APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE CB3 0RD UNITED KINGDOM
+44 1223 515010 • Fax +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

ANSA Phase III

DIMMA Tracing

Richard Hayton

Abstract

DIMMA has a comprehensive tracing system, that allows the degree of tracing to specified independently for each thread, task, code module and object. Tracing may be controlled dynamically by setting tracepoints (analogous to break points) and telnet access is provided to allow modification to the tracing during execution. This document describes how to program using the tracing system, and how to examine a program instrumented in this way.

1980.02	**Approved** Technical Report	03 April 1997

Distribution:
Supersedes:
Superseded by:

TABLE OF CONTENTS

1 PROGRAMMING WITH TRACING

1 .1 Introduction

DIMMA has a comprehensive tracing system. This consists of a set of macros used to insert trace statements at relevant points in the code, and a Tracer class that can interpret these statements and output detailed tracing information.

Output can be controlled to a fine level, including interactive or automatic changes in the amount of information output, according to the current thread being executed, module of code, method being called or object in use. Trace statements themselves are typed, and output can be controlled dependant on the type of trace statement.

Programming with tracing is straightforward. There are a number of tracing statements that should be inserted into code in order that it may be traced. At compile time, a number of macros are used to indicate if tracing should be compiled in, or trace statements ignored. When tracing is disabled in this way, there is no run time performance overhead.

There are three concepts the programmer should be aware of. Code is logically split into *modules* for tracing purposes, there are a number of *trace statements* that may be used to instrument code, and each statement may indicate a tracing event of one or more *classes.* These are described in turn.

1 .2 Modules

To aid control of logging, each trace statement is associated with the module of code that contains it. Modules are simply textual tags given to pieces of source code to group them for identification purposes.

Each source or header file containing tracing information should define the C pre-processor macro **`MODULE`** to the name of the current module. Typically this is the file name, but in some circumstances it is useful to group a number of files within a module. Both the file name and module name are available in the output log.

Whenever **MODULE** is defined, it should first be undefined, to prevent compiler warnings if a number of included header files also contain traced code. **MODULE** should be defined after #includes in case they redefine it.

For example

```
#undef MODULE
#define MODULE "task"
```

1 .3 Trace Statements

There are a number of statements that can be used to indicate tracing events. They are of two kinds, those that generate a single trace event, and those that generate two trace events; one at the entry, and one at the exit of a region of code. The full list is given here:

- **TRACE(***event class, message***)**

  This is the basic trace statement. *Event class* may be a single event class, or collection of classes, specified by using the |. The prefix **TR_** should be added to each event class, this is used to help ensure C++ macro names are unique. For example

  ```
  TRACE(TR_BUG, "Obsolete function called");

  TRACE(TR_CREATE | TR_ENTRY, "Creating a session");
  ```

  A description of event classes is given in the following section.

- **TRACE_METHOD(***message)*

  This should be used at the start of a method that is to be traced. It generates two trace events, one at the entry and one at the exit of the method. For example

  ```
  TRACE_METHOD("iiop::init");
  ```

  This statement makes use of the C++ variable 'this' to record the identity of the object making the call. For this reason it cannot be used with static methods. For these, the more general statement **TRACE_ENTRYEXIT** should be used.

- **TRACE_CONSTRUCTOR(***message***)**

  A version of **TRACE_METHOD** for use in constructors.

- **TRACE_DESTRUCTOR (***message***)**

  A version of **TRACE_METHOD** for use in destructors.

- **TRACE_ENTRYEXIT(***message***)**

This should be used at the start of a procedure that is to be traced. It generates two trace events, one at the entry and one at the exit to the procedure or block of code.

For example

```
TRACE_ENTRYEXIT("iiop::init");
```

- **TRACE_CREATE(***message***)**

  A version of **TRACE_ENTRYEXIT** for use in creation routines where **TRACE_CONSTRUCTOR** cannot be used.

- **TRACE_DESTROY(***message***)**

  A version of **TRACE_ENTRYEXIT** for use in destruction routines where **TRACE_DESTRUCTOR** cannot be used.

- **THROW(***throw expression***)**

  A wrapper for throw that creates a trace statement first. Expands to

```
TRACE(THROW, "expr"); throw expr
```

- **ERR(***error number***)**

  A wrapper to be used when returning an integer error number. For example if tcp_returncode had the value 27, the statement

```
return ERR(tcp_returncode)
```

  would expand to (the equivalent of)

```
TRACE(ERROR, "tcp_returncode = 27");return 27;
```

1 .4 Classes of Trace Events

There are a number of different classes of trace event. Each trace event may be an instance of any number of these. The primary purpose of event classes is to allow the user to control which events are displayed in different circumstances. In addition some event classes give the tracing system clues to enable it to keep track of other parameters, such as procedural nesting depth. The statements described in the previous section all implicitly identify the event class, apart from the **TRACE** statement.

- **BUG**

  This indicates a trace statement that should never be reached.

- **ERROR**

  This is used to indicated an error (for example a broken connection).

- **THROW**

This special event class is created by the **THROW**() statement. The latest **THROW** event is recorded for each thread, to aid debugging.

- **ENTRY**

  This is used to indicate entry to a procedure, constructor or other block of code. The current nesting level is recorded for each thread.

- **EXIT**

  This use used to indicate exit from a block of code. Often **ENTRY** and **EXIT** statements are generated by the macro **TRACE_ENTRYEXIT()**, or **TRACE_METHOD**. When these are used, the line number of the end of the block is not known by the pre-processor, so is unavailable for use by the tracer. By convention the pseudo line number (-start line) is used in these cases.

- **CREATE**

  This indicates the creation of something. The macros **TRACE_CREATE** and **TRACE_CONSTRUCTOR** may be used within constructors and generate events of class **TR_CREATE | TR_ENTRY** at the start of the constructor and events of class **TR_EXIT** at the end.

- **DESTROY**

  Similar to **CREATE**, but used to indicate destruction. The macros **TRACE_DESTROY** and **TRACE_DESTRUCTOR** may be used in a similar way to **TRACE_CREATE**.

- **TRACE**

  This is used as a general purpose event class.

- **USER1**,**USER2**,**USER3**

  These are available for use in user code and have no predefined meaning.

1 .5 Compiling with/without Tracing

There are four relevant pre-processor macros:

- **GLOBAL_TRACING**

  This should be defined when compiling the distributed object nucleus (DON) if *any* files have tracing enabled.

- **TRACING**

This should be defined for all files where tracing is to be compiled in. Note tracing may be compiled in but monitoring disabled, or output suppressed. The default is for monitoring to be disabled.

- **TRACEWITHPURIFY**

  This flag should be set when compiling the DON if Pure Atria's Purify program is also being used. This will combine tracing output with Purify's output.

- **DEBUG_TRACE_TRACING**

  This should be set when compiling the DON if the tracing system itself is to be traced. This 'meta tracing' uses a simple scheme with no user control.

2 TRACING A PROGRAM

2 .1 Introduction

When tracing is compiled in, a traced program will examine a number of environment variables and resource files on initialisation to control which events are output, and in what circumstances. In addition, a user may connect to the program via telnet to dynamically examine or modify the tracing state.

2 .2 Tracing Configuration

When a traced program starts up it will read in a tracing resource file to define its initial configuration. It may also start a background task to allow telnet connections to be used to dynamically control the tracing level during program execution.

2 .3 Environment Variables and Resource Files

At initialisation, a traced program will examine the **`DIMMATRACERC`** environment variable to determine the resource file to use. If this is not set the default file **`.tracerc`** will be used if it exists. If there is no resource file, tracing is disabled.

The resource file may indicate a port to be used for telnet access. If it does not, the environment variable **`DIMMATRACEPORT`** is consulted. Telnet access is disabled if neither is set. If the port specified is 0, then a port will be picked at random. To allow the user to locate the chosen port, the port number will be reported in the log file, and also written to a file

`.traceport.`*`pid`* (where *pid* is the process id)

in the current directory. This file is deleted if the program exits normally.

The log file should be specified in the resource file, and is set to `stderr` by default.

Note: Telnet access is disabled in a single threaded environment.

2 .4 Controlling Tracing During Execution

The current tracing configuration may be examined or changed by using a `telnet` connection to the running program. The majority of commands may be used in the resource file or via `telnet`, the only difference being that the output of some commands is set via `telnet`, whilst others cause changes to the output log.

2 .5 Long running Applications

For long running applications it is possible to turn the tracing system off to increase performance. When disabled in this way, there is still an overhead for each trace statement, but this is kept to a minimum. As tracing is disabled, the tracing system will lose track of procedural nesting depths, and other useful information.

3 CONTROLLING THE LEVEL OF TRACING

3 .1 Introduction

There is considerable flexibility as to when tracing is enabled and disabled, and the level of information provided. In particular it is possible to set *tracepoints* so that the degree of tracing is changed when a particular method, or line of code is executed. It is also possible to track the execution of particular objects.

3 .2 Modules

Each source file contains a macro defining the MODULE that it is part of. A module is simply a conceptual grouping of source code to aid the control of tracing. It is possible to indicate different levels of tracing for different modules, to reduce the amount of unnecessary event reporting.

3 .3 Threads and Tasks

DIMMA has both the notion of tasks and threads. A thread is an independent thread of control. Most client applications are only aware of threads. A task is an 'engine' used to execute threads. If there are more threads than tasks, then each thread must wait until there is a task to execute it.

It is possible to indicate that one thread should be traced whilst another is not. Equally, tracing may be specified on a per-task basis. User code should normally be controlled in terms of threads. However, the DIMMA nucleus is written in terms of tasks, and tasks should therefore be used to control tracing within the nucleus. For ease, it is possible to control tracing of both threads and tasks simultaneously. For example it is possible to state

> *when statement x is executed, change the tracing level for the current thread and the current task.*

A trace event will be reported provided

> *It is an instance of a class of event enabled in the current module*

and

It is an instance of a class of event enabled for the current thread

or

It is an instance of a class of event enabled for the current task.

3 .4 Tracepoints

A trace point is analogous to a break point in a debugger. Whenever a thread of control passes through a trace point, the level of tracing is updated. Trace points may be set at any line of source code that contains a tracing macro. They may also be set at the return from traced functions. A trace point may set the tracing level for the thread directly, may selectively add or remove event classes from the current tracing level, or may PUSH the current tracing level onto a stack and later POP this back off.

For convenience it is possible to simultaneously set a tracepoint at the start and end of a method. For example a user may wish to trace all calls that originate from a particular method. It is also possible to restrict a tracepoint so that it is only active when a method on a particular object is called.

By convention, trace events resulting from the exit of a procedure or method are given pseudo line numbers, equal to the negation of the start line number. This allows the tracing of procedure exit to be easily controlled. Note the line number is the line number of the *trace* statement, not the start of the actual procedure.

3 .5 Objects

The tracing statements **TRACE_METHOD**, **TRACE_CONSTRUCTOR** and **TRACE_DESTRUCTOR** make the identity of the calling object available to the tracing system. Trace points may specify a particular object identity, to limit their effect. For example a user may wish to enable tracing whenever a particular method is called on a particular object.

Object addresses are mapped to small integer identifiers for ease of use. Generally a user will use object identifiers (OIDs) if working purely with the tracer, and object addresses (Objects) if the output is to be compared with that of a debugger or other development tool.

Care must be taken when interpreting object identifiers. When sub-typing is used (particularly with multiple inheritance) the address of an object may change depending on the type.

For example consider

```
class A : public B, public C;
A *a;
```

The pointer `a` may or may not equal `(B*) a`, or `(C*) a.`

As the tracing system is unaware of the type of objects that it assigns OIDs to, it cannot determine that `a`, `(B*) a` and `(C*) a` actually refer to the same object, and will given them different OIDs. As a rule of thumb, if `A`, `B` and `C` all have constructors, then these will be called in turn when an instance of `A` is created. Therefore, if the constructors are traced, the OIDs generated for `a`, `(B*)a` and `(C*)c` are likely to be sequential.

3.6 Exceptions

A wrapper for the C++ throw command is provided in the form of the **`THROW`** macro. In addition to inserting a trace statement, this command stores information about the most recent throw encountered by each task and by each thread. A summary of the most recent exception in the current task/thread may be printed by calling the macro **`TRACE_THROWINFO`**. It is also available via the **`lastthrow`** command described in the next section.

4 TRACE COMMANDS

Tracing commands may be specified in an resource file read at initialisation, or given over a telnet connection during program execution. It is possible to cause a resource file to be (re)read by using the load command. Resource files may recursively cause the nested loading of other resource files. For example a user may have a personal resource file, and then a more specific resource file for each program being traced.

Each tracing command is specified on a single line, and are case insensitive.

4.1 Commands

- **#** *<comment>*

  Comment, ignored by the command parser. Comments may appear on lines of their own, or at the end of any other command.

- **display** *<field>*

  Turn on output of the given field. Fields include module name, file name, current task, etc. A full list is given later.

- **dump**

  Dump the current configuration state. This is identical to save, but the state is displayed via telnet (or to `stderr`, if the command is specified in a resource file).

- **hide** *<field>*

  Turn off the output of a given field.

- **init**

  Reinitialize the tracer. This will cause the reloading of the resource file and the tracer will attempt to restart cleanly. This command is provided to help if a program corrupts the tracing state. For this reason, it does not attempt to release resources (primarily memory) before reinitialisation. It should be used sparingly.

- **lastthrow**

  Dump information about the most recent exception for the current thread and task.

- **lastthrow task** <*id*>

  Dump information about the most recent exception for the named task.

- **lastthrow thread** <*id*>

  Dump information about the most recent exception for the named thread.

- **load** <*file name*>

  Load the specified resource file. If this statement is included in a resource file, it will cause the nested loading of the specified file.

- **logfile stderr**

  Send future tracing output to stderr.

- **logfile** <*file name*>

  Send future tracing output to the specified file.

- **module** <*name*> <*class spec*>

  Set the tracing level for a particular module.

- **module all** <*class spec*>

  Set the tracing level for all modules.

- **off**

  Turn off tracing. This not only prevents output, but it also stops logging of procedural nesting, monitoring of **THROW** statements and everything else. This should only be used to speed up long running programs. The preferred method is to turn off output by disabling tracing for all modules or threads/tasks.

- **on**

  Enable tracing. By default, tracing is disabled for efficiency, so **on** should normally be specified in a resource file. If the program has been allowed to run with tracing disabled, the procedural nesting depth cannot be trusted, as it will not have been monitored.

- **port** <*no*>

  Set the tracing port to <*no*>. This command is only meaningful before the tracer is activated (i.e. in the resource file). Setting the port to a value of 0 will cause the tracer to pick a port at random. The chosen port number is then written to the log file, and to a file **.traceport.*pid*** in the current directory.

- **proc** <*file*> <*line*> <*mod class spec*>

  Control tracing for the procedure/method/constructor/destructor starting with a trace statement at <*file*>,<*line*>. This changes the tracing level for the duration of the procedure (and all nested calls), and then resets

the tracing level when the procedure is exited. A **proc** command equivalent to a pair of **tracept** commands.

proc *file line class* ≡ **tracept** *file line class* **PUSH**
tracept *file* **-***line* **POP**

- **proc** <*file*> <*line*> <*OID*> <*mod class spec*>
- **proc** <*file*> <*line*> <*object*> <*mod class spec*>

  As above, but for a specified object.

- **save** <*file name*>

Save the current configuration state in a resource file

- **task** <*id*> <*class spec*>

Set the tracing level for a named task.

- **task all** <*class spec*>

Set the tracing level for all current and future tasks.

- **task new** <*class spec*>

Set the tracing level for newly created tasks.

- **thread** <*id*> <*class spec*>

Set the tracing level for a named thread.

- **thread all** <*class spec*>

Set the tracing level for all current and future threads.

- **thread new** <*class spec*>

Set the tracing level for newly created threads.

- **tracept** <*file*> <*line*> <*mod class spec*>

Set a trace point at the indicated source file and line. <*file*> should be a file name, not a complete path name, and <*line*> must be the position of a tracing statement. <*mod class spec*> is an event class modifier (described below).

- **tracept** <*file*> <*line*> <*OID*> <*mod class spec*>
- **tracept** <*file*> <*line*> <*object*> <*mod class spec*>

Set a trace point as above. The specified location must contain an object tracing statement (**METHOD**, **CONSTRUCTOR** or **DESTRUCTOR**). The trace point will only be activated by threads/tasks executing methods on the indicated object. <*OID*> is a decimal integer object identifier, <*object*> is a hexadecimal object address preceded by 0x.

- **traceptmode both**
- **traceptmode task**

- **traceptmode thread**

  Set the interpretation of **tracept** statements following this statement. 'Task' trace points update the tracing level for the current task when control passes through a trace point. 'Thread' updates the level for the current thread. 'both' updates the level for both the task and thread.

4 .2 Specifying Event Classes.

When specifying the event classes to be displayed, for some commands the specified level may be combined with the level for the current thread or task. The current level may be ignored, saved for later on a per thread/task stack, restored from a previous save and combined with the specified level.

<class spec> = *<class> <class> ....* [- *<class> <class>*]

<mod class spec> <mclass> <mclass> .... [- *<class> <class>*]

<class> = **NONE** | **TRACE** | **ERROR** | **BUG** | **ENTRY** | **EXIT** | **THROW** | **CREATE** | **DESTROY** | **USER1** | **USER2** | **USER3** | **ALL**

<mclass> = *<class>* | **PUSH** | **POP** | **OR** | **AND**

- none

  Set the active event classes to those indicated.

- **OR**

  Set the event classes to the union of the current level and all the specified classes (bitwise OR)

- **AND**

  Set the event classes to the intersection of the current level and the specified classes (bitwise AND).

- **PUSH**

  Save the current level. This is saved before the current statement is evaluated.

- **POP**

  Restore the previous level (after evaluating the current statement).

Modifications may be combined, for example a trace point set to **ALL PUSH POP** will ensure that the current trace statement is logged, but will not modify the trace level.

Examples

ERROR THROW BUG	Events indicating unexpected circumstances
ALL - ENTRY EXIT	All events other than entry or exit.
PUSH NONE	Store the current classes and turn off tracing
POP	Restore the previous classes

5 OUTPUT FORMAT

Each tracing command creates at most one line of output. This line consists of a number of fields of information. Each may be turned on or off individually (for example to keep output more compact). In addition a tag can be added to indicate the meaning of each field. This is also useful if the output is to be post processed automatically.

5 .1 Output Fields

- **`event`**

  The class of the trace statement, for example ENTRY, or BUG

- **`filename`**

  The name of the source file in which the trace statement is located.

- **`lineno`**

  The line number at which the trace statement is located. For method or procedure exits, this is *-start*, where start is the line number at which the procedure started.

- **`message`**

  The message associated with the trace statement. Generally the name of the current method, or other useful information.

- **`module`**

  The name of the module in which the trace statement is located.

- **`object`**

  The address of the object whose method invocation generated the trace statement. Only printed for **`METHOD`**, **`CONSTRUCTOR`** and **`DESTRUCTOR`** trace statements.

- **`oid`**

  An arbitrary integer identifier assigned to represent the object (easier to read than pointers). OIDs are not reused, but object addresses might be and OID is simply hash(object address).

- **`rawevent`**

The class of the trace statement in a raw format. (not all combinations have associated keywords)

- **tag**

  Add a tag to each field indicating it's type. This is particularly useful when the output is to be post-processed, for example by awk.

- **task**

  The identity of the currently executing task

- **taskdepth**

  The current procedural nesting depth of the current task.

- **taskgraph**

  Display the task depth and event class in a compact wave that is only 4 characters wide. This wraps if nesting is deeper than 4. Different event classes are displayed as different characters. (Display both EVENT and GRAPH to see the effect).

- **taskwave**

  Display the task depth in terms of a 'wave' of indentations of the message.

- **thread**

  The identity of the currently executing thread.

- **threaddepth**

  The current procedural nesting depth of the current thread.

- **threadgraph**

  Display the thread depth and event class in a compact wave.

- **threadwave**

  Display the thread depth in terms of a 'wave' of indentations of the message.

6 EXAMPLES

These examples are taken from the ODP Tiny Bank service.

The output shown, is a portion of the log produced when the server was run using the example resource file.

6 .1 Example Resource File

```
###### tracing resource file for Tiny Bank server

# Enable tracing
on

# enable telnet access
port 0

# set display format
logfile /tmp/server.log
display oid
display taskgraph
hide tag
hide filename
hide lineno
hide task

# default output
module all all
task   all all
thread default none

# coarse grain control - never trace these modules
module mc         none
module SocketObj  none
module SocketPool none
module cdr        none
module rpc        none

# fine grain control - tracepoints
# ignore the details of invocations
#  (only show errors)

proc odp__ClientStub.cc 54 bug error
```

6 .2 Example Output

```
                Module Event Depth Object    Method
                       Class       ID                                    .
                binder IN   \     4 0002->iop_Binder::svrBind(ServerStub)
                  iiop IN    \    5 0003->ModuleIIOP::openMaster
             Condition NEW    +   6 0026->Condition
             Condition OUT    /   6 0026->Condition
               Session NEW    +   6 0027->Session
               Session OUT    /   6 0027->Session
                  iiop OUT   /    5 0003->ModuleIIOP::openMaster
                binder OUT  /     4 0002->iop_Binder::svrBind(ServerStub)
                binder IN   \     4 0028->iop_Binding::iop_ifref
           ChanellIIOP IN    \    5 0029->ChannelIIOP::profile_data
        odp_RefCounter NEW    +   6 0030->odp_RefCounter
        odp_RefCounter OUT    /   6 0030->odp_RefCounter
        odp_RefCounter IN     \   6 0030->odp_RefCounter++ (=1)
        odp_RefCounter OUT    /   6 0030->odp_RefCounter++ (=1)
                  iiop IN     \   6       << (odp_Transmitter, ProfileBody)
                  iiop OUT    /   6       << (odp_Transmitter, ProfileBody)
        odp_RefCounter NEW    +   6 0031->odp_RefCounter
        odp_RefCounter OUT    /   6 0031->odp_RefCounter
        odp_RefCounter DEL    -   6 0030->odp_RefCounter
        odp_RefCounter OUT    /   6 0030->odp_RefCounter
           ChanellIIOP OUT   /    5 0029->ChannelIIOP::profile_data
        odp_RefCounter NEW   +    5 0032->odp_RefCounter
        odp_RefCounter OUT   /    5 0032->odp_RefCounter
         odp_Reference NEW   +    5 0033->odp_Reference(count,data[])
        odp_RefCounter NEW    +   6 0034->odp_RefCounter
        odp_RefCounter OUT    /   6 0034->odp_RefCounter
        odp_RefCounter IN     \   6 0031->odp_RefCounter++ (=2)
        odp_RefCounter OUT    /   6 0031->odp_RefCounter++ (=2)
        odp_RefCounter IN     \   6 0034->odp_RefCounter++ (=1)
        odp_RefCounter OUT    /   6 0034->odp_RefCounter++ (=1)
         odp_Reference OUT   /    5 0033->odp_Reference(count,data[])
                binder OUT  /     4 0028->iop_Binding::iop_ifref
     odp_GenServerStub OUT     /  3 0025->odp_GenServerStub
     odp_GenServerStub IN      \  3 0025->odp_GenServerStub::ref
                binder IN   \     4 0028->iop_Binding::iop_ifref
                binder OUT  /     4 0028->iop_Binding::iop_ifref
     odp_GenServerStub OUT     /  3 0025->odp_GenServerStub::ref
  odp_GenInvocationRef OUT    /   2 0022->odp_GenInvocationRef::reference
              odp_ref IN     \   2 0033->odp_Reference::to_string
              nucleus IN      \  3       Nucleus::cdr_buffer_factory
              nucleus OUT     /  3       Nucleus::cdr_buffer_factory
              odp_ref IN      \  3       odp_Transmitter<<
                  iop IN   \     4       << (odp_Transmitter, TaggedProfile)
                  iop OUT  /     4       << (odp_Transmitter, TaggedProfile)
              odp_ref OUT     /  3       odp_Transmitter<<
              odp_ref OUT    /   2 0033->odp_Reference::to_string
         trader_client OUT   /    1 0024->odp_ansa_Trader_Client::export
  odp_GenInvocationRef DEL   -    1 0022->odp_GenInvocationRef
        odp_RefCounter IN     \   2 0023->odp_RefCounter-- (=2)
        odp_RefCounter OUT    /   2 0023->odp_RefCounter-- (=2)
  odp_GenInvocationRef OUT   /    1 0022->odp_GenInvocationRef
```