The Axis C++ Trace Guide

<!-- --> <!-- --> <!-- -->

1. Introducion

This documents describes the trace facilities within Axis C++, how to enable them and how to make use of the produced trace.

2. Contents

3. Runtime Trace

To aid the development team in understanding the problems of users, the Axis C++ engine has been instrumented with trace.

3.1. Enabling runtime trace

Trace is enabled at runtime by adding the following to `axiscpp.conf`:

```
ClientLogPath:<path to log file>
```

3.2. Reading runtime trace

The Axis C++ runtime trace is produced in a format compatible with the Trace Analyzer for WebSphere Application Server, which is available from here:

Trace Analyzer for WebSphere Application Server

However, it is also possible to use any text editor. Each line is made up of the following columns:

- date/time
- thread ID
- Class name, a - indicates a c-style gloval function
- Trace type:
  - > (entry)
  - < (exit)
  - X (exception)
  - I (information)
- Method name
- `this` pointer for instance methods
- comma seperated list of parameters
  - [nnnnnnnn] Data in hex
  - <...> Data in ascii
  - For pointers address is given before data
  - For void*, only 1st byte of data is given, as there is no way of knowing size of data.

So, a typical entry would be:

```
[27/03/2006    16:21:48:945    Time]    4220    AxisConfig    >
getAxisConfProperty @00376358,[00000000] <....>
```

Some other things to watch for are:

- An entry containing: `------------>` indicates that it is a multiline entry
- `<BADPOINTER>` means the trace attempted to dereference a pointer causing a SIGSEGV which the trace has caught.
- `<UNKNOWNTYPE>` means the trace tool and runtime have got out of step. You have found a bug!
- `<ANONYMOUS>` means that it is unnamed in the method signature.

4. Startup Trace

One shortcoming of the runtime trace is that it only begins tracing after the configuration file has been loaded and the `ClientLogPath` entry read. This means any problems that occur before this, for example while reading the configuration file, are missed. To overcome this the Axis C++ engine has also been instrumented with startup trace, this starts tracing at the first call into the Axis engine and everything up to the point at which runtime trace starts.

Startup trace uses the same formatting as the runtime trace.

4.1. Enabling startup trace

Startup trace is enabled by setting the following environment variable:

```
AXISCPP_STARTUP_TRACE=<path to startup log file>
```

5. Interesting Trace Entries

Here are a few entries well worth looking for when you first receive a trace file.

5.1. Stub Destructor

Check lines in the trace file immediately preceding the entry for ~Stub, as this is typically the first method run following a `SIGSEGV`.

5.2. Exception Entries

Check any exception entries, which is indicated by an X in the 3rd column. These actually indicate that an exception has been caught. If you see `@<number>` this indicates which catch block, if more than one present in the current method. You can use this, along with the trace lines immediately preceding, to determine fairly closely exactly where the trace was thrown. You will also see trace entries for the exception constructor if it is an Axis C++ exception.

5.3. HTTPChannel::operator<<

The entry trace for `operator<<` on `HTTPChannel`shows the message that will sent on the wire. Although the HTTP Header and SOAP message tend to be in seperate calls to this method. This allows for diagnosis of some problems with the message, without requiring the use of a tcp/ip monitoring tool.