libIDL2

Andrew T. Veliath

1 Overview

libIDL is a library licensed under the GNU LGPL for creating trees of CORBA Interface Definition Language (IDL) files, which is a specification for defining portable interfaces. libIDL was initially written for ORBit (the ORB from the GNOME project, and the primary means of libIDL distribution). However, the functionality was designed to be as reusable and portable as possible.

It is written in C, and the aim is to retain the ability to compile it on a system with a standard C compiler. Preprocessed parser files are included so you are not forced to rebuild the parser, however an effort is made to keep the parser and lexer compatible with standard Unix yacc and lex (although bison and flex are more efficient, and are used for the preprocessed parsers in the distribution).

With libIDL, you can parse an IDL file which will be automatically run through the C preprocessor (on systems with one available), and have detailed error and warning messages displayed. On a compilation without errors, the tree is returned to the custom application. libIDL performs compilation phases from lexical analysis to nearly full semantic analysis with some optimizations, and will attempt to generate meaningful errors and warnings for invalid or deprecated IDL.

libIDL exports functionality used to generate detailed conforming error and warning messages in gcc-like format, and also comes with a default backend to generate IDL into a file or string (useful for customized messages or comments in the output). The IDL backend is complete enough that most generated IDL can be reparsed by libIDL without errors. libIDL returns separate syntax and namespace trees, and includes functionality to hide syntactical information from the primary tree, while keeping it accessible through the namespace for type information and name lookup.

Optional extensions to standard IDL can be enabled using parse flags. These include node properties, embedded code fragments, and XPIDL. Nodes can also have declarations tags which assign particular attributions to certain IDL constructs to further facilitate custom applications.

2 Usage

The following C program using libIDL will parse an IDL file and print the Repository IDs of the interfaces in the IDL module.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <libIDL/IDL.h>

gboolean
print_repo_id (IDL_tree_func_data *tfd, gpointer user_data)
{
char *repo_id = NULL;

if (IDL_NODE_TYPE (tfd->tree) == IDLN_INTERFACE)
repo_id = IDL_IDENT_REPO_ID (IDL_INTERFACE (tfd->tree).ident);

if (repo_id)
printf ("%s\n", repo_id);

return TRUE;
}

int
main (int argc, char *argv[])
{
IDL_tree tree;
IDL_ns ns;
char *fn;
int rv;

if (argc < 2) {
fprintf (stderr, "usage: %s <file>\n", argv[0]);
exit (1);
}
fn = argv[1];

rv = IDL_parse_filename (fn, NULL, NULL, &tree, &ns, 0, IDL_WARNING1);

if (rv == IDL_ERROR || rv < 0) {
if (rv < 0)
perror (fn);
exit (1);
}
IDL_tree_walk_in_order (tree, print_repo_id, NULL);
IDL_ns_free (ns);
IDL_tree_free (tree);
```

```
return 0;
}
```

3 Reference

4 Data Types

- IDL_tree

  A semi-opaque tree which encapsulates an IDL tree node. Must be freed with IDL_tree_free (see Chapter 5 [Functions], page 9).

- IDL_ns

  A semi-opaque structure which encapsulates the IDL module namespace. Must be freed with IDL_ns_free (see Chapter 5 [Functions], page 9).

- IDL_msg_callback

  Defined as typedef int (*IDL_msg_callback)(int LEVEL, int NUM, int LINE, const char *NAME, const char *ERR). A function of this type can be optionally passed to IDL_parse_filename to be called when a parse warning or error occurs.

- IDL_tree_func

  Defined as typedef gboolean (*IDL_tree_func) (IDL_tree_func_data *TREE_FUNC_DATA, gpointer DATA). A function of this type is passed to IDL_tree_walk_in_order to traverse the tree. TREE_FUNC_DATA contains an up traversal hierarchy of the current traversal, as well as some state information. The current node being processed is given by TREE_FUNC_DATA->tree.

5 Functions

- Function: int IDL_parse_filename (const char *NAME, const char *CPP_ARGS, IDL_msg_callback CALLBACK, IDL_tree *TREE, IDL_ns *NS, unsigned long FLAGS, int MAX_MESSAGE_LEVEL)

  Parse an file containing an IDL definition into a parse tree. Returns IDL_SUCCESS if successful, or IDL_ERROR if there was a parse error. If -1 is returned, errno will be set accordingly. Usually, if IDL_ERROR is returned, all one needs to do is exit with a non-zero status, since libIDL will probably have made the reason for failure explictly known.

  - NAME: required, specifies the filename to be parsed.
  - CPP_ARGS: optional, if non-NULL, specifies extra arguments to pass to the C preprocessor. The most common type of string would be in the form of -I<dir> to include additional directories for file inclusion search, or defines in the form of -D<define>=<value>.
  - CALLBACK: optional, if non-NULL, this function will be called when a warning or error is generated (see Chapter 4 [Data Types], page 7). If not given, warnings and errors will be sent to stderr. All errors and warning, including callbacks, are subject to MAX_MESSAGE_LEVEL as described below.
  - TREE: optional, if non-NULL, points to an IDL_tree * to return the generated tree which must be freed with IDL_tree_free. If NULL, the tree is freed and not returned.
  - NS: optional, if non-NULL, points to an IDL_ns * to return the namespace tree which must be freed with IDL_ns_free. If NULL, the tree is freed and not returned. If TREE is NULL, then NS must also be NULL, since the namespace is created as the AST is generated.
  - FLAGS: optional, specifies extra flags for parsing or 0. The various flags are described here.
  - General Parse Flags
    - IDLF_NO_EVAL_CONST: instructs the parser not to evaluate constant expressions.
    - IDLF_COMBINE_REOPENED_MODULES: instructs the parser to combine modules defined later in the IDL code in the first module node in the tree.
    - IDLF_PREFIX_FILENAME: instructs the parser to prefix the filename to the namespace.
    - IDLF_IGNORE_FORWARDS: instructs the parser to not try to resolve and print messages for unresovled forward declarations.
    - IDLF_PEDANTIC: instructs the parser to display stricter errors and warnings.
    - IDLF_INHIBIT_TAG_ONLY: only tag inhibited nodes, do not remove them. Use IDL_tree_remove_inhibits to remove them at a later time.
    - IDLF_INHIBIT_INCLUDES: causes libIDL to automatically inhibit IDL trees in included files.

  - Syntax Extension Flags
    - IDLF_TYPECODES: understand the 'TypeCode' keyword extension.
    - IDLF_XPIDL: enable XPIDL syntax.
    - IDLF_PROPERTIES: enable support for node properties.
    - IDLF_CODEFRAGS: enable support for embedded code fragments.
  - MAX_MESSAGE_LEVEL:

    This specifies the maximum message level to display. Possible values are -1 for no messages, IDL_ERROR for errors only, or IDL_WARNING1, IDL_WARNING2 and IDL_WARNING3. A typical value is IDL_WARNING1, which will limit verbosity. IDL_WARNINGMAX is defined as the value in which all messages will be displayed.

- Function: void IDL_tree_walk_in_order (IDL_tree ROOT, IDL_tree_func FUNC, gpointer DATA)

  Walks an IDL_tree, calling FUNC for every node. If the FUNC returns TRUE for a particular node, that particular node will also be traversed, if FALSE is returned, that particular node will be skipped, in the assumption that the function has taken care of it.
  - ROOT: required, specifies the IDL_tree to traverse.
  - FUNC: required, specifies the callback function (see Chapter 4 [Data Types], page 7).
  - DATA: optional, specifies the callback data.
- Function: void IDL_tree_free (IDL_tree TREE)

  Frees the memory associated with TREE.
- Function: void IDL_ns_free (IDL_ns NS)

  Frees the memory associated with NS.

6 Extensions

This page documents extensions to standard IDL which libIDL will understand. To maintain portability, it is recommended that these extensions are only used with some sort of C preprocessor define so they can be conditionally omitted.

- __declspec (<spec>)

  This token assigns special attributions to particular IDL constructs.

  - inhibit

    If __declspec (inhibit) is placed before a definition or export, that module or interface definition will be removed from the tree. The tree is only deleted when the IDL_ns component is freed, so it can be traversed from the namespace component for extended information, but will be omitted from the primary tree.

7 Tree Structure

8 Function Index

Table of Contents