libcsdbg

1.27

Generated by Doxygen 1.8.5

Wed Apr 2 2014 17:38:01

Contents

Chapter 1

Overview

1.1 Introduction

1.1.1 Description

Project **libcsdbg** is a **C++ exception (and generic) stack tracer**. When an exception is thrown, caught and handled, libcsdbg offers the tools to create, process and output the exception stack trace, the path the exception has propagated up the call stack, unwinding it, up to the section were it was handled. The traces are fully detailed with demangled function signatures and additional **addr2line** information (the source code file and line that each function was called). Libcsdbg will perform flawlessly with single-thread and multi-thread (or multi-process) programs, with any **dynamically linked shared object** (**DSO** - Dynamic Shared Object) and with any statically linked library. Additionally, libcsdbg helps the developer/tester create sophisticated stack traces **of any thread, at any given moment** during the execution of a process, or create **a dump of the call stacks of all threads** as a snapshot of the runtime call graph. This is useful in cases of fatal errors, reception of terminating signals (such as SIGSEGV) or process abortion/termination. It can also prove a great tool to detect and resolve deadlocks.

In an object oriented programming paradigm, it's usually not enough to catch and handle an exception. It's essential to know at runtime, *where* in the code and under *what circumstances* the exception was thrown and also the path that the exception followed from the throw point up to the point where it was caught. Although much of this information can be embedded inside the exception object, this has several drawbacks, namely:

- It's not a complete solution
- It needs lots of code to keep track (the exception objects must have small footprint)
- It can't function with STL exceptions and integral types
- It's difficult to make it generic and portable

In Java, method **java.lang.throwable.prinStackTrace** can provide this additional exception information, because the Java language is interpreted, not native, as C++ is. To implement this with **GNU g++**, libcsdbg exploits the compiler code generation features, to inject code for **function instrumentation**. Compiling code with the *-finstrument-functions* flag, commands g++ to inject calls to the instrumentation functions (**__cyg_profile_func_enter**, **__cyg_profile_func_exit**) at the beginning and end of all the instrumented user functions, respectively. The compiled code is then linked with libcsdbg that implements these two functions.

At runtime, libcsdbg uses these functions to transparently **simulate** the call stack of each thread of execution within the debugged process, adding only **minimum overhead**. When an exception is thrown the actual thread stack is unwound until a catch section is found or until the process aborts, because of the unhandled exception. The stack simulator detects there is an exception being thrown and does not unwind, so the libcsdbg user can obtain the **exception stack trace** and print, store or process the trace data any way seem fit.

Libcsdbg **transparently loads the symbol tables** of the executable and of any chosen dynamic shared objects, **demangles function symbols** to complete signatures and binds function names to **runtime addresses** (even for

relocatable, position independent, **DSO** symbols). Libcsdbg can use all the well-known objective code file formats (a.out, elf, coff, ecoff e.t.c), works with 32 and 64 bit systems and with both big and little endian architectures. The library API can easily be used as the base for your own instrumentation code. The library exports (through its namespace, **csdbg**) a **rich user API** to create and process stack traces, to store them to files or send them (via **ethernet** or **serial** media) to a remote debug workstation, to track threads and lookup process symbols, to process command line arguments and shell variables that configure the library at runtime and a host of other useful utility functions. One of the most useful library tools is the csdbg::plugin API. Using the methods exported from this class the user can register multiple function profilers (either inline or modular) to be run by libcsdbg. A g++ shortcoming is that the names of the default instrumentation functions are **hard-coded**, so only a unique implementation of these can exist at linkage time, therefore only one profiler can be used at a time. The plugin class is the solution to this problem.

The library is also equipped with a parser, usable with generic automata (POSIX extended regular expressions) and grammars. The default grammar specifies a C++ stack trace. Using this default parser a trace can be tokenized, processed and printed using custom syntax highlighters. The library supports a default stack trace **syntax highlighter** for **VT100** compatible terminals (XTerm, RXVT, GNOME terminal e.t.c) with **configurable styles**.

The following is a sample exception stack trace (produced by one of the example executables that are shipped withing package libcsdbg-extra):

```
at thread 0x7f94d9c79740 {
  at main
  at csdbg_extra::level1(char const*, unsigned char) (csdbg_step1.cpp:149)
  at csdbg_extra::level2(char const*, unsigned short) (csdbg_step1.cpp:121)
  at void csdbg_extra::level3<csdbg::string>(csdbg::string&, unsigned int&) (csdbg_step1.cpp:116)
  at csdbg_extra::level4(char const*, unsigned long long volatile*, void (*)(double)) (csdbg_step1.cpp:102)
  at csdbg_extra::dso_main(char const*) (csdbg_step1.cpp:89)
  at csdbg_extra::dso_inner(char const*) (libcsdbg_test.cpp:53)
}
```

1.1.2 Features

This is a comprehensive, albeit not complete, list of libcsdbg features:

- Create sophisticated exception stack traces (with addr2line support)
- Works with multi-thread processes and with multi-process programs
- Create detailed **POSIX thread** stack traces (with addr2line support)
- Full support for **Position Independent Code (PIC)** (for DSO)
- Works with **generic throwables and user defined exception types**
- Support for most of the objective code formats (elf, a.out, ecoff e.t.c)
- Support for both **32 and 64 bit** systems
- Support for both **big and little endian CPUs**
- Compiled/tested for x86, x86_64, ARM, AVR32, Leon

Project libcsdbg exports an extended, well documented and reusable **library API** to provide:

- Easy and minimal code interface, transparent library integration
- Easy library **runtime configuration**
- Transparent but configurable loading of symbol tables of any executable, DSO or other module
- Instrumentation algorithms that only add **minimal overhead** and minimal **memory footprint**
- The high level API provides **thread safety**
- Smart API to output trace or generic data to files, network peers and serial lines, using **LDP (Libcsdbg Debug Protocol)** and minimal code

- An evergrowing API to support **multiple output interfaces** through **different media** (ethernet, serial, UNIX domain sockets, pipes, e.t.c)
- Support for multiple function profilers either inline or loaded from a **DSO module (plugin)**
- Stack trace **syntax highlighting** for **VT100** console output
- Custom dictionaries and grammars for the syntax highlighters
- Custom **VT100** configurable **styles**
- Library API can be used as the base for custom instrumentation code

Libcsdbg is currently available for **GNU/Linux/uClibc** platforms. There is development going on to provide a version for **Windows** (32 and 64 bit) systems (through **MinGW**) and Unix, **FreeBSD/OpenBSD/NetBSD** systems along with an **GNU autotools**-based build system and plugins for various **IDE** (Eclipse, Code::Blocks, e.t.c).

Project **jTracer** is a libcsdbg sister project, a portable **LDP** server implemented with Java. Each application that uses the libcsdbg LDP API can implement a jTracer client. This can be essential for **cross-platform** development with **embedded** devices and **development boards**. Most often target platforms such as these don't have a screen or other resources to visualize output and data collection during the development and debugging cycles is controlled at a workstation through ethernet or serial ports. Instead of cluttering the IDE, console or debugger with trace data, LDP is designed to isolate these generated data, collect them with jTracer (**even from multiple target hosts with diverse architectures**) and provide an easy way to navigate through them.

1.1.3 Licence and details

Libcsdbg is currently published as an Open Source project, licenced under the **GNU Library or Lesser General Public License version 3.0 (LGPLv3)**. **Tasos Parisinos** develops and maintains libcsdbg and the project documentation, while **Antonis Kalamaras** develops and maintains the build system. Version numbering uses three numbers, major, minor and subminor. **The current library version is 1.27**. Major library updates are related with multiple feature addition, significant changes in the library interface, major bug fixes or overall optimization. Minor library updates usually add minor features (or major features that were *almost* ready to be included in the major release, but didn't), minor bug fixes, typos and documentation updates or entities that need to be tested to gather feedback. Subminor releases are stable beta snapshots of the current code development status, with the last additions since the last minor update.

The following are locations essential to the development of libcsdbg:

- **Homepage**
- **libcsdbg@sourceforge**
- **libcsdbg@freecode**
- **Downloads**
- **Forum**
- **Support**
- **jTracer homepage**
- **jTracer@freecode**

1.2 Obtaining the sources

The latest version of libcsdbg is 1.27 (release date April 3rd 2014).

This is a major maintenance version considered as the final form of all current features and documentation. Therefore, to fully exploit the library features you should read the latest documentation overview, the API reference or the pdf user manual, that fully describe all the latest major API changes and all of the new features.

The latest **source** distribution is **libcsdbg-1.27.tar.bz2**

Alternatively, if you've already got the previous version (1.26) you can patch it to upgrade it to the latest release:

- **libcsdbg-1.26.tar.bz2**
- **upgrade-1.27.patch**

If you have both these files in the same directory, to apply the patch, execute:

```
tar -xjpf libcsdbg-1.26.tar.bz2
cd libcsdbg-1.26
patch -p1 < ../upgrade-1.27.patch
```

The project **documentation** can be recreated (with **doxygen**) in the source code distributions. The doxygen version used to produce the documentation for this version is **1.8.5**. It can also be separately downloaded in **HTML** or as a single **hyperlinked PDF** document (for the user manual):

- **api-ref-libcsdbg-1.27.tar.bz2**
- **user-manual-libcsdbg-1.27.pdf**

Project code examples, tutorials, example DSO and instrumentation plugin modules have moved to a new package **libcsdbg-extra-1.25.tar.bz2**

Finally, if you want an overview of **all** project files check the libcsdbg **repository** at sourceforge.net. Click on any file's info icon and obtain its **SHA1** or **MD5** checksum to verify the file you've downloaded.

Note

After you download any project file, remember to subscribe/follow libcsdbg in order to receive notification about new releases, updates, news, announcements and other libcsdbg related material in your mailbox. The easiest way to do so is to visit the **libcsdbg project page at freecode.com** and click on *'follow'*

1.3 Installation

1.3.1 Compile the sources (Unix/Linux)

If you downloaded the source distribution, you need at least the following to build the library, the example executables and the project documentation:

- **GNU g++**
- **GNU make**
- **GNU binutils** (at least libbfd.so, strip and addr2line)
- UNIX standard tools such as rm, echo, touch, mkdir, cd, cp, ln, mv, grep, id, tar, ldconfig, sudo e.t.c
- You will need **doxygen** and **graphviz**, to recreate the project documentation (doxygen-1.8.5)

Unpack the archive, unless you already have done that. If you want to recreate the documentation you must unpack the tarball in /**devel** (or change the paths in the doxygen configuration files (doc/docgen_html and doc/docgen_tex) with a simple text editor or **doxywizard**:

```
mkdir -p /devel
mv libcsdbg-1.27.tar.bz2 /devel
cd /devel
tar -xjpf libcsdbg-1.27.tar.bz2
```

Then compile and install it:

```
cd libcsdbg-1.27
sudo sh ./build
```

If you install as a simple user (not root) use of **sudo** is essential if you want to keep the default prefix (/**usr**/**local**). Installing in another prefix may not need root privilege, but the dynamic loader configuration file (**ld.so.conf**) will not be updated if sudo is not used.

Currently **GNU autotools** are not supported but this is under development. The *build* script should compile without problems and install libcsdbg library, headers and other project files with /**usr**/**local** as the prefix. For an overview of all *build* script options and modes of operation use:

```
sh ./build -h

Project libcsdbg installer
Usage: build [-c] [-u] [-m] [-s] [-d] [-h]

'build' will compile and install its target by default.
The following options change the default behaviour:

-c  Clear the source tree (make clean)
-u  Uninstall the package (make uninstall)
-m  Compile but don't install
-s  Don't parallelize compilation on multicore systems
-d  Create/update the documentation (make doc)
-h  Show this message
```

Although the *build* script can do all that you may need, to compile and install or uninstall, recreate the documentation or clear the source tree by calling the equivalent *Makefile* targets (in the **right** order), it is essential to describe these individual targets.

First of all clear the source tree from built binaries:

```
make clean
```

Compile the sources (the result binaries are stored in **./.build**):

```
make
```

Install the library, header files, miscellaneous resource files (stack trace syntax highlighter dictionaries, utility scripts e.t.c) and pkg-config file (under /**usr**/**local** by default):

```
make install
```

Optionally generate the documentation. This will create the **API reference** in **HTML** and the user manual in **LaTex** and in **hyperlinked PDF** (all generated under **./doc**). To generate the PDF version of the manual you will need a running distribution of **LaTex** and **pdflatex**, **makeindex** and **egrep**. To see the HTML documentation, just point a browser to *index.html* in the **./doc**/**api-ref-libcsdbg-1.27** folder. The PDF manual *refman.pdf* will be located in the user manual directory of the distribution (**./doc**/**user-manual-libcsdbg-1.27**). Just view and print it via the acrobat reader.

```
make doc
```

Finally, if you need to uninstall all libcsdbg-related files from your system:

```
make uninstall
```

To clear the source directory from built binaries **and** all the compiled documentation files (this target is not called from *build*):

```
make distclean
```

As said earlier, no **autotools** support exists for the project but this is under way. So, currently all you can do is edit *Makefile* to change some options that may give you a headache. The *Makefile* options you may want to alter are:

$PREFIX

Where to install libcsdbg (copy changes to include/config.hpp)

$PLATFORM

The target prefix you need to cross compile the package (for example *mips-linux-*). It is used as the prefix for various binaries in the toolchain

$IPATHS

Additional paths to search for header files

$DOPTS

Define various preprocessor macros such as debug or release mode, optimizations, include or discard modules e.t.c

$GOPTS

Various g++ options such as target machine architecture, C++ standard used throughout compilation e.t.c

The *Makefile* **$DOPTS** variable defines the main build configuration. Through this variable you can define which modules you want to include in the library and which features to support. The directives in **$DOPTS** are formatted as CSDBG_WITH_feature to add support for a module or feature, or CSDBG_WITHOUT_feature to exclude it:

_REENTRANT

Support thread safety

CSDBG_WITH_DEBUG

Include debugging code

CSDBG_WITH_COLOR_TERM

Include support for color terminals (for debug text only). This coloring of debug messages makes it easy to discriminate between the different debug levels

CSDBG_WITH_STREAMBUF

Include code for buffered output streams

CSDBG_WITH_STREAMBUF_FILE

Include code for buffered file output streams (this is valid only if the CSDBG_WITH_STREAMBUF directive is also defined)

CSDBG_WITH_STREAMBUF_TCP

Include code for buffered TCP/IP socket output streams (this is valid only if the CSDBG_WITH_STREAMBUF directive is also defined)

CSDBG_WITH_STREAMBUF_STTY

Include code for buffered serial tty output streams (this is valid only if the CSDBG_WITH_STREAMBUF directive is also defined)

CSDBG_WITH_PLUGIN

Include code for instrumentation plugins

CSDBG_WITH_HIGHLIGHT

Include code for trace C++ syntax highlighting

The complete library, with all its features enabled has a memory footprint of approximately 263Kb. The complete release library is marginally smaller (239Kb). If you keep only the core library functions and exclude all advanced features (buffered output streams, instrumentation plugins, trace syntax highlighter) the release library shrinks down

to ∼115Kb.

1.4 Troubleshooting

1.4.1 Bug reports

Bugs are tracked in the **tickets** section of the libcsdbg site at **sourceforge.net**. Before submitting a new bug, first search through **all** the tickets (open and closed), if the same bug has already been submitted by others. If you are unsure whether or not something is a bug, you may ask help on the **users forums** first (subscription is **not** required, the forums are moderated).

So, you think you found a bug? You should report it either on the user forums or by sending an email to the project admin **tasos42@users.sourceforge.net**.

If you send only a (vague) description of a bug you are usually not very helpful and it will cost much more time to figure out what you mean. In the worst-case, your bug report may even be completely ignored, so always try to include the following information in your bug report:

- The version of libcsdbg you are using (use *pkg-config –modversion libcsdbg* if you are not sure)
- The name and version number of your operating system (*uname -a*)
- The libcsdbg-related shell and CLI variables
- All libcsdbg output (to the console, files, network) that is specific to the problematic scenario

The easiest way for us to fix bugs is if you can attach a small example that demonstrates the problem you have to the bug report, so we can reproduce it on our machines. Please make sure the example is valid source code and that the problem is really captured by the example. If you intend to send more than one file please zip or tar the files together into a single file for easier processing.

You can (and are encouraged to) add a patch for a bug. If you do so please use **PATCH** as a keyword in the bug entry form or in the email subject. If you have ideas how to fix existing bugs and limitations please discuss them on the **users forums**. For patches please use *diff -uprN* or include the files you modified.

1.4.2 How to contribute

Donate time

You can contribute your time by helping with programming, testing and filing bug reports, improving documentation, translations or by answering questions on the mailing list. We always welcome users whose only contribution is simply using libcsdbg, giving us feedback on how to improve it and telling others about it. Thank you for supporting libcsdbg.

Donate money

If you don't have time to help but do find libcsdbg useful, then please consider making a financial donation. This will help to pay the bills and motivate us to continue working on libcsdbg. You can do so using the Paypal account indicated on the project site at sourceforge.net, or contact us for other payment methods.

1.5 Usage

1.5.1 Compiling with libcsdbg support

Integration with libcsdbg is transparent and simple, just compile your code with some g++ mandatory flags and link with libcsdbg (-lcsdbg). From these g++ flags the most prominent are *-finstrument-functions* and *-g[format]*. You may need to invoke:

```
pkg-config --cflags libcsdbg
```

```
-fPIC
-g
-finstrument-functions
-finstrument-functions-exclude-file-list=/usr/include
-finstrument-functions-exclude-file-list=iostream
-finstrument-functions-exclude-file-list=ios
-finstrument-functions-exclude-file-list=istream
-finstrument-functions-exclude-file-list=ostream
-finstrument-functions-exclude-file-list=/usr/local/include/csdbg
-I/usr/local/include
```

to see **all** the additional flags you must pass to the compiler. Similarly:

```
pkg-config --libs libcsdbg
```

```
-L/usr/local/lib
-lcsdbg
-ldl
-lbfd
-lpthread
```

will print the flags you must pass to the linker. You may need to add **$PREFIX/lib/pkgconfig** to your pkg-config path.

Stack traces created with libcsdbg can only contain calls to functions that are instrumented by libcsdbg. For example if you write code for an application and instrument that code with libcsdbg only these functions you instrument will appear in traces. If you link your application with third-party DSO, the DSO function calls will not appear in the trace by default! You need to recompile those DSO to add libcsdbg support. As a rule of thumb, you should recompile code to add support for libcsdbg **if that code throws exceptions** and if it is essential to you to see how those exceptions propagated inside the module call graph.

Even if you don't call a single API method, if libcsdbg is compiled with debug support, it is easy to see if everything linked properly by just running your application: libcsdbg will at least attempt to load its symbol table and it will print various debug messages

1.5.2 Configuring libcsdbg at runtime

Before you run a program linked with libcsdbg there are some things you may want to configure, apart from the dynamic linker path, to help libcsdbg decide what to load and what to ignore, as well as some other runtime configuration tokens.

The symbol table of the program is always loaded by libcsdbg. The user can select which DSO symbol tables to load and which to discard by declaring the **$CSDBG_LIBS** shell variable as a ':' delimited list of POSIX extended regular expressions. The absolute path of each DSO is matched against each regexp. If one matches then the symbol table of the DSO is loaded to the libcsdbg namespace and the DSO functions are instrumented throughout execution. If **$CSDBG_LIBS** is not set, all linked DSO symbol tables will be loaded. If it is set with a void value, all DSO are filtered out from instrumentation. You should only load DSO that are compiled with libcsdbg support. Loading non instrumented DSO is not a problem, apart from additional overhead. For example to run the example programs (in libcsdbg-extra) you must execute something like:

```
export CSDBG_LIBS=csdbg_test
```

Generally, to get the value of such shell variables, parsed to their components, you need to use the csdbg::util-::getenv method instead of the equivalent **libc** function.

Libcsdbg can also accept command line arguments. To avoid conflicts with the application and its command line arguments, the ones for libcsdbg are prefixed with **–csdbg-**. To feed those arguments to libcsdbg you must call, early in your code the csdbg::util::init method. This method parses the command line argument vector, identifies the arguments for libcsdbg, canonicalizes (removes the –csdbg- prefix), stores them internally and removes them from

the vector, so the application never knows they were there. These arguments may just be flags or pass values to the library. Currently no such argument is utilized and the mechanism is there only for the users to use, but in the future the runtime configuration of libcsdbg will be affected by them and/or a configuration file.

If you plan to develop and use any instrumentation plugin DSO modules, the best practice is to store them in **$PREFIX/lib/modules/libcsdbg**. At runtime, for the dynamic linker to be able to locate them you must add this path to the linker search path either by editing **ld.so.conf** or by adding the path to the **$LD_LIBRARY_PATH** shell variable. The examples and tutorials that demonstrate the plugin API work in exactly this way.

1.5.3 Using the csdbg::tracer API

To be able to do anything with libcsdbg, even when you use its higher level objects (csdbg::filebuf, csdbg::tcpsockbuf e.t.c) you first need to obtain a csdbg::tracer object. The word is obtain, not create, because the class constructors, destructor and assignment operator are not public. The class itself provides you with interfacing tracer objects. You must call the csdbg::tracer::interface static method, as seen in the following snippet, to obtain a pointer to an interfacing tracer object.

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  ; // abort
```

If this call returns NULL, that means that library initialization failed to load at least one symbol table. You should check the debug and generic output to see why this occured. For example this can happen if all modules and the executable itself are stripped of symbols! You don't need to release or otherwise clear the obtained interface object after you are done. The library will take care of it internally.

Read the code and comments in libcsdbg-extra package examples and documentation. This will help you understand how to use libcsdbg, how to interface with it with code and use its facilities. The interface is really very simple and usually it takes **one or two lines of code to produce a trace!**

1.5.3.1 Exception stack traces

The best place to create and output (or process) an exception stack trace is in the **catch** section that is handling it. There are two ways to output such a trace. The first is to feed the tracer object to an STL output stream, using the insertion operator, for example:

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  return;

try {
  ...
}

catch (exception &x) {
  std::cout << x << "\r\n" << *iface << "\r\n";
}

iface->unwind();
```

This is the simplest way possible! Although inside the tracer object lots of things are happening behind the scenes, things that can throw exceptions, you don't need to worry, they are taken care of internally. Moreover the output to an STL stream is **atomic** in the scope of the library (no other library output method will interleave). The other way to get the stack trace of the currently handled exception is to call the interface tracer object to store it in a csdbg::string buffer, as in the following example:

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  return;
```

```
try {
  ...
}

catch (exception &x) {
  try {
    string buf;
    iface->trace(buf);
  }

  catch (...) {
  }
}

iface->unwind();
```

As you see in this last example, creating a stack trace in this way may throw exceptions that are the caller's responsibility to handle. The call to csdbg::tracer::unwind is only **mandatory** when the exception stack trace is ignored (in order to dispose that stored trace). If you don't properly unwind the simulated stack, the stored trace will mess with the next attempt to obtain a stack trace. Nevertheless, if the trace was actually created, a call to unwind doesn't affect the tracer object state at all (nothing to dispose), so it is not an error to call it once or even more times even when the trace was produced. Mind you, all calls to csdbg::tracer::trace even from the *std::ostream* insertion operator **unwind** the stack on their own, even if they fail to produce a trace due to some error.

A final point is, if you code a function to do the trace creation and output, this function must not be instrumented, otherwise it will corrupt the exception stack trace. Here's an example:

```
using namespace csdbg;

void foo(exception&) __attribute((no_instrument_function));

void foo(exception &x)
{
  tracer *iface = tracer::interface();
  if ( unlikely(iface == NULL) )
    return;

  try {
    string buf;
    iface->trace(buf);
  }

  catch (...) {
  }
}

...

try {
  ...
}

catch (exception &x) {
  foo(x);
}

iface->unwind();
```

Due to C++ polymorfism **all csdbg::string subclasses** (the buffered output stream types of the library, and the stack trace parser/highlighter) work in the same way!

1.5.3.2 Thread stack traces

You can create a thread stack trace anywhere in your code, at any moment of execution. You can create a stack trace for the currently running thread or for any process thread, running, stopped, resumed or blocked with a call at the two argument variant of csdbg::tracer::trace as in the next example:

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  return;

string buf;
iface->trace(buf, pthread_self());
```

In case you have stored the thread ID or name of any thread, you then can obtain its stack trace from any other thread. Although the libcdbg higher lever library calls are thread safe, you should be vigilant for data races that **you** may create. Another consideration, is that the above code may throw exceptions, that of course you need to handle somewhere. Libcsdbg maintains a list of simulated threads in a csdbg::process object obtained with csdbg::tracer::proc. There are four methods to obtain the csdbg::thread object pointer for a thread (all thread safe):

- **csdbg::process::current_thread** - Get the current thread descriptor
- **csdbg::process::get_thread(pthread_t)** - Get the thread descriptor for a specific ID
- **csdbg::process::get_thread(const i8*)** - Get the thread descriptor for a specific name. Threads are anonymous by default, libcsdbg adds a name attribute to thread handlers for easy identification. You may set or retreive a thread's name using csdbg::thread::set_name and csdbg::thread::name methods respectively
- **csdbg::process::get_thread(u32)** - Get the thread descriptor at an offset in the thread enumerator

This is an example of creating a stack trace for the third thread in the list:

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  return;

thread *thr = iface->proc()->get_thread(2);
string buf;
iface->trace(buf, thr->handle());
```

Another version of thread stack tracing is to create a batch trace for all the threads within a process. This feature is delivered by calling the method csdbg::tracer::dump. This can prove **extremely** helpful when you want to debug an application that has totally crashed or deadlocked. Instead of loading and debugging the **core dump** (**if** the OS supports this!) you can solve the problem quicker by taking a look at the thread stack traces, to see what each was doing (and what the then executing thread was doing) before the program crashed. Here's an example scenario:

1. The application setups and registers a handler for the **SIGSEGV** signal
2. A thread execution creates a **memory access violation**
3. The process receives a SIGSEGV because of the segmentation fault
4. The registered handler creates a stack trace dump for all the threads and stores it in a file using a csdbg::filebuf object
5. The process aborts

There are additional methods in the csdbg::process public API, they are designed to be called by the instrumentation functions (these functions are not part of the **csdbg** namespace, so they too need to obtain a tracer interface and use its public methods). These methods are used for thread management and symbol lookup and they **may** be called by the library user. One of them, csdbg::process::cleanup_thread is specifically for the library user. This method should be called from **thread cancellation handlers** to release resources. If you don't cleanup the thread descriptor, though it becomes useless when the actual thread has exited, it continues to occupy memory and will also inject junk, empty traces in dumps or in explicit trace requests. This method may also be called just before a thread exits. If you don't call cleanup_thread, it is not a bug, it's just poor coding. When you have called this method you should call no more methods for the deleted thread ID. Again, this will be handled by the library, but again, it is poor coding.

1.5.4 The Libcsdbg Debug Protocol (LDP)

Ok, you have created some traces and learned how to output them to a console or store them in buffers. **LDP (Libcsdbg Debug Protocol)** is an application level protocol designed to transmit trace data, together with other, process, thread and exception descriptive data, through any media and transport layer protocol (**TCP/IP**, **UDP/IP**

e.t.c). LDP is a **unidirectional**, **client-server** protocol. Unidirectional because only the client sends (trace) data to the server, the server need not aknowledge or reply in any way. The only role that the server plays in LDP, is to collect these data from multiple (and possibly of variant architecture and OS) client peers and store them or let a user collectively navigate through them and process them. LDP is designed as a message oriented protocol (like **HTTP**). A single connection, kept alive, can be used to send multiple messages but this is not mandatory, it is up to the implementation. Nevertheless, two LDP messages are not connected in any way and can be thought as two separate LDP sessions. This is a pseudo-BNF description of a message:

- The message consists of a **head** and a **body** separated by an empty line (\r\n)
- The **head** consists of a number of **headers**
- Each **header** is formatted as '**key**: *value*\r\n'
- Header numeric values are **hexadecimal** (no 0x prefix)
- The message **body** is the whole trace
- The message is terminated by an empty line

This is the generic layout:

```
key1: value1\r\n
key2: value2\r\n
...
keyN: valueN\r\n
\r\n
trace data\r\n
\r\n
```

The mandatory protocol headers are for:

- executable absolute path
- process ID
- thread ID
- timestamp (in microseconds)

The non mandatory protocol headers are for:

- exception data
- other, user and OEM headers

The following is an LDP message created by one of the example executables in package libcsdbg-extra:

```
path: /usr/local/bin/csdbg_step6
pid: 3b3
tid: 7f9870ca8700
tstamp: 4f264e66740f9

at child_1 thread (0x7f9870ca8700) {
  at csdbg_extra::pthread_main(void*)
  at csdbg_extra::level1(char const*, unsigned char) (csdbg_step6.cpp:205)
  at csdbg_extra::level2(char const*, unsigned short) (csdbg_step6.cpp:187)
  at void csdbg_extra::level3<csdbg::string>(csdbg::string&, unsigned int&) (csdbg_step6.cpp:182)
  at csdbg_extra::level4(char const*, unsigned long long volatile*, void (*)(double)) (csdbg_step6.cpp:169)
}
```

Project **jTracer** is a libcsdbg sister project, a portable LDP server implemented with Java. Each application that uses the libcsdbg LDP API can implement a jTracer client. This can be essential for **cross-platform** development with **embedded** devices and **development boards**. Most often target platforms such as these don't have a screen or other resources to visualize output and data collection during the development and debugging cycles is controlled at a workstation through ethernet or serial ports. Instead of cluttering the IDE, console or debugger with trace data, LDP is designed to isolate these generated data, collect them with jTracer (**even from multiple target hosts with diverse architectures**) and provide an easy way to navigate through them.

1.5.5 Buffered output streams

Subclassing the abstract class csdbg::streambuf is the standard way to create objects that output traces and other data to various media. A streambuf-derived object is both a string buffer (an object of class csdbg::string) and an output stream. The media that are supported are those that can be handled with an integer descriptor (files, character devices, terminals, sockets, pipes e.t.c). The libcsdbg project is currently shipped with three streambuf subclasses, csdbg::filebuf is used to output traces to files, csdbg::tcpsockbuf is used to transmit traces through a TCP/IP network and csdbg::sttybuf is used to send traces to serial devices. Other classes to support UDP/IP and Unix sockets, pipes, FIFOs and other will be added in the future or contributed by users. Class streambuf apart from providing the common base functionality it also implements a part of the **Libcsdbg Debug Protocol (LDP)**. These classes aren't thread safe, but class csdbg::streambuf implements basic stream locking methods.

1.5.5.1 Using csdbg::filebuf

A csdbg::filebuf object is a buffered output stream used to output **LDP (Libcsdbg Debug Protocol)** data (protocol headers and traces) or generic data to a file. This class is not thread safe, the caller must implement thread synchronization. A filebuf object inherits all the csdbg::string methods designed for text manipulation (append, clear, set e.t.c). You use these methods to process the buffer data, or the csdbg::tracer methods to store traces to the buffer. Then, you may open the underlaying file stream and flush/clear the buffer. You can re-fill and re-flush as many times as you wish before you close the filebuf. Closing is just the opposite of open, it doesn't release the object and its buffer, so you may re-open it. Based on the unique identifiers of the instrumented process, a filebuf object can assign file names in an unambiguous way. The steps to use a filebuf object are:

1. Create a filebuf object for a path
2. Open the file
3. Append/set text data in the buffer
4. Flush the buffer
5. Repeat steps 3-4 until completion
6. Close
7. Repeat steps 2-6 until completion
8. Release the object

Step 2 in this list can be inserted anywhere within steps 2-4, you don't need to open the file in order to process its buffer, it must be opened before you flush the buffer. To name files in an unambiguous way, method csdbg::filebuf-::unique_id can come in handy. It takes a printf-style format string (the default format is *%e_%p_%t_%s*) and uses the following specifiers:

- %e - executable name
- %a - executable absolute path
- %p - process ID
- %t - thread ID
- %s - timestamp (in microseconds)

The following is an example of using the filebuf class:

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  return;

/* Log a thread stack trace to a file, with a unique, trace describing name */
```

```
string *nm = filebuf::unique_id("%e_%p.trace");
filebuf fout(nm->cstr());

/* Add a header that describes the trace */
fout.header();
fout.append("\r\n");
iface->trace(fout, pthread_self());
fout.append("\r\n");

fout.open();
fout.flush();
fout.close();
```

1.5.5.2 Using csdbg::tcpsockbuf

A csdbg::tcpsockbuf object is a buffered output stream than can be used to implement the client side of **LDP (Libcsdbg Debug Protocol)** or a generic **TCP/IP client socket**. Nevertheless, this class is optimized for LDP and generally for **unidirectional** protocols. If you need to implement a **bidirectional** protocol you must subclass tcpsockbuf. This class is not thread safe, the caller must implement thread synchronization. A tcpsockbuf object inherits all the csdbg::string methods designed for text manipulation (append, clear, set e.t.c). You use these methods to process the buffer data, or the csdbg::tracer methods to store traces to the buffer. Then, you may connect the socket to its peer and flush/clear the buffer. You can re-fill and re-flush as many times as you wish before you close/disconnect the tcpsockbuf. Disconnecting (closing) or shutting down the socket is just the opposite of open/connect, it doesn't release the object and its buffer, so you may re-connect it. The steps to use a tcpsockbuf object are:

1. Create a tcpsockbuf object for a peer IP address and TCP port
2. Connect to the peer
3. Append/set text data in the buffer
4. Flush the buffer
5. Repeat steps 3-4 until completion
6. Disconnect
7. Repeat steps 2-6 until completion
8. Release the object

Step 2 in this list can be inserted anywhere within steps 2-4, you don't need to connect the socket in order to process its buffer, it must be connected before you flush the buffer. The following is an example of using the tcpsockbuf class:

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  return;

tcpsockbuf *client = NULL;
chain<string> *peer_info = util::getenv("CSDBG_PEER");
if ( likely(peer_info == NULL) )
  client = new tcpsockbuf(NULL);

else {
  i32 port = g_ldp_port;
  if ( likely(peer_info->size() > 1) )
    port = atoi(peer_info->at(1)->cstr());

  client = new tcpsockbuf(peer_info->at(0)->cstr(), port);
}

/* Log a trace to a socket connected to an LDP server at port 4242 */
client->header();
client->append("\r\n");
iface->trace(*client, pthread_self());
client->append("\r\n");

client->open();
client->flush();
client->close();
```

1.5.5.3 Using csdbg::sttybuf

A csdbg::sttybuf object is a buffered output stream used to output **LDP (Libcsdbg Debug Protocol)** data (protocol headers and traces) or generic data to a serial device (RS-282, RS-485, USB or other serial interfaces, terminals, pseudoterminals e.t.c). The interfaces are configured for 8N1 transmition, the baud rate is configurable (throughout a session). This class is not thread safe, the caller must implement thread synchronization. An sttybuf object inherits all the csdbg::string methods designed for text manipulation (append, clear, set e.t.c). You use these methods to process the buffer data, or the csdbg::tracer methods to store traces to the buffer. Then you may open the device node and flush the buffer. You can re-fill and re-flush as many times as you wish before you close the sttybuf. Closing is just the opposite of open, it doesn't release the object and its buffer, so you may re-open it. The steps to use an sttybuf object are:

1. Create an sttybuf object for a serial port and baud (8N1 configuration)
2. Open the device
3. Append/set text data in the buffer
4. Flush the buffer
5. Repeat steps 3-4 until completion
6. Close
7. Repeat steps 2-6 until completion
8. Release the object

Step 2 in this list can be inserted anywhere within steps 2-4, you don't need to open the port in order to process its buffer, it must be opened before you flush the buffer. The following is an example of using the sttybuf class:

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  return;

sttybuf tty("/dev/ttyS0", 115200);
tty.header();
tty.append("\r\n");
iface->trace(tty, pthread_self());
tty.append("\r\n");

tty.open();
tty.flush();
tty.close();
```

1.5.6 Using the instrumentation plugin API

A **plugin** object is the way to declare a pair of instrumentation functions and register them with libcsdbg to be run upon function call and return. A plugin can be created by passing it the addresses of the instrumentation functions (**inline plugin**) or by loading a plugin **DSO** module that implements and exports these two functions. A plugin invokes the system dynamic linker (**ld**) to find, load and link the module, so it must reside in one of the linker search directories (see **ld.so.conf** and the linker manual to understand its path search algorithm).

The names of the default profiling functions (**__cyg_profile_func_enter** and **__cyg_profile_func_exit**) are **hard-coded into g++**, so only a unique implementation of these can exist at linkage time, therefore only one profiler can be used at a time. For example, if a user needed to use libcsdbg he/she could not use any other function profilers at the same time. The plugin class is the fix to this g++ shortcoming. Now you can use libcsdbg and let the library run the secondary profilers by registering to it the proper plugins.

The plugin class supports both **C and C++ ABIs**. To resolve C++ functions the user must pass the symbol name and its full scope (as a separate argument in the form namespace::class). To resolve C functions omit the scope argument altogether. The module callback functions must be named **mod_enter** and **mod_exit**, take two **void***

arguments and return **void** for the plugin object to resolve them correctly. All plugin functions (as all libcsdbg functions) must **NOT** be instrumented by libcsdbg, as this will result in an infinite recurse and a stack overflow.

You don't really need to instantiate plugin objects yourself. Just use the plugin API exported by csdbg::tracer to register or unregister plugins, like in the following code example:

```
using namespace csdbg;

void init() __attribute((constructor, no_instrument_function));

void init()
{
  tracer *iface = tracer::interface();
  if ( unlikely(iface == NULL) )
    return;

  try {
    string path("path/to/module/plugin.so");
    iface->add_plugin(path.cstr(), "outer::inner::class");
  }

  catch (exception &x) {
    std::cerr << x;
  }

  catch (std::exception &x) {
    std::cerr << x;
  }
}
```

Like in this example, by registering a plugin in a function marked with the constructor g++ attribute (**__attribute((constructor))**) lets the plugin get called even before main and other compiler generated initialization functions, to profile them as well. Note: if you register the plugin inside main the profiler **mod_exit** callback will be called for main when it returns! As you see in this example the *scope* argument, can be as complicated as you need it to be. The plugin object code will mangle the callback symbols and resolve them correctly, as long as their scope is correctly given and the callbacks are correctly coded, exported and linked.

1.5.7 Using the stack trace parser (syntax highlighter)

Class csdbg::parser is a string buffer that can parse its contents using any arbitrary grammar (POSIX extended regular expressions). The predefined, default grammar defines a C++ stack trace as structured by libcsdbg both for exceptions and threads. A parser object equipped with this grammar can be used to perform **stack trace syntax highlighting** for VT100 terminals (XTerm, RXVT, GNOME terminal e.t.c). Subclasses of the default parser can be implemented to create highlighted stack traces for any rich text format (HTML, XML e.t.c). The default highlighter uses custom styles that describe how to render each type of token (function name, C++ keyword, intrinsic type e.t.c). This is implemented using the classes csdbg::dictionary and csdbg::style.

Class csdbg::dictionary represents a named collection of words. Its tokens can be loaded from a **simple text file**. A parser may be equipped with multiple dictionaries and use them to lookup and identify tokens. The default parser object loads three dictionary data files, one for **C++ keywords**, one for **C++ intrinsic types** and one for **C++ file extensions**. The contents of a dictionary can be looked up as literals or as POSIX extended regular expressions.

A csdbg::style object is a named set of VT100 text style attributes (foreground and background colors and text style indicators). The user of a parser object can register a style for each type of token, to create custom syntax highlighters. The following is an example of using the predefined libcsdbg syntax highlighter:

```
using namespace csdbg;

tracer *iface = tracer::interface();
if ( unlikely(iface == NULL) )
  return;

...

try {
  ...
}

catch (exception) {
  try {
    parser *p = parser::get_default();
```

```
    p->clear();

    iface->trace(*p);
    std::cerr << x << "\r\n" << *p << "\r\n";
  }

  /*
   * In non instrumented code sections, like the above (all called functions
   * are not instrumented) you don't need to create stack traces or unwind
   * the simulated stack
   */
  catch (...) {
  }
}

iface->unwind();
```

The following snippet shows how the default parser gets initialized, as an example on how to clone and customize your own syntax highlighters:

```
/* Create the default parser */
m_default = new parser;

/*
 * Equip the default parser with dictionaries for C++ keywords, intrinsic
 * types and file extensions
 */
string path("%s/etc/keywords.dict", util::prefix());
m_default->add_dictionary("keywords", path.cstr(), false);

path.set("%s/etc/types.dict", util::prefix());
m_default->add_dictionary("types", path.cstr(), false);

path.set("%s/etc/extensions.dict", util::prefix());
m_default->add_dictionary("extensions", path.cstr(), true);

/*
 * Create the default, fallback style. When a highlighter can't determine or
 * create/obtain the correct style for a token, it uses the fallback
 */
m_fallback = new style("fallback");

/* Add styles for all kinds of trace tokens to the default parser */
style *s = m_fallback->clone();
s->set_name("delimiter");
m_default->add_style(s);

s = m_fallback->clone();
s->set_name("number");
s->set_fgcolor(HLT_NUMBER_FG);
s->set_attr_enabled(style::BOLD, true);
m_default->add_style(s);

s = m_fallback->clone();
s->set_name("keyword");
s->set_fgcolor(HLT_KEYWORD_FG);
m_default->add_style(s);

s = m_fallback->clone();
s->set_name("type");
s->set_fgcolor(HLT_TYPE_FG);
s->set_attr_enabled(style::BOLD, true);
m_default->add_style(s);

s = m_fallback->clone();
s->set_name("file");
s->set_fgcolor(HLT_FILE_FG);
m_default->add_style(s);

s = m_fallback->clone();
s->set_name("scope");
s->set_fgcolor(HLT_SCOPE_FG);
m_default->add_style(s);

s = m_fallback->clone();
s->set_name("function");
s->set_fgcolor(HLT_FUNCTION_FG);
s->set_attr_enabled(style::BOLD, true);
m_default->add_style(s);
```

To ease foreground/background color selection for your custom syntax highlighter styles, script **$PREFIX/vtcolors** prints all available colors of VT100 compatible terminals.

1.5.8 Using the internal libcsdbg API

Although now you know everything you need, to produce and process any kind of **stack trace**, some libcsdbg types may still come in handy, for other, more generic purposes, although these types are quite specific and optimized for the project. This is why, some of these classes have some rather tricky details, so instead of getting lost in the code and comments to get an idea, some of them are shortly described here, with some of their most uncommon features.

Class csdbg::exception

This type is used internally by libcsdbg to propagate and process errors. It is also used by the library to report internal errors to the user. Users may use it or even subclass it for their own needs. An exception object can be constructed using **printf-style formatting** and a variable argument list (for its error message). Inside the constructor, copy constructor and assignment operator other formatting or allocation exceptions may be recursively thrown but they are internally caught and silently ignored. In these cases the object is still safe to use by ignoring its error message. The std::ostream insertion operator implementation and all exception methods take this into account. Although an exception object is not thread safe by itself, all overloaded std::ostream insertion operator implementations that output exceptions synchronize thread access.

Class csdbg::string

A string object is mainly used to create trace text. Text is easily appended using **printf-style format** strings expanded with variable argument lists. Memory is allocated in blocks (aligning) to reduce overhead when appending multiple small strings. It is comparable against **POSIX extended regular expressions**. By creating traces in string buffers it is easy to direct library output to any kind of stream (console, file, serial, network, plugin, device e.t.c). Apart from traces a string can be used for generic dynamic text manipulation. If the library is compiled with plugin support (**CSDBG_WITH_PLUGIN**) or with support for stack trace syntax highlighting (**CSDBG_WITH_HIGHLIGHT**) a string object gets equipped with a method to tokenize it using POSIX extended regular expressions and other advanced text processing methods. This class is not thread safe, the caller must implement thread sychronization.

Data structure classes csdbg::node, csdbg::chain and csdbg::stack

A node object, through its m_link member variable, can be linked to a single node (**direct addressing**), or to two nodes (**XOR linking**). Class csdbg::stack uses singly-linked nodes, class csdbg::chain is a doubly-linked list. A node can be instantiated only through the public methods of a chain or stack object. A node can point to data of any type (intrinsic or user defined) except arrays. When a node is released it also calls delete (not delete[]) on its data pointer, unless it's previously detached. Therefore each node must point to a single T and not a T[], otherwise memory leaks are bound to happen. When a node is copied or assigned, only its data are copied. Data copying invokes T(const T&) or T::operator=(const T&), exceptions thrown from these methods are not handled by the node nor its container, they are propagated up the call stack.

A doubly-linked list is a great optimization compared with a singly-linked one in node access times and memory references, especially in very big lists. The XOR linking implementation, although a bit more complex, uses the same ammount of memory (per node) as a singly-linked list. The chain supports shared data (multiple chains can point to the same data) but it's not thread safe, callers should synchronize thread access. This implementation does not allow a node with a NULL or a duplicate data pointer. A node can be detached (dispose the node without deleting its data) or removed (dispose both node and data). A chain can be traversed using simple callbacks and method chain::foreach.

The stack supports shared data (multiple stacks can point to the same data) but it is not thread safe, callers should synchronize thread access. This implementation doesn't allow a node with a NULL or a duplicate data pointer. A stack can be traversed using simple callbacks and method stack::foreach. Apart from the legacy **push**/**pop** functions, node data can be accessed using stack offsets.

Class csdbg::symtab

A symtab object can load code from executables or dynamic shared objects with absolute addressing or position independent. It supports all the binary formats supported by the **libbfd backends** on the host (or target) machine (elf, coff, ecoff e.t.c). To optimize lookups the symbol table (as structured in libbfd) is parsed, the non-function symbols are discarded and function symbols are demangled **once** and stored in simpler data structures. A symtab can be traversed using simple callbacks and method csdbg::symtab::foreach. The access to a symtab is not thread safe the caller must implement thread synchronization.

Class csdbg::dictionary

A dictionary object is used to create a collection of tokens, under a common name. Dictionary data can be loaded from regular text files (.dict extension). Each non-empty line in the source file is translated as a single token. A line with only whitespace characters is considered an empty line. The tokens are trimmed to remove leading and trailing whitespace characters. If the source file is empty no tokens are loaded, but the dictionary object remains valid. The dictionary class inherits from csdbg::chain (T = csdbg::string) all its methods for item management. A dictionary can be looked up for literal strings or for POSIX extended regular expressions (with or without case sensitivity). If a word appears more than once, its first occurence is used. A dictionary is not thread safe, users must implement thread synchronization.

Class csdbg::process

An object of this class is an abstraction of the actual debugged process. It stores the whole instrumented namespace and the details of all the simulated threads and their stacks. The namespace consists of a number of symbol tables, one for each objective code module (executable and selected DSO libraries). A process object offers methods to perform batch symbol lookups, inverse lookups (given a resolved symbol find the module that defines it) and thread handling. A lookup cache is used internally to optimize symbol resolving. Access to the process object **is thread safe**.

1.6 Changelog

1.6.1 Version 1.10

- Header limits.h included to provide constants for library runtime configuration
- Memory address bus default width was set to 64 bit
- High level user API was made thread safe
- RTTI checks were reworked in class util, chain and stack
- Added checks for chain and stack overlapping
- Class symbol_table was almost rewritten and got fully optimized by parsing the original BFD tables, discard non-function symbols, demangle the rest and store them in a chain. This change allows the symbol_table to be copied, assigned and cloned, so these methods were made public and functional. The same changes were propagated to class name_space
- Added an optional symbol lookup cache (with its profiler) to class namespace and defined a default name for unresolved symbols
- Added methods exception::header, node::detach, chain::detach, chain::detach_node, namespace::get_table_count, call_flow::enabled and various thread accessor methods
- Removed the debug enumerator from class chain, stack, thread, symbol_table, name_space and call_flow
- Inverse lookup was removed from class namespace and symbol_table
- Namespace symbol lookup and current thread lookup was moved from class util to class call_flow
- Function simulation methods and simulated stack unwinding were moved from class call_flow to class thread
- Document generation system updated
- Various code, comment and documentation typos corrected
- Various minor bugs fixed
- Script stats was reworked to collect objective code size only
- Man page documentation was dropped
- Added a signal handler (to demonstrate stack dumps) to the examples
- Added user defined exceptions to the examples

1.6.2 Version 1.11

- Removed class call_flow
- Added class call, string and tracer
- Radically **simplified, beautified and optimized the whole internal and exported library API**. Now, a global tracer object is used as the simple interface to the library for all its features, from library construction and destruction to DSO filtering and symbol table loading and parsing and from symbol and thread lookup to trace creation in string buffers and output to std::ostream objects using the **insertion operator**. The sum of changes **dramatically** reduces the interfacing code from whole functions to a couple of lines!
- **Heavily** optimized **all code** and removed all unused and obsolete code
- Added **addr2line** support and all needed infrastructure code in class name_space and symbol_table (for inverse lookups)
- Removed all RTTI checks from class object, util, chain and stack
- Class name_space cache made mandatory, removed its profiler
- Fixed a common memory leak in class chain, stack, name_space and thread
- Added CSDBG_LIBS shell variable for DSO filtering
- Overloaded std::ostream insertion operator for std::exception and csdbg::exception objects
- Heavily reworked the build system for portability, dropped the stats script, added support for the example DSOs, for the new documentation system and to reflect the new classes and the new API interface
- Heavily reworked the documentation generation system in general and created the first draft of the documentation overview and user manual pages
- Started the **LaTeX**/**PDF** API reference documentation
- Added complete example documentation
- Heavily reworked the examples to simplify them and accomodate the radical libcsdbg API changes
- Added an example DSO that throws demo exceptions and code in the examples to use the example DSO
- Moved all example code to a new namespace (csdbg_extra)

1.6.3 Version 1.12

- Added class streambuf and filebuf (buffered output streams)
- Added method tracer::init to load the library runtime configuration from the command line arguments
- Added method tracer::thread_by_id and tracer::cleanup_thread
- Overloaded tracer::trace to create **thread** stack tracer
- Added class throwable (as a more transparent interface API)
- Updated the user manual (for PDF)
- Added file documentation
- Added example for buffered output streams and throwables

1.6.4 Version 1.13

- Reworked the API of the buffered output streams
- Implemented the first draft version of LDP (Libcsdbg Debug Protocol)
- Added method filebuf::unique_id and streambuf::header
- Added class tcpsockbuf
- Added example code to test/demonstrate the new output APIs
- Minor changes to the documentation system (upgraded to doxygen 1.8.5)

1.6.5 Version 1.14

- This is a maintenance version
- Added an iterator method (foreach) to class chain, stack, symbol_table and name_space
- Added methods to the thread interface API (thread_count, thread_at) of class tracer
- Added methods to process the CLI arguments (opt_count, getopt_at) to class tracer and made getenv public
- Added function try blocks to fix potential memory leaks in the constructors of class symbol_table, name_space, tracer, streambuf, filebuf and tcpsockbuf
- Finalized the implementation of LDP (filebuf::unique_id now takes a printf- style format argument and timestamps are given in microseconds)
- Added the final optimizations, fixes and beautifications to the code, to the library API and the documentation for all current code
- Dropped class throwable
- Added assertions
- Finalized the HTML documentation generation system
- Added the 'Usage' section (with its 10 subsections) and an 'Examples and Tutorials' section to the User Manual (both for HTML and PDF)
- Removed all old examples and added the new Step examples (4) and the new Tutorial examples (4) all with their complete documentation

1.6.6 Version 1.15

- Added class plugin
- Added a user API to register/unregister and manage instrumentation plugins, to class tracer (two register_plugin and unregister_plugin methods, method plugin_count and plugin_at)
- Dropped the PDF API reference manual and finalized the document generation system for the PDF User Manual
- Added a tutorial that demonstrates the use of the plugin API both for inline plugins and for DSO modules
- Added a tutorial plugin module DSO (mod_null.so)

1.6.7 Version 1.16

- Added a generic, adaptive, syntax parser/tokenizer in class string (method split). This method is used for the plugin symbol mangler and is the base for the stack trace syntax highlighter
- Added support for the **C++ ABI** to class plugin and updated the class tracer plugin management API
- All typedefs where added to the csdbg namespace
- API reference and User Manual are brought up to date (both HTML and PDF)
- Added the full class plugin documentation to the User Manual under a new section
- Added the 'Changelog' section to the User Manual
- Implemented a profiling module (plugin), mod_callgraph.so, that constructs and prints to console the complete call graph of any multi-thread process. This module was created both as an elaborate example of the plugin mechanism and as a demonstration and proof of the power of the libcsdbg API as a base for generic, user instrumentation code
- Added an example step (inserted as step 4) to demonstrate to the full the features of class plugin, using mod_callgraph as the plugin module
- Tutorial plugin mod_null.so renamed to mod_test.so
- Finalized the code and documentation for all current example code
- Added three syntax highlighter vocabularies (for C++ types and keywords and for source, header, assembler, inline and other C/C++ file extensions)

1.6.8 Version 1.20

- Added class dictionary
- Added class highlighter
- Added class style
- Added case sensitivity flag in string::cmp
- Added method string::trim
- Fixed Makefile and header includes to reflect the Makefile::$DOPTS build configuration
- Added Makefile target distclean
- Added example and tutorial for the trace syntax highlighter
- Added utility script **vt100_colors** (color sampler for highlighter customization)
- Added dictionary source files for C++ keywords, integral types and file extensions

1.6.9 Version 1.21

- Class highlighter renamed to parser
- Added thread names and all supporting infrastructure
- Added lookup mode to class dictionary
- Heavily reworked class style
- Added methods plugin::destroy, string::insert and thread::foreach
- Added method util::header and COLOR_TERM support for debug text

- Added macros for the default syntax highlighter colors
- Added various code optimizations and beautifications
- Updated all examples to use/demonstrate thread names
- Updated examples to make them more uniform
- Added Makefile target distclean (clean the source tree from all generated files)
- Added scripts vt100_bgcolors and vt100_fgcolors (VT100 palette samplers)
- Added parser and trace syntax highlighter documentation
- Added Makefile $DOPTS documentation

1.6.10 Version 1.22

- Changed LDP to support multiple messages through a unique session socket
- Scripts vt100_fgcolors and vt100_bgcolors fused into vtcolors
- Reconfigured the default syntax highlighter
- Reworked examples to make them compatible with the new LDP architecture

1.6.11 Version 1.23

- Added class csdbg::sttybuf
- Added method style::is_attr_enabled and style::set_attr_enabled
- Removed method style::has_attributes and style::add_attributes
- Reworked script vtcolors

1.6.12 Version 1.24

- Reworked the build system
- Reworked the documentation generators
- Reworked the library compile-time configuration
- Detection of memory bus width and endianity made automatic (and portable)
- Recode of class csdbg::chain to make it a doubly-linked list (using XOR linking)
- Added method filebuf::sync, filebuf::seek_to, filebuf::truncate
- Added method util::min
- Added XOR linking capability to csdbg::node (node::link, node::operator^)
- Renamed class csdbg::symbol_table to csdbg::symtab

1.6.13 Version 1.25

- Removed class csdbg::name_space
- Added class csdbg::process
- Added method dictionary::set_name
- Added method filebuf::seek_to and filebuf::resize
- Added method parser::get_fallback_style
- Added method streambuf::sync, streambuf::lock and streambuf::unlock
- Added method sttybuf::set_baud, sttybuf::is_tty, sttybuf::open and sttybuf::sync
- Added method util::memcpy, util::is_regular, util::is_chardev, util::is_readable and util::is_writable
- All examples, tutorials and example DSO and plugin modules are moved to a separate package (libcsdbg-extra)
- Added code to process plugin stray exceptions
- Object filebuf now appends to file by default and added file checks
- Fixed a bug in parser::highlight
- Fixed various minor bugs and omittions
- Fixed a bug in dictionary 'extensions.dict'
- Replaced name_space with process in csdbg::tracer, to declutter and simplify

1.6.14 Version 1.26

- Added useful assertions to various methods, as needed
- Added method node::link_to and node::unlink_from
- Added method parser::remove_all_dictionaries, parser::get_dictionary_names parser::remove_all_styles and parser::get_style_names
- Added method overload parser::lookup
- Added method streambuf::is_opened
- Added method overload string::set
- Added method tcpsockbuf::is_connected and tcpsockbuf::set_option

1.6.15 Version 1.27

- Method streambuf::sync was made abstract
- Added filebuf::sync(bool) overload
- Added tracer::get_plugin(const i8$*$) overload
- Added string::bufsize, string::available, string::shred
- Added streambuf::config, streambuf::discard, streambuf::sync
- Added thread::is_current
- Replaced process::thread_by_$*$ and process::thread_at with multiple process::get_thread overloads

- Variable tracer::m_plugins and the plugin interface were made non-static
- Removed tracer::lookup, tracer::current_thread
- Moved getenv, init, argc and argv methods from class tracer to util
- Added class util library constructor and destructor and moved m_config static member variable from class tracer to util
- Added more system call checks (for **EINTR and EAGAIN**) to streambuf, filebuf and sttybuf
- Made class process API methods thread safe
- Added plugin stray exception handling
- Minimized accepted baud rates to a useful common set in sttybuf
- Added lots of assertions to all classes
- Fixed various minor bugs