and *source*) for their own usage. See the primary scripts descriptions.

2.6. Markup management

This service resides in file *Markup.pm*. It acts as a driver for syntax-highlighting files or strings.

2.6.a. Driver routines

There are only three accessible service functions. All others are support routines.

Routine name	Arguments	Description
<code>markupfile</code>	File handle Output function	The first argument is a filehandle for the file to syntax-highlight. The markup result is processed, piece after piece, by the second argument which is a reference to a one string-argument procedure. <i>This procedure is called from scripts source and diff.</i>
<code>markupstring</code>	String Virtual root	This function is intended to be used for “local” highlighting rules (defined in file <i>Local.pm</i>). It tries to build hyperlink after having

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
		detected URL (server access or mail addresses), identifier-like symbols or some files. The second argument provides (<i>HTML</i>) base directory for files.
freetextmarkup	String	This function is used to markup URL in comments or plain text (passed in the string argument). <i>Its only out-of-module use is in the HTML parser to hyperlink non-file URL. It should be considered a support routine as much as possible.</i>

markupfile tries to find a language parser for the file.

If the call to the parsing service succeeds, the fragment parser is initialised and the file is split into homogeneous fragments. According to the category, the fragment is handed over to the processing method of the language parser.

Images (categorised by their extension filtered by configuration parameter 'graphicfile') are tentatively sent as an element. The success depends on the browser capabilities.

Supposed unidentified text files are scanned with freetextmarkup for URL.

“Binary” files are associated with an hyperlink allowing to dump them in raw mode. It is unlikely to work always unless printhttp in HTTP management service *Common.pm* is changed to handle more file extensions.

Note:

The output function may “write” into a string instead of a standard file if display needs to be deferred such as in the *diff* case.

markupstring is a convenience function to help highlight URL encountered in file or directory “descriptions”. These descriptions are built or extracted when script *source* calls functions *dirdesc* or *descexpand* to insert some sort of comment in directory listing. Both functions reside in *Local.pm* and are supposed to be freely adapted to suit local needs.

2.6.b. Support routines

The last two routines are very short and could be inlined for performance. But, having them as individual pieces of code ensures that a consistent policy is enforced.

Routine name	Arguments	Description
is_linkworthy	String	The argument is a symbol found in plain text (usually a comment). The function decides if further highlighting should be attempted. The decision is made on symbol length (at least 6 characters) and on appearance (if _ or capital letters after initial character are used, it may be an identifier). Names containing README are excluded. For performance, database is not interrogated, which means many unknown symbols are given a “green light”.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
		Result is 1 for “go ahead” and 0 for “discard”.
markspecials	String	Meaningful <i>HTML</i> characters in the string (<, > and &) are prefixed with a NUL character (\0) to be later identified.
htmlquote	String	The previously marked characters (see <code>markspecials</code>) are replaced by their entity names (<, > and &). Since this function is called before outputting the marked-up stream, it also removes the special “start-of-line” flags.

2.7. File parsing

This is a generic context free parser based on pattern matching. The service resides in file *SimpleParse.pm* as a set of procedures or functions.

2.7.a. Support routines

There are an initialisation routine, the parsing function and two auxiliary routines.

Routine name	Arguments	Description
<code>init</code>	Filehandle Tab hint Array of <i>references</i> to key/value pairs	This routine initialises the package global variables. The first argument is a filehandle pointing to the source file. The second argument is the default tab width. The third argument, usually extracted from file <i>generic.conf</i> , defines the specific parsing rules to apply.
<code>nextfrag</code>		Returns a pair of strings. The first one is the fragment category, the second one the sequence of characters for the fragment. <i>To be called repetitively until it returns undef.</i>
<code>untabify</code>	String	Returns its argument where TAB (\t) characters are replaced by an appropriate number of spaces.
<code>requeuefrag</code>	String	Stores its argument so that its content will be scanned first on next call to <code>nextfrag</code> . It acts as a stack; it can then be called multiple times with the effect of considering the strings in reverse order of their entry.

The package-private global variables are:

<code>\$fileh</code>	filehandle to the scanned source file
<code>@frags</code>	queue for strings waiting to be processed; those extracted from the file are entered at the tail, while those requeued are entered at the head.
<code>\$next</code>	current run of characters under scrutiny (in fact, it is a cache for the first string in <code>@frags</code> to avoid pushing and popping it repetitively, thus achieving some speed optimisation)
<code>@bodyid</code>	list of category names (<code>comment</code> , ...)

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

@open	list of all opening delimiter regular expressions
@term	list of all closing delimiter regular expressions
@stay	list of all <i>escape</i> regular expressions
\$split	regular expression consisting of the union of elements of @open (separated by the alternative operator)
\$open	regular expression similar to \$split but arranged so that the matching delimiter can be identified
\$continue	the special 'atom' regular expression
\$tabwidth	tabulation spacing for this file

There is also a package public variable:

\$doutab	flag requesting to expand tab characters
----------	--

Their use is explained below.

2.7.b. Parsing algorithm

LXR does not try to parse source files as a compiler would do it. Though it could benefit from fine-grained parsing, notably improving detail and accuracy of symbol description, two factors must be accounted for.

1. *Perl* is targeted at regular expression processing and is not really fit for efficiently and easily implementing finite state automata.
2. A framework common to all languages is desired.

Consequently, a two-tier parsing strategy is implemented. The parser will recognise abstract items, namely comments, strings and *include* constructs. These items are then handed over to language-specific sub-parsers.

CAUTION!

genxref performance depends heavily on the parser efficiency. It is the most frequently called procedure during this phase and every modification has a direct noticeable impact on total running time (for good or evil). It is also very easy to get it wrong, as has experienced the main developer while adding new features, even without optimisation goal.

Therefore, if you want to implement the super-smart lightning-fast algorithm, check first that it still provides the intended result in ALL circumstances.

The item classes are defined as delimiter-bounded runs of characters. The delimiters are described by regular expressions. See the 'spec' arrays in *generic.conf* for examples. *Code* (also named *unclassified* below) consists of everything which cannot be classified in those three classes.

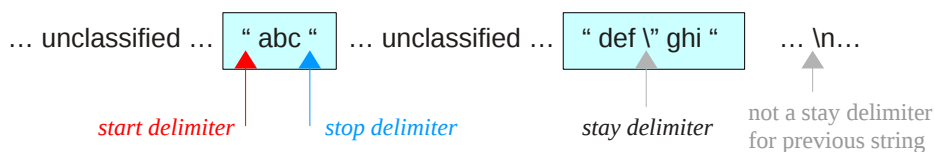
To avoid memory flooding, the source file is read one line at a time when data is needed. Tab characters are expanded to their equivalent number of space characters if variable \$doutab is

true⁷. Due to progressive processing, a line may be split into smaller chunks which are queued in array @frags, waiting to be parsed. Similarly classified chunks are concatenated into local variable \$frag which is one of the two output variables (the other one is \$btype).

Parsing proceeds through pattern-matching to find the bounds of the runs in the file. But there is no notion of left-to-right processing or precedence between the regular expressions (other than order of applications). Consequently, special attention must be paid to the limits of regular expression application when *escape* or *lock* regular expressions are present (stored in the @stay array).

On each invocation, the parser initialises its default state and enters a parsing loop. This loop is iterated as long as the category for the run of characters remains the same. When a *border delimiter*⁸ is detected, the collected fragment is returned. The loop is travelled under a single [parsing] state. State transition is allowed only during the first iteration when the output buffer is empty and an open delimiter lies at the head of the input buffer.

The figure below is provided as an aid in trying to follow the algorithm on C-style string discovery. Look at the two backslashes. C-string definition says a backslash is an escape character INSIDE the string. When parsing the second string has started, pattern-matching will tell it sees two of them, but it sees also an end delimiter (in fact it sees two, read below how the algorithm handles this ambiguity). A backslash is indeed an escaped character only if it is located before the end delimiter.



Drawing 3: Parsing issue

- The “parser” is initialised in the *unclassified* state, *i.e.* local variables \$term (closing delimiter regular expression) is undefined, \$stay (*escape* delimiter regular expression) is set to the 'atom' regular expression and \$btype (returned class or category) is undefined.

Note: The *unclassified* run of characters may be empty. This is the case when an opening delimiter is located at the current scanning position in the source file.

- The rest of the processing steps takes place inside a main “infinite loop” (it is left only on a parsing event, before a state transition).
 - The processing buffer (\$next), if empty, is loaded from queued strings in array @frags or, when exhausted, from a new line of the source file.

When the first two lines of the file are read, they are scanned for an *Emacs* tabulation definition. If one is found, it overrides the tab hint passed through an argument.

⁷ \$doutab is set false by *genxref* to increase speed by avoiding this expansion. *genxref* does not record column numbers where symbols are found and does not display files, consequently visual fidelity is unnecessary.

⁸ Here, the term *border delimiter* is used for short. Usually, a category fragment is delimited by its end delimiter. However, some categories, notably the *unclassified* (or *code*) one have no end delimiter. They are implicitly terminated on encountering a start delimiter for any other category.

Tab characters in the line are expanded if variable `$doutab` requests it.

- First, if *escape* delimiters may be encountered (`$stay` is defined), a loop is entered to check the position of this *escape* delimiter relative to other delimiters. See flowchart at right.

This loop terminates when no more *border* delimiter can be found or the position of one is known, meaning we already went through the loop without doing something useful.

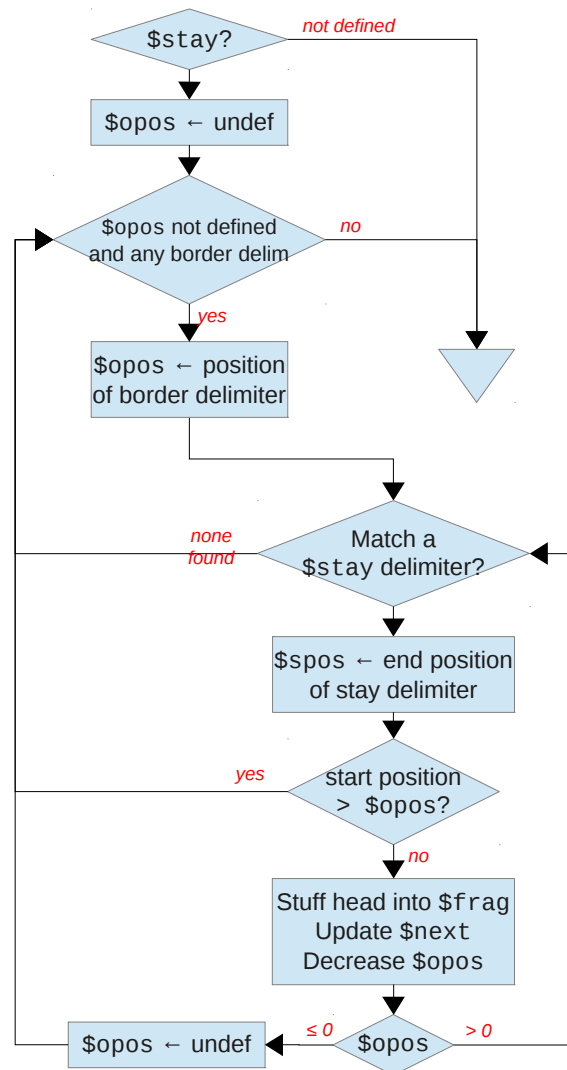
If one is found, its position must be repetitively compared with the position of all subsequent *stay* delimiter. This is the job of second nested loop, entered on detection of a *stay* delimiter.

The inner loop is left as soon as a *stay* delimiter starts at the right of the *border* delimiter, *i.e.* lies outside the currently scanned fragment.

A valid *stay* fragment causes concatenation into the output buffer (`$frag`) of the beginning of the processing buffer (`$next`) up to and including the complete *stay* delimiter. The processing buffer is then truncated and the position of the *border* delimiter updated.

If this position becomes non-positive, it is set undefined and the inner loop is exited to start another iteration of the outer loop.

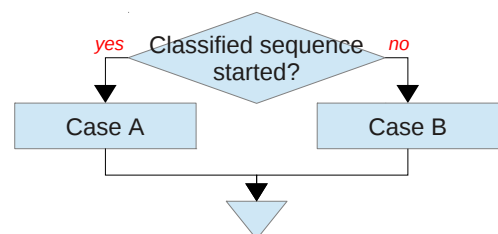
At this point, the current chunk no longer contains "active" escape delimiters. The stay regular expressions will not be considered for the rest of the main loop.



Note:

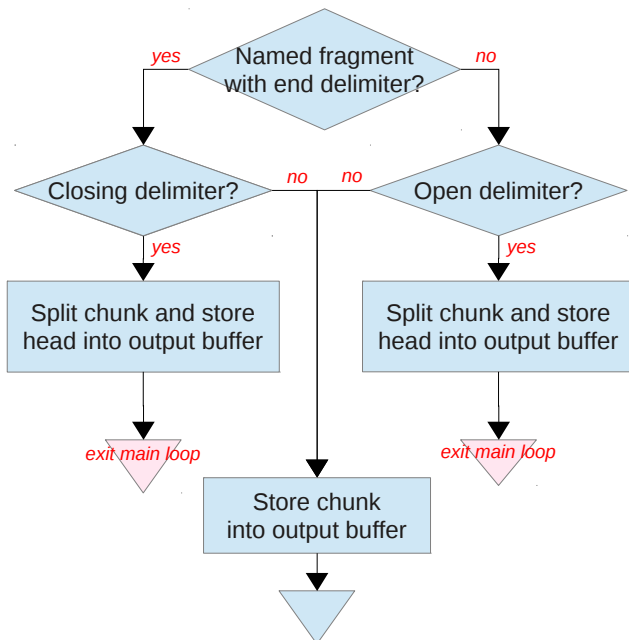
Beginning of the current chunk (up to a *border* delimiter if one is present, the complete chunk otherwise) must be added to the output buffer. Two cases are considered.

If the output buffer already contains meaningful data⁹, the initial part is added as a continuation of the current fragment (case A)



⁹ So-called *meaningless* data is composed exclusively of empty lines. This is an optimisation choice to decrease the number of calls to the parser. Without it, a *code* fragment without real significance would be returned to the caller for second-level parsing. It is more efficient to keep the empty lines as a prefix for another category. This prefix can be very easily processed with low overhead in *Markup.pm*.

Otherwise, we begin a new fragment and must determine its category (case B).



- **Case A**, the output buffer is not empty. See flowchart at left.

The pending fragment may be terminated either by a closing delimiter if its specification defines one, or implicitly by any opening delimiter for a new fragment (as is the case for *unclassified* or *code fragment*). This is the purpose of the initial test.

If such a delimiter is present, the fragment is augmented with the appropriate run of characters, the rest is requeued and the main loop is terminated for return processing.

In the absence of delimiter, the full chunk is added to the output buffer.

Note:

Since delimiter detection gives a wealth of information, we care to prepare variable \$next to avoid pushing the full unused chunk part into @frags to pop it back upon next parser call.

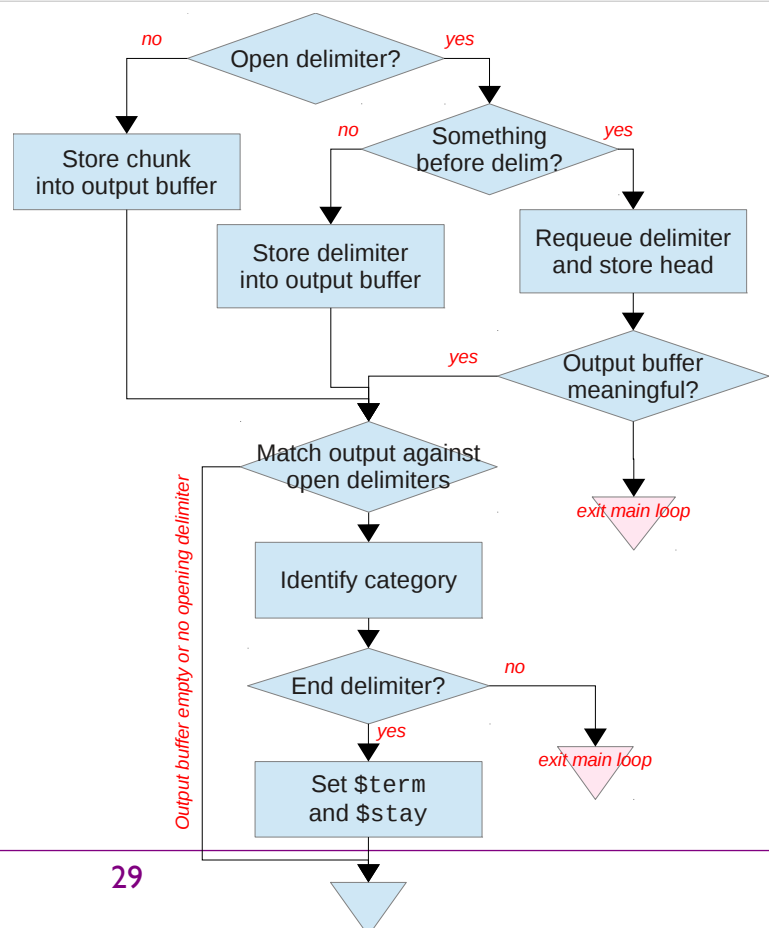
- **Case B**, the output buffer is empty. See flowchart at left.

A new sequence is started, but the current chunk may contain an opening delimiter pertaining or not to this new sequence.

If there is no opening delimiter, the full chunk is added to the output buffer and we proceed to fragment identification.

Depending on its location, an opening delimiter is either the effective beginning of the new sequence (if at the very chunk beginning) or the end of a short *non-classified* (aka. *code*) segment.

The first case is processed like the previous no-delimiter case.



Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

The second case must split the chunk at the delimiter and store the initial part into the output buffer. The delimiter and the end of the chunk are requeued (more precisely, for optimisation sake, the delimiter is kept in `$next` to avoid a push-pop sequence). If the output buffer contains meaningful data, exit from the main loop is forced to return the short assembled fragment.

The new sequence category is identified through a very *Perl*-ish trick retrieving the index of the delimiter in `@open` into variable `$btype`. If no delimiter was found (*unclassified* case), `$btype` will be undefined.



CAUTION!

If you want to modify this section, read at least ten times the `grep` line and be sure you understand all its implications.

If the current category is defined only by its opening delimiter¹⁰, exit from the main loop is forced.

Variables `$term` and `$stay` are loaded with the regular expressions for the closing and *escape* delimiters respectively setting up the parser state for the next iteration through the main loop.

Note:

To be able to cope with some context-sensitive environment, `$term` may be provided as a sub. This is detected here. The sub is invoked and its result `eval`d (computed) to give the actual regular expression.

- The main loop is exited when there is a change in the sequence category. Then, `$btype` is changed from numeric value to the symbolic category name in `@bodyid`. Both `$btype` and the sequence in `$frag` are returned to the caller.

Special notes:

Lines are prefixed with an `\xFF` byte to mark the beginning of a line. This is necessary since the start anchor `^` in regular expressions is supposed to point to the start of the line, which may not coincide with the first character in the buffer (and the first character in the chunk buffer is not always the first character of a line). This special sentinel should be tested for start-of-line in regular expressions instead of start anchor `^`. For the same reason, end-of-line should be checked with `\n` instead of end anchor `$`.

`\xFF` is erased before returning to caller



CAUTION!

This marker may conflict with *Unicode* `ÿ` U+00FF (Latin small letter y with diaeresis).

¹⁰ Apart from this specific case, all categories described in 'spec' have a closing delimiter. Distinction between implicit *unclassified* (or *code*) category which has neither opening nor closing delimiters is made here. For this reason, test at the beginning of case A could be simplified. However, profiling showed no gain.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

2.7.c. Algorithm limitations

As can be seen from the preceding descriptions, regular expressions for the delimiters match against the current chunk. This chunk contains at most one line. This means that **constructs extending further than the current end-of-line can never be matched**. Any “remarkable” delimiter must be completely developed in the current chunk.

This is not an issue for single-character delimiters or the usual “escaped” character pairs. However, it is impossible to capture a sequence spanning several lines if it is defined by the single opening delimiter (special case without ending delimiter). This is also true for 'atom' definition: if the construct to keep inside the unclassified sequence crosses a line boundary, the end of 'atom' regular expression will never match because no extra data is read at this time beyond the chunk end.

IMPORTANT!

This limitation applies only to 'spec' regular expressions. It does not apply to 'include' generic processing (do not mistake this hash with the classification vector of the same name in 'spec') because the internal buffer contains the complete include statement loaded by the parser.

Some language construct definitions need to be anchored at the beginning of a line. Since the beginning of the chunk is not necessarily the same as the beginning of the line, a special character (x\FF) is inserted when lines are read to remember the position.

CAUTION!

Depending on the order of the definitions in *generic.conf*, a *Latin small letter y with diaeresis* (Unicode U+00FF) in column 1 may be erroneously mistaken for a start-of-line and erased. This mishap is very difficult to notice if you do not know what to expect in this line.

TODO:

Use a longer unlikely sequence for start-of-line.

The algorithm is absolutely context-free. This is the main cause of *out-of-sync* situations.

Take HTML as an example. HTML tags may have attributes whose values are strings. But data text (anything outside the <elements>) may contain string delimiters quote and double quote as normal punctuation or grammar diacritic marks. In human languages, they do not necessarily occur in pairs and will cause out-of-sync situation through tag attribute interference.

Lastly, performance is very poor because characters are scanned several times before being sent to the output buffer. This is inherent to pattern-matching and requeueing of unused chunk bits.

2.8. Language parsing

This service resides in file *Lang.pm* and in directory *Lang/*. It complements the file parsing service with language-specific editing actions on the classified runs of characters extracted by *SimpleParse.pm* nextfrag.

2.8.a. Initialisation

The language object is created by method `new`. Its arguments are:

- a string containing the name of the file to parse;
- a string containing the version for this file;

Note:

Considering how this initialisation method is used, global variable `$releaseid` could be used as well and this argument could be dropped.

- an array containing 3 strings to generate the `<A>` links for the identifiers found in the file (the identifier name is simply inserted between every pair of elements to obtain the effective link).

Selection of an appropriate language manager is driven by data from file *filetype.conf*. It is first attempted by matching the file name against one of the file patterns in the 'filetype' list (second item in element). If none is found, the first line of the file is read to check for a *shebang* (`#!`) defining an interpreter, which can eventually be linked to a language through the 'interpreters' configuration parameter. The last fallback is to scan this first line for an *emacs*-style `mode:` definition which is compared against an interpreter name.

In case of match, the language manager named in the third item in element is activated.

Note:

'filetype' and 'interpreters' parameters are listed in file *filetype.conf*.

2.8.b. Public methods

Some of these methods are “dummy” or skeletal stubs which print an error message for an unimplemented language-specific mandatory method.

Routine name	Arguments	Description
<code>processcomment</code>	String reference	Highlight the string as <code>comment</code> with <code>multilinetwist</code>
<code>processtring</code>	String reference	Highlight the string as <code>string</code> with <code>multilinetwist</code>
<code>processextra</code>	String reference	Highlight the string as <code>extra</code> with <code>multilinetwist</code>
<code>processinclude</code>	String reference Directory name	Stub <i>The string contains the full “include” statement. Supposed to create the hyperlink for an included file from the</i>

Routine name	Arguments	Description
		<i>directory argument if file name has none.</i>
<code>processcode</code>	String reference	Stub <i>The string contains the full "code" block which is not synchronised with statement boundaries. Supposed to create hyperlinks for the identifiers and highlighting for keywords.</i>
<code>processreserved</code>	String reference	Stub <i>Initially supposed to create highlighting for keywords. Not used because keyword are handled in processcode.</i>
<code>indexfile</code> <i>For genxref use</i>	LXR file name OS file path File id Reference to DB object Reference to configuration object	Stub <i>Supposed to parse a file and collect the definitions. LXR file name and OS file path differ when files are stored in a VCS (in this case OS path is a temporary file).</i>
<code>referencefile</code> <i>For genxref use</i>	LXR file name OS file path File id Reference to DB object Reference to configuration object	Stub <i>Supposed to parse a file and collect the references. Returns the number of lines or 0 if file not processed. LXR file name and OS file path differ when files are stored in a VCS (in this case OS path is a temporary file).</i>
<code>language</code>		Stub <i>Supposed to return the language name associated with the object. It can then be used to reference the correct descriptor in 'langmap' if the object is derived from the generic parser Generic.pm.</i>

Note:

Remember that `comment`, `extra` and `string` are prefixes for families of category.

2.8.c. Support routines

These functions offer various services related to language handling.

Routine name	Arguments	Description
<code>parseable</code>	File name	Simplified version of initialisation method <code>new</code> without object creation; this function returns 1 if some language parser could handle the file content, 0 otherwise
<code>multilinetwist</code>	String reference CSS class name	Returns the string argument inside a <code> </code> block with the given CSS class attribute name. If the block spans several lines, the block is closed before the end of line and reopened after the end of line. The eventual final empty block is removed. <i>This creates the HTML highlighting.</i>

Routine name	Arguments	Description
<code>_linkincludedirs</code>	File link File name (encoded) Language-specific path separator File path (encoded) Directory name	Returns the link argument modified to hyperlink every directory comparison of the path. The file name argument is written following the rules of the language, with the separator defined by the next argument. The file path argument is URL %-encoded so that it can be directly used in the <A> tags. The directory argument is the default directory for file-only paths. <i>Note that the file name, file path and directory arguments are used to call <code>incdirref</code>.</i>
<code>_incfindfile</code> <i>Also used by <code>incref</code> from <code>Common.pm</code></i>	File/directory flag File path Extra directory array	Tries to resolve the file path argument as a file (flag 1) or a directory (flag 0) among the directories given or those of ' <code>incprefix</code> ' configuration parameter. The resolved path is returned or <code>undef</code> .
<code>incdirref</code>	Name to display CSS class Directory path Extra argument array	Returns <code>filerref</code> (first three arguments) if <code>incfindfile</code> succeeds, the name to display otherwise

2.9. File access management

This service resides in file *Files.pm* and in directory *File/s*. It provides an abstraction layer for file access, no matter how they are stored.

Initialisation `new` is only a dispatcher towards the specialised `new` methods of the specific managers. The appropriate manager is selected based on the prefix in the first argument:

```
rep_type: absolute_OS_path_to_repository
```

where

- `rep_type` is one of `CVS`, `git`, `svn`, `hg` or `bk` for *CVS*, *GIT*, *Subversion*, *Mercurial* or *BitKeeper*¹¹. For plain files, no prefix is needed and the colon (:) separator is also omitted.
- `absolute_OS_path_to_repository` is the source tree repository. For plain files, it is the master level directory containing the version subdirectories.

The second optional argument is a *hash* reference for key/value pairs passing options to the specific manager. This hash is usually taken from '`sourceparams`' in *lxr.conf*.

2.9.a. Public methods

The following methods are generally meant to be overridden in the specific managers.

¹¹ Usage of *BitKeeper* is discouraged because the module has not been updated since at least 2005 and not even tested because it is now proprietary.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
<code>getdir</code>	File name Version	Stub <i>Supposed to return an array enumerating the directory content or undef if the directory does not exist in this version</i>
<code>getfile</code> <i>Primarily used by genxref and by some specific managers when there is no other means to compute file size</i>	File name Version	Returns the content of the designated file in the requested version as a long string or <code>undef</code> if the file does not exist. <i>This method should be avoided as much as possible since it imposes a heavy load on memory.</i>
<code>getnextannotation</code>	File name Version	Stub <i>Supposed to return the annotation¹² for the next line</i>
<code>truncateannotation</code>	String reference Length	The string contains an annotation (as retrieved from the previous method) which must be truncated to the number of screen positions given by the length argument. Extra characters on the left side are replaced by a truncation indicator; the truncated annotation occupies <i>length+1</i> screen positions! The returned value is the final number of screen positions. The length argument unit is <i>screen positions</i> , not “computer” characters, which means that <i>HTML</i> editing tags or elements can be added freely since they use no screen positions. <i>This method must be overridden if the most meaningful characters are on the left-hand side and truncation must be done at right.</i>
<code>getauthor</code>	File name Version Annotation	Stub <i>Supposed to return the name of the annotation committer</i>
<code>filerev</code>	File name Version	Stub <i>Supposed to return the latest file revision</i> <i>A revision is more specific than a version and uniquely identifies a file content.</i>
<code>getfilehandle</code>	File name Version	Stub <i>Supposed to return a handle to the requested file for further content access, or undef if the file does not exist</i>
<code>getfilesize</code>	File name Version	Stub <i>Supposed to return the file size in bytes</i> <i>For some SCM, this may require to extract the file</i>
<code>getfiletime</code>	File name Version	Stub <i>Supposed to return the file latest modification time</i> <i>For some SCM, this may require to extract the file</i>
<code>isdir</code>	File name Version	Stub <i>Supposed to return “true” if the designated file name is an existing directory</i> Note: <i>Testing for directory is rather time-consuming; consequently, after LXR initialisation in <code>httpinit</code>, directory paths are suffixed with / so that a trailing slash is a signature for a directory</i>

¹² An annotation is whatever information kept with a line, usually the revision number the line was entered into the file.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
		<i>afterwards.</i> <i>This method is used to check a path other than the currently displayed file, for example an include directory.</i>
isfile	File name Version	Stub <i>Supposed to return "true" if the designated file name is an existing file</i> Notes: <i>This method is used to check a path other than the currently displayed file, for example an included file.</i> <i>If the file should subsequently be accessed, it is simpler and more efficient to call <code>getfilehandle</code>.</i>
realfilename	File name Version	Returns the name of a real file with the same content as the designated path or <code>undef</code> if the copy operation failed. File content is extracted from the repository and copied into a temporary file. <i>This method should be overridden if plain files can be accessed without the copy operation.</i>
releaserealfilename	File name	Erases the designated file. The method attempts to assert that the file name looks like it has been created by <code>realfilename</code> but this is not 100% guaranteed. IMPORTANT! <i>If <code>realfilename</code> has been overridden, override this method also to revert accurately the actions; otherwise you risk destroying an original file.</i>

2.9.b. Support methods

These methods offer various services related to language handling.

Routine name	Arguments	Description
getannotations <i>Deprecated</i>	File name Version	Stub <i>Supposed to return an array of all annotations, each element corresponding to a line</i> <i>To preserve memory, source uses now <code>getnextannotation</code>;</i> <i>from release 1.1 on, this method is considered "internal" in a specific manager when the underlying SCM provides only file annotations and cannot provide individual line annotation.</i>
_ignoredirs	Directory path Node	Processes ' <code>ignoredirs</code> ' parameter against node and ' <code>filterdirs</code> ' against directory and node; returns 1 if the directory should be ignored, 0 otherwise. Node is the last segment of the full path (<i>i.e.</i> a name relative to the directory argument). <i>This filter is to be called from method <code>getdir</code> to filter out unwanted nodes.</i>
_ignorefiles	Directory path	Processes ' <code>ignorefiles</code> ' parameter against node and

Routine name	Arguments	Description
	Node	'filterfiles' against directory and node; returns 1 if the file should be ignored, 0 otherwise. Node is the last segment of the full path (<i>i.e.</i> a name relative to the directory argument). <i>This filter is to be called from method getdir to filter out unwanted nodes.</i>

2.10. Database management

This service resides in file *Index.pm* and in directory *Index/*. It provides an abstraction layer for database access, no matter the underlying engine. Query language is *SQL*. As long as a database engine is standard-compliant, there is no need to override a method. The main reason to do so is related to auto-increment features.

Method *new* initialises the database object. Its argument is a reference to the configuration object (so that the module may access any needed parameter).

It routes the call to the appropriate specialised database manager which creates a *self* object containing engine-specific transaction templates. This object is augmented with generic transaction templates (independent of the underlying engine), taking care not to erase an overridden template.

Global variables implement caches to avoid frequent accesses to the database thus improving performance at the cost of larger memory footprint.

`%files` remembers the unique file identifications (internal base version designation) for a version of a file;

Disabled in release 2.0:

It does not look really necessary since a file is met only twice (collecting the definitions then the references). In the case of the kernel, this ends up with a huge *hash* (`%files`) putting a heavy stress on memory. On small test cases, there seem to be no difference in indexing time, or maybe a little advantage when the cache is disabled.

`%symcache` remembers the unique internal symbol identification to avoid database lookup (the value may be `undef` for symbols without declaration); used in `setsymreference`, `issymbol`, `symid`, `flushcache` and `purgefile`, checked for consistency in `setsymdeclaration`;

`%cntcache` remembers the reference count for the symbol (to spot changed values, reference counts are negated when read from the database and set back positive when incremented or decremented); used in `setsymdeclaration`, `setsymreference`, `issymbol`, `symid`, `flushcache` and `purgefile`;

Note:

As a trade-off between performance, memory footprint and database consistency, it is suggested to flush the symbol caches %symcache and %cntcache at the end of every file processing.

`$database_id` running counter incremented by *genxref* every time it opens a new database (also done by *httpinit* but this less important because an HTTP request sees only one database); this can be used by any module needing to initialise or inspect the database once only.

Note:

This is used by *Generic.pm* initialisation routine to write the types mapping into the database on its first instantiation with this database.

`$fileini, $symini, $typeini`
 value of the counters at the beginning of the session

`$filenum, $symnum, $typenum`
 running counters incremented every time it opens a new file, symbol or language type is entered into the database. If they differ from their initial value at the end of the session, their final value is recorded in the database for later use by another session.

2.10.a. Support methods

These methods offer various services related to manipulate the database.

Routine name	Arguments	Description
<code>uniquecountersinit</code>	DB table prefix	Retrieves the latest stored values for file, symbol and type ids <i>To be used when the built-in features for fields with unique attribute lead to poorer performance.</i> <code>uniquecounterssave</code> writes back the final values to the database.
<code>fileid</code> <code>fileidifexists</code>	File name Revision	Returns a unique id for a file with a given revision. <code>fileid</code> creates this record if it does not exist, while <code>fileidifexists</code> would return <code>undef</code> . CAUTION! <i>Revision is the kind of data returned by function <code>filerrev</code> from the file management service, not a "user-visible" version.</i>
<code>getallfilesinit</code>	Version	Initialises an internal iterator for retrieving all files present in the requested version through execution of the <code>allfiles_select</code> transaction. <i>Individual records are retrieved with <code>nextfile</code> method.</i>
<code>nextfile</code>		This an iterator running over all files making up a version of the source-tree. Each call retrieves a file description until it returns <code>undef</code> , at which time it must no longer be called. The returned record is a list containing a file id, a filename, a revision (<i>remember, NOT a version!</i>) and the number of versions

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
		associated with this revision.
setfilerelease	File id Version	Associates the given version with the file id. Version is any tag by which the file in this state is known by the VCS. As noted in <code>fileid</code> description, a file id is a canonical unique identification of a state of the file, but this state may be shared by several versions of the source-tree.
removerelease	File id Version	Removes the given version from the association list to file id
fileindexed	File id	Returns 1 if the file id has been “indexed”, otherwise 0 <i>Indexation is the definitions collecting phase</i>
setfileindexed	File id	Marks the file id as been indexed
filereferenced	File id	Returns 1 if the file id has been “referenced”, otherwise 0 <i>Referencing is the references collecting phase</i>
setfilereferenced	File id	Marks the file id as been referenced <i>This method also updates the indexing time-stamp.</i>
filetimestamp	File id	Returns the time when this file was referenced
symdeclarations	Symbol name version	Returns an array containing the set of declarations for the given symbol. Every element is list consisting of a file name, a line number, a string describing the type and an eventual symbol id if this declaration is nested in another one (such as a field of a structure).
setsymdeclaration	Symbol name File id Line number Language id Symbol type id Related symbol	Records this declaration as described by the arguments <i>Related symbol is omitted if this declaration is not part of a larger one.</i> <i>Reference counts to the symbols are automatically updated.</i>
symreferences	Symbol name Version	Returns an array containing the set of declarations for the given symbol. Every element is list consisting of a file name and a line number.
setsymreference	Symbol name File id Line number	Records this usage as described by the arguments <i>If the symbol does not exist in the database (no declaration for it has been encountered), nothing is recorded.</i>
issymbol	Symbol name Version	Returns 1 if the symbol exists in the database for the given version, 0 otherwise IMPORTANT! <i>This method is intended only to decide symbol highlighted. It MUST NOT be used during indexation.</i>
symid	Symbol name	Returns a unique id for the symbol, entering it into the database if it is unknown.
symname	Symbol id	Returns the symbol name corresponding to the given id
decid	Language id Text	Returns an integer identifying the text for a type declaration If the record does not exist in the database, create it and return its

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
		integer code. Note: <i>There is no id to text retrieval function because it is coded inside symdeclarations method, its only meaningful use.</i>
commit		Secures the last set of operations in the database and starts a new transaction
forcecommit		Commits the database now, even if auto commit mode is in effect IMPORTANT! <i>This method should not be overridden in specific managers.</i>
emptycache <i>Deprecated</i>		Erases the internal symbol cache (without writing to the database)
flushcache	Full flush flag	Flushes the cached symbol reference counts to the database <i>Method to be called at the end of each file processing</i> Implementation: To minimise I/O when adding declarations or references, initial reference counts are negated when entered into the cache. They are turned back positive when they need to be incremented. Strictly positive values show which symbols have been referenced. Only these are flushed to the DB. However, when deleting symbols, the reference count may eventually decrement to zero. In this case, it is necessary to also write back zero-reference symbols otherwise the database loses its integrity. The flag is set to 1 to request full cache write-back. <i>The cache is erased before returning to the caller.</i>
purgefile	File id Version	Deletes data associated with the given version of the base file (designated by its id) The method carefully decrements first the reference counts in the higher-level symbols (the <i>related</i> symbols). Then the definitions and references in this file can be deleted in any order.
purge	Version	Deletes all data pertaining to a version
purgeall		Completely wipes out the database
uniquecountersreset	Forced initial value	Resets the counters to 0 Initial counters are forced to the argument value. If it is different from 0, it will cause automatic write to the database in the following method. Note: it is recommended to call this method with argument 0 to disable the feature after the save to the database.
uniquecounterssave		Writes in the database the final value of the “unique” counters if it differs from the stored value
dropuniversalqueries		Deactivates the query handles defined in <code>new</code> to avoid warning messages about still active <i>SQL</i> statements before disconnecting from the database. Called from <code>final_cleanup</code> . Note: the warning message is harmless but may disturb the casual user.

Routine name	Arguments	Description
final_cleanup		Commits the pending transactions, cleans up internal DBD state and disconnects from the database IMPORTANT! <i>MUST be called before the object disappears.</i>

2.11. Local customisation

File *Local.pm* is the home for (limited) local customisation of directory listing. What is presently inside was contributed by Dawn Endico (*aka.* dme) a long time ago. Code targets *C* and *Java* source files.

The two exported support functions are *template editing functions* (see 2.5 HTML stream generation management) which are referenced from template text as:

Routine name	Macro name	Description
dirdesc	description or desc <code>text</code>	Returns a description for a directory; if none can be found, returns at least a non-breaking space to force the browser not to skip this element (otherwise it may mess up screen lay-out)
filedesc	desc <code>text</code>	Returns a description for a file; if none can be found, returns at least a non-breaking space to force the browser not to skip this element (otherwise it may mess up screen lay-out)

descexpand is a dispatching routine selecting the final editing function depending on the file or directory nature of the node. It expands \$desc`text` in its template argument with dirdesc for a directory and fdescexpand for a file. The returned string is provided by the called function.

Arguments for this function are:

- the template string (should contain \$desc`text` to have an effect),
- file or directory name (last path segment only),
- parent directory name,
- version name.

dirdesc may be called as a result of expanding either \$description¹³ in a header area above the directory content area or \$desc`text` in a \$description template block in *subdirectory* context. Arguments for this function are:

- full directory path,
- version name.

¹³ To be honest, it should rather be considered an implicit expansion of \$desc`text`. The expansion rule might be modified in a future release to unify both contexts.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

`filedesc` is called as a result of expanding `$desctext` in a `$description` template block in *file* context. Arguments for this function are:

- file name (last path segment only),
- parent directory name,
- version name.

Note:

To disable this feature which may be time-consuming or too difficult to implement/customise, write functions returning only string `' ; '` (a single non-braking space).

`filedesc` reads a reasonable amount of lines at the beginning of a *Java* or *C-family* file. If this is a *Java* file, it keeps only the last comment before declarations. It looks in the remaining lines for the file name or `Description:` marker. In this case, the corresponding paragraph is returned after elementary editing. Otherwise, common text usually present (such as licence, copyright, ...) is removed. What is left after other filtering is returned.

`dirdesc` looks for different flavours of *README* files and hands it over to `descreadmehtml` or `descreadme`.

Internal support routines are:

Routine name	Arguments	Description
<code>descreadmehtml</code>	Directory name File name File handle	Returns a description from the <i>HTML</i> file; see below for the definition of this description
<code>descreadme</code>	Directory name File name File handle	Tries to discover descriptive text in this text file and returns it as an <i>HTML</i> element.
<code>convertwhitespace</code>	String	This auxiliary routine helps to preserve text lay-out when mapped to <i>HTML</i> Line breaks become <code>
</code> . Bullet-list paragraphs (which, by convention, start with lowercase letter <code>O</code> followed by whitespace) are rendered with <code>
</code> , two non-breaking spaces and lowercase letter.

Within *README.html* files, a `` element is considered description intended to be displayed by *LXR*. However, the scanner is very simplistic and stops at the first `` tag. This means no `` block can be nested inside the *LXR* description, with the exception of a `` element. If a nested `` is detected, nothing will be displayed. Priority is given to the “short” description if it is present.

2.12. Derived language parsers

The specific parsers are stored in directory *Lang/*. They are derived classed of *Lang.pm*.

Presently, there is only one derived parser *Generic.pm*¹⁴ which is table-driven to parse many languages. The tables come from file *generic.conf*. Specialised versions are derived from it to provide an easier handling of some language constructs, mostly *include* statements.

As hinted in section 2.8.b Public methods, *new* and *stub* methods must be overridden (*i.e.* all except *processcomment*, *processextra* and *processstring*). *new* is a special case because it is referenced indirectly through *Lang.pm* *new* and has different arguments from the latter.

2.12.a. Generic parser

The private global variable `$generic_config` contains a complete copy of file *generic.conf* once initialised. This allows to avoid reading again the configuration file when several instances of the parser are simultaneously created.

The second private global variable `$seenDB` contains a copy of the global counter `$database_id` (located in *Index.pm* and incremented by *genxreef* or *httpinit*). If both values are equal, the database did not change since last invocation and no initialisation is needed. If they disagree, a full initialisation is necessary if `$generic_config` is undefined, otherwise only the types table is written to the database; finally, `$seenDB` is updated to the counter current value.

The generic parser overrides the following methods:

Routine name	Arguments	Description
<code>new</code>	File name Version Language name	Returns the object structure with the complete configuration file content. <i>In the present implementation, argument file name is not used. To spare memory, only the language relevant part of the configuration file could be kept in the object structure.</i>
<code>indexfile</code> <i>For genxref use</i>	LXR file name OS file path File id Reference to DB object Reference to configuration object	Launches <i>ctags</i> to collect the declarations and enters them into the database
<code>processinclude</code>	String reference Directory name	Splits the string according to the 'include' definition in file <i>generic.conf</i> and changes it with keyword highlighting and hyperlink insertion <i>Uses <code>_linkincludedirs</code> for hierarchical link editing</i> Note: <i>If no 'include' definition is supplied in the configuration file, default statement syntax is supposed to be "keyword file-without-</i>

¹⁴ All other parsers in directory *Lang/* are derived from *Generic.pm*.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
		<i>delimiters</i> ". If a statement terminator follows file without intervening spaces, it is considered part of the filename.
processcode	String reference	Highlights all reserved keywords and hyperlinks the known symbols
referencefile <i>For genxref use</i>	LXR file name OS file path File id Reference to DB object Reference to configuration object	Using <i>SimpleParse.pm</i> <code>nextfrag</code> , it scans unclassified fragments non-reserved symbols which are entered into the database Reminder: symbols which were not entered during the declaration phase are filtered out by <code>setsymreference</code> ; consequently, there is no need to look up the database before submitting the symbol.
language		Returns the effective language name

Auxiliary methods are:

Routine name	Arguments	Description
read_config		This internal <code>sub</code> (not a method) reads into global variable <code>\$generic_config</code> the complete content of file <i>generic.conf</i> and stuffs into the database the human-readable type declarations corresponding to the <i>ctags</i> one-letter types found in the 'typemap' sections.
parsespec		Returns an array containing the category definitions as found in the 'spec' section of <i>generic.conf</i> for this language
flagged	Flag name	Returns the value (0/1) of the given flag from the 'flags' section of <i>generic.conf</i> for this language
isreserved	Symbol name	Returns 1 if the symbol is a reserved word Reserved words are listed in the 'reserved' section of <i>generic.conf</i> for this language
langinfo	Item name	Returns the requested item of the 'langmap' section of <i>generic.conf</i> for this language

2.12.b. C parser

Derived from the generic parser, *C.pm* only reimplements method `processinclude` for syntax accuracy and speed performance.

2.12.c. COBOL parser

Derived from the generic parser, *Cobol.pm* is yet in an experimental state. In particular, code layout (margins A and B) is not taken into account. It reimplements `referencefile` as an empty method and `processcode` for case-insensitivity keyword and identifier detection.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

2.12.d. HTML parser

Derived from the generic parser, *HTML.pm* only reimplements method `processinclude` for handling targets in `<A>` or `` tags. If targets are URL, *i.e.* they start with *scheme:*, the target is handed over to `freetextmarkup` which will highlight the URL.

2.12.e. Java parser

Derived from the generic parser, *Java.pm* only reimplements method `processinclude` for processing package and import statements. Both variants of `import` are handled. The tail is requested for eventual extra *Java* code.

2.12.f. Make parser

Derived from the generic parser, *Make.pm* only reimplements method `processinclude` for speed efficiency and iterating on the file list.

2.12.g. Pascal parser

Derived from the generic parser, *Pascal.pm* reimplements methods `new` and `processinclude`.

Method `new` calls the generic method and captures the file extension which varies with OS and compiler. This extension will be added to the *Pascal*-filename designation to obtain the OS-filename in `USES` statements (*include* feature name in *Pascal*).

Method `new` iterates on the `USES` file list and suffixes the *Pascal* filename with the current extension before creating an hyperlink.

2.12.h. Perl parser

Derived from the generic parser, *Perl.pm* only reimplements method `processinclude` to cope efficiently with both variants of the statements. The tail is requested for eventual extra *Perl* code.

2.12.i. Python parser

Derived from the generic parser, *Python.pm* only reimplements method `processinclude` for handling `import` statement efficiently. Unfortunately, the `from ... import ...` variant is processed in two independent passes: the first one deals correctly with `from`; the second one manages the requested `import` tail (having “forgotten” the directory defined in the `from` part). This results in incorrect hyperlinks being created if any.

2.12.j. Ruby parser

Derived from the generic parser, *Ruby.pm* only reimplements method `processinclude` for speed efficiency.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

Be aware that due to the very dynamic nature of *Ruby* LXR does not manage accurately *Ruby* syntax.

2.13. Specialised file access managers

The specific managers are stored in directory *Files/*. They are all derived classed of *Files.pm*.

The oldest managers are Plain.pm for ordinary files and CVS.pm for CVS repositories. BK.pm is no longer maintained because BitKeeper changed to proprietary licence around 2005.

2.13.a. BitKeeper manager

Derived from *Files.pm*, *BK.pm* implements all stub methods. Its private support methods are:

```

openbkcommand(command)
    Executes command and returns a handle to the pipe result
insert_entry(...)
    Inserts an entry in the internal cache (sub, not method)
fill_cache(version)
    Fills the internal cache with directory contents
get_tree(version)
    Returns the entire tree as an array
cachename(version)
    Returns the cache file name
canonise(file name)
    Returns the file name with initial / removed (sub, not method)
file_exists(file name, version)
    Returns 1 if file exists (checking data in the internal cache)
get_fileinfo(file name, version)
    Fills the internal cache and returns cached data for the file

```

2.13.b. CVS manager

Derived from *Files.pm*, *CVS.pm* implements all stub methods.

Method *new* checks that *rcs diff* is GNU compliant because retrieving differences requires GNU arguments. The boolean result is kept in global variable `$gnu_diff`.

Method *getdir* handles the case of the *Attic/* directories, the location where *CVS* keeps removed files.

Method *getannotations* builds the annotation for the designated file in global array `@anno`. It proceeds by reading the file, keeping only the number of lines. It tells which version a line was entered by scanning the difference directives between older and older versions. It must both read *CVS* internal data and *diff* output. *This method should be considered "internal" only called from getfilehandle*.

Method *getnextannotation* only pops the head element of array `@anno`.

Method *getauthor* retrieves the author from *CVS* internal data.

Method *filerev* returns "standard" numeric revisions (but for "import" branches folded to 1.1 because they cause problems) or translate version symbols to revisions through internal data

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

scanning.

Method `getfilehandle` proceeds through checking out the requested file.

Method `getfilesize` reads the file to compute its length.

Method `getfiletime` retrieves the commit time from *CVS* internal data.

Its private support methods are:

Routine name	Arguments	Description
<code>toreal</code>	File name Version	Returns the name of a real file containing the change history of the requested file (usually just suffixed with <code>,v</code> but may be located in an <i>Attic/</i> directory) This involves decoding <i>CVS</i> internal data with <code>parsecv</code> s to check if this revision is dead . <i>Virtually called by all other methods.</i>
<code>getdiff</code>	File name Version 1 Version 2	Returns a file handle to the patch directives transforming version 1 into version 2 with <code>rcsdiff</code>
<code>dirempty</code>	Directory name Version	Returns 1 if directory is “empty” <i>Here, empty means directory contains only “empty” subdirectories or files not belonging to the designated version.</i>
<code>cleanstring</code> <i>Candidate for deletion</i>	String	Returns a string with all “dangerous” characters removed <i>Dangerous characters are those allowing an XSS attack or otherwise disturbing LXR operation. This, of course, limits the set of possible filenames.</i> <i>This method is used only to prune a file name before launching a shell command (no XSS risk). It has been commented out to experiment.</i>
<code>allreleases</code>	File name	Returns a list of all release tags of which this file is a member <i>A release tag is a symbolic identifier, not a numeric revision number.</i>
<code>allrevisions</code>	File name	Returns a list of all revision numbers of which the file is a member
<code>byrevision</code>		Comparison function (<i>not a method</i>) to sort according to revision numbers
<code>parsecv</code> s	File name	Method to parse internal <i>CVS</i> data Note: <i>Since this method is called from many other methods, among which <code>toreal</code> which calls <code>parsecv</code>s, and it itself calls <code>toreal</code>, global variable <code>\$cache_filename</code> is used as a cache to prevent unneeded parsing AND infinite recursion.</i> The method read change history data and stores it in global hash <code>%CVS</code> .(see below).

CVS change history files are composed of *paragraphs*: a *paragraph* is a group of non-empty lines or `@`-delimited strings¹⁵ (which may span several lines and contain blank lines). *Paragraphs* are

¹⁵ `@` inside strings are written `@@`.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

separated from each other by blank lines.

Paragraphs start with a *tag* and consist of a list of items (keyword followed by an optional list of tags or values and terminated by a semicolon ;). If a value needs to span several lines or may contain a semicolon, it is coded as an @-string.

The first paragraph is a header (keyword head¹⁶). The items are:

- the latest revision number (considered as item head);
- access (not used by *LXR*);
- symbols defining the correspondence between release tags and revision numbers as a list of *tag:revision*, stored in {'header'}{'symbols'}{*tag*} with *revision* value:

Note:

An undocumented feature allows to customise the release tags for display purpose in the *version* list (variable 'v'). Configuration function 'cvsversion' (in file *lxr.conf*) is applied to *tag* (passed as an argument) and returns an alias to be used from now on. Example to replace hyphens by spaces:

```
'cvsversion' => sub
    { my ($tag) = @_
      ; $tag =~ s/-/ /g
      ; return $tag
    }
```

- locks (not used by *LXR*);
- strict (not used by *LXR*);
- comment (not used by *LXR*), value is an @-string.

With the exception of symbols, all item *values* end up stored in {'header'}{*tag*}.

The following paragraphs describe the revisions. Their keyword is the *revision* number. They contain the following items:

- date with commit time and date;
- author;
- state (only dead value tested);
- branches giving the revision numbers of the lateral branches stepping out of this revision;
- next giving the “ancestor” revision number (this defines a reverse chronological order of the revision or describes the tree “upside-down” starting from the leaves and ending in the root).

All item *values* end up stored in {'branch'}{*revision*}{*tag*}.

¹⁶ Not sure if the header should not be considered as an “anonymous” paragraph starting immediately with items. Nevertheless, it works well as is.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0



CAUTION!

The next relation on a lateral branch points in the opposite direction compared to the main trunk: from the branching point to the leaves. Consequently, `{'header'}` `{'symbols'}``{branch symbol}` is modified to retain the latest revision number on the branch so that selecting this symbol will list the file in a different revision than that of the branching point.

This may eventually give an inaccurate result if other tags have been added later on the branch.

The next paragraph desc contains a description in an @-string. There is no terminating semicolon. This string is not used by **LXR**.

The following paragraphs with keyword equal to the *revision* number contain the needed information to rebuild the file as it was in the *revision* state. The items¹⁷ are:

- log containing the commit message in an @-string;
- text being either the file original content for the *head* revision or the patch directives for transforming a revision into the *next* (according to the tree relationship from the header).

These paragraphs are not recorded in the %cvs hash. A standard check-out is done when a file revision is needed.

2.13.c. GIT manager

Derived from *Files.pm*, *Git.pm* implements all stub methods.

Method `getdir` builds the content array with data returned from command `git ls-tree`.

Method `getnextannotation` pops the head element of array `@{'annotations'}`, eventually causing it to be refilled by `loadline`.

Method `truncateannotation` is reimplemented for truncation at right.

Method `getauthor` pops the head element of array `@{'authors'}`, eventually causing it to be refilled by `loadline`.

Method `filerev` relies on command `git rev-list` to return the latest revision id.

Method `getfilehandle` returns a “file handle” to the designated file version. It is a usual file handle as returned by command `git cat-file` if no annotations are required. If annotations are requested, it is a fake file handle because annotation, author and source content are all presented in a unique line. This requires processing to separate information. An *ad hoc* `getline` method is added to this fake file handle (the **GIT** object itself) to transparently simulate a standard file handle behaviour.

¹⁷ Here, the items are not terminated by a semicolon.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Method `getfilesize` reads the result of command `git cat-file -s`.

Method `getfiletime` extracts the commit time from command `git cat-file commit`.

Methods `isdir` and `isfile` rely on command `git ls-tree`.

Its private support methods are:

Routine name	Arguments	Description
<code>loadline</code>		Strictly internal method to grasp the next annotated line and split it into annotation, author and line respectively queued into object arrays <code>{'annotations'}</code> , <code>{'authors'}</code> and <code>{'nextline'}</code> . On end of file, array <code>{'nextline'}</code> and the real file handle (to the <i>GIT</i> pipe) are deleted.
<code>getline</code>		Method used to make the <i>GIT</i> object a pseudo-file handle when annotated source is required. The “next” line is taken from array <code>{'nextline'}</code> . End of file causes <code>undef</code> to be returned.
<code>_git_cmd</code>	Command Argument array	Returns a handle to a pipe from which command output can be read <i>GIT directory location is forced with a --git-dir option.</i> IMPORTANT! The returned pipe must be explicitly closed by the caller.
<code>_git_oneline</code>	Command Argument array	Wrapper method for cases when a single line is expected from the <i>GIT</i> command The pipe is closed by the method before returning the result.

`loadline` and the three object arrays simulate three independent files (namely annotation, author and source) from a single physical source.

Note:

All paths in *LXR* are relative to the repository directory defined by configuration parameter 'sourceroot'. Since this directory is passed to *GIT* commands with option `--git-dir`, paths **MUST** look like relative paths and the initial slash is removed from the names in these commands.

2.13.d. Mercurial manager

Derived from *Files.pm* (and similar to *GIT.pm*), *Mercurial.pm* implements all stub methods.

Method `getdir` builds the content array with data returned from command `hg ls-onelevel`.

Method `getnextannotation` pops the head element of array `@{'annotations'}`, eventually causing it to be refilled by `loadline`.

Method `getauthor` pops the head element of array `@{'authors'}`, eventually causing it to be refilled by `loadline`.

Method `filerev` relies on command `hg id -n` and `hg log` to find the requested revision. If an exact match is not found, it returns the time-closest revision number.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Method `getfilehandle` returns a fake “file handle” to the output pipe of command `hg cat` which returns “decorated” lines (complete with annotation and author). This requires processing the lines to separate information. An *ad hoc* `getline` method is added to this fake file handle (the *Mercurial* object itself) to transparently simulate a standard file handle behaviour.

Method `getfilesize` reads the result of command `hg fsize`.

Method `getfiletime` extracts the commit time through parsing the output of command `hg log` with method `parsehg`.

Methods `isdir` and `isfile` check the last character in the file name argument for the presence of `/`.

Its private support methods are:

Routine name	Arguments	Description
<code>loadline</code>		Strictly internal method to grasp the next annotated line and split it into annotation, author and line respectively queued into object arrays <code>{'annotations'}</code> , <code>{'authors'}</code> and <code>{'nextline'}</code> . On end of file, array <code>{'nextline'}</code> and the real file handle (to the <i>Mercurial</i> pipe) are deleted.
<code>getline</code>		Method used to make the <i>Mercurial</i> object a pseudo-file handle. The “next” line is taken from array <code>{'nextline'}</code> . End of file causes <code>undef</code> to be returned.
<code>parsehg</code>	File name	Builds the object hashes <code>{'changeset'}</code> and <code>{'date2rev'}</code> which give respectively the commit date for a revision and the revision for a commit date

`loadline` and the three object arrays simulate three independent files (namely annotation, author and source) from a single physical source.

Note:

All paths in *LXR* are relative to the repository directory defined by configuration parameter `'sourceroot'`. The current directory is switched to the latter before launching any *Mercurial* commands. Paths **MUST** also look like relative paths and the initial slash is removed from the names in these commands.

To check:

Directory listing incurs very poor performance. The cause has not yet been identified. Since directory listing is the only context where many files are interrogated, could the implementation of `parsehg` be blamed? `{'changeset'}` and `{'date2rev'}` are never erased, thus resulting in bigger and bigger hashes, which may also cause erroneous answers for `filerrev`.

Complimentary methods for version selection in configuration file:

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
alltags	File name	Returns a list of all tags valid for the designated file The list is built with command <code>hg tags</code> .
allbranches	File name	Returns a list of all branches valid for the designated file The list is built with command <code>hg branches</code> .

2.13.e. Plain files manager

Derived from *Files.pm*, *Plain.pm* implements all stub methods.

Method `getdir` builds the content array with data returned from standard *Perl* function `readdir`.

Method `getnextannotation` returns `undef` because plain files have no VCS attributes.

Method `getauthor` returns `undef` because plain files have no VCS attributes.

Method `filerev` returns a kind of “signature” made of the file size and its last modification time.

Method `getfilehandle` returns a real file handle to the source file.

Method `getfilesize` uses standard *Perl* function `-s`.

Method `getfiletime` does a `stat` on the source file.

Methods `isdir` and `isfile` use standard *Perl* function `-d` and `-f`.

Method `realfilename` is reimplemented to return the true source file name.

Method `releaserealfilename` is reimplemented to do nothing, so it does not destroy the source file.

Its private support methods are:

Routine name	Arguments	Description
toreal	File name Version	Returns the real OS file name associated with the arguments, <i>i.e.</i> <code>value('sourceroot')/Version/File name</code>

2.13.f. Subversion manager

Derived from *Files.pm* (and similar to *GIT.pm*), *Subversion.pm* implements all stub methods.

Method `getdir` builds the content array with data returned from command `svn list`.

Method `getannotations` returns an array containing annotations for all lines of the source file as retrieved from command `svn blame`. (*deprecated*)

Method `getnextannotation` pops the head element of array `@{'annotations'}`, eventually causing it to be refilled by `loadline`.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Method `getauthor` pops the head element of array `@{ 'authors' }`, eventually causing it to be refilled by `loadline`.

Method `filerev` returns the numeric part of the version argument if it is not head. In this latter case, command `svn info` is used to retrieve the revision.

Method `getfilehandle` returns a fake “file handle” to the output pipe of command `svn blame` which returns “decorated” lines (complete with annotation and author). If no annotations are required, the command is `svn cat`. This requires processing the lines to separate information. An *ad hoc* `getline` method is added to this fake file handle (the *Subversion* object itself) to transparently simulate a standard file handle behaviour.

Method `getfilesize` reads the result of command `svn list -v`.

Method `getfiletime` extracts the commit time through parsing the output of command `svn info`.

Methods `isdir` and `isfile` use command `svn info` to check the presence of directory or file respectively.

Its private support methods are:

Routine name	Arguments	Description
<code>loadline</code>		Strictly internal method to grasp the next annotated line and split it into annotation, author and line respectively queued into object arrays <code>{ 'annotations' }</code> , <code>{ 'authors' }</code> and <code>{ 'nextline' }</code> . On end of file, array <code>{ 'nextline' }</code> and the real file handle (to the <i>Mercurial</i> pipe) are deleted.
<code>getline</code>		Method used to make the <i>Mercurial</i> object a pseudo-file handle. The “next” line is taken from array <code>{ 'nextline' }</code> . End of file causes <code>undef</code> to be returned.
<code>revpath</code>	File name Internal revision	Converts the <i>LXR</i> file designation (name and version) into a <i>Subversion</i> location This involves selecting the appropriate subdirectory of the repository (such as <i>trunk/</i> , <i>branches/</i> , ...) and setting the revision number. The returned string is used as the file designation in <code>hg</code> commands.

`loadline` and the three object arrays simulate three independent files (namely annotation, author and source) from a single physical source.

Complimentary methods for version selection in configuration file:

Routine name	Arguments	Description
<code>allreleases</code>	File name	Returns a list of all tags valid for the designated file The list is built with command <code>svn log</code> on the file in the <i>trunk/</i> subdirectory.
<code>alltags</code>	File name	Returns a list of all tags valid for the designated file The list is built with command <code>svn list</code> on the <i>tags/</i> subdirectory.

Project LXR	The LXR Developer's Manual 2 LXR Engine	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
allbranches	File name	Returns a list of all branches valid for the designated file The list is built with command <code>svn list</code> on the <i>branches/</i> subdirectory.

2.14. Specialised database managers

The specific managers are stored in directory *Index/*. They are all derived classed of *Index.pm*.

2.14.a. MySQL

Method `new` connects to the *Perl* DBI module and overrides transactions `files_insert`, `symbols_insert` and `langtypes_insert`.

Nothing else is modified.

2.14.b. Oracle

*This manager has been blindly updated to parallel the organisation of other managers. It has not been tested because **Oracle** has a proprietary licence.*

Method `new` connects to the *Perl* DBI module and overrides transactions `files_insert`, `symbols_insert`, `langtypes_insert` and `purge_all`.

Nothing else is modified.

2.14.c. PostgreSQL

PostgreSQL has no auto-increment attribute for fields in tables. It offers however a sequence feature associated with a `nextval` function which can be called to provide a sequence-unique running counter. Transactions are added and others modified to use this feature.

Note;

Experiments showed that further performance improvement was possible adopting the same incrementation method as in *SQLite*. Consequently, the same variant was implemented.

Method `new` connects to the *Perl* DBI module, sets explicit commit mode¹⁸, creates transactions `filenum_nextval`, `symnum_nextval`, `typeid_nextval`, `reset_filenum`, `reset_symnum` and `reset_typenum` and overrides transactions `files_insert`, `symbols_insert`, `langtypes_insert`, `delete_definitions` and `delete_usages`.

¹⁸ Several parameters seem to determine global performance. However, setting commit mode looks like the most reasonable way to achieve good performance without resorting to writing sophisticated transactions. This results in a 10-times improvement over basic auto-commit mode.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	2 LXR Engine	Document revision 1.0

Other methods need to be overridden to make use of the new transactions: `fileid` (unique internal file identification), `symid` (unique internal symbol identification) and `decid` (unique language type declaration identification). `purgeall` is overridden to reset the counters. `commit` does nothing (to suppress the “auto commit” warning message). `final_cleanup` is updated to clean the new transactions.

2.14.d. SQLite

SQLite has no auto-increment attribute for fields in tables and does not offer any substitute feature. Consequently, unique numbering must be simulated through incrementation of an independent counter stored in a dedicated table (so that numbering sequence remains consistent across executions).

Method `new` connects to the *Perl* DBI module, sets explicit commit mode¹⁹, creates transactions `filenum_newval`, `symnum_newval` and `typenum_newval` and overrides transactions `files_insert`, `symbols_insert` and `langtypes_insert`.

Methods `fileid`, `symid` and `decid` are overridden to use the specific numbering transactions.

Since there is no `truncate` statement in the SQL, method `purgeall` must be reimplemented with new transactions `purge_definitions`, `purge_usages`, `purge_langtypes`, `purge_symbols`, `purge_releases`, `purge_status` and `purge_files` based on `delete` statements.

`final_cleanup` is updated to clean the new transactions.

¹⁹ This results in 40-times performance improvement factor!

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	3 Index Generator	Document revision 1.0

3 Index Generator

*Indexing a source-tree is done with script `genxref`. It shares as much code as possible with the **LXR** engine to remain consistent with it.*

3.1. Process outline

The script first reads the **LXR** configuration file. Its file name is retrieved from `Config.pm`.

Part 1

It then checks different software tools needed to use **LXR**. The *Perl* interpreter is a special case because the script already runs inside it. It is easy to test the version number to insure all syntactic constructs will be correctly understood. The other tools are checked with subroutine `check_tool`. The arguments are:

- configuration file parameter name containing the path to the executable, *e.g.* 'ectagsbin';
- tool name (without directory), *e.g.* `ctags`: used to search for the tool in the standard system locations;

If the configuration parameter does not exist, `check_tool` tries to find the tool in the system in order to proceed as much as possible with indexing. In case of success, this is called a “forced” tool situation.

- tool option to print version, *e.g.* `--version`;
- minimal version required if not '0'
- optional additional constraint on tool name

If present, this is a regular expression which must be satisfied by the tool name as returned by version printing (see third argument). This test is checked before version number.

Example:

Some Linux distributions still use plain *ctags* instead of *exuberant ctags*. This goes unnoticed, unless regular expression `qr/exuberant/i` or equivalent is passed as fifth argument. It results in a [FAILED] status if *Exuberant* is not present in the resulting line.

Status of the tool is printed on standard output and a numeric result is returned for decision making:

- 2 = version too low for a “forced” tool,
- 1 = version too low,
- 0 = tool not found,
- 1 = OK, “forced” tool,
- 2 = OK.

The tested tools are *ctags*, *glimpse* (and its buddy *glimpseindex*) and *swish-e*. Global consistency is

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	3 Index Generator	Document revision 1.0

checked according to the values returned by `check_tool`.

If *genxref* runs in checking mode (option `--checkonly`), it stops here. It also stops if fatal errors were encountered unless “recovery” attempts are accepted (option `--accept`). The recovery is limited to using the locations suggested for missing tools (as discovered by `check_tool`).

Part 2

Indexing the source-tree(s) begins here.

Processing option `--allurls` implies looping on all tree sections in configuration file (or equivalently on all elements of array `@config` since the global section was removed in part 1). On the contrary, option `--url=` means to process the option value without looping (this ensures that a typo will be reported as a missing source-tree). In order to merge both cases in a single block, array `@config` is changed to single-element list (1) in the `--url` case. It is not void then the loop is executed once and the element is not a reference to a tree section but this is irrelevant according to the initial statements in the loop.

At the head of the loop, a distinction is made between both cases to call `Config->new` with the correct parameters. Minimal checks are made, namely existence of parameters `'sourceroot'` and `{'variables'}{'v'}` before creating files and database objects. The running counter `$database_id` is incremented so that modules may detect a change of database.

The set of versions to index is then defined by checking options `--version` and `--allversions` (default if nothing specified) taking into consideration the *CVS* case.

Indexing for *CVS* and `--allversions` is special because *CVS* offers no centralised version list and every source file version will have to be collected and the final set can be dumped to build a static list.

Otherwise, the first task is to determine how to purge the database to prevent stale data from remaining in the tables. This also avoids useless database growth. Preference is given to total database erasure whenever possible (*i.e.* `--reindexall` on all existing versions) with `purge_all()`. For a one-version `--reindexall`, a flag is set for a purge. In a one-version incremental indexation, another flag is set for a careful examination of all tables.

An internal loop on the selected versions is started. The above flags select the purge method, either database management `purge(version name)` or support routine `cleanindex(version name)`. If enabled, the plain text search index is built with `gensearch(version name)`. Declarations are collected with `directorytreetraversal(...)` applied with `processfile`. Usages are collected with `directorytreetraversal(...)` applied with `processrefs`.

A call to database management `final_cleanup()` at the end of the outer loop resets everything for a new iteration.

3.2. Internal support routines

The following support routines are written inside file *genxref*:

Routine name	Arguments	Description
check_tool	Parameter Executable name Version option Minimal version Additional constraint	<i>Described in previous section</i>
dirbannerprint	Bullet string Version name Directory name File name	Prints the arguments to monitor progress across the source-tree To spare screen space, this line is overwritten if no files were processed in the previous directory.
directorytreetra versal	Processing function Version name Directory name "File" name	Recursive routine! If "file" is a directory, do a <code>directorytreetra</code> on this directory with adapted arguments. Commit changes to database before returning to caller (to "freeze" the new records from this directory. If "file" is really a file, the processing function is called. For a <i>CVS</i> repository in automatic version enumeration mode, configuration function <code>varrange('v')</code> is called and the processing function is iterated on every version. <i>The processing function must return undef in case it was unable to handle the file (e.g. no parser could be found) or 1 otherwise.</i>
gensearch	Version name	Driver routine to generate free-text search indexes. For <i>glimpse</i> engine, hand over the task to <code>glimpseindex</code> . For <i>swish-e</i> engine, open a pipe to <code>swishbin</code> and traverse the source-tree with <code>feedswish</code> .
feedswish	Current path Version name <i>swishbin</i> pipe File handle	If the current path names a directory, <code>feedswish</code> is iterated on every member. If the current path names a text file ²⁰ , it is sent to the pipe prefixed with an HTTP-like set of headers. <i>The fourth argument is a handle to a file where processed file names are written for later use by swish-e.</i>
dump_versionset	Prefix string Hash reference	Writes into <code>custom.d/</code> a file enumerating the <i>CVS</i> version set. The versions are the keys to the hash reference (a hash is used to obtain easily a list without duplicates). The file name is built with the prefix, virtual root and tree name. The last two items are URL-encoded to avoid possible issues with path separators.
cleanindex	Version name	Scans all files in this version to determine if it is up-to-date. The list is obtained from <code>getallfilesinit</code> . If the file is member of only one version of the source-tree (the present one), definitions and usages can be erased. In the other case, this information must be kept and the file is skipped. Erase is done by <code>purgefile</code> and the version descriptor can then be removed. Note: <i>Symbols are not erased, nor the file descriptor because it is</i>

²⁰ This is where `File::MMagic` is used. The criteria file name is taken from configuration parameter `'magicmime'` or `lib/magic.mime` if the parameter does not exist (backward compatibility with previous releases).

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	3 Index Generator	Document revision 1.0

Routine name	Arguments	Description
		<i>expected that the removal is caused by a newer version of the file which will be indexed again. This saves time to erase and recreate symbols and file descriptor for the most frequent case. File erasure, renaming or replacement is considered less frequent.</i>

Progress is monitored through printing various lines. The most obvious is the currently scanned file. But its name is directory-less to spare space and avoid as much as possible multi-line printing because it is easier to read information on a single line when it has a semi-organised lay-out. Directory name is printed by `dirbannerprint` when it is entered and repeated every so often.

To spare space again, a directory line is overwritten when none of its files are processed. This is controlled by variable `$printdirbanner`. Set to 1, it forces a full line while when undefined the cursor backs up a line and the last path segment is replaced by the new directory name. When a full line is printed, variable `$printdirbanner` is then forced to undefined value. Also, the counter `$repeatbannercountdown` is reset to its initial value `$repeatbannerevery`. The control variable `$printdirbanner` is set to a defined value after every successful file processing. This is necessary in order not to overwrite a file status line.

To cause repetition of the directory line, `$repeatbannercountdown` is decremented after every successful file processing. When its value is detected as zero or negative before file processing, variable `$printdirbanner` is forced to 1 and `dirbannerprint` is called.

Directory name printing is also needed when processing a file after exiting a subdirectory (this is an implicit directory change). In the directory loop (map function), variable `$needbanner` keeps track that a subdirectory was entered (which caused printing its name). When a file is encountered, if this variable is still defined, `dirbannerprint` is called and `$needbanner` is reset to undefined value.

3.3. External support routines

These routines (files) are located in the *scripts/* directory.

3.3.a. VTescape.pm

This file defines ANSI escape codes (also called VT100 codes very long ago) as a set of variables for static sequences or functions when dynamic parameters need be inserted. All names are prefixed with VT. Function names use the official acronym, e.g. VTCUU for *CUrsor Up CUU* except when there is none: VTprRM *private Reset Mode*, VTprRSM *private ReStore Mode*, VTprSM *private Set Mode*, VTprSVM *private SaVe Mode*, VTSSR *Set Scrolling Region*.

This simplifies message editing on the Linux console.



CAUTION!

Not all ANSI codes are implemented.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	3 Index Generator	Document revision 1.0

3.3.b. *Tagger.pm*

This file contains the processing functions passed as references in the first argument to recursive function `directorytreetraversal`. Their returned value is `undef` if something prevents processing a file (no parser, unknown revision, ...). It is 1 if a file has been successfully processed²¹.

The main role of these functions is to call the parser-specific processing method.

Changes to the database are committed after processing.

Routine name	Arguments	Description
<code>processfile</code>	File name Version name Configuration object Files object Database object	Records in the database definitions obtained from parser method <code>indexfile</code>
<code>processrefs</code>	File name Version name Configuration object Files object Database object	Records in the database symbol usages obtained from parser method <code>referencefile</code>

Presently, the configuration object argument is not used but it is offered in case the functions would need to access configuration parameters.

Performance issues:

The definition collecting pass (`processfile`) is currently implemented in *Generic.pm* with *exuberant ctags*. This utility provides compiled parsers for most of the supported languages. The exceptions are *Ant*, *COBOL*, *HTML*, *Matlab*, *S-lang* and overridden or added languages (*SQL* and *Virtual BASIC*) which are parsed with regular expressions.

The reference collecting pass (`processrefs`) ends up in the same parsers as those used for file highlighting. They are derived from *Generic.pm* and reference *SimpleParse.pm* which is based on regular expressions. The speed can be considered acceptable when file display is requested, though some delay may be seen on long files. But it clearly results in low performance when huge trees are scanned (*e.g.* the *Linux* kernel). It could be improved only through the implementation of compiled automaton-based parsers similar to *exuberant ctags*.

3.3.c. Multi-threaded attempt

In the hope to improve performance, an experimental development implemented *multi-threading* for both definition and reference collecting passes. This required dispatching the work through a queue

²¹ To be honest, it rather means *something has been printed on screen* and the fact that the file has already been processed is not considered an error (it adds to an existing message only).

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	3 Index Generator	Document revision 1.0

to subordinate workers. To avoid database inconsistency, workers locked the database while they were batch-writing to it and committing changes was necessary after each file processing.

It was quite successful in parsing simultaneously several files (temporarily storing the symbols in an array) but database locking resulting in serialisation of access. Moreover, the commit high frequency finally gave a poorer global performance than single-thread processing.

This optimisation track was given up.

4 Database Architecture

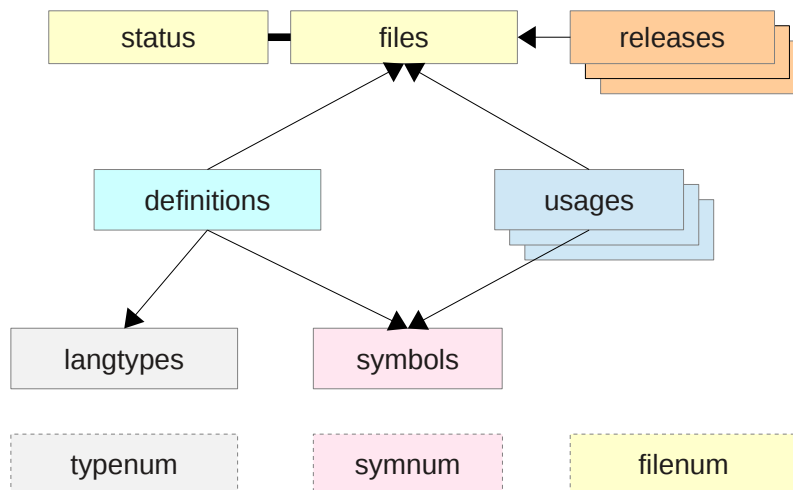
For performance reason, cross-reference data is kept in a database instead of a set of files²². As long as the database engine is SQL-compliant, only minor adjustments are needed to cope with SQL dialect differences.

4.1. Tables

Tables store the various entities relevant in cross-references. A *symbol* is *declared* somewhere in a *file* and is *used* in *files*. A *file* comes in different flavours: it is part of *releases* (or more commonly version) corresponding to different *base revisions*. When several *releases* map to a single *base revision*, reindexing is avoided when another *release* in the set is encountered. A *release* is a user-visible name for a file version while the *base revision* is an internal unique name given to this file state.

The base revision name may or may not be related to the version name, depending on the repository type.

Relations between tables are summarised in the following figure:



Drawing 4: Relationship between DB tables

status and *files* tables logically make up a single table. *status* has been isolated because it contains only a small amount of mutable data while *files* contains (long) text immutable data. When file name is not relevant to the current task (which is the most frequent case), it is thus more efficient and memory-friendly to load only *status* data.

²² The initial LXR releases used files but that did not scale very well: indexing big projects resulted in huge files; accessing data meant reading the whole file into memory; selective random access was not easy and not optimised for large reference data. All this ended up in poor performance when project size grew.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	4 Database Architecture	Document revision 1.0

langtypes provides human-readable text for a declaration type.

filenum, *symnum* and *typenum* are optional tables depending on the selected strategy for uniquely numbering files, symbols and language types respectively.

A table is made of fields to store project information. In some tables, database functions may be associated with fields to automate tasks. Fields and functions are described in the following sections.

Special sequence-simulating tables may be added in some database engines.

4.1.a. *files* and *status* tables

These two tables describe a unique base revision file. No other file in the source-tree has an identical content.

files is the master table:

<code>fileid</code>	internal unique number identifying this file (primary key)
<code>filename</code>	file path in the source-tree, maximum 255 characters ²³
<code>revision</code>	unique revision string (provided by repository support method <code>filerrev</code>), maximum 255 characters ²⁴

Combination of `filename` and `revision` form a secondary key with *unique* attribute.

status is the associated table:

<code>fileid</code>	internal unique number identifying a file (primary key), identical to <i>files</i> value
<code>relcount</code>	number of <i>releases</i> associated with this revision
<code>indextime</code>	time of last indexation on this file revision (allows to detect stale reference data when displaying a file)
<code>status</code>	only two bits of this integer are of interest: bit 0 (value 0/1) for “file has been scanned for declarations” and bit 1 (value 0/2) for “file has been scanned for usages”

An automatic function `remove_file` is triggered by deletion of a *status* record. It causes deletion of the *files* record with same `fileid` key.

4.1.b. *releases* table

This table enumerates the different user-visible version names mapping to a base revision file.

<code>fileid</code>	internal unique number identifying a file in the <i>files</i> table
<code>releaseid</code>	version name, maximum 255 characters

Combination of `fileid` and `releaseid` form a primary key. `fileid` creates the link with the *files* table.

²³ This may need to be increased in some deeply nested projects with long subdirectory names. Unfortunately, there is no truncation warning during indexation.

²⁴ This should not be a problem since revisions tend to be rather short.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	4 Database Architecture	Document revision 1.0

Two automatic functions maintain consistency of the `relcount` counter in the `status` table.

`add_release` is called after insertion of a `releases` record to increment the `relcount` counter.

`remove_release` is called after deletion of a `releases` record to decrement the `relcount` counter.

4.1.c. *langtypes* table

This table records text for a declaration type in a given language.

<code>typeid</code>	internal unique number identifying this type (primary key)
<code>langid</code>	language identifier (usually taken from <code>generic.conf</code> parameter 'langid')
<code>declaration</code>	free descriptive text, maximum 255 characters

Combination of `typeid` and `langid` form the primary key.

4.1.d. *symbols* table

This table records all unique symbols names.

<code>symid</code>	internal unique number identifying this name (primary key)
<code>symcount</code>	counter for <code>definitions</code> and <code>usages</code> records referencing this name (protection against in-use name purge)
<code>symname</code>	symbol name, maximum 255 characters

Function `decsym` decrements the `symcount` counter. It is used in the automatic functions associated with `definitions` and `usages` tables.

4.1.e. *definitions* table

This table is the list of variable, function or otherwise interesting entity declarations.

<code>symid</code>	numeric identifier for the name
<code>fileid</code>	numeric identifier for the file where the declaration appears
<code>line</code>	line number of the declaration
<code>typeid</code>	numeric identifier for the type
<code>langid</code>	numeric identifier for the language
<code>relid</code>	optional numeric identifier for an outer symbol name

Two indexes are built to speed up access, one on `symid`, the other one on a combination of `typeid` and `langid`.

Links to other tables are created through the following fields: `symid` and `relid` to the `symbols` table, `fileid` to the `files` table, combination of `typeid` and `langid` to the `langtypes` tables.

Field `relid` is used when the current declaration is in “relation” with another one. Think for example of the sub declarations of a `struct` record in C.

An automatic function `remove_definition` is triggered after a definition deletion to call function `decsym` which maintains reference counter consistency in `symbols` table.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	4 Database Architecture	Document revision 1.0

4.1.f. usages table

This table is the list of variable, function or otherwise interesting entity usages.

symid	numeric identifier for the name
fileid	numeric identifier for the file where the declaration appears
line	line number of the declaration

An index on symid is built to speed up access.

Links to other tables are created through the following fields: symid to the *symbols* table, fileid to the *files* tables.

An automatic function remove_usage is triggered after an usage deletion to call function decsym which maintains reference counter consistency in *symbols* table.

4.1.g. Unique numbering tables

These tables are present with *SQLite* and with other engines if user management is chosen for unique numbering of files, symbols and languages types. They do not appear if unique numbering relies on internal algorithms.

Note:

Experiments have shown a performance boost with the user management method. This improvement comes from the decrease of commits to the database. It is possible because the only *genxref* needs write access and multi-threading has been dropped.



CAUTION!

If any of these requirements are violated, database integrity is lost. Never attempt to refresh the same tree cross-references from two sessions, it results in garbage. There should be only one *LXR* administrator responsible for *LXR* server maintenance.

The tables contain a single record:

rcd	record number (equal to 0)
xid	running counter

xid is fid, sid and tid for the *filenum*, *symnum* and *typenum* tables respectively. These counters are cached in memory, incremented when a new associated record is created and saved to the database when the session terminates.

4.2. Queries

The elementary queries defined in *Index.pm* will not be commented. A query is considered *elementary* if it involves a single table. In this context, *files* and *status* are parts of a single logical table.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	4 Database Architecture	Document revision 1.0

Note:

The descriptions below refer to *arguments*. These *positional anonymous* arguments are part of the *Perl* DBI calling interface. In case query source lines are modified, take great care to check that the order in which arguments are evaluated did not change vis-a-vis the argument list.

`allfiles_select` is used to retrieve the files which are members of a version (passed as argument to the query). The version argument filters the *releases* records on field `releaseid`. Field `fileid` indirectly selects the *files* records on the corresponding key field.

`related_symbols_select` is used to retrieve the “outer” declaration symbol before deleting a file to correctly maintain reference count integrity. The argument is the internal identifier of the to-be-deleted file. The file identifier argument filters the *definitions* records on field `fileid`. Field `relid` indirectly selects the *symbols* records on key field `symid`. Note that a `relid` equal to 0 means “no relation to an outer definition” and that no symbol has a `symid` key equal to 0.

`definitions_select` is used to find definitions for an identifier (first argument) in a version (second argument). First argument filters the *symbols* records on field `symname` which gives the key value `symid`. Second argument filters the *files* records on field `releaseid` which gives the key value `fileid`. Finally, both keys filter the *definitions* records on fields `symid` and `fileid`.

`delete_definitions` is used to delete all *definitions* records pertaining to a version (database purge with `--reindexall` option). The set of records to delete is computed by a nested selection query. The version argument filters the *releases* records on field `releaseid`. Field `fileid` indirectly filters the *status* records on the corresponding key field with the condition that the release counter `relcount` is equal to 1 (no other version references the base revision, it is thus safe to delete data). The *definitions* records are then selected on field `fileid`.

`usages_select` is used to find usages for an identifier (first argument) in a version (second argument). First argument filters the *symbols* records on field `symname` which gives the key value `symid`. Second argument filters the *files* records on field `releaseid` which gives the key value `fileid`. Finally, both keys filter the *usages* records on fields `symid` and `fileid`.

`delete_usages` is used to delete all *usages* records pertaining to a version (database purge with `--reindexall` option). The set of records to delete is computed by a nested selection query. The version argument filters the *releases* records on field `releaseid`. Field `fileid` indirectly filters the *status* records on the corresponding key field with the condition that the release counter `relcount` is equal to 1 (no other version references the base revision, it is thus safe to delete data). The *usages* records are then selected on field `fileid`.

4.3. Database engine specifics

Differences in underlying concepts and *SQL* implementation between database engines lead to adjustments of architecture and queries.

The most divergent area is the auto-numbering of records. This implied the inability to define a

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	4 Database Architecture	Document revision 1.0

common query to insert *files*, *symbols* and *langtypes* records into the database. These are the only operations involving such controlled value assignment to fields.

Thanks to *LXR* limitations (it is a read mostly process), auto-numbering was fully simulated in memory with the advantage of drastically decreasing the number of commits. However, it is possible to revert to built-in mechanisms by uncommenting the relevant lines in the database-specific interfaces and commenting out the common simulation lines which was initially developed for *SQLite*.

4.3.a. MySQL

`files_insert`, `symbols_insert` and `lang_types` rely on auto-increment feature.

Foreign key constraints remain active in `truncate table` statements, thus preventing from using the common `purgeall()` method. The method is overridden to temporarily disable foreign key checks while erasing the database.

For performance reason, table descriptions force engine *MyISAM*.

4.3.b. PostgreSQL

Auxiliary table indexes must be explicitly created with `create index` statements.

PostgreSQL has no auto-increment feature but it offers sequence objects from which a running value can be retrieved through function `nextval`. Three “sequences” are created in the database description: `filenum`, `symnum` and `typenum` for numbering *files*, *symbols* and *langtypes* records respectively. In parallel, new queries are defined for retrieving the current number (`filenum_nextval`, `symnum_nextval` and `typeid_nextval`) and resetting the numbers after a purge (`reset_filenum`, `reset_symnum` and `reset_typenum`).

Queries `files_insert`, `symbols_insert` and `langtypes_insert` are adapted to this mechanism.

Queries `delete_definitions` and `delete_usages` must be rewritten due to a difference in syntax (no nested select allowed).

Trigger functions need also some adjustments: Functions `incr1`, `decr1` and `decsym` are used by trigger functions to increment/decrement database counters.

Methods involving auto-numbering (`fileid`, `symid` and `decid`) are overridden to use the appropriate queries.

4.3.c. SQLite

Auxiliary table indexes must be explicitly created with `create index` statements.

SQLite has neither auto-increment feature nor `truncate` statement. The latter issue is solved by substituting a set of `purge_XXX` (with `XXX` equal to a table name) to `purge_all`. These queries use a `delete` statement without record target, which completely erases the table. The solution to the

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	4 Database Architecture	Document revision 1.0

former issue is simulation of auto-increment.

Three new tables are added to the database: *filenum*, *symnum* and *typenum* which will remember the highest number reached for files, symbols and declaration types during the last execution. These tables contain a single record which is cached in memory while connected to the database. Queries *filenum_lastval*, *symnum_lastval* and *typenum_lastval* are used only during initialisation to retrieve the previous values into the cache. Queries *filenum_newtval*, *symnum_newval* and *typenum_newval* are used only during disconnection to store the current values into the database.

Queries *files_insert*, *symbols_insert* and *langtypes_insert* are adapted to this mechanism.

Methods involving auto-numbering (*fileid*, *symid* and *decid*) use the internal caches to generate numbers.

4.3.d. Oracle

CAUTION!

Since **Oracle** is released under a proprietary licence, the implementation has not been tested on a real case. It is a “best guess” based on publicly available documentation, comparison with other databases and evolution of “historical” code. Global changes are translated in a generic way.

*Please report your experience if you ever install **LXR** under **Oracle**.*

Auxiliary table indexes must be explicitly created with `create index` statements.

Oracle has no auto-increment feature but it offers sequence objects from which a running value can be retrieved through special variable *nextval*. Three “sequences” are created in the database description: *filenum*, *symnum* and *typenum* for numbering *files*, *symbols* and *langtypes* records respectively. Queries *files_insert*, *symbols_insert* and *langtypes_insert* are adapted to this mechanism.

It is very likely that this implementation is incomplete. Some common methods need probably to be customised.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	5 LXR Main Scripts	Document revision 1.0

5 LXR Main Scripts

The so-called “main” scripts are the driver scripts `diff`, `ident`, `search`, `showconfig` and `source` which are the entry gates into **LXR** realm.

5.1. source script

This is the most used script. It implements source display.

It initialises the **LXR** engine through a call to `httpinit`.

An undocumented filter is applied on the file name to discard unwanted files.

Note:

This undocumented feature has been present for ages. It is associated with configuration parameter `'filter'` whose final value is a regular expression describing which file (or directory) names should be kept. *Final value* means it can be a literal regular expression or a sub returning such a regular expression.

Since it is quite difficult to set it right (because it is ALWAYS applied to the raw path taken from the URL – both to directories AND files), do not use it. Configuring parameters 'ignorefiles' and 'ignoredirs' (or their respective regular expression counterparts 'filterfiles' and 'filterdirs') lead to the same effect.

If URL parameter `_raw` is defined, `printfile(1)` is called to display the file “as is”; otherwise, header and footer are built around `printfile(0)`.

Finally, the **LXR** engine is cleared through a call to `httpclean`.

Dedicated support routines are:

Routine name	Arguments	Description
<code>iconlink</code>	HTML element Path name	Wrapper function around <code>fileref</code> to allow insertion of a real <i>HTML</i> element (which would otherwise be disabled in <code>fileref</code> as protection against XSS). Note that the first argument is passed without <code>< ></code> delimiters (internally added). <i>Used by <code>diricon</code> and <code>fileicon</code> only.</i>
<code>diricon</code>	Template string Directory name Parent directory	Expands to an <code><A></code> link around an <code></code> element for the directory icon
<code>dirname</code>	Template string Directory name Parent directory	Expands to an <code><A></code> link around the name of the subdirectory
<code>fileicon</code>	Template string File name	Expands to an <code><A></code> link around an <code></code> element for the file icon

Project LXR	The LXR Developer's Manual 5 LXR Main Scripts	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
	Parent directory	
filename	Template string File name Parent directory	Expands to an <A> link around the name of the file
filesize	Template string File name Parent directory	This template function expands its argument, substituting markers \$bytes, \$kbytes or \$mbytes with the file size Note: <i>The present implementation makes no distinction between the preceding markers; the file size is internally scaled based on its string length.</i>
_edittime	UTC time in seconds	Returns a human readable date/time string or a single dash if the argument is undefined
modtime	Template string File name Parent directory	Expands to the last modification time of the file
indextime	Template string File name Parent directory	Expands to the last indexation time of the file
descexpand	Template string Node name Parent directory Version number	This template function expands its argument, substituting marker \$desctext with a description for the node (a file or directory) The called function is either filedesc or dirdesc depending on the nature of the node. These functions are located in <i>Local.pm</i> . <i>They should return at least a non-breaking space to force the browser to keep the element when laying out the page.</i>
rowclass	Template string Line number	Expands to a CSS class name for this line in the directory listing
direxand	Template string Directory name	This template function expands its argument to create a directory listing, calling the editing functions for the various markers, making a difference between a file and a subdirectory <i>It takes care to force head version for a CVS repository since CVS does not manage directory version.</i>
printdir	Directory name	Driver routine for directory listing It retrieves template 'htmlmdir' and expands it, associating markers \$description and \$files to functions dirdesc and direxand respectively.
next_annot	Version number Previous version Background flag	Returns a decorated HTML block containing revision and author information <i>Version number</i> is the requested revision; lines belonging to this revision have their annotation specifically highlighted. <i>Previous version</i> is the revision for the previous line. If the previous and current lines share the same revision, annotation is suppressed in order not to clutter screen (however, the background has the correct colour). <i>Background flag</i> toggles between 0 and 1 between consecutive change sets to use different highlighting colours.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0		5 LXR Main Scripts

Routine name	Arguments	Description
		<i>The last two arguments are references to caller variables so that their values survive across calls.</i>
printfile	Raw flag	The node to display is defined by global variable \$pathname and version by \$releaseid Dispatches to printdir for a directory. If raw flag is 1, the requested file is sent "as is" to the browser. Otherwise, a file handle for the requested version is retrieved from the repository manager and passed to markupfile for highlighting. An output function is built to merge annotations with lines.

5.2. ident script

This is the second most used script. It implements identifier search.

It initialises the *LXR* engine through a call to httpinit.

The value of \$defonly (display only definitions) is determined from URL variables or configuration parameter 'identdefonly'.

Header and footer are built around printident.

Finally, the *LXR* engine is cleared through a call to httpclean.

Dedicated support routines are:

Routine name	Arguments	Description
varinputs		Expands to a sequence of <INPUT> element describing the current value of all 'variables' <i>This is part of the state preserving feature between invocations.</i>
countfiles	Search result array	Expands to the number of files in the result
checkvalidref	File name	Expands to CSS class name identinvalid if the file has been changed since last indexation Side effect: increments global variable \$bad_refs
expandwarning	Template string	Expands its templates argument if global variable \$bad_refs is non zero; returns an empty string otherwise
ref_in_file	Description CSS class File path Line number	Front-end to fileref (same arguments) Line numbers are set negative on a case-approximative match. Line numbers are then corrected and CSS class name is augmented with identapprox. For exact matches (positive line numbers), arguments are simply forwarded to fileref.
refsexpand	Template string Search result array	This template function has two expansion variants for its argument: several references per line if marker \$lines is present, single reference per line otherwise

Routine name	Arguments	Description
		Every reference is inserted in the expanded template argument by an appropriate loop.
<code>cmprefs</code>	Two array arguments	Comparison function for sorting search results The arguments are arrays. The elements are taken in order: 0 file name, 1 line number and for definitions 2 type, 3 higher level definition. <i>The last comparison may not be human-meaningful since the data is an index into the database.</i>
<code>defsexpands</code>	Template string	Expands its template argument for every definition of the identifier defined by global variables <code>\$identifier</code> and <code>\$releaseid</code> Definitions are retrieved both against “native” case and uppercase versions of the identifier. The lists are merged, removing duplicates, and the remaining occurrences are sorted with <code>cmprefs</code> . Template expansion for marker <code>\$refs</code> is done by <code>refsexpand</code> .
<code>usesexpands</code>	Template string	Expands its template argument for every usage of the identifier defined by global variables <code>\$identifier</code> and <code>\$releaseid</code> Usages are retrieved both against “native” case and uppercase versions of the identifier. The lists are merged, removing duplicates, and the remaining occurrences are sorted with <code>cmprefs</code> . Template expansion for marker <code>\$refs</code> is done by <code>refsexpand</code> .
<code>printident</code>		Retrieves the 'htmlident' template and expands it with the previous support routines

5.3. *diff script*

It implements difference display between two versions of the same file. It relies on the availability of *rcs diff*.

Since two version references are needed, two passes through this script are necessary. The first pass will transfer the current 'variables' values into “remembered value” arguments and request the second variant from the user. The second pass can then do the job with “current” and “remembered” designations.

It initialises the *LXR* engine through a call to `httpinit`.

The “difference arguments” `@dargs` are built from URL query arguments of the form `~var_name` capturing the “remembered” (from first pass) value of the corresponding 'variables'.

Header and footer are built around `printdiff`.

Finally, the *LXR* engine is cleared through a call to `httpclean`.

Dedicated support routines are:

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0		5 LXR Main Scripts

Routine name	Arguments	Description
fflush		Sets STDOUT in autoflush mode <i>Candidate for deletion</i>
htmljust	HTML string Maximum width	Returns a justified <i>HTML</i> string occupying exactly the requested width To guarantee correctly balanced <i>HTML</i> , tags are copied blindly to the output string (without checking for matching opening/closing tags) considering they need no screen position. <i>HTML</i> entity references, supposed to be one screen position wide, and ordinary text are copied only if there is room for them.
prindiff	Difference arguments array	Since standard URL (in other scripts) designate only one version, two passes are necessary to grab versions to compare. On first pass (where the difference array is undefined), user is requested to name a second version. The current ' <i>variables</i> ' values are implicitly transferred into <i>remembered</i> values by the variables link generator (see <i>varlinks</i> in <i>Template.pm</i>). These <i>remembered</i> values will be put into the difference array on re-entry. On second pass, both versions are described by the variables value sets but only the file name in <i>\$pathname</i> points to an adequate file. To get the second one, ' <i>maps</i> ' rules must be inverted (in the remembered environment) and reapplied (in the current environment). Then, patch directives can be computed by <i>diff</i> (on real files). Highlighted sources can be displayed side-by-side under control of these patch directives.

5.4. search script

It implements free-text search. It relies on the presence of a free-text search engine, presently either *glimpse* or *swish-e*.

It initialises the *LXR* engine through a call to `httpinit`.

Header and footer are built around search after having checked that free-text search is allowed.

Finally, the *LXR* engine is cleared through a call to `httpclean`.

Dedicated support routines are:

Routine name	Arguments	Description
varinputs	Template string	Expands its template argument to a sequence of <code><INPUT></code> element describing the current value of all ' <i>variables</i> ' <i>This is part of the state preserving feature between invocations.</i> Note: implementation is different from <i>ident</i> and some decision should be made in favour of one or the other for consistency sake.
filename_matches	String Pattern flag	Returns 1 if the string is part of the file name The string is a regular expression if pattern flag is non-zero, an

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0		Document revision 1.0

5 LXR Main Scripts

Routine name	Arguments	Description
	Case-sensitive flag Filename to check	ordinary string otherwise. Case-sensitive flag, if non zero, will cause case-sensitive comparison. Otherwise, comparison is case-insensitive. <i>This function is useful to discard results from files not matching the name criterion.</i>
<code>glimpsearch</code>	String to find Filename filter Filename pattern flag Case-sensitive flag	Launches <i>glimpse</i> to find the string within the current version of the source-tree and returns an array of the resulting hits Search if normally made case-sensitively (flag equal to 0), but can be made case-insensitively (flag non-zero). Results can be restricted to specific files with a non-empty file name filter. This filter is either an ordinary string (flag equal to 0) or a regular expression (flag non-zero).
<code>swishsearch</code>	String to find Filename filter Filename pattern flag Case-sensitive flag	Launches <i>swish-e</i> to find the string within the current version of the source-tree and returns an array of the resulting hits Search if normally made case-sensitively (flag equal to 0), but can be made case-insensitively (flag non-zero). Results can be restricted to specific files with a non-empty file name filter. This filter is either an ordinary string (flag equal to 0) or a regular expression (flag non-zero).
<code>checkvalidref</code>	File name	Expands to CSS class name <code>searchinvalid</code> if the file containing a search hit has been modified since last indexation or was never indexed
<code>printresults</code>	Template string Search text Result array	Expands its template argument with the results of the search Since the two supported search engine return different data, processing is adapted to the search engine (expanding appropriately the template).
<code>search</code>		Main driver for free-text search It retrieves the template and the URL query parameters. It takes care search variants: search text with or without file filter to be handled by <code>glimpsearch</code> or <code>swishsearch</code> , or file filter only handled by scanning a private search engine file containing the names of the scanned files. Results are edited through template expansion.

5.5. *showconfig* script

It allows to check the configuration file from the browser.

It initialises the *LXR* engine through a call to `httpinit`.

The parameter-group sections from the configuration file are read into array `@pgs` and the requested parameter group number is determined from URL query argument `_parmgroup` or, if not given, from configuration parameter `'parmgroupr'` or defaults to 1.

URL query argument `_confall` controls the amount of information displayed:

Project LXR	The LXR Developer's Manual 5 LXR Main Scripts	Language en_UK
Software release 2.0		Document revision 1.0

- 0 or not specified: only parameters present in the current tree-specific and global sections
- 1: all parameters ever used in any tree-specific and global sections
- 2: like 1 with the addition of parameters in the configuration object

Note:

This last argument value is not disclosed in the *User's Manual* since it is intended for developer's use, allowing to dump value of derived or internally generated parameters.

Template 'htmlconfig' is retrieved. Header and footer are built around the template expansion.

Finally, the *LXR* engine is cleared through a call to `httpclean`.

Dedicated support routines are:

Routine name	Arguments	Description
dumphash	Reference to a <i>hash</i> Left indent	This recursive function returns a ready-to-display string representing the <i>hash</i> key/value pairs
parmvalue	Parameter name Reference to a parameter group <i>hash</i> Reference to general configuration object	Dumps a parameter value if it exists in the parameter group Actual editing depends on the parameter type (<i>hash</i> , array or other) Third parameter is present only for the special developer view.
parmexpand	Template string Script name Reference to the parameter group array Parameter group number	Expands its template argument for every "authorised" parameter A parameter is "authorised" when it is present in the designated group. All parameters (in any group) are "authorised" when URL query argument <code>_forceall</code> is non-zero. "Internal" parameters 'confpath' and 'parmgroupnr' are always skipped.
parmgrouplink	Parameter group number Reference to parameter group array	Expands to an <A> link to have <i>showconfig</i> display the parameters in the designated group

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

6 Configuration Wizard

Configuring LXR, i.e. creating one's own lxr.conf file, has always been a tedious task owing to the large number of configuration parameters and their sometimes obscure relationships. It is also very easy to forget an important parameter. To circumvent these difficulties, a configuration wizard is available. It handles all configuration situations, from the simplest to the most elaborate.

The configuration wizard, written in *Perl*, is stored in the *scripts/* directory. Its components are:

- *configure-lxr.pl*: main configuration driver;
- *ContextMgr.pm*: context file utilities;
- *LCLInterpreter.pm*: interpreter for the *LXR Configuration Language* (LCL) macro statements;
- *QuestionAnswer.pm*: user interaction manager;
- *VTescape.pm*: ANSI escape codes definitions (see 3.3.a VTescape.pm).

6.1. Process Outline

Once initialised, the wizard is driven by macro statements found in a configuration file template (stored in the *templates/* directory). These macro statements are interpreted, some requesting input from the user, and results may be inserted in the output configuration file.

First, it checks its command line arguments and the LXR root directory environment.

It then determines the general context of this configuration. If this is an initial run (no `--add` option), the user is asked for his choice of single/multiple trees preference, web-server and database engine. These choices are saved in a *context* file (extension *.ctxt*) for an eventual later session. If this is an addition session (option `--add` given), the previous choices are retrieved from the *context* file to guarantee consistency.

A dictionary of symbols corresponding to options and context parameters is built for use by the LCL statements.

Eventually, free-text search is disabled if the search engines cannot be found.

The effective configuration begins here.

During the initial session, web-server and SCM configuration or auxiliary files are customised and copied into the *custom.d/* directory through `copy_and_configure_template` procedure.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0



CAUTION!

Subdirectory and file names are hard-coded. If any modification is done in the *templates/* directory, it must be forwarded in the configuration wizard.

The main configuration file *lxr.conf* is built in two passes.

The **first pass** exists only during the initial session: it builds the global section part through a call to procedure `copy_and_configure_template`.

The **second pass** deals with the source-trees (one iteration per tree). It starts by reinitialising a few symbols in the dictionary to make each iteration independent. The tree section part is built through a call to procedure `pass2_hash`. On return, the tree-specific parameters have been recorded in the symbol dictionary. They can be used to customise the web-server configurations (*Apache*, *lighttpd* and *nginx* need this) with a call to `pass2_hash`. The database description is also recorded in the symbol dictionary. *Shell* and *SQL* statements for creating the tree database are added to the output script with a call to `expand_slash_star` under control of an *initdb-x-template.sql* template.

After all trees are configured, the wizard ensures that all known scripts are executable.

6.2. Support library

VTescape.pm has already been described in 3.3.a.

6.2.a. ContextMgr.pm

This package manages the so-called *context* in which **LXR** is executed. The context contains “constant” parameters, valid in any tree, describing the general behaviour of **LXR**. They are:

- single-tree/multiple-trees operation flag,
- URL components (host name, port and aliases),
- location of **LXR** service within the server document hierarchy,
- position of tree designation in URL (multiple-trees operation),
- virtual root policy (and eventually common virtual root),
- database engine and policy (single universal database or dedicated databases),
- global database parameters if any (user name, password, table prefix, flags, ...).

The context is implemented by a set of exported package-global variables. When it is necessary to remember this context, the set is saved into a *Perl* source file as a sequence of assignments to the global variables so that an `eval` of this file restores everything. The set is augmented with a variable containing the context version number in order to detect incompatibilities.

CAUTION!

The detailed definition of the context can change without notice. It is considered internal to the configuration wizard and is kept in a file only to allow to split the configuration process into multiple sessions. It is not intended to contain user-visible data. What is in the context is not meant either to be long term information: though every reasonable effort is made to maintain upward compatibility, context reloaded by a newer version of the configuration wizard is likely to be rejected.



The context support routines are:

Routine name	Arguments	Description
contextReload	Verbose flag Context filename	Reads the context file and <code>evals</code> its content. If no format version check variable is found, the file is probably not a context file and processing is aborted. If recorded and expected format versions are beyond compatibility tolerance, processing is aborted. Warning! <i>The name of the output configuration file is not checked against the recorded name. It is anticipated that the average user will use the default names and that the power user will care not to mess its files.</i> With verbose flag non zero, the decoded reloaded context is printed on screen. Returned value is 0 if context reloading succeeded or 1 if manual context restore may be tentatively attempted.
contextSave	Context filename Output configuration filename	Writes context into the context file Warning! <i>The initial comment line containing the output filename is parsed in contextReload. If this comment is changed, the regular expression in contextReload must be kept in sync.</i>
contextTrees		Asks the user for the operational mode: single or multiple trees
contextDB	Verbose flag	Asks the user for his choice of database and policy
contextServer	Verbose flag	Asks the user his URL structure (host name and aliases, document hierarchy, tree designation policy, ...)

6.2.b. LCLInterpreter.pm

This package contains a parser and interpreter for **LXR** configuration-language **LCL** (see 6.3). LCL is embedded in comments so that it does not interfere²⁵ with file data. Moreover, only specialised wrapper routines are exported; they can be thought of as instantiations of two base routines depending on the lexical appearance of comments. Two are meant for pass 1, `expand_hash` and `expand_slash_star`, two for pass 2, `pass2_hash` and `pass2_slash_star`. They look for LCL statements in comments started by a hash # (up to the end of line) or delimited by `/*` and `*/` respectively.

²⁵ Really? That could probably be the case when LCL only substituted parameters for their values. But now, the macro language offers a selection feature between exclusive alternatives. These alternatives cannot be kept simultaneously in the output file since they have conflicting meanings.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

The core routine is `expand`. It is responsible for parsing file, locating LCL statements and translating their semantics. The routine handles nested constructs by indirect recursive calls. The arguments are:

- `$source`: reference to a sub returning the next input line
A sub allows very versatile input. The most common case references a file handle (with a little loss in efficiency due to the call overhead). But it is very easy to handle iterative constructs where the statement block is put aside in an array when encountered and the sub returns each line when needed. More complex constructs store their sub-blocks in arrays and the adequate sub-block is afterwards selected according to context.
- `$dest`: output file handle (where transformed input is written)
- `$markers`: reference to symbol dictionary (stored in a *Perl hash*)
- `$verbose`: verbose flag (0 = silent, 1 = standard verbosity, 2 = detailed verbosity)
- `$comstart`: a **regular expression** defining the opening delimiter of a comment
- `$comend`: a **regular expression** defining the closing delimiter of a comment or an empty string if comment is limited by the end of line

Note:

Even if in simple cases the previous two delimiters may seem to reduce to strings, they are regular expressions, which means characters with special semantics must be quoted. For instance, *SQL* comment delimiters must be passed as `'/*'`, `"/**"` or `qr(/**)` (and the like for the closing delimiter).

- `$end_label`: a **regular expression** describing an LCL statement where expansion stops
This regular expression is internally prefixed with the comment opening delimiter and the LCL sentinel `@`.

Notes:

The note about simple comment delimiters applies also here.

If interpretation of the whole input file (or stored block) is requested, use an improbable string such as `'~~~TO~EOF~~~'`. Submitting an empty string would cause stopping interpretation on the first LCL statement.

CAUTION!

When this regular expression is applied, LCL statement parsing has not yet taken place. It means the match target of the regular expression must be found in the initial line of the statement and cannot be seen if it is located in continuation lines. Be careful when you develop your template configuration files.



`expand` loops through its input (calling `&$source()` to get the next line) until input is exhausted. Its return value is `undef`, though internal recursive calls to `expand` rely on another value. The loop

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

is described in the following paragraphs.

It checks for the terminating LCL statement (the associated comment MUST start at the beginning of the line fragment²⁶. In case of a match, it returns the value of `parse_statement`.

“Erasable” comments are removed. Erasable comments use delimiters extended with a dash (-), *e.g.* #- and (unmodified) end of line or /* - and - */. In the latter case, the erasable commented block may even contain ordinary comments²⁷.

Note:

The GPL licence is written inside erasable comments because the templates are open-source. However, the output file, after transformation by the configuration wizard contains user data whose openness status is unknown. The GPL licence may no longer be applicable and is thus removed. If this output file is released in the public domain, an adequate licence should be added, remembering that the original template source was GPLed (v3).

“Ordinary” lines, *i.e.* not an LCL statement, are copied into the output file after substituting the value for `%symbol%`. An unknown symbol is an error.

Lines starting with an opening comment delimiter followed by @ are LCL statements. They are decoded by `parse_statement` which returns in `$args`, `$var`, `$command` and `@labels` respectively the argument string, the associated variable, the command (statement name) and the labels of the LCL statement. This statement is interpreted by `interpret_statement`. If the statement does not need further processing, skipping symbol substitution and insertion into output is forced with `next` instead of going through the end of the loop.

- Label (void command): ignored but copied to output file

This should normally not happen since a label is a target for an action and absorbed during action processing. However, it does not harm to keep unused labels. It may even be a way to debug a template.

- U (potentially Unknown): this is rather a warning flag

In case the line contains undefined substitution symbols, the error indication is suppressed and the line is unchanged. If all substitutions succeed, the comment delimiters are removed, leaving an “ordinary” data line.

Interpretation for pass 2 (and others) is launched by `pass2` (from wrapper routines).. The arguments are the same as for `expand` with the exception of the end label, which is always `ENDP2` and is not given.

The output file is opened for input and will be copied/changed into a temporary file with the same name and extension `.LXR`. The input template file is repeatedly scrolled to the next `PASS2` LCL

²⁶ It is highly recommended to write any LCL statement on its own dedicated line starting it in column 1. The present implementation for “floating” comments (such as /* ... */) accepts an LCL statement immediately after a multi-line erasable comment, but this comes from a side-effect rather than from deliberate design.

²⁷ For a tricky use of this feature, read carefully `templates/initdb/initdb-m-template.sql`. The LXR manager has the possibility to create the databases either under the master account or under a user account by writing/erasing an erasable closing delimiter - */ at the end of two lines (only one delimiter must be present).

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

statement, which is parsed with `parse_statement` to extract the label to use. The original output file is copied to the temporary file until the target label is met. Interpretation of the `PASS2` block is requested through a call to `expand` with an `ENDP2` stop label. The target label is rewritten if the block was not marked with option `R` (remove).

The loop is left when reaching end-of-file on the input template file. The remaining lines from the original output file are copied to the temporary file. The original output file is deleted and the temporary renamed to the original file.

`interpret_statement` is in charge of interpreting an individual LCL statement or its subordinate block for compound constructs.

- **XQT** (execute): if generating a *shell* script, insert the rest of the line (and substitute value for symbols); otherwise, do nothing
- **ERROR**, **REMIN**D, **LOG** and **MSG**: print the arguments after symbol substitution
LOG needs `$verbose` at least 1 and **MSG** at least 2.
- **ASK**: get data from user through `ask_question` and store answer into symbol `%var%`, where *var* is the variable associated to the LCL statement
- **KEEPON**: get data from user until an empty answer and expand the block terminated by `ENDK`
All components of the **KEEPON** block are kept in hash `%keep`. Key `q` (question) contains the argument (after symbol substitution) to pass to `ask_question` after ensuring an empty answer is possible. Key `v` (variable) is the name of the associated variable. The block itself is scanned to store its components under adequate keys: `=none`, `=first` and `=epilog` for **ON NONE**, **ON FIRST**, **ON EPILOG** sub-blocks. The sub-block is delimited by an `ENDON` statement. The standard body, terminated at `ENDK`, is stored under key `=body`.
The question is asked a first time to see if there is no answer at all, leading to a call to `expand` on sub-block with key `=none`. Otherwise, sub-block with key `=first` is expanded. Then a loop is entered, calling `expand` on sub-block with key `=body` for every answer to the question until a empty string is returned. At this time, a last call to `expand` is made for block with key `=epilog`.
- **CANON** and **CANONR**: apply replacement rules to a variable content
- **IF**: conditional block interpretation
Only one of the **IF**, **ELSEIF** or **ELSE** blocks must be interpreted. A loop which stops on `ENDIF` statement is entered. It checks if command is **ELSE** or if the expression is true to decide for block interpretation. On a positive decision, `expand` is re-entered in function mode (to keep the characteristics of the last LCL statement) with a stop sentinel of **ELSEIF**, **ELSE** or **ENDIF**. The statements are then skipped until `ENDIF` if this statement has not been reached, matching any new **IF** with the corresponding `ENDIF`. On a negative decision, `skip_until` is called for **ELSEIF**, **ELSE** or **ENDIF**, matching any new **IF** with the corresponding `ENDIF` and the loop is iterated again.
- **CASE**: block selection
The argument is evaluated with `evaluate_expr` to give the target case. An infinite loop is then entered.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

Each iteration begins with a call to `skip_until` for a label, matching any new CASE with the corresponding ENDC. Reaching statement ENDC means no label was found, which is an error causing loop exit. If the target case matches any label of the stop statement, an extra call to `interpret_statement` is made for the matching labelled one (if any). `expand` is re-entered in function mode (to keep the characteristics of the last LCL statement) with a stop sentinel of label or ENDC. After this expansion, lines are skipped until ENDC, matching any new CASE with the corresponding ENDC unless we already reached ENDC. The loop is terminated.

- **ARRAY:** iterate block on every element of designated array(s)

Since several arrays (with same size) can be scanned in parallel, the statement argument is parsed for symbol names (access to the actual array through `%markers` dictionary) and its optional variable name. This information is stored in hash `%array` with key equal to variable name and value equal to symbol name (array name). A check is made for the existence of the arrays.

The block is scanned until ENDA to store its components in the `%array` hash under keys `=none`, `=prolog` and `=epilog` for sub-blocks ON NONE, ON PROLOG, ON EPILOG. These sub-blocks are terminated by ENDON. The standard body, terminated at ENDA, is stored under key `=body`.

All designated arrays must have the same number of elements or an error is issued.

Empty arrays cause expansion of sub-block with key `=none`.

For non-empty arrays, sub-block with key `=prolog` is interpreted by `expand` with all associated variables set to the size of the array. One expansion of sub-block with key `=body` is done with all associated variables set to the value of the current element. Finally, sub-block with key `=epilog` is interpreted with all associated variables set to the size of the array.

- **DEFINE:** create or change a symbol

The statement argument is split into variable (symbol) name, = operator (dropped) and expression. If the symbol name starts with `_`, it is considered read-only and an error is issued. The symbol, surrounded with % is entered into the dictionary with a value returned by `evaluate_expr`.

- **ADD:** include a file

Argument first character is examined for single quote (') or double quote ("). If this is the case, it is retained as the file name delimiter. This quoting feature is very unusual syntactically speaking since the parser grabs anything between the opening delimiter and the last delimiter (preceding white space before the end of line). There is thus no need of escape mechanism, even to include the delimiter.

The eventual substitution variables are replaced by their values with `substitute_markers`.

The resulting file name is used "as is" if it starts with `/`, `./` or `../`. In the latter two cases, the file name is relative to whatever directory is current (usually it is the *LXR root directory*). Otherwise, the file is prefixed with the value of `%LXROvrdir%` (defined by option `--tmpl-ovr=`) if it exists, or with the value of `%LXRtmpldir%` (defined by option `--tmpl-dir=` with default value *templates/*).

The designated file is scanned with `expand` and closed on return.

- **PASS2:** block for later execution

Location in the output file is marked with a label generated from the command argument unless this is a pass in an additional session and option R (**R**emove) is set on the statement.

Lines are skipped without interpretation until the next ENDP2 statement.

- Unmatched block delimiters (ELSE, ELSEIF, ENDF, ENDA, ENDC, ENDK, ON, ENDON, ENDP2) and unknown commands give an error message.

The support routines are:

Routine name	Arguments	Description
parse_statement	Reference to input function Line to parse Comment start Comment end	Assembles the statement from continuation lines and splits the line into its components: labels, command name, associated variable or option, argument string (<i>i.e.</i> rest of the line after command) Returned value is a list: argument, variable, command, labels in this order. Note there may be no labels. By default, variable is A.
interpret_statement	Reference to input function Output file handle Reference to symbol dictionary Verbose flag Comment start pattern Comment end pattern Reference to line LCL argument LCL variable LCL command	Interprets an LCL statement and eventually causes a block to be expanded The first set of arguments is needed to call <code>expand</code> . Reference to the current line allows to transform it for insertion into the output file. The last set of arguments comes from <code>parse_statement</code> and drives interpretation. Returned value is 1 if the statement has been fully interpreted (and can be “forgotten” by caller) or <code>undef</code> if the current line should be further processed/
substitute_markers	Reference to line Reference to symbol dictionary Comment start Comment end	All <code>%name%</code> symbols are replaced by their value from the dictionary An error is issued if <code>name</code> does not exist in the dictionary, unless the line is an U LCL statement in which case the line is left unmodified. The U command (and comment delimiters) are erased if no errors are reported.
evaluate_expr	Expression string Reference to symbol dictionary	Submits the expression string to <code>eval</code> after some transformations Sub-expressions involving <code>%symbols%</code> should be limited to comparisons for equality/inequality (<code>eq</code> or <code>ne</code> for string, <code>==</code> or <code>!=</code> for numbers). Since the implementation does not strictly enforce this rule, other operators may work but this is not supported. A list of all <code>%symbols%</code> is collected. With it, a sequence of <i>Perl</i> variable definitions is built as <code>my \$_symbols = value;</code> . <i>Value</i> is taken from the dictionary. If the symbol is an array, its value is the number of array elements. This sequence is followed by the expression string where every <code>%symbols%</code> is replaced by <code>\$_symbols</code> and then passed to <code>eval</code> which checks the expression correctness and computes it, using <i>Perl</i> rules. The function returns the computed value.
skip_until	Reference to input	Skips lines until condition is met

Project LXR	The LXR Developer's Manual 6 Configuration Wizard	Language en_UK
Software release 2.0		Document revision 1.0

Routine name	Arguments	Description
	function Stop sentinel pattern Begin nesting command pattern Ending nesting command pattern Comment start pattern Comment end pattern	<p>The stop, begin and end patterns are converted into regular expressions matching on complete LCL lines.</p> <p>Lines are read in a loop. If a begin nesting command is encountered, nesting level is incremented and stop sentinel identification is disabled. When an ending nesting command is encountered, nesting level is decremented and stop sentinel identification is re-enabled if level is zero. The loop is exited either on recognising the stop sentinel or an ending nesting command at level 0 (<i>this is based on the assumption that a stop sentinel is an internal "command" of a begin/ending nesting block</i>).</p> <p>The returned value is computed by <code>parse_statement</code> on the command causing loop exit.</p> <p>Restriction:</p> <p>This procedure does not handle <code>ADD</code> commands to get input from another file. Also, if the stop sentinel is not found before the end of the current input file, no attempt is made to pop the input file stack. Consequently, the skip target must be located within the current file.</p>
<code>grab_block</code>	See <code>skip_until</code>	<p>Returns an array of lines from current position to the stop sentinel. This function is similar to <code>skip_until</code>. The lines are stored in an array instead of being skipped. When the stop sentinel is found, the lines are returned. Finding an ending nesting command at level 0 is an error.</p> <p>The same restriction applies.</p>
<code>ask_question</code>	LCL argument string	<p>Interface to <code>get_user_choice</code></p> <p>The argument string is split at <code>;</code> into question, default answer, optional choices and optional normalised answers. The last two components are split at <code>,</code> to be stored into arrays.</p> <p>These components become arguments of <code>get_user_choice</code>.</p>

6.2.c. QuestionAnswer.pm

This package offers a simplified interface to ask a question on the terminal and get an answer. Question may be *closed*, if answer must be taken from an exhaustive list of choices, or *open*, when answer is not constrained. *Closed questions* may have a default answer which is selected when the user just hits "return" (empty answer).

The exported function is `get_user_choice`. It returns the user answer (string). Arguments are:

- `$question`: a string containing the question to ask
- `$default`: an integer describing the default answer as
 - 3 open question, empty answer allowed,
 - 2 open question, no default answer, empty answer not allowed (*user answer mandatory*),
 - 1 open question with default answer,

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

- 0 closed question, no default answer (*user answer mandatory*),
- >0 closed question, position of default answer in list (first is 1).

- `$choices`: optional reference to list of answers (not present for open question, mandatory for closed question)

Open question have no limited choice. `$choices` becomes a reference to a list of even number of strings. The first string in a pair is a regular expression (without delimiters) to match the answer. The second string is a message to print if the answer does not match. The list may contain several criteria which are tested one after the other. Any failure causes the question to be asked again.

- `$answers`: optional reference to list of “normalised” answers

This argument is present for closed question when a “normalised” answer is desired and for open question to define the default answer. The choices may then be written in a user-friendly style while the more processing-oriented associated “answer” is returned to caller. For example, the choices may describe in detail the resulting effects and the answers return a mnemonic, like:

Choices of databases:

MySQL, Oracle, PostgreSQL, SQLite

Normalised answers:

m, o, p, s

In practice, \$answers is systematically specified, notably in LCL, because a choice can be selected with the smallest unique prefix, leading to unpredictable answer length. Perl processing can use substr function to keep only the prefix (though its length may vary with the specific answer) but this cannot be done in LCL.

Quick checks are made to insure consistency between the arguments. Fatal errors cause process termination with status 2.

The choices for a closed question are transformed into a list of regular expressions, one for each choice, by function `find_unique_prefix`. The choices are then converted to lowercase into a local array.

Note:

Since `$choices` is a reference, this conversion modifies the original list (side-effect!). It does not matter in the current implementation since all arguments are not named variables but dynamic anonymous arrays which are recomputed on every call.

In the case of an open question with default answer, the local array contains the default answer.

The string corresponding to the default answer is converted to uppercase and prefixed with an ANSI escape sequence to display it green.

The ask-and-check infinite loop is entered. `$question` is printed “as is” on the terminal; no highlighting is done and nothing is suffixed to the string (any needed punctuation must be provided in the string). The list of choices is printed between square brackets, separating each item with a solidus (/). Items are highlighted yellow, the optional default choice green. Finally a yellow blinking > prompt is sent. User entry is read and checked.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

If it is void and a default answer is allowed, the default normalised string is returned. An empty string is returned for an open question with empty answer allowed. Otherwise, the empty answer is illegal and a new iteration is started.

A non-void answer to an open question is first checked against the validation regular expressions and is returned as is if it passes the tests. Otherwise, a new iteration is started.

Lastly, the regular expressions for the closed answers are taken one at a time. If the user entry matches the expression, the corresponding normalised answer is returned. If no match occurred, the answer is invalid and a new iteration is started.

Function `find_unique_prefix` computes the shortest unique prefix for each element of its array argument (it is a reference to a choice list).

The list is first “flattened” to the concatenation of all its elements, everyone prefixed with `#`²⁸. This string will be a test case for the candidate regular expressions.

A loop examines each choice string in its turn. A prefix, starting with `#`, is progressively extended with the next character of the choice. As each character is added, a match is attempted against the flattened string. If there is a single match, the remaining characters of the choice are added as optional match²⁹, the initial `#` is replaced by anchor `^`, the regular expression is stored in a list and the loop proceeds with the next choice. When the characters are exhausted without a single match, no valid unique prefix exist for this choice set; this is a fatal error.

The function returns the list of computed regular expressions.

Example:

Choices are between `file` and `function`. The computed regular expressions will be:

`^fi(1(e)?)?` and `^fu(n(c(t(i(o(n)?)?)?)?)?)?`

6.3. LXR Configuration Language (LCL)

This rudimentary macro language is used to drive substitutions in the configuration templates based on input from the user. This way, rather elaborate configurations can be constructed and the resulting file is not cluttered by unused options.

6.3.a. Syntax

LCL statements are found inside comments. The comments follow the syntax of the language for the generated data, *e.g.* from `#` to end of line for *Perl*, web server configuration or *shell*, from `/*` to `*/` for *SQL* or *C*.

²⁸ **CAUTION!** This pound character `#` is supposed never to occur in the strings. If this is not the case, another delimiter should be chosen.

²⁹ Strictly speaking, there is no need to match beyond the shortest unique prefix because all choices can be differentiated with these head characters. However, this is a fool-proof safety measure. If a legal choice is `file` with unique prefix `fi` and user types `find`, maybe he is thinking to something else. Matching only `fi` would not detect a possible error.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

Some comments are *erasable*: they are not copied to the output file. They are identified with an hyphen (-) immediately following the comment start delimiter. If comment syntax needs an ending delimiter, the matching delimiter is immediately preceded by an hyphen.

Examples;

```
#- Erasable comments when comment ends at end of line
#- The - marker must be repeated on every erasable line

/*- With comments capable of extending
*- on multiple lines, strictly only the delimiters
*- need to bear the - marker
-*/
```

If an erased comment leaves a blank line (no other data remains after erasure), the blank line is also removed to avoid excessive vertical spacing.

A comment is an LCL statement if it starts in column 1 and the statement marker @³⁰ immediately follows the comment start delimiter. No other non-comment data may be present after the comment.

Examples:

```
#@MSG This is an LCL statement

/*@MSG This is also an LCL statement */
```

Non-LCL examples:

```
some data #@MSG This is a standard comment
           #@MSG This is not an LCL statement
# @MSG neither this one

some data /*@MSG standard comment*/
           /*@MSG comment not LCL */
/* @MSG comment also */
/*@MSG message */ This data cause downgrading to ordinary comment
```

When needed, LCL statements may span several lines. The form is slightly different depending on the existence of a closing comment delimiter.

For shell-style comments, a backslash (\) immediately preceding newline (no whitespace allowed) requests continuation. Continuation lines start with #@ but are not anchored to column 1. The statement is continued starting with the first non-whitespace character after the statement marker.

```
#@MSG Beginning of \
#@      message (note: initial \
```

³⁰ This marker has been chosen as a tribute to a famous (and remarkably efficient) OS and hardware line of the '70s and '80s. Do the old-timers remember?

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

```
#@ spaces on continuation lines \
#@ are dropped and continuation \
#@ are not anchored to column 1.)
```

For C-style comments, no special arrangement is necessary. The statement ends at comment terminator. Initial whitespace in continuation lines is dropped.

```
/*@MSG Beginning of message
   (note that a space
   precedes newline to allow word separation)*/
```

Though it is not directly part of LCL, any sequence *%name%* found anywhere is replaced by the value of *name* from the symbol dictionary. *Name* is a run of alphanumeric or underscore (`_`) characters.

An LCL statement is either a label or a command. Labels and commands are made of a sequence of alphanumeric or underscore characters.

A label statement is written without any space as:

```
@name:
```

Several labels can be specified on the same same statement without separating spaces as `@name1:name2:name3:.`

A command statement has a more elaborate structure:

```
@ command_name,var_name rest of line
@name: command_name,var_name rest of line
```

There may be spaces between `@` (or `@label:`) and *command_name*. *command_name* is case-insensitive.

, *var_name* is optional. If present, there is no space between *command_name* and the comma, nor between the comma and *var_name*. The result of the command, if any, will be stored in variable *var_name* (by default, A like Answer). For some commands, this is an option field and every character designates an option.

Spaces separate the command field from the eventual arguments in *rest of line*.

Note:

Some commands are not allowed to be labelled: ON, PASS2 and all those beginning with END and ELSE because it does not make sense or would break correct nesting.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

6.3.b. LCL commands

6.3.b.1. Error suppression

```
@U data line with %symbols%
```

`%symbols%` are replaced by their values. If all substitutions were made, the data line (without `@U`) is inserted into the output file. If some substitution is impossible, the full command is inserted as a comment into the output file. No error is reported to the user.

Tip:



This is useful when some symbol is known to have no value. For instance, when using *glimpse*, parameter 'swishbin' is not defined. An alternate way of doing the same would be to use `@IF`, `@ELSE`, `@ENDIF` but this become tedious even with a small number of `%symbols%` in a line.



CAUTION:

This statement is not a real LCL command. It should be considered as a “protected” data line and, as such, cannot have a label.

6.3.b.2. Shell command insertion

```
@XQT shell command
```

Insert shell command into output file if generating a shell script; otherwise do nothing.

6.3.b.3. Message display

```
@ERROR error text
```

Print unconditionally the error text prefixed with `ERROR:`.

```
@REMIND advisory text
```

Print unconditionally the text prefixed with `Reminder:`.

```
@LOG message
```

Print the message under any verbosity (message not printed if no `-v` or `--verbose` options).

```
@MSG message
```

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

Print the message only under full verbosity (-vv or --verbose options).

6.3.b.4. User interaction

```
@ASK, var question;dft;choice,list;answer,list
```

Ask a question and get answer from user.

var is the variable name where the answer is stored., A (case significant) if not specified.

The four command arguments are separated from each other by a semicolon (;).

- *question* is the text displayed to the user.
- *dft* defines the type of query and the default answer:
 - 3 open question (no choice list nor answer list), empty answer allowed
 - 2 open question (no choice list nor answer list), non-empty answer mandatory
 - 1 open question (no choice list) with default answer (defined in answer list, choice list is empty)
 - 0 closed question (both lists present), non-empty answer mandatory
 - >0 closed question (both lists present) with default answer (*dft* value is the default answer index in the lists; first one has index 1)
- *choice list* is a comma-separated (,) list of human-readable proposals; the items are displayed lowercase with the default answer highlighted uppercase.

Tip:



For an open question (*dft* negative), *choice list* may be replaced by a list with an even number of elements to drive a validation filter for the answer. The first string of the pair is a regular expression without delimiters, the second string is a message to print if the regular expression did not match. If the answer does not pass the test, the question is asked again until it is valid.

- *answer list* is a comma-separated (,) list of “normalised” answer to be returned in variable *var*.

choice list may contain lengthy descriptive text because it is intended to be read and understood by a human while *answer list* contains only abbreviated symbols intended for processing.

Examples:

```
#@ASK,C Enter a comment ;-3
#@ASK,N Enter your name ;-2
#@ASK Host name? ;-1;;http://localhost
#@ASK Processor variant? ;0;32 bits, 64 bits;5,6
```

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

```
#@ASK Database engine? ;1;MySQL,Oracle,PostgreSQL,SQLite;m,o,p,s
```

The first command requests a comment in variable C. An empty comment is accepted.

The second command requests the user's name into N. The user cannot elude the question.

The third command requests a host name into A. The answer is arbitrary. But if the user just hits “return”, `http://localhost` is entered into A.

The fourth command requests the width of processor bus into A. Since there is no obvious default, the user must explicitly choose one of the proposed values. The power of two of the bus width will be returned into A.

The fifth command requests a database engine name from the list. The corresponding single letter is returned into A. If the user just hits “return”, MySQL is chosen and `m` is returned.

Examples of validating open questions:

```
#@ASK Host name? ;-1;^(https?:)?//,not an HTTP URL;http://localhost
```

This is a variant of the third command above. The answer is accepted now only if it begins with `//`, `http://` or `https://`. More than one validation may be requested, as in:

```
#@ASK Host name?;-1\  
#@ ;^(https?:)?//\  
#@ ,not an HTTP URL\  
#@ ,//[w-]+(\.[w-]+)*(:\d+)?/?$\  
#@ ,invalid URL host syntax\  
#@ ;http://localhost
```

The second validation criterion tells the answer must end in a dot-separated host name with optional port number and optional final solidus (`/`).



CAUTION!

The argument set is split at semicolon (`;`) and then the bits at comma (`,`). If you need these characters inside the strings, protect them with an escaping backslash (`\`).

```
@KEEPON,var question  
@ON none  
...  
@ENDON  
@ON first  
...  
@ENDON  
@ON epilog  
...  
@ENDON  
... (loop body block)  
@ENDK
```

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

Ask a question and process the answer until an empty answer is given. See `@ASK` for the command arguments.

Note:

From a logical point of view, `@KEEPON` is an iterated `@ASK` with *dft* equal to `-3`. This selector is the only possible one (any other would not allow to exit from the loop).

The optional `@ON/@ENDON` are interpreted under specific circumstances. They may be listed in any order but must precede the loop body block.

`@ON none` is interpreted if the first answer is empty, then the `@KEEPON` block is left.

`@ON first` is interpreted before the first loop iteration (non-empty first answer).

`@ON epilog` is interpreted after the empty answer requesting loop exit. The main loop body is not interpreted.

The loop body block is interpreted for each new answer which is stored into variable *var* (by default A).



Tip:

Since `@KEEPON` is strictly equivalent to an iterated `@ASK`, answers may also pass through a validation filter with the same syntax as `@ASK` (but is restricted to `-3` selector).

Example:

```
#@KEEPON Alias;-3 \
#@      ;^(https?:)?//,not an HTTP URL\
#@      ,//[a-z-]+(\.[a-z-]+)*(:\d+)?/?$,invalid URL host syntax
, '%A%'
#@ENDK
```

This insures that the generated list contains only valid URL host names.



Tip:

`@KEEPON` implements the *0 or more instances* paradigm. If you want the *1 or more instances* paradigm instead, precede the `@KEEPON` block with an `@ASK` statement with a `-1` or `-2` selector (*i.e.* open question with mandatory answer implicit or not), like:

```
#@ASK,vn ---Version name? ;-2
%vn%
#@KEEPON,vn ---Version name? (hit return to stop)
%vn%
#@ENDK
```

After receiving data from the user, it is usually good practice to “canonise” or slightly transform it into a standard expected form. For instance, some parts of the answer may be omitted (such as the `http:` prefix in an URL) but it is more comfortable for the *LXR* processing scripts to always deal

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

with the full format. Another important case is adaptation from human expression to supporting machine language (*Perl* in *lxr.conf* case), inserting escape characters where necessary. This is possible with @CANON and/or @CANONR.

```
@CANON, var pattern, replacement
```

```
@CANONR, var pattern, replacement
```

Apply transformation rules to a variable content (default variable is A).

@CANON rules are applied to the first occurrence of the patterns, while @CANONR (R for **R**epeat) applies them to all occurrences. If a pattern does not match against variable text, the variable is not changed and the next rule is tried.

- *pattern* is a regular expression without delimiters.
- *replacement* is the string to substitute when pattern is found.

CAUTION!

ALL characters are significant after the first non-whitespace character following the command name. Do not use spaces to pretty write your rules: these spaces will be taken literally in the patterns or replacements. The first sentence also means the first pattern cannot begin with significant whitespace.



Tips:

If you need a first (or single) pattern starting with a literal space escape it with a backslash.

To erase the run matched by the pattern, do not write any replacement, but leave the separating commas.

Examples:

Prepare answer in default variable A to be inserted in a single-quote delimited *Perl* string (all single quotes must be backslashed-escaped):

```
#@CANONR ' , \ '
```

Erase trailing slashes in directory name and make sure there is one initial slash:

```
#@CANON /*$, , ^/*, /
```

Note that if the rules were applied in reverse order, we could end up with an empty string instead of a single /.

Replace spaces with underscores:

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

```
#@CANONR \ , _
```

6.3.b.5. Conditional interpretation

```
@IF expression
...
@ELSEIF expression
...
@ELSE
...
@endif
```

Interpret only one of the blocks depending on the value of expressions.

Note:

The alternative test command is @ELSEIF, not @ELSIF as in *Perl* or *C*.

Presently, the *expression* is evaluated by *Perl* but it is highly recommended to use only simple expressions. A single %symbol% is a test for existence. If %symbol% is used with an operator like eq or ne, it is a comparison to a value³¹ or to another %symbol% and an error is issued if the symbol is unknown. An array %symbol% has the number of elements for its value. These primary expressions can be combined with “or” (||) and “and” (&&) operators and organised for precedence with parentheses. A non-zero final value means “true” and a zero value “false”.

CAUTION!

%symbol% are symbolically substituted in the expression text before *Perl* eval processing. This substitution is not delimited by any specific character. This will very likely lead to syntax errors when decoded. To avoid such errors, put single or double quotes around %symbol% because a *string* is usually the desired substitution result.



Example:

```
#@ASK Hide LXR release number? ;2; yes,no; y,n
#@IF %A% eq 'n'
, 'release' => '2.0'
#@ENDIF
```

is wrong because the expression passed to *Perl* is *y eq 'n'* and *y* is neither a variable nor a string. You must write:

```
#@ASK Hide LXR release number? ;2; yes,no; y,n
#@IF '%A%' eq 'n'
, 'release' => '2.0'
#@ENDIF
```

³¹ A value may be a single-or double-quote delimited string. The difference is only relevant to the *Perl* evaluation.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

The expression is now 'y' eq 'n', a comparison between constants.

```
@CASE expression
@label:label:
...
@label:label:
...
@ENDC
```

Select one the cases conditioned by an expression value matching a label.



IMPORTANT:

Since label may only contain alphanumeric and underscore characters, take care that the expression, usually a %symbol% value, does not introduce “foreign” characters which cannot be matched to a label.

Note:

The preferred layout style is to write the labels on their own line, especially if the corresponding case consists of several lines.

6.3.b.6. Array content insertion

```
@ARRAY array1,var1 array2,var2...
@ON none
...
@ENDON
@ON prolog
...
@ENDON
@ON epilog
...
@ENDON
... (loop body block)
@ENDA
```

Retrieve array content in a variable and use it for expanding a block of statements for each element. The elements of several arrays may be retrieved simultaneously if they are declared on the @ARRAY command. They must all have the same size.

If a variable name is not specified, the element is stored into E (Element). All variable names must be different lest some array content becomes inaccessible.

The optional @ON/@ENDON are interpreted under specific circumstances. They may be listed in any order but must precede the loop body block.

@ON none is interpreted if the array is empty, then the @ARRAY group is left.

@ON prolog is interpreted before the first loop iteration (non-empty array).

Project LXR	The LXR Developer's Manual 6 Configuration Wizard	Language en_UK
Software release 2.0		Document revision 1.0

@ON epilog is interpreted after the iteration on the last array element.

The loop body block is interpreted for each element set where the elements are stored into the variable associated with each array.

Example:

```

, 'host_names' =>
  [ '%scheme%/%hostname%'
#@ARRAY schemealiases,S hostaliases,A portaliases,P
#@  ON  none
      # Put here aliases for host name, such as
      # , '//localhost'
      # , 'https://192.168.1.1'
      # , 'http://mycomputer.outside.domain:12345'
#@  ENDON
      , '%S%/%A%:%P%'
#@ENDA
  ]

```

Note the difference between array or variable specification/declaration (without % characters) and usage of the value (with % characters).

6.3.b.7. Variable assignment

```
@DEFINE var = expression
```

Define a new scalar variable and set it to the value of an expression. Can also be used to change the value of an existing scalar variable.

Note:

There is no way to change the value of an array element, nor to define new arrays.

As previously stated, do not write too complex expressions. The expression evaluator might someday be internally coded instead of handing over the computation to Perl eval.

Avoid using *Perl* concatenation operator (.). If you need to glue together two variables A and B, write it as:

```
#@DEFINE result = '%A%B%'
```

To keep a user answer, do not write:

```
#@ASK Which colour? ; 0; diamonds,clubs,hearts,spades; d,c,h,s
#@DEFINE colour = '%A%'
```

but write:

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

```
#@ASK,colour which colour? ; 0; diamonds,clubs,hearts,spades; d,c,h,s
```

It is simpler, more elegant and more efficient.

6.3.b.8. File inclusion

```
@ADD file_name
```

Continue interpretation in the designated file.

The argument³² is interpreted as a file name according to the following rules:

- if the name begins with /, ./ or ../, it is used “as is”;

CAUTION!

Use the ./ and ../ variants at your own risk because it depends on the current working directory setting. The configuration wizard may be launched in many different ways, while the template designer has rather precise (and often implicit) ideas on its template usage.

- otherwise, the file name is prefixed with the directory defined by option `--tmpl-ovr=`, the secondary template repository, or else by the standard template directory (whether the default one or defined by option `--tmpl-dir=`).

@ADD may be dynamically nested up to an implementation-defined limit. A minimum of 5 is guaranteed.

Important semantic restriction:

This command is not interpreted when skipping lines (hunting for the active @IF or @CASE part or exiting these constructs after successful interpretation) or when collecting code samples (@ARRAY and @KEEPON sub-blocks).

This has severe consequences.

1. When looking for a specific command in compound constructs (@ELSEIF, @ELSE or @ENDIF; label or @ENDC; @ENDP2), the target command must be found in the current input file or its parents and cannot be located in any @ADD'ed file.
2. The sentinels for parts of differed constructs (@ON, @ENDON, @ENDA or @ENDK) must also be found in the current input file or its parents and cannot be located in any @ADD'ed file.
3. After the initial pass (during the “passes 2”), @PASS2 commands can be found only in the original input file since input is fully reinitialised.

These restrictions do not prohibit @ADD usage in compound or differed constructs, it only means

³² Actually, the first word of the argument. If the file name is expected to contain spaces, it should be surrounded by single- or double-quotes but no mechanism is offered to “escape” any character identical to a delimiter.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

@ADD commands are active when statements are interpreted. @ADD commands present in @ON blocks are stored in the sample and rightfully interpreted when the @ON block is activated.

To be considered:

The restrictions can be removed if @ADD processing is done in an input layer (with a performance penalty because every command would then be parsed and the result discarded most of the time).

6.3.b.9. Pass 2 interpretation

```
@PASS2,R name
...
@ENDP2
```

Mark a group of lines for interpretation during pass 2.

During the initial pass, label *@name*: (with comment delimiters) is inserted into the output file and the block is skipped.

When a pass 2 is started, the input file is scanned for @PASS2 commands. When one is found, the output file is “scrolled” to the corresponding label. If option R (Remove) is specified, the label is erased. Input lines are interpreted until @ENDP2. When this command is reached, search for @PASS2 is resumed.

6.3.c. Standard symbol dictionary

The dictionary is initially loaded with symbols describing the environment and the context of execution. All symbols are kept in a “ready-to-substitute” form, *i.e.* *ad %name%*. If *name* begins with an underscore (*_*), the symbol is read-only and its value cannot be changed with @DEFINE.

Options and environment:

<i>%_add%</i>	1 if adding a tree (either second tree in configuration or --add session), 0 for the initial tree
<i>%_shell%</i>	1 if generating a <i>shell</i> output file (presently only used for <i>initdb.sh</i>)
<i>%_singlecontext%</i>	1 if in single tree context; 0 in multiple-trees context
<i>%_createglobals%</i>	1 to tell the database templates to generate the global (shared by all trees) tables, parameters or database
<i>%_dbengine%</i>	name of database engine as m, o, p or s
<i>%_dbpass%</i>	password common to all databases
<i>%_dbprefix%</i>	table prefix common to all databases
<i>%_dbuser%</i>	user common to all databases
<i>%_dbuseroverride%</i>	set to 1 before pass 2 on <i>initdb.sh</i> if another user/password should be used for this tree
<i>%_globaldb%</i>	1 if all trees share the same database, 0 if every tree has its own database

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

<code>_%_nodbuser%</code>	1 if user name is not shared among the databases, undefined otherwise
<code>_%_nodbprefix%</code>	1 if table prefix is not common to all databases, undefined otherwise
<code>_%_routing%</code>	how the URL is parsed to route requests to LXR as A (argument), E (embedded in section name), H (specific host name), P (prefix in host name), S (section name) or N (none, <i>i.e.</i> single tree)
<code>_%_shared%</code>	1 if LXR is part of a wider server; 0 if server is dedicated to LXR
<code>_%_virtrootpolicy%</code>	defined only if <code>_%_routing%</code> is equal to E, value is b for built-in URL decoding and c for custom decoding
<code>_%_commonvirtroot%</code>	non zero if virtual root in URL common to all trees

Note:

Most of the above symbols have a *boolean* nature. Their value should not be compared to constant 0 or 1 since *truth* may be implemented in many ways; *falsehood* itself may be represented by 0 or undef. The correct way to use the symbols in expressions are:

`_%_add%` is *true* if the symbol is *true*,

! `_%_add%` (logical negation) is *true* if the symbol is *false*.

String content environment:

<code>%LXRconfuser%</code>	login name of the user configuring LXR (supposed to be the same as the one initialising the databases)
<code>%LXRroot%</code>	LXR root directory
<code>%LXRtmpldir%</code>	templates directory
<code>%LXRovrdir%</code>	templates override directory (from <code>--tmpl-ovr</code> option)
<code>%LXRconfdir%</code>	output directory for configuration (relative to <code>%LXRroot%</code>)
<code>%scheme%</code>	scheme for URL (<code>http:</code> or <code>https:</code>)
<code>%hostname%</code>	primary host name
<code>%port%</code>	TCP port (numeric only, without colon)
<code>%schemealiases%</code>	array of scheme for aliases
<code>%hostaliases%</code>	array of aliases for host name
<code>%portaliases%</code>	array of port for aliases
<code>%virtrootbase%</code>	virtual root base in URL
<code>%glimpse%</code>	path to <i>glimpse</i> executable
<code>%glimpseindex%</code>	path to <i>glimpseindex</i> executable
<code>%glimpsedirbase%</code>	directory for <i>glimpse</i> internal databases
<code>%swish%</code>	path to <i>swish-e</i> executable
<code>%swishdirbase%</code>	directory for <i>swish-e</i> internal databases
<code>%search_engine%</code>	selected search engine as <i>glimpse</i> or <i>swish</i>
<code>%ctags%</code>	path to <i>ctags</i> executable
<code>%DB_name%</code>	common database name
<code>%DB_user%</code>	shared user name for database access
<code>%DB_password%</code>	share database password
<code>%DB_globalprefix%</code>	common database table prefix
<code>%DB_tree_user%</code>	if <code>%DB_user%</code> has been overridden during tree description in <i>lxr.conf</i> ,

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	6 Configuration Wizard	Document revision 1.0

passes this user name to *initdb.sh* generator

`%DB_tree_password%` if `%DB_password%` has been overridden during tree description in *lxr.conf*, passes this password to *initdb.sh* generator

`%DB_tbl_prefix%` if `%DB_globalprefix%` has been overridden during tree description in *lxr.conf*, passes this table prefix to *initdb.sh* generator

Note:

These symbols are not protected *read-only* with an initial underscore, but they should be considered as such lest the configuration process becomes unreliable.

6.4. Standard templates

Templates used by the configuration wizard are stored in the *templates/* directory. Those involved with server configuration are fairly simple:

- in *Apache/* directory: *apache-lxrserver.conf* (server configuration), *apache2-require.pl* (Perl library initialisation) and *htaccess-generic* (model for *.htaccess*)
- in *Mercurial/* directory: *hg.rc* (declaration of *Mercurial* plugin for **LXR**)
- in *Nginx/* directory: *nginx-lxrserver.conf* (server configuration) and *nginx-fastcgi.conf.part* (@ADD'ed part for **FastCGI** parameters)
- in *initdb/* directory: *initdb-x-template.sql* where **x** covers the set of values of `%_dbengine%` (*shell* script for creating the databases) and eventual files @ADD'ed by the previous templates
- in *lighttpd/* directory: *lighttpd-lxrserver.conf* (server configuration)
- *thttpd-lxrserver.conf* (server configuration)
- *lxr.conf*: driver for user source-tree configuration
- *lxrkernel.conf*: driver for *Linux* kernel source-tree configuration
- files to be @ADD'ed by the previous driver templates: *global.conf.part* (global parameters section), *tree-server1.conf.part* (specific URL description for a tree), *tree-server2.conf.part* (final URL or *HTML* parameters for a tree), *tree-ignore.conf.part* (ignored directories for a tree) , *datastorage.conf.part* (database description for a tree)

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	7 Auxiliary Scripts	Document revision 1.0

7 Auxiliary Scripts

These scripts are stored in the *scripts/* directory. They provide convenience or complementary services in order to facilitate configuration or maintenance of an *LXR* deployment.

7.1. Linux kernel exploration script

This is a *shell* script. Its components are:

- *kernel-vars-grab.sh*: main driver;
- *ANSI-escape.sh*: ANSI escape codes definitions (does not contain all possibilities offered by 3.3.a VTescape.pm).

The script purpose is to gather the 'range' of values for 'v' (version), 'a' (architecture) and sub-architectures or variants 'variables'. The sets are stored in *custom.d/* files which are later read in from *lxr.conf* by procedure *readfile*.

CAUTION!

This script is not guaranteed to give the expected result. It has been crafted by A. Littoz after pragmatic observation of a 3.1 kernel. Using it on another kernel release surely needs some adaptation.

7.1.a. Process outline

The script target is an *LXR* source-tree with a subdirectory for each version.

The version list is built from the names of the subdirectories.

Looping from this list, the subdirectory names in each version *arch/* directory are added to the list of architectures. Some subdirectories, known to contain sub-architectures, are scanned for nested directories matching some pattern. Those found are stored in a specific file corresponding to a sub-architecture or variant 'variables'.

Finally, the files are sorted and duplicates are removed.

7.1.b. Support routines

The functions in the script are:

Routine name	Arguments	Description
<code>scan_one_version</code>	Version directory	Checks that the version directory has roughly the expected organisation for a <i>Linux</i> kernel Enumerates the subdirectories in <i>arch/</i> of the version directory

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0		7 Auxiliary Scripts

Routine name	Arguments	Description
		Calls <code>collect_sub_arch</code> on a selected list of subdirectories
<code>collect_sub_arch</code>	Version directory <i>arch/</i> subdirectory Prefix	In the <i>arch/</i> subdirectory, keeps only the directories whose name begins with the given prefix followed by a dash (-). The tail of the name is added into a <i>custom.d/</i> file. The name of this file is composed of the <i>arch/</i> subdirectory name, an underscore (_) and the prefix.

7.1.c. Interaction with *lrxkernel.conf*

lrxkernel.conf prepares *lrx.conf* based on several assumptions.

The following items must be checked in the script:

- List of versions

The 'range' of variable 'v' is read from file *version_list.txt*. The content may need to be updated if version directories are manually erased after script execution. 'default' may also need an update.

- List of architectures

The 'range' of variable 'a' is read from file *arch_list.txt*. The content of this file is reliable (no assumption needs to be made about kernel source organisation except for the existence of the *arch/* directory).

- Sub-architectures

There is no automatic method to identify architectures with variants. The structure of variant name vary from one architecture to the other. Some architectures may contain several variant families. This is determined only through human scrutiny of the architecture directories.

The names of the relevant architectures are hard-coded in the script.

There is one call to `collect_sub_arch` per variant.

The arguments describing the variant are hard-coded in the script.

Example:

The ***mn10300*** architecture exhibits variants in processors and units. Code for them is contained in subdirectories *proc-mn103e010/*, *proc-mn2ws050/*, *unit-asb2303/*, *unit-asb-2305* and *unit-asb2364/*. Two calls are necessary to enumerate the value ranges.

```
collect_sub_arch "$1" "mn10300" "proc"
collect_sub_arch "$1" "mn10300" "unit"
```

`$1` is the first argument to `scan_one_version`, *i.e.* the version directory. The collected list will be stored into *custom.d/mn10300_proc_list.txt* and *custom.d/mn10300_unit_list.txt*.

The following items must be checked in *lrx.conf*:

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	7 Auxiliary Scripts	Document revision 1.0

- Sub-architecture 'variables'

They must be hard-coded in the template.

Their name is composed of the *architecture*, an underscore (`_`) and the *variant*. It is the same as the filename without the `_list.txt` suffix.

To reduce screen cluttering, display is filtered by a 'when' clause testing the architecture.

Example with the *mn10300* architecture:

```
, 'mn10300_proc' =>
  { 'name'   => 'MN10300 processor'
    , 'when'  => '"$a" eq "mn10300"'
    , 'range' => [ readfile('custom.d/mn10300_proc_list.txt') ]
  }
, 'mn10300_unit' =>
  { 'name'   => 'MN10300 unit'
    , 'when'  => '"$a" eq "mn10300"'
    , 'range' => [ readfile('custom.d/mn10300_unit_list.txt') ]
  }
```

- Mapping *include* directories

The 'maps' transformation rules make use of the sub-architecture 'variables' to point to the correct subdirectory. There is one rule for each variant family.

```
, '^/arch/architecture/%=LVL2x=%/'
  =>      '/arch/architecture/variant-${var_name}/'
```

where:

- *architecture* is the architecture name (e.g. mn10300)
- *x* is a running letter starting from A for the first rule in this architecture
- *variant* is the directory prefix, e.g. proc or unit
- *var_name* is the corresponding 'variables', e.g. mn10300_proc or mn10300_unit



CAUTION!

Due to 'maps' implementation limitations (cumulative effect without backtracking), there must be one 'incprefix' path for each `%=LVL2x=%` template name.

There are presently only two (A and B) because no architecture had more than two variant families. If it happened that a new architecture contained more than two, new lines should be added into the 'incprefix' list to match the number of variant families:

```
, '/arch/%=ARCH=%/%=LVL2C=%/include'
```

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	7 Auxiliary Scripts	Document revision 1.0

7.2. Database reconstruction script

This script is written in *Perl*. Its components are:

- *recreatedb.pl*: main driver;
- *ContextMgr.pm*: context file utilities;
- *LCLInterpreter.pm*: interpreter for the *LXR Configuration Language* (LCL) macro statements;
- *VTescape.pm*: ANSI escape codes definitions (see 3.3.a VTescape.pm).

Note:

It also references *LXR/Files.pm* and *LXR/Common.pm* only to prevent errors when evaluating *lxr.conf* in the event the configuration file contains 'range' functions using built-in procedures from *LXR* library.

7.2.a. Process outline

Schematically, this script is a simplified configuration wizard where the interactive phase is replaced by reading a configuration file.

The global parameters are used to initialise global database context symbols in the dictionary.

Each tree section is scanned for its database definition and symbols are updated or created. *Shell* and *SQL* statements for creating the tree database are then added to the output script with a call to `expand_slash_star` under control of an *initdb-x-template.sql* template, where *x* is obtained from the database driver name.

7.2.b. Maintenance issue

Every time the configuration wizard is updated or improved, the database reconstruction script should be checked for a parallel update.

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	8 Release Tool	Document revision 1.0

8 Release Tool

The release tool, `makerelease.pl`, a small Perl script to automate the release procedure with **SourceForge**, is part of module `lxr-tools`. It must be explicitly downloaded since it is separate from the public **LXR** release.

8.1. Command line

```
$ makerelease.pl --option ...
```

When launching the script, the current working directory should be an **LXR** directory.

Note:

From code analysis, this is not the correct criterion. The script needs a `../lxx-tools/tests/` directory to run the tests and command `./scripts/set-lxx-version.sh`. It will also create a source release directory as `../lxx-x.y.z/`, which means the current directory cannot be the personal user directory but at least a sub-directory.

Options are:

<code>--help</code>	print help text and exit
<code>--cvsuser=name</code>	name of a SourceForge CVS user allowed to make release (write permission granted on CVS repository)
<code>--devel</code>	create a development tarball without tagging the CVS repository, nor uploading it
<code>--noex</code>	“dry-run” mode, commands are not executed
<code>--notest</code>	skip the tests
<code>--tag=tag_name</code>	define the CVS release tag (must be in the form <code>release-x-y-z</code>)

`devel`, `noex` and `notest` are flags defining the tool sub-tasks.

`cvsuser` and `tag`, if not provided on the command line, are requested from the user.

8.2. Process outline

The file environment is first checked to make sure the test suite and script `set-lxx-version.sh` can be reached. If this is not the case, some fall-back processing may be attempted but, most of the time, the process is terminated, leaving to the user the responsibility to fix the problem.

Tests are run unless option `--notest` was specified. Tests have not been updated for long but they

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0	8 Release Tool	Document revision 1.0

still make a good checking base.

For a public release, a tag is created from `--tag` or from user entry and marks the *CVS* repository. The change log is computed. The *CVS* repository is exported into a “parallel” directory and the change log is added to it. The directory is then compressed as a “tarball” and uploaded to *SourceForge stable/* directory using *rsync*.

For a development (intermediate) release, when option `--devel` is provided, The *CVS* repository is exported from head (the current most recent state) and compressed as a “tarball”. There is no tagging, change log nor upload.

8.3. Support routines

All specific support routines are written in the script file.

Routine name	Arguments	Description
<code>get_tags</code>		Returns a list of all symbols found in <i>cvs rlog</i> output for module <code>lxr</code> <i>Called from get_all_release_tags</i>
<code>tag_to_val</code>	Tag name	Returns a list of the 3 numbers in the tag, which must be in the form <code>release-x-y-z</code> , otherwise the list is (0, 0, 0) <i>Called from get_last_release_tag and tag_release</i>
<code>make_release_tag</code>	Major Minor Point	Returns a tag in the form <code>release-x-y-z</code> from the 3 integer arguments (inverse of <code>tag_to_val</code>) <i>Called from tag_release</i>
<code>get_last_release_tag</code>		Returns the highest <code>release-x-y-z</code> tag The release tags are read from <i>CVS</i> with <code>get_tags</code> and compare to the current target tag with the help of <code>tag_to_val</code> . <i>Called from tag_release</i>
<code>check_environment</code>		Checks the file environment to see if the required constraints are met or dependencies present
<code>run_tests</code>		Launches <code>../lxr-tools/tests/TestRunner.pl</code> as an independent process to run the tests Aborts if any test fails
<code>tag_release</code>		If <code>--tag=</code> is not specified, suggests a release tag based on <code>get_last_release_tag</code> , <code>tag_to_val</code> and <code>make_release_tag</code> and requests confirmation from user Tags the repository with <i>cvs tag</i>
<code>create_release_tarball</code>	Tag name Version string	Creates a release directory, exports <i>CVS</i> with <i>cvs export</i> into it, add to it <i>ChangeLog</i> unless <code>--noex</code> or <code>--devel</code> , sets the version string into <i>Template.pm</i> with <i>scripts/set-lxr-version.sh</i> and compresses the release directory in a tarball <i>Version string is a “classical” human-readable version which may or may not be related to the tag name. Tag name is the internal CVS release-x-y-z</i>

Project LXR	The LXR Developer's Manual	Language en_UK
Software release 2.0		8 Release Tool

Routine name	Arguments	Description
create_changelog	Tag name Version string	Creates the change log with <i>cvs2cl</i> <i>Arguments not used</i>
upload_release	Tag name Version string	Uploads the resulting tarball to <i>SourceForge</i> <i>/home/frs/project/lxr/stable/</i> directory using the version argument Important! --CVSUSER= <i>must be granted write access.</i>

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this *License* is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non commercially. Secondly, this *License* preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This *License* is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the *GNU General Public License*, which is a copyleft license designed for free software.

We have designed this *License* in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this *License* is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this *License* principally for works whose purpose is instruction or reference.

I. APPLICABILITY AND DEFINITIONS

This *License* applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this *License*. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the *Document* means any work containing the *Document* or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the *Document* that deals exclusively with the relationship of the publishers or authors of the *Document* to the *Document's* overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the *Document* is in part a textbook of mathematics, a *Secondary Section* may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain *Secondary Sections* whose titles are designated, as being those of *Invariant Sections*, in the notice that says that the *Document* is released under this *License*. If a section does not fit the above definition of *Secondary* then it is not allowed to be designated as *Invariant*. The *Document* may contain zero *Invariant Sections*. If the *Document* does not identify any *Invariant Sections* then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as *Front-Cover Texts* or *Back-Cover Texts*, in the notice that says that the *Document* is released under this *License*. A *Front-Cover Text* may be at most 5 words, and a *Back-Cover Text* may be at most 25 words.

A "**Transparent**" copy of the *Document* means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise *Transparent* file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not *Transparent*. An image format is not

Transparent if used for any substantial amount of text. A copy that is not "*Transparent*" is called "**Opaque**".

- Examples of suitable formats for *Transparent* copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG.
- *Opaque* formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this *License* requires to appear in the title page. For works in formats which do not have any title page as such, "*Title Page*" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the *Document* to the public.

A section "**Entitled XYZ**" means a named subunit of the *Document* whose title either is precisely *XYZ* or contains *XYZ* in parentheses following text that translates *XYZ* in another language. (Here *XYZ* stands for a specific section name mentioned below, such as "**Acknowledgments**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the *Document* means that it remains a section "*Entitled XYZ*" according to this definition.

The *Document* may include **Warranty Disclaimers** next to the notice which states that this *License* applies to the *Document*. These *Warranty Disclaimers* are considered to be included by reference in this *License*, but only as regards disclaiming warranties: any other implication that these *Warranty Disclaimers* may have is void and has no effect on the meaning of this *License*.

2. VERBATIM COPYING

You may copy and distribute the *Document* in any medium, either commercially or non commercially, provided that this *License*, the copyright notices, and the license notice saying this *License* applies to the *Document* are reproduced in all copies, and that you add no other conditions whatsoever to those of this *License*. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the *Document*, numbering more than 100, and the *Document's* license notice requires *Cover Texts*, you must enclose the copies in covers that carry, clearly and legibly, all these *Cover Texts*: *Front-Cover Texts* on the front cover, and *Back-Cover Texts* on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the *Document* and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute *Opaque* copies of the *Document* numbering more than 100, you must either include a machine-readable *Transparent* copy along with each *Opaque* copy, or state in or with each *Opaque* copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete *Transparent* copy of the *Document*, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of *Opaque* copies in quantity, to ensure that this *Transparent* copy will remain thus accessible at the stated location until at least one year after the last time you distribute an *Opaque* copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the *Document* well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the *Document*.

4. MODIFICATIONS

You may copy and distribute a *Modified Version* of the *Document* under the conditions of sections 2 and 3 above, provided that you release the *Modified Version* under precisely this *License*, with the *Modified Version* filling the role of the *Document*, thus licensing distribution and modification of the *Modified Version* to whoever possesses a copy of it. In addition, you must do these things in the *Modified Version*:

- A. Use in the *Title Page* (and on the covers, if any) a title distinct from that of the *Document*, and from those of previous versions (which should, if there were any, be listed in the *History* section of the *Document*). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the *Title Page*, as authors, one or more persons or entities responsible for authorship of the modifications in the *Modified Version*, together with at least five of the principal authors of the *Document* (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the *Title Page* the name of the publisher of the *Modified Version*, as the publisher.
- D. Preserve all the copyright notices of the *Document*.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the *Modified Version* under the terms of this *License*, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of *Invariant Sections* and required *Cover Texts* given in the *Document's* license notice.
- H. Include an unaltered copy of this *License*.
- I. Preserve the section *Entitled "History"*, *Preserve its Title*, and add to it an item stating at least the title, year, new authors, and publisher of the *Modified Version* as given on the *Title Page*. If there is no section *Entitled "History"* in the *Document*, create one stating the title, year, authors, and publisher of the *Document* as given on its *Title Page*, then add an item describing the *Modified Version* as stated in the previous sentence.
- J. Preserve the network location, if any, given in the *Document* for public access to a *Transparent* copy of the *Document*, and likewise the network locations given in the *Document* for previous versions it was based on. These may be placed in the *"History"* section. You may omit a network location for a work that was published at least four years before the *Document* itself, or if the original publisher of the version it refers to gives permission.
- K. For any section *Entitled "Acknowledgments"* or *"Dedications"*, *Preserve the Title* of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.
- L. Preserve all the *Invariant Sections* of the *Document*, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section *Entitled "Endorsements"*. Such a section may not be included in the *Modified Version*.
- N. Do not retitle any existing section to be *Entitled "Endorsements"* or to conflict in title with any *Invariant Section*.
- O. Preserve any *Warranty Disclaimers*.

If the *Modified Version* includes new front-matter sections or appendices that qualify as *Secondary Sections* and contain no material copied from the *Document*, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of *Invariant Sections* in the *Modified Version's* license notice. These titles must be distinct from any other section titles.

You may add a section *Entitled "Endorsements"*, provided it contains nothing but endorsements of your *Modified Version* by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a *Front-Cover Text*, and a passage of up to 25 words as a *Back-Cover Text*, to the end of the list of *Cover Texts* in the *Modified Version*. Only one passage of *Front-Cover Text* and one of *Back-Cover Text* may be added by (or through arrangements made by) any one entity. If the *Document* already includes a

cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the *Document* do not by this *License* give permission to use their names for publicity for or to assert or imply endorsement of any *Modified Version*.

5. COMBINING DOCUMENTS

You may combine the *Document* with other documents released under this *License*, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the *Invariant Sections* of all of the original documents, unmodified, and list them all as *Invariant Sections* of your combined work in its license notice, and that you preserve all their *Warranty Disclaimers*.

The combined work need only contain one copy of this *License*, and multiple identical *Invariant Sections* may be replaced with a single copy. If there are multiple *Invariant Sections* with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of *Invariant Sections* in the license notice of the combined work.

In the combination, you must combine any sections *Entitled "History"* in the various original documents, forming one section *Entitled "History"*; likewise combine any sections *Entitled "Acknowledgments"*, and any sections *Entitled "Dedications"*. You must delete all sections *Entitled "Endorsements"*.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the *Document* and other documents released under this *License*, and replace the individual copies of this *License* in the various documents with a single copy that is included in the collection, provided that you follow the rules of this *License* for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this *License*, provided you insert a copy of this *License* into the extracted document, and follow this *License* in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the *Document* or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the *Document* is included in an aggregate, this *License* does not apply to the other works in the aggregate which are not themselves derivative works of the *Document*.

If the *Cover Text* requirement of section 3 is applicable to these copies of the *Document*, then if the *Document* is less than one half of the entire aggregate, the *Document's Cover Texts* may be placed on covers that bracket the *Document* within the aggregate, or the electronic equivalent of covers if the *Document* is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the *Document* under the terms of section 4. Replacing *Invariant Sections* with translations requires special permission from their copyright holders, but you may include translations of some or all *Invariant Sections* in addition to the original versions of these *Invariant Sections*. You may include a translation of this *License*, and all the license notices in the *Document*, and any *Warranty Disclaimers*, provided that you also include the original English version of this *License* and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this *License* or a notice or disclaimer, the original version will prevail.

If a section in the *Document* is *Entitled "Acknowledgments"*, *"Dedications"*, or *"History"*, the requirement (section 4) to

Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the *Document* except as expressly provided under this *License*. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this *License*.

However, if you cease all violation of this *License*, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this *License* (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this *License*. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the *GNU Free Documentation License* from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the *License* is given a distinguishing version number. If the *Document* specifies that a particular numbered version of this *License* "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the *Document* does not specify a version number of this *License*, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the *Document* specifies that a proxy can decide which future versions of this *License* can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the *Document*.

11. RELICENSING

"**Massive Multiauthor Collaboration Site**" (or "**MMC Site**") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "*Massive Multiauthor Collaboration*" (or "*MMC*") contained in the site means any set of copyrightable works thus published on the MMC site.

"**CC-BY-SA**" means the *Creative Commons Attribution-Share Alike 3.0* license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"**Incorporate**" means to publish or republish a *Document*, in whole or in part, as part of another *Document*.

An MMC is "**eligible for relicensing**" if it is licensed under this *License*, and if all works that were first published under this *License* somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this *License* in a document you have written, include a copy of the *License* in the document and put the following copyright and license notices just after the title page:

- Copyright (©) YEAR YOUR NAME.
- Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
- A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have *Invariant Sections*, *Front-Cover Texts* and *Back-Cover Texts*, replace the "with...Texts." line with this:

- with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have *Invariant Sections* without *Cover Texts*, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains non trivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the *GNU General Public License*, to permit their use in free software.