

# AppTrace User Guide

## vDicos

---

### USER GUIDE

**Copyright**

© Ericsson AB 2015–2017. All rights reserved. No part of this document may be reproduced in any form without the written permission of the copyright owner.

**Disclaimer**

The contents of this document are subject to revision without notice due to continued progress in methodology, design and manufacturing. Ericsson shall have no liability for any error or damage of any kind resulting from the use of this document.

**Trademark List**

All trademarks mentioned herein are the property of their respective owners. These are shown in the document Trademark Information.



# Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	Target Groups	1
1.2	Prerequisites	2
1.3	Terminology	2
1.4	Related Information	4
<b>2</b>	<b>AppTrace Basics</b>	<b>5</b>
2.1	Trace Domain	5
2.2	Trace Control State	7
<b>3</b>	<b>AppTrace Sessions</b>	<b>9</b>
3.1	Step 1: Collecting and Verifying Trace Domains	9
3.2	Step 2: Defining an AppTrace Session	11
3.3	Step 3: Executing an AppTrace Session	14
3.4	Step 4: Cleaning Up	17
<b>4</b>	<b>Filters - Detailed Trace Programming</b>	<b>19</b>
4.1	Execution Model for Trace Programs	19
4.2	Special Characters in Expressions	21
4.3	Trace Expressions	21
4.4	Forlop Tracing	32
4.5	String Expressions	34
<b>5</b>	<b>Examples</b>	<b>41</b>
5.1	Example 1: The Simplest Possible Trace Session	41
5.2	Example 2: Add Several Processors to the Scope	44
5.3	Example 3: Add Several Process Types to the Scope	44
5.4	Example 4: Use Wildcard to Add Several Similar Domains to the Scope	45
<b>6</b>	<b>Reference Manual for AppTrace Commands</b>	<b>47</b>
6.1	ls_domains.sh	47
6.2	cat_domain.sh	47
6.3	collect_domains.sh	47
6.4	verify_domains.sh	48
6.5	begin_session.sh	49



6.6	display_session.sh	49
6.7	include_processors.sh (edit session)	51
6.8	ls_processtypes.sh	51
6.9	add_process_type.sh (edit session)	51
6.10	add_process_instance.sh (edit session)	52
6.11	insert_expression.sh (edit session)	53
6.12	route_output.sh (edit session)	54
6.13	upload_session.sh	55
6.14	start_trace.sh	56
6.15	stop_trace.sh	57
6.16	unload_session.sh	57
6.17	exclude_processors.sh (edit session)	58
6.18	remove_process_type.sh (edit session)	58
6.19	remove_process_instance.sh (edit session)	59
6.20	delete_expression.sh (edit session)	59
6.21	end_session.sh	60
6.22	reset.sh -f	60



# 1 Overview

Application Trace (AppTrace) is a service available in the vDicos execution environment. AppTrace is used for tracing vDicos applications selectively and safely.

Selectively, in the sense that the trace user can choose precisely what to trace. Safely, in the sense that any reasonably specified trace session does not itself cause significant disturbance to the system.

The main purpose of AppTrace is to provide practical assistance in the troubleshooting of applications on live systems. Using AppTrace, a trace user can gain insight into the current behavior of an application. This is important when an application does not behave as expected. In some cases, AppTrace provides a better understanding of how an application works.

The inherent problem with observing the behavior of a system by tracing is the consumed capacity of the tracing itself. If the cost is too high, it can interfere with the primary function of the system, at worst even causing system failure. Effort has been put into minimizing the consumed capacity of AppTrace.

Some of the responsibility of keeping down the consumed capacity of tracing also falls on the trace user. AppTrace provides tools to scope the trace and to filter the output at the source. It is ideal to minimize the trace, not only to save resources, but also to remove as much “noise” as possible from the data that is of interest.

Although AppTrace can also be useful in function test and debugging, it is not a source code debugger. The use of AppTrace does not require knowledge of or access to the source code. The only documentation needed, besides this document, is the documentation on the trace model of the application to enable you to select relevant trace domains.

## 1.1 Target Groups

This document is intended to be read by the following target groups:

- Specially trained Ericsson external customers (operator personnel), when using AppTrace for live-site troubleshooting.
- Ericsson support personnel, when using AppTrace for live-site troubleshooting.
- System testers, when using AppTrace for troubleshooting test systems.
- Application programmers when preparing the application software for the possibility to use the AppTrace service.



- Application programmers using AppTrace during function testing.

It is important that the operator personnel and the Ericsson support personnel, intending to use AppTrace on a live system, have a good understanding of AppTrace and its use. They need to be sure of what they are doing. Misuse of the related tools can have severe impact on the in-service performance (ISP). It is recommended to read and understand this document before trying to run a trace session. This is to avoid unnecessary problems at a live site and to avoid unnecessary support cases.

**Note:** If you want to use AppTrace on a test system, you can jump directly to the examples in Section 5 on page 41. If you run into problems or have questions on AppTrace, consult this User Guide thoroughly before contacting support.

Consult the reference manual, see Section 6 on page 47, if there is any problem or question regarding a particular AppTrace command.

## 1.2 Prerequisites

To create and run an AppTrace session, the following conditions must be met:

- The application to be traced must be prepared, by the application programmer, for AppTrace. This includes documentation describing the trace domains of the application.
- Before attempting to trace, read the application documentation describing the trace domains.
- You must have access through IP to a node of the target system. The apptrace command line utilities are located under the `/opt/lpmsv/bin/apptrace` directory. Go to the directory to log on through the Secure Shell Service (SSH) to one of the control nodes of the target system.

## 1.3 Terminology

### Trace user

The intended reader of this document, see Section 1.1 on page 1. More generally, it is a person with access to a target system, with a need to obtain information on the behavior of that system. The trace user includes vDicos ADE users, but also Ericsson support using AppTrace on live sites. The term “AppTrace user” is a synonym.



<b>Trace domain</b>	Denotes a class of trace events, that is, a set of trace events of a particular kind. The trace domain determines the name and type of parameters used to present a trace event. A trace domain is identified by a hierarchical domain name, for example, <code>appl.test1.call11</code> . For details, see Section 2.1 on page 5.
<b>Trace event</b>	A trace event is a set of parameter values describing one occurrence of a certain piece of functionality that is traced. A trace event belongs to a trace domain. The parameters (name, type, and order) are determined by this trace domain. A trace event is generated by a trace point.
<b>Trace point</b>	A statement in the source code that conditionally generates a trace event. A trace point always belongs to a trace domain that determines its type (trace event type). There can be many trace points of the same type. All such trace points belong to the same trace domain. The application programmer has prepared the application source code with trace points. At runtime, the conditional of the trace point is sensitive to the selection criteria chosen by the trace user.
<b>Trace session</b>	A period of interaction between a trace user and the AppTrace service. A trace session is started by the <code>begin_session.sh</code> command described in Section 6.5 on page 49, and terminated by the <code>end_session.sh</code> command described in Section 6.21 on page 60. In other words, the trace session is “edited” by the trace user, meaning that the user edits the scope and filter for the session.
<b>Trace expression</b>	<p>Trace expressions are added to the trace session, by the trace user, to capture the information they need. A trace expression consists of the following:</p> <ul style="list-style-type: none"><li>• Trace domain</li><li>• Condition (optional)</li><li>• Actions (optional)</li></ul> <p>A trace expression matches a trace point if the trace domain of the expression matches the trace domain of the trace point and if the condition evaluates to true. If the trace expression matches, the actions are executed.</p>



<b>Trace program</b>	An ordered list of trace expressions belonging to the trace session and created by the trace user. When a trace point is encountered during execution, the trace expressions of the trace program are evaluated in sequence. The first match found is selected and its actions executed.
<b>Trace control state</b>	The cluster-global and coordinated state of the AppTrace service. This state is expressed as a value taken from a small set of labeled states, see Section 2.2 on page 6 for details. The state determines which commands are currently accepted by AppTrace.
<b>AppTrace-CLU</b>	The set of command line utilities used to manage AppTrace. The AppTrace-CLU tools are available in the <code>/opt/lpmsv/bin/appttrace</code> directory of those target system nodes the vDicos is installed on.
<b>Software version</b>	The version of software existing in the system. A system upgrade creates new software version.

## 1.4 Related Information

Trademark information, typographic conventions, definition, and explanation of acronyms and terminology can be found in the following documents:

- *Glossary of Terms and Acronyms*
- *Trademark Information*
- *Typographic Conventions*





## 2 AppTrace Basics

This section describes the basics of AppTrace such as trace domain and the registration of trace domain.

### 2.1 Trace Domain

The most central idea for understanding AppTrace is the concept of a trace domain. A specific trace domain can be seen as a “region” of tracing.

Trace domains are identified by a domain name. A trace user interested in obtaining trace output from a trace domain adds the name of the domain to the trace session.

The trace domains available at a site, can be fetched using the `verify_domains.sh` command, see Section 6.4 on page 48, or the `ls_domains.sh` command.

The top domain `hlr` can, for example, correspond to the Home Location Register application (this is a fictional example, not a currently existing AppTrace domain).

Immediately under the application top domain, there is usually a fixed set of subdomains representing tracing categories. One possibility is to have a subdomain `hlr.use_case` intended for tracing use cases. Below this, each use case has its domain. For example, `hlr.use_case.location_update` corresponds to the “location update” use case of the HLR application. Using this domain in a trace session is appropriate for tracing the logic controlling the location update use case.

Other parts of the application software are associated with resources rather than use cases. There could be a `hlr.subscriber` domain with subdomains for the operations that are applied to HLR subscribers, such as `hlr.subscriber.create.sh` or `hlr.subscriber.location_update.sh`.

#### 2.1.1 Trace Domain as Data Type

Each application defined trace domain can also have parameters. Each parameter has a name and a type. The sequence of parameters defined for a trace domain is also called the “signature” of the domain. A trace domain can be seen as a struct or record type, where the parameters are the attributes (data members).

A trace domain denotes a set of traceable events, but in addition, each such event is described by the values of the parameters.



When a trace user selects a trace domain for tracing, AppTrace generates output for each event belonging to the trace domain, including the values of the trace domain parameters.

In more refined trace sessions, the trace events can be filtered based on the values of the parameters. The trace user can also restrict the output to contain only some of the parameters (a projection).

Parameters are inherited from domain to subdomains. If `hlr.subscriber` has a parameter `SubscriberId` defined on the data type “integer”, `hlr.subscriber.location_update.sh` also has that parameter. A subdomain can have its own parameters. These are logically “appended” to the inherited parameters.

If a trace user selects the `hlr.subscriber` domain for tracing, all trace events that match `hlr.subscriber` are output, including events of any subdomain to `hlr.subscriber`. The output for each event by default includes the value for the `SubscriberId` parameter. The values of parameters specific to subdomains are not output, unless the subdomains were also selected explicitly to be part of the session (inserted into the trace program).

It is not certain that parameters are defined for all trace domains. Some domains are abstract and used only for classification. This is usually the case for higher-level domains. It is possible to select an abstract domain for tracing, but the output does not include any application-specific parameters.

All trace domains, even abstract domains, have a predefined set of system parameters. The domain name is one of the system parameters. The system parameters can be seen as defined in an anonymous root domain, from which all top domains inherit. The predefined system parameters are explained in Section 4.3.5 on page 28.

## 2.1.2 Registration of Trace Domains

The trace domain is defined in the load module. Normally, the trace domain is available for AppTrace immediately after this load module is loaded into a processor. The load process happens after cluster reload or processor reload. Such trace domains are registered in processor memory at load time.

If the traces session stops, it is mandatory to reproduce the same trace session and execution scenario a second time. It is important to recollect and reverify the domains to see if there is any inconsistent domain.

The problem is that during an ongoing trace session there is no way to verify the consistency of the redundant domains. For more information about the signatures of the domains, see Section 4.3.4 on page 25. About collecting and verifying domains, see Section 3.1 on page 9.



## 2.2 Trace Control State

The trace control state is a cluster-global and coordinated state of the AppTrace service. The trace control state determines which commands are currently accepted by AppTrace. Use the `display_session.sh` command, see Section 6.6 on page 49, to print the trace control state.

The observed state is one of the following:

Unknown	This is the initial state. AppTrace has not established any cluster-global state yet. There cannot be any existing session. You can see this state after a processor reload, a cluster reload, or a reset, see Section 6.22 on page 60. As soon as you do a <code>begin_session.sh</code> , see Section 6.5 on page 49, you transit to the “Clean” state.
Clean	This is the normal “idle” state. AppTrace is “healthy” and there is no ongoing resource consuming AppTrace activity. If any trace session exists, it is open for editing. This is the state you are in when you are finished with the tracing activity.
Ready	AppTrace is ready to start trace, whenever the <code>start_trace.sh</code> command is issued, see Section 6.14 on page 56. This state is reached from the “Clean” state by the <code>upload_session.sh</code> command, see Section 6.13 on page 55, or from the “Running” state by the <code>stop_trace.sh</code> command, see Section 6.15 on page 57. This state also means that there is a frozen trace session that has been uploaded to (installed on) all included processors. Depending on the routing variant, AppTrace can also have started a dynamic gateway process on each processor. When you have finished the tracing, remember to transit to the “Clean” state through the <code>unload_session.sh</code> command, see Section 6.16 on page 57, before disconnecting. This is to avoid having unused resources allocated indefinitely at one or more processors in the cluster.
Running	AppTrace has an executing trace session. The only way to reach this state is by the <code>start_trace.sh</code> command, see Section 6.14 on page 56, from the “Ready” state. You can return to the “Ready” state by the <code>stop_trace.sh</code> command, see Section 6.15 on page 57. If there is an overload, AppTrace automatically returns to the “Clean” state. If there is a fault, such as a processor reload, AppTrace does an automatic reset, dropping to the “Unknown” state.



If the trace has a control state that is not mentioned, the AppTrace is faulty.  
A `reset.sh -f`, see Section 6.22 on page 60, returns AppTrace to the  
“Unknown” state.



## 3 AppTrace Sessions

There are four distinct steps the trace user needs to perform to correctly manage a trace session:

1. Collecting and verifying trace domains
2. Defining an AppTrace session
3. Executing an AppTrace session
4. Cleaning up

These four steps are described in the following sections.

### 3.1 Step 1: Collecting and Verifying Trace Domains

This section describes how to collect and verify trace domains.

#### 3.1.1 General

The trace domains must be collected and verified to be consistent globally over the cluster. The domains are registered at loadtime at the individual processors. This is required to create any trace session.

After establishing the set of consistent domains for the cluster there is no need for collection and verification again. These actions are required if something alters the raw set of registered trace domains at one or more processors. This can only happen as a result of a system upgrade, when there is a change of software version.

The collected and verified trace domains are stored persistently in Database. Therefore a cluster reload does not cause a need to recollect and reverify trace domains (assuming that a backup has been taken).

The collected, verified, and saved domains are also “stamped” with the current software version. The trace control mechanism detects whether a software upgrade has been performed **after** the last `collect_domains.sh`. In this case the mechanism refuses to allow a trace session to be used if it was based on an old software version.

#### 3.1.2 Collecting Domains

The `collect_domains.sh` command, see Section 6.3 on page 47, gathers the trace domains registered at all processors in the cluster. Duplicate domain definitions (using the same domain name) are consolidated into one database



object. Collected domains can be displayed using the `ls_domains.sh` command.

The command involves distributed processing and database transaction processing. Depending on the number of processors and trace domains, it can take some time to complete, normally a few seconds. In rare cases (if there is a processor reload during the execution of the command), the command could fail to register some domains. If this happens, you see error messages in the output from the command. Try the command again.

### 3.1.3 Verifying the Collected Domains

The `verify_domains.sh` command, see Section 6.4 on page 48, verifies global consistency for all collected domains. Only domains that have been verified “OK” can be used in trace sessions. The command updates the persistent data for each domain with the verification status and possible error messages.

On completion, the command outputs a summary of the verification status for all domains. The command can be re-executed simply to obtain this summary. More detailed information on each domain can be found by doing a `cat_domain.sh` on the domains returned by `ls_domains.sh`.

A domain cannot be verified “OK” and is not traceable if any of its subdomains fail verification. However, a domain can be verified “OK” even if its parent domain is not verified “OK”.

For more details about how to obtain information about the status of domains and parameters, see Section 4.3.4 on page 25.

### 3.1.4 Potential Problems in Step 1

This section describes potential problems in step 1, such as missing domains.

#### 3.1.4.1 Missing Domains

There are a few reasons why a domain could fail to be collected.

If the domain name is not well-formed, it is ignored with a clear error message from the `collect_domains.sh` command. A well-formed domain name consists of a top domain name followed by zero or more names separated by dots (“.”). Each name can only contain letters (a-z) or numbers (0-9) or underscore (“\_”). The top domain does not allow upper case letters and does not allow underscore.

In some cases, expected specific domains are not listed by `ls_domains.sh` and no relevant errors are reported by `collect_domains.sh`. This can be because the application load modules of the current software version do not have these domains.



If there are too many domains, the system can fail to register some of them because of a memory limit being reached. AppTrace has a limit on the amount of kernel memory that it can use for registering trace domains. If there are many domains, this limit can be reached. There is no explicit alarm for this error case. Check in the `syslog` or console logs for error messages instead. If there is a memory problem, contact vDicos support for help.

Finally, domains can be missing because they are not registered at load time and, therefore, need dynamic registration. For the risks of dynamic registration of domains, consult the documentation of the application.

To collect dynamically registered domains, perform the steps described in this section for setting up a session. When defining the scope of the session, described in Section 3.2.2 on page 12, include all processors and all process types expected to use the missing domains. Since the expected domains are missing, they cannot be included in the scope. The scope can even be empty of any trace domains. Despite the incomplete scope, upload the session and let it execute for a while. For a domain to get registered, execution has to pass a trace point using the domain. Therefore, let the session execute long enough for the relevant use cases to get executed. Stop the session and unload it.

Execute `collect_domains.sh` again. The relevant domains are now printed by `ls_domains.sh`. It can happen that you need to repeat the procedure. Once you start using the new domains in sessions, additional domains can get registered dynamically. There is no risk or problem caused by an “unnecessary” order to collect or verify domains.

## 3.2 Step 2: Defining an AppTrace Session

This section describes how to define an AppTrace session.

### 3.2.1 General

There are two basic difficulties for the trace user in this step:

- Understanding the application trace domains including the **meaning** of the events they generate.
- Understanding how to limit the trace by scope and filter.

If the application has provided a rich set of trace domains and parameters describing the domains, the trace user can scope and filter the information space, producing a successful trace.

The `begin_session.sh` command, see Section 6.5 on page 49, creates an empty trace session. The trace session is “edited” to define the scope and filter.

In principle, it is possible for the trace user to use a maximal scope and no filters, generating and postprocessing a flood of output. This approach does not work in most cases, particularly on live systems. Even if this approach can



work on a test system, time can be saved in post processing, if the trace is trimmed at the source.

### 3.2.2 Defining the Scope of the Trace

The scope of the trace is the most important part of determining the cost (execution, memory, communication) of the trace. It defines which trace points are **turned on** when the session is executed:

- When execution encountering trace points that are off (the normal case), produces low cost.
- When execution encounters trace points that are on (in scope), evaluation of parameters, evaluation of expressions, and generation of output increase the cost significantly.

The scope consists of the intersection of three sets, as follows:

- The set of processors to trace.
- The set of processes to trace.
- The set of trace domains to trace.

The scope of a session also includes a fourth criteria: the **desired trace level**. However, assigning the desired level is done in Step 3, when starting the trace and not as part of editing the trace session. This is explained in Section 3.3.2.1 on page 14.

A trace point is turned on only in the following cases:

- The processor, on which it is executing, is included.
- The process, in which it is executing, is included.
- The trace domain, to which it belongs, is included.
- The level value of the trace point is lower than the desired level of the session.

Where the first three are defined by editing the trace session, use the following commands:

- Add which processors to include in the session by using the `include_processors.sh` command, see Section 6.7 on page 51.
- The processes to include can be specified to be entire process types, by `add_process_type.sh`, see Section 6.9 on page 51. This means that all processes of this type are traced, even processes that do not exist yet, but are created at runtime when the session is running.
- Alternatively, specific process instances can be specified using `add_process_instance.sh`, see Section 6.10 on page 52. Such





processes must exist in both cases, when the session is being specified and when the session is running. Short lived dynamic processes cannot practically be traced by specifying the instance.

- The trace domains to be included are specified using the `insert_expression.sh` command, see Section 6.11 on page 53.

At any time, the current state and contents of the session can be viewed by `display_session.sh`, see Section 6.6 on page 49.

### 3.2.3 Filtering the Trace

All trace points that are included in the scope and are encountered during execution contribute to the execution and communication cost of the trace. It is possible to reduce the volume of output and the communication cost, by adding filters. This is done by elaborations on the `insert_expression.sh` command, see Section 6.11 on page 53.

**Note:** Filter evaluation does add a cost, it is the cost of execution.

An effective filter is a filter that discards many trace events and those events where there is no way to eliminate the same events by tighter scope. Eliminating trace events by scoping is much more efficient than any filter.

A filter can express selections, projections, or assignments.

A filter can select trace event parameters with certain values that are of interest. Events with other values on the parameters are discarded. The parameters used for selection are usually the specific parameters of the trace domain, but can also include system parameters, see Section 4.3.5 on page 28, such as `$forlop`.

A filter can project the result so that only some parameter values are logged to the output. By default, the output includes all application-specific parameters and no system parameters. A projection filter specifies exactly which parameters are of interest. This can include system parameters, such as source file, source line, and process name.

A filter can assign values to variables, for example, a forlop identity can be injected to be used later in selections.

The details on filtering can be found in Section 4 on page 19.

### 3.2.4 Output Routing

Use the `route_output` command to adjust the output routing for the session, see Section 6.12 on page 54.



## 3.3 Step 3: Executing an AppTrace Session

When the editing of the trace session is finished, it is possible to execute the trace. On live systems, it is recommended that a first try is done with only one processor included. Use a processor in a relevant pool. If the trace uploads and runs without any problems, in particular without significantly increasing the execution load, it is safe to add more processors (if necessary).

### 3.3.1 Upload the Session

The `upload_session.sh` command, see Section 6.13 on page 55, freezes the trace session and installs it at the included processors. The freezing prevents the trace user from further editing the trace session. Only when the trace session is unloaded is it open again for editing.

If the `upload_session.sh` command has completed successfully, the trace control state is `Ready`.

### 3.3.2 Start the Trace

The `start_trace.sh` command, see Section 6.14 on page 56, releases the trace for execution.

The `start_trace.sh` command enables the desired trace level to be set. Only trace points with a level value lower than the desired level selected at `start_trace.sh` are part of the scope.

Not all applications utilize the level parameter. For more information, consult the trace model documentation of the application.

#### 3.3.2.1 Trace Level

The trace level is a system parameter. Each trace point in the application assigns a value for level, along with the values of other parameters. The level parameter is a secondary classification of trace points, independent of trace domain.

The level parameter orders the trace points into categories of detail. The higher the value of the level parameter, the more detailed the trace is.

The meaning of “detailed” is intentionally a bit vague. It is roughly equal to Frequency of events across the cluster. Thus there are more events in the cluster occurring at level  $N+1$ , than at level  $N$ .

The screening provided by the level is the fourth dimension of the scope. The other three are: processors, processes, and domains. With the other three dimensions fixed, the trace user can, in a figurative sense, turn a “volume dial” to adjust the volume of output. On live systems, it is recommended that for the



first attempt to start a trace session, the trace user uses a relatively low value for the desired level.

Some applications can decide to ignore the level. All trace points of such an application must then have a default value in one of the ranges, explained in the following sections.

### 3.3.2.2 Ranges of Levels

There are 64 levels allowed. These are divided into four ranges of 16 values each. Each range has predefined default level in the middle of the range. Each range is also associated with a predefined generic meaning, that is, the meaning of the ranges is the same for all applications. Within each range, each application is free to define if and how to use the detailed levels. The four ranges and their defaults are:

Range label	Default value	Meaning
-----	-----	-----
Debug-trace	55	Trace points in the range from 48 to 63 are only intended for function testing and debugging (non-live site tracing). If you work on a live site, do not trace in this range, because the trace is likely to compromise the real-time characteristics of the application.
Minor-trace	39	Trace points in the range from 32 to 47 are intended for detailed subsystem live site tracing. Effort by the trace user is needed to limit the scope and to use filters. The documentation of the application must be consulted before starting the session.



Major-trace	23	Trace points in the range from 16 to 31 are intended for broad, system-wide, live site tracing. The trace user can use a large scope and no filters. Although consulting the application documentation about how to set filtering is not necessary, the documentation can help in understanding the output.
Critical-trace	7	Trace points in the range from 0 to 15 are intended for live site tracing of only exceptional events. Trace points in this range can even be statically compiled in production code.

If the trace user does not explicitly choose any desired level in the `start_trace.sh` command, the chosen default for the desired trace level is 32. This means only trace points in the Major-trace and Critical-trace are included.

If the documentation of the application states that its trace model does not support the level parameter, all public trace points of the application must have a certain level value in the Minor-trace range. To trace such an application, the trace user needs to set explicitly the desired level in `start_trace.sh` to a higher value, otherwise all trace points are out of scope. However, the trace user also needs to set the other dimensions of scope restrictively and possibly use filters. For more information, consult the documentation of the application.

Applications that support major trace (in the range Major-trace and Critical-trace) ease the burden on the trace user. Trace points in this range have been designed not to overwhelm system resources, independently of scope. For more information on the relevance of domains for Major-trace, consult the documentation of the application.

**Note:** Do not trace in the Debug-trace range at live sites, except in emergencies, and even then only with direct assistance from application designers.

### 3.3.3 Stop the Trace

Use the `stop_trace.sh` command to stop the trace when the desired information has been obtained, see Section 6.15 on page 57.

A trace that has only been stopped is still uploaded and can be restarted, possibly with a different value for the desired level.



### 3.3.4 Potential Problems in Step 3

Try to judge the impact of the session that is specified. On live systems, include only one processor for the first upload. When starting `start_trace.sh`, try to monitor the load at the processor and any other problems such as crashing processes.

A running trace can always be stopped manually by a `stop_trace.sh` or by the `reset.sh -f` command. The latter is more drastic in that it restarts all trace control processes, and the contents of the trace session can be lost.

The reset mechanism can get started automatically because of AppTrace-internal communication problems due to a process crash or a processor reload.

The trace session is also ended because of an interfering upgrade. This is the case when the trace program is using a trace domain that is defined in a load module which is being upgraded.

AppTrace is not a source code debugger. Trace session data has a message size limit of 64 KB. If the session data exceeds this limit, displaying and uploading the session fails. In this case try to use less or shorter domain names and trace expressions.

## 3.4 Step 4: Cleaning Up

Use the `unload_session.sh` command when the trace session is finished, see Section 6.16 on page 57.

The command frees resources across the cluster and disables the processes that have been enabled for trace. If you do not unload, the trace session remains uploaded until the next processor reload occurs in the system.

The `end_session.sh` command removes the session. However, a session that is not uploaded does not occupy any significant resources. Not deleting it can be useful if you want to execute the same session again soon. The session, however, is not persistent, so the next processor reload drops it.





## 4 Filters - Detailed Trace Programming

This section explains the details of how to set up filters to reduce the communication cost of a trace session, beyond that of minimizing the scope. The scope of the trace session consists of four components: processors, processes, trace domains, and the desired trace level. Only trace points that agree on all four are “turned on”. When execution encounters such a “turned on” trace point, a trace event is generated.

Additional restrictions, which discard trace events generated in the scope, or modify their output, can be expressed in the **trace program** of the session as filters. Such additional restrictions are the topic of this section.

This section can also be described as “advanced use of the `insert_expression.sh` command”.

### 4.1 Execution Model for Trace Programs

The trace user specifies the trace domains to be included in a trace session using the `insert_expression.sh` command. The result is a trace program, represented as an ordered list of trace expressions. The order reflects the order of insertion. By default, each `insert_expression.sh` appends an expression to the end of the program.

When a trace event occurs, the trace program is executed with the generated trace event as context. The execution evaluates the trace expressions of the trace program, starting with the top expression and continuing down the list. However, as soon as a **matching** expression is found, any actions of that expression are executed and the event terminates. This means that expressions below the matching expression are never evaluated for that event.

The simplest trace program is where each trace expression is only a trace domain name and none of the trace domains in the program overlap. In this simple case the trace program can be seen as just the set of domains defining part of the scope. A set has no order and no duplicates. Indeed, for this simplest case, it does not matter in which order domains are inserted and duplicate insertions of the same domain have no effect on the trace result.

Example:

```
[0]                hlr.use_case.send_routing_informati
                    on
[1]                hlr.use_case.location_update
```

For this simplest form of trace program, the order does not matter. The only match is the expression where the domain matches the domain of the trace



event. An event matches one or the other, or none of them, but never both. It is only when an event matches multiple trace expressions that the order between the expressions is an issue, and this cannot happen in the program example.

The following principle remains intact independently of how complicated the trace program becomes: the order of insertion never matters between distinct domains. Put in another way: the order of insertion of two trace expressions can only matter if the trace domains are the same, or one is a subdomain of the other.

For the two overlapping domains, `hlr.use_case` and `hlr.use_case.location_update` the order does matter. If the trace program is as follows, no event ever reaches the second trace expression:

```
[0]                hlr.use_case
[1]                hlr.use_case.location_update
```

Any event that matches `hlr.use_case.location_update`, also matches `hlr.use_case`. Since execution stops with the first match, the second expression is never reached. The effect is that only the parameters defined for `hlr.use_case` are printed, even if the event is of type `hlr.use_case.location_update`.

The program makes more sense if the order is reversed as follows:

```
[0]                hlr.use_case.location_update
[1]                hlr.use_case
```

Only events of type `hlr.use_case.location_update` match the first line, while events of type `hlr.use_case` that are not events of type `hlr.use_case.location_update` match the second. The effect is that events of type `hlr.use_case.location_update` print all parameters defined for that type, including any parameters specific to `hlr.use_case.location_update`.

A program like the first one (with an expression based on a more general domain inserted before an expression based on a more specific domain), makes sense only if the first expression is augmented with a restricting filter of some form. This is explained in the next section.

In summary, the execution model for the trace program, applied on a trace event is as follows:

1. Evaluate each expression of the program, one at a time in sequence, until a match with the event is found.
2. When a match is found, the actions of the expression are performed and execution stops for that event.





**Note:** The execution model is a model and not the exact and actual way that the trace program is represented and evaluated. The execution model describes to the trace user how to think when creating a trace program. The trace program is compiled and reorganized to optimize the execution.

In particular, for any given event, the only trace expressions evaluated are the ones that actually match the trace domain. This is why trace domains are part of the scope of the trace, rather than part of the filter. It is more efficient if the trace user can discriminate between two desired cases expressed on distinct domains. The other case is when the two expressions are on the same domain but with different filters.

## 4.2 Special Characters in Expressions

Always keep in mind that the AppTrace commands and expressions are entered in Linux shell, therefore special characters of the given shell have to be well known. In case the expression has to contain such a special character, then the appropriate use case has to be used on the special character for parsing so the special character is recognized as a regular one. Use cases are for example quoting and escaping. For more information on use cases, see Section 4.5 on page 33.

## 4.3 Trace Expressions

The general form of a trace expression is as follows:

```
<domain_name> ( <condition> ) => <action>, <action>, ..
```

The `<condition>` is a C-style expression constructed from elements described in Section 4.3.2 on page 22. The elements are combined using the normal numeric, logical, and bit-wise operators, see Section 4.3.3 on page 24. Other operators or functions are also allowed if the types are used in a correct way. The `<condition>` as a whole must be a numeric expression, but not all elements need to be numeric.

The `<action>` could generally be any kind of expression. It is typically an expression with side effects, such as logging parameters to the trace output, assignment to a variable, assignment of forlop ID, or a trigger. For more information, see Section 4.3.3 on page 24.

If a trace event matches the `<domain_name>`, and the `<condition>` evaluates to true (not zero), a match has been found. Each `<action>` in the trace expression, if there is any, is executed. The event then terminates.

If an event does not match `<domain_name>` OR if the `<condition>` evaluates to zero, the actions are not executed, and evaluation continues with the next expression of the trace program.

Variants of the general form of trace expression are possible:



Unfiltered domain, action implicit	<code>&lt;domain_name&gt;</code>
Filtered domain, action implicit	<code>&lt;domain_name&gt;(&lt;condition&gt;)</code>
Unfiltered domain with actions	<code>&lt;domain_name&gt; ⇒ &lt;action&gt;, &lt;action&gt;, ...</code>
Filtered domain with actions	<code>&lt;domain_name&gt; (&lt;condition&gt;) ⇒ &lt;action&gt;, &lt;action&gt;, ...</code>

The implicit action, if no actions are specified, is to log all the application parameters defined for `<domain name>`. If no parameters have been defined for `<domain name>`, that is, the domain is abstract, only the system parameter `$id` containing the concrete domain name is logged.

### 4.3.1 Data Type of Expressions

All expressions, both elements and more complex, have an expression type. The expression type determines how the expressions can be combined with other expressions using operators and functions. AppTrace recognizes three types, as follows:

- Numeric expressions. These are probably the most common. The value of the expression is interpreted as an integer. It can be expressed as a decimal, hex, or octal literal. The source can also be a parameter or variable.
- Address expressions. The value of the expression is interpreted as a memory pointer. The value can be used, to some extent, in expressions as a numeric value, but not always. It can be expressed as a hex constant, or the source can be a parameter or variable. Depending on context, an address expression can be pretty-printed when logged. See Section 4.3.4.1 on page 27.
- Data expressions. The value of the expression is interpreted as a memory pointer and an associated length value. The typical example of a data expression is an ASCII string. The source is a string literal, a string parameter, or a string variable. Floating point types are also handled as data expressions.

The trace user can usually stay ignorant of the data type of expressions. The `insert_expression.sh` command, see Section 6.11 on page 53, rejects expressions that do not have the correct type. When this happens, reformulate the expression. Determining the type of the components can then help in deciding how to reformulate the expression.

### 4.3.2 Trace Expression Elements

The trace expression elements are the “atoms” of trace expressions.



The following table describes the allowed kinds of elements. The token “abc” is to be interpreted as a string of alphanumeric characters. The token “n” is to be interpreted as a positive integer (decimal). The token “nnn” is to be interpreted as a string of digits, including A to F if hexadecimal.

Element	Description
-----	-----
nnn	Decimal literal (numeric).
0xnnn	Hexadecimal literal (numeric).
0nnn	Octal literal (numeric).
@nnn	Memory address literal in hex (address).
true	Boolean literal evaluates to 1 (numeric).
false	Boolean literal evaluates to 0 (numeric).
string(abc)	ASCII string literal (data).
"abc"	ASCII string literal (data).
[_abc]	ASCII string literal (data).
[nnn]	Binary data or string literal, nnn denotes here an even number of hexadecimal digits (data).
\$abc	Parameter of any type, see Section 4.3.4 on page 25, or numeric variable, see Section 4.3.6 on page 30 where abc is the name.
\$_abc	Address variable where abc is the name (address).
\$_abc	ASCII string variable where abc is the name (data).
\$_[abc]	Binary string variable where abc is the name (data).
\$n	Parameter of type numeric where n is the ordinal number of the parameter as determined by the trace domain (numeric).
\$_n	Parameter of type address where n is the ordinal number of the parameter as determined by the trace domain (address).
Vn	Numeric variable, n must be one decimal digit (numeric), see Section 4.3.6 on page 30.
F	Forlop Id system parameter, same as \$forlop (numeric).
P	Process Id system parameter, same as \$pid (numeric).
T	Thread Id system parameter, same as \$tid (numeric).
C	Capsule Id system parameter, same as \$cid (numeric).
c	Capsule Type Id system parameter, same as \$rtid (numeric).

The parameter is predefined by a trace domain or as a system parameter, see Section 4.3.5 on page 28 and Section 4.3.4 on page 25. The variable is defined in the trace program, see Section 4.3.6 on page 30. Most parameters



can only be read and not assigned to from the trace program. Parameters are instead assigned by the event context. An exception to this rule is the system parameter `$forlop` (or “F”).

String literals can be expressed in several ways. This is clarified in Section 4.5 on page 33.

### 4.3.3 Trace Expression Operators and Functions

Trace expression elements are combined using operators and functions to build more complex trace expressions. The following table describes the operators and functions currently supported, in order of precedence. The token “X” or “Y” is to be interpreted as an expression of any type. The token “N” or “M” is to be interpreted as an expression of numeric or address type. The Token “S” or “T” is to be interpreted as an expression of data or string type. The data type of the return value of the operator or function is given in parentheses - (numeric, address, data) - in the “Description” column.

Operator or Function	Description
-----	-----
<code>L</code>	Log domain name and all application parameters (default action).
<code>L(x, y, ...)</code>	Log elements and expressions x, y, ...
<code>sample(N)</code>	Returns true once every Nth occurrence (numeric). It is state-dependent, see Section 4.3.6 on page 30.
<code>size(S)</code>	Returns the length of the string or data (numeric).
<code>head(N, S)</code>	Returns the prefix string of length N from the beginning of string S (data).
<code>tail(N, S)</code>	Returns the postfix string of length N from the end of string S (data).
<code>seek(S, T)</code>	Searches from the front of T for the first match of S. Returns position and length from the <b>end</b> of T where the first match of S starts. If there is no match, 0 is returned.  It can be used as a boolean indicator, where S is part of T.  It can be used as input to <code>tail(N, S)</code> to extract the substring of T that starts with S (numeric).
<code>start</code>	Start trigger (numeric), state dependent, see Section 4.3.6.1 on page 31.
<code>stop</code>	Stop trigger (numeric), state dependent, see Section 4.3.6.1 on page 31.
<code>- N</code>	Negation (numeric).



$\sim N$	Bitwise NOT (numeric).
$!N$	Boolean NOT (numeric).
$X = Y$	Assign Y to X.
$N * M$	Multiplication of N and M (numeric).
$N / M$	Division of N by M (numeric).
$N \% M$	N Modulo M (numeric).
$N + M$	Addition of N and M (numeric).
$N - M$	Subtraction of M from N (numeric).
$X == Y$	Equality comparison (numeric).
$X != Y$	Inequality comparison (numeric).
$N <= M$	Less-than-or-equal comparison (numeric).
$N < M$	Less-than comparison (numeric).
$N >= M$	Greater-than-or-equal comparison (numeric).
$N > M$	Greater-than comparison (numeric).
$N \& M$	Bitwise AND (numeric).
$N   M$	Bitwise OR (numeric).
$N \wedge M$	Bitwise XOR (numeric).
$N \&\& M$	Boolean AND (numeric).
$N    M$	Boolean OR (numeric).
$N \Rightarrow X$	Conditional. If (N!=0) then evaluate X (numeric).

Parentheses “(” and “)” are allowed to group expressions and override precedence.

#### 4.3.4 Parameters

The parameters of a trace domain are defined by the application developer. For a description of the trace domains of the application, including parameter names and type, refer to the documentation of the application.

The trace user can also obtain basic information about domains by running first the `ls_domains` tool. The tool lists the successfully collected domains by running the `cat_domain` tool that shows information about the indicated domain. Perform the following to display the domain information:

```
cat_domain.sh <domain name>
```

The following is displayed:



Condition of the domain	Shows whether it is concrete or abstract and if it is verified “OK”. A domain that is not verified cannot be put in the scope of a trace session. If <code>verify_domains.sh</code> , see Section 6.4 on page 48, has been successfully executed, yet a domain remains not verified, this is because there is some consistency problem with the domain. See Errors in the bottom row.
Signatures	Shows the raw signature of the domain. This is provided by the application developer in the definition of the trace domain and its parameters. The load module and file or line where the definition is located are also shown. There can be several signatures if there are several different definitions for the same domain, in the cluster. This is one source of possible inconsistency.
Parameter types	This is the format type, see Section 4.3.4.1 on page 27, of the parameters. This controls how the parameter is printed to the log. The format type also determines the expression data type, see Section 4.3.1 on page 22, of the parameter.
Parameter names	These are the names of the parameters. The trace user refers to parameters by their name in trace expressions (a “\$” symbol is then prepended before the name).
Subdomains	This lists the names of the immediate subdomains of the domain.
Errors	Any error or warning messages produced by <code>verify_domains</code> are kept and shown here. If the domain is not verified “OK”, the explanation is found here.

A domain that cannot be verified is inconsistent in some ways. If the domain has duplicate definitions, the number of parameters, their order, and type must be identical.

Make sure that parameter names are identical (even though name discrepancies are allowed). A warning is attached to such a domain and the condition is “sloppy”, but verification status is still “OK”. Discrepancies on parameter name are not dangerous, but can cause confusion for the trace user in formulating trace expressions. A mismatch of parameter **type** can be dangerous in that a trace point could cause a crash of the process, if it were allowed. This is why such domains are not allowed to be verified “OK”.

A domain must also be consistent with its super domain and its subdomains.

The parameters (if there is any) of the super domain must agree with the leading sequence of parameters in this domain. A domain can be consistent



with its super domain and verified “OK”, even if the super domain is not verified “OK”. The super domain can be inconsistent in itself or inconsistent with some other subdomain.

All subdomains of a domain must be verified “OK”, for this domain to be verified “OK”. The reason for this is that a user putting a domain in scope also puts all subdomains in scope.

#### 4.3.4.1 Format Type for Parameters

Each domain has a “signature” which is a string defining the names and types of the parameters. The signature is designed by the application developer when defining a trace domain. If you are familiar with C or C++, you recognize the string as a `printf`-style format string, where each parameter is defined like: “%X” where “X” is an ASCII character acting as the type tag for the parameter.

The `verify_domains.sh` command extracts the name and the format type of each parameter. The format type is identified by this type tag. In AppTrace the basic set of format types is the same as the type tags for `printf`. AppTrace also uses some additional tags identifying some common data types used in vDicos. This enables AppTrace to pretty-print such types directly, without the application developer having to do the formatting.

The following table lists the current set of defined format types:

Format type	Expression data type	Formatted as
-----	-----	-----
u	Numeric expression	Unsigned int decimal
d	Numeric expression	Signed int decimal
i	Numeric expression	Signed int decimal
o	Numeric expression	Unsigned int octal
x	Numeric expression	Unsigned int hexadecimal (abcdef)
X	Numeric expression	Unsigned int hexadecimal (ABCDEF)
c	Numeric expression	Char
p	Address expression	Pointer (hex)
s	Data expression	ASCII string
f	Data expression	Float or double decimal
F	Data expression	Float or double decimal
e	Data expression	Float or double exponent
E	Data expression	Float or double exponent
g	Data expression	Float or double adjusts to f or e



G	Data expression	Float or double adjusts to F or E
b	Numeric expression	Boolean (true or false)
B	Numeric expression	Boolean (TRUE or FALSE)
S	Data expression	DicosString as ASCII string
Y	Address expression	DicosBCD_String 0-9 a-f
Y	Address expression	DicosBCD_String 0-9 A-F
z	Address expression	DicosBCD_StringArray 0-9 a-f
Z	Address expression	DicosBCD_StringArray 0-9 A-F
t	Address expression	DicosTime {hour:minute:second :millisecond}
T	Address expression	DicosTime {month day hour:minute:second year}
q	Address expression	DicosDBObjectDID (Database reference)
a	Address expression	DicosStringArray {{ASCII string}}{ASCII string},...
r	Address expression	DicosOctetString 0-9 a-f
R	Address expression	DicosOctetString 0-9 A-F
m	Address expression	DicosOctetArray 0-9 a-f
M	Address expression	DicosOctetArray 0-9 A-F
K	Data expression	DicosOctetArray as ASCII string
!	Address expression	DicosIntegerArray +/- 0-9
#	Address expression	DicosUnsignedArray 0-9
v	Address expression	DicosDuration (to be implemented)
V	Address expression	DicosDuration (to be implemented)
\$	Address expression	DicosFloatArray (to be implemented)
&	Address expression	DicosFloatArray (to be implemented)

### 4.3.5 System Parameters

The system parameters are defined for all trace domains. To illustrate, the system parameters are as if they were defined in an invisible root domain, from which all application top domains inherit. The set of system parameters is the following:





System Parameter -----	Format type -----	Description -----
<code>\$id</code>	s	The actual domain identity of the event. The matching expression can be based on a super domain of the actual domain. For abstract domains, where the developer has not defined any parameters, the value of <code>\$id</code> is what is printed by default.
<code>\$level</code>	u	The level value of the event. Not to be confused with the desired level value of the session. The level value of the event has to be less than the desired level value of the session, for there to be an event.
<code>\$forlop</code>	d	The current forlop identity associated with the thread. The <code>\$forlop</code> can be assigned in a trace expression and filtered on in others.
<code>\$file</code>	s	The source file in which the trace point resides. This is not necessarily the same file as where the trace domain is defined.
<code>\$line</code>	u	The source line on which the trace point resides, in the source file. This is not necessarily the same file or line as where the trace domain is defined.
<code>\$function</code>	s	The function in which the trace point resides.
<code>\$lm</code>	s	The load module identity of the load module where the trace point resides. This is not necessarily the same load module as where the trace domain is defined.
<code>\$pid</code>	u	The process identity of the process in which the trace event occurs.
<code>\$tid</code>	u	The thread identity of the thread in which the trace event occurs.
<code>\$cid</code>	u	The capsule identity of the capsule in which the trace event occurs.



<code>\$rtid</code>	<code>u</code>	The Run Time Identity (RTID) of the process in which the trace event occurs.
<code>\$processname</code>	<code>s</code>	The string name of the process in which the trace event occurs.
<code>\$processorname</code>	<code>s</code>	The string name of the processor on which the event occurs.
<code>\$time</code>	<code>t</code>	The time (hour, minute, second, millisecond) when the event occurred.
<code>\$date</code>	<code>T</code>	The date (year, month, day, hour, minute, second) when the event occurred.

### 4.3.6 Variables and State

If the trace user wants to capture a combination of trace events, there is usually a need to maintain some form of state between the evaluation of expressions for several events. This is accomplished by the use of variables. A variable, such as `$tmp`, can be assigned a value in one event, that is later retrieved in another. Using variables in a trace program raises the expressive power (and complexity) of a trace program to a new level.

The most important part about variables is their scope. In AppTrace the scope of a variable is the vDicos process.

A variable, such as `$tmp`, then has a larger scope than the expression in which it occurs.

**Note:** If `$tmp` is referred to in several expressions in the trace program, then all refer to the same actual variable, as long as the expressions are evaluated in the same process.

Variables are not the only states possible in a trace program. Each occurrence of the `sample` function uses an internal counter. The following trace expression produces output once for every 1000 occurrences of an `hlr.use_case.location_update` event, in **each** enabled process.

```
hlr.use_case.location_update(sample(1000))
```

**Note:** If the `sample` function is used in several places, in the same expression or in different expressions, each use maintains its own counter.

Keep in mind that the trace program is being evaluated in all trace enabled processes concurrently and independently. Each process has its own copy of the variables and other state. The only common part in all enabled processes and all included processors is the trace program and desired session level. This means that the trace program logic has global scope, but the state has



process scope. The only exception to this is the `$forlop` system parameter, that has a special scope. For more information, see Section 4.4 on page 32.

#### 4.3.6.1 Trigger Expressions

Another feature based on process state is trigger expressions.

When the trace user starts the `start_trace.sh` command, usually the trace session starts producing output immediately. If the user is only interested in obtaining information **after** some particular event occurs, this can be expressed as a trigger.

A **start trigger** is an expression that contains the function `start.sh` as an action. Example:

```
h1r.use_case.location_update ($location == 123456) =>
start
```

A **stop trigger** is a corresponding expression that contains the `stop.sh` function as an action.

Trigger expressions “float to the top” of the trace program. Trigger expressions are inserted in order but always above any non-trigger expressions. This is handled automatically by `insert_expression`. A trace program can have several start triggers and several stop triggers. It can have only start triggers, but it cannot have only stop triggers. A trace program with stop triggers only never starts in the current implementation of AppTrace.

If a trace program contains any triggers, the evaluation of that program is influenced by a hidden state variable that can have the following three states:

Running-pending	No start trigger has fired. Only start triggers are evaluated. No stop triggers or normal expressions are evaluated.
Running-started	A start trigger has fired. No start triggers are evaluated. Stop triggers (if there are any) are evaluated. If no stop trigger matches, normal expressions are evaluated.
Running-stopped	A stop trigger has fired. Nothing is evaluated.

Any trace point that is encountered in the scope of a trace session starts to evaluate the trace program of the session. If the trace program has no triggers, the program is evaluated normally as explained earlier. But if the trace program has triggers, the process state is first Running-pending.

In the Running-pending state only start triggers are evaluated. If a trace event does not match any start trigger, nothing more happens for that event. If, however, any one of the start triggers matches and reaches the `start` action, the Running-started state is entered in that process. Subsequent events in that process are evaluated according to the Running-started state.



In the Running-started state, all stop triggers are first evaluated. If no stop trigger matches, normal expressions are evaluated, generating the desired output. Any event that matches a stop trigger causes a transition to the Running-stopped state for the process. Subsequent events in that process are evaluated according to the Running-stopped state.

In the Running-stopped trace, no expressions are evaluated. All events are filtered out.

Start triggers are suitable if the events of interest occur at a time much later than when the user is preparing the session. Another case is when it is not known when an event occurs, but when it does, the user wants as much information as possible.

A stop trigger is suitable if there is at least one start trigger and the user wants the trace output to stop when a certain event occurs. Again, the output is only stopped in the process where the stop trigger fired.

The start and stop triggers of AppTrace are local one-shot. That is, once fired in a process they do not fire again in that process. The only way to reset the trigger control state of all processes is by doing an `unload_session`.

## 4.4 Forlop Tracing

The `$forlop` system parameter allows the trace user to inject an identity at one trace event, and then to use that identity in filters to capture related subsequent events. The basic idea is to allow the trace user to trace an entire “call” (use case), as it propagates through the system by passing through various processes on several processors. To understand what is meant by “related event”, one needs to understand:

- What a forlop identity **is**.
- How a forlop identity is **injected**.
- What is **tagged** by the injected forlop identity.
- How an injected forlop identity **propagates** through the system.

A forlop identity **is** simply an integer value. The value of zero is predefined to mean “anonymous forlop”. It is also the default value.

A forlop identity is **injected** when a trace expression assigns a value to the `$forlop` system variable. Example:

```
h1r.use_case.location_update ($location == 123456) =>  
$forlop=123456
```

In this example, the forlop identity is assigned a value which happens to be the matched location. In general, the forlop identity can be any non-zero integer value.



**Note:** The AppTrace system variable assigned to `$forlop` is limited to 20 bits.

As long as the session only attempts to trace one forlop at a time, the exact non-zero value of the forlop identity is unimportant. Create unique forlop identities in case the session attempts to trace several forlops concurrently, for the sessions to distinguish the different “calls” from each other. In the previous example if there are concurrent independent update location calls for the location 123456, then using the same forlop identity can produce a trace that is a confusing mixture of several calls. Use a variable that is incremented for each call and then assigned to the `$forlop` system variable.

```
h1r.use_case.location_update ($location==123456) =>
  $n=$n+1, $forlop=$n
```

The entity which is **tagged** by a forlop identity is the current thread. An untagged thread simply has the default forlop identity zero. A thread gets tagged when the thread executes a trace point, that generates a trace event, that matches a trace expression, that assigns the forlop identity. The thread remains tagged with the forlop identity until the thread reaches empty stack (goes idle), which resets the forlop identity to zero. This makes sense since the thread is finished with one job and ready to accept a new job, presumably involving a different call or forlop.

The real leverage of the forlop concept derives from the way forlop identity **propagates** between threads.

- If a tagged thread sends a message over TIPC (uses a vDicos dialogue), the receiving thread becomes also tagged. A forlop propagates over TIPC only inside the vDicos boundaries, but not outside of it (for example, between a vDicos and a process running outside of it).
- If a tagged thread spawns or forks a process or a thread, the new thread or threads become also tagged. A forlop propagates over thread creation.
- If a tagged thread creates a timer, the thread is tagged again when the timer expires. A forlop propagates over timers.

**Note:** Communication over IP does not, by itself, propagate forlop identity. It is, however, possible for the application to propagate forlop identity at the application protocol level refer to AppTrace documentation of the applications if this is supported by the application.

Assuming the trace user can find an appropriate way to inject a forlop identity, they can trace the forlop by defining the appropriate scope and then adding filters such as the following:

```
h1r.use_case.location_update ($forlop==123456)
```

In the example, the same trace domain is used both when injecting the forlop, and when tracing on it. This was only to keep the example as simple as possible.



## 4.5 String Expressions

Filters on strings are the most **costly** to evaluate. Parameters of type string are probably the most common, so the need to express filters on strings is inevitable.

The underlying data type for strings in AppTrace is **data expression**. A data expression has a value that is a sequence of bytes with a known length. The contents can be anything. In this User Guide and for AppTrace in general, the emphasis is on **ASCII strings**. However, it is possible to express and manipulate strings as binary data. This is described at the end of this section, see Section 4.5.4 on page 39.

### 4.5.1 ASCII String Literals

There are three ways to express ASCII string literals in AppTrace. For example, the string `Hello-world` can be created as an ASCII string literal by any of these three forms:

- `string(Hello-world)`
- `"Hello-world"`
- `[_Hello-world]`

The first form is probably more convenient and easier to remember for most users. If the string to be created itself contains many parentheses, the second form can be convenient, but it actually requires an additional level of quoting as described later. The third form is mainly used as the internal form, also understood as an ASCII string literal.

An expression using the first or the second form is converted to the third form before the whole expression is compiled. In cases where an expression has parse errors, you can see an error message that tries to indicate the error in terms of the third form. This happens even if the expression input by the user used one of the other forms. The `display_session.sh` command can be used to browse the compiled representation of the trace program. In this representation, ASCII string literals are also in the third form. This is why a user needs to know about the third form.

The reason for having more than one way for expressing ASCII string literals is convenience in terms of a complication for ASCII string literals. The complication is how to express parse controlling characters as part of the ASCII string. Examples of parse controlling characters are blanks and closing parenthesis. The problem with blanks is that they are regarded as “white space” by AppTrace and as argument separator by AppTrace-CLU, unless they are quoted. Similarly, the closing parenthesis “)” is interpreted as the termination of the ASCII string initiated by “string(...)”, unless quoted.

For example, the ASCII string `Hello world` could not be expressed as:



`string(Hello world)` since it could be interpreted as:

`string(Helloworld)` and printed as:

Helloworld

To express blanks successfully as part of an ASCII string literal, the blanks have to be quoted. Quoting is done using the normal shell escape characters: backslash (\), double quotes (" "), or single quotes (' ').

So the string `Hello world` could be expressed as:

`string(Hello\ world)` which could be printed correctly as:

Hello world

Alternative ways to express the same string are the following:

- `string(Hello' 'world)`
- `string(Hello" "world)`
- `string("Hello world")`
- `string('Hello world')`
- `"string(Hello world) "`
- `'string(Hello world) '`
- `'"Hello world"'`
- `\ "Hello\ world\"`

The two levels of quoting in example 7 and 8 are needed because the double quote character (") is here intended as the AppTrace delimiter of the ASCII string literal. If the double quote characters are not themselves in the scope of quotes, the double quotes are consumed as quotes by the AppTrace-CLU shell. They are shown in examples 2, 3 and 5.

If the scripting variant of AppTrace-CLU is used, three levels of quoting is sometimes necessary:

- The outermost level for the UNIX shell
- The next level for the AppTrace-CLU shell
- The innermost quotes for the AppTrace expression

As for the complications of quoting, the best advice is to try until the expression successfully compiles and the intended string is created. When compilation is not successful, there is an error message that helps. The error message on not successful compilation can not be used as help in case some special characters were used in the command that were not quoted or escaped, in



this case the output is not displayed on the standard output but redirected. If compilation is successful, it still can be the not intended string that is created. It is usually a good idea to start a `display_session.sh` to inspect the trace program. The command displays the compiled expressions included in the session. One can usually discern from the compiled expressions if the ASCII string literal is what is intended.

## 4.5.2 ASCII String Operators and Functions

Trace events that contain string parameters can be filtered by using operators and functions to compare string expressions.

### 4.5.2.1 String Equality Comparison

Using the equality operator (`==`), two string expressions can be compared for equality. Equality means equal length and equal contents. Example:

```
tsp.dbn.tdc.setstatus($processname == string(MyProcess))
```

In this example the value of the system parameter `$processname` is compared with the string literal `MyProcess`. Only processes with the name `MyProcess` pass the filter.

**Note:** For this case, a more efficient filter could use the numeric system parameter `$rtid` to achieve the same effect.

Processes with the name `MyProcess` have to be in the scope of the session for there to be any matches.

### 4.5.2.2 String Inequality Comparison

Examine the following example for the semantics of the string inequality operator `!=`:

```
tsp.dbn.tdc.setstatus($processname != string(MyProcess))
```

Only processes in scope and with a value for `$processname`, not equal to `MyProcess` match.

### 4.5.2.3 String Size Function

The function `size(S)` returns the length of the string expression `S`. Example:

```
tsp.dbn.tdc.setstatus(size($message))
```

This filter matches trace points in scope where the string parameter `$message` is not of zero length. Thus the filter avoids generating any trace output when the `$message` parameter is empty.





#### 4.5.2.4 String Head Function

The function `head(N, S)` returns a string expression of length `N` starting at the front of the string expression `S`, in other words, the first `N` characters of `S`. If `size(S)` happens to be less than `N`, the entire string `S` is returned. Example:

```
tsp.dbn.tdc.setstatus(head(3,$processname) == string(Hlr))
```

This filter matches trace points in scope where the name of the process starts with “Hlr”. Thus any trace points in scope being in a process with a name that does not start with “Hlr” are filtered out.

#### 4.5.2.5 String Tail Function

The function `tail(N, S)` returns a string expression of length `N` starting at `N` bytes from the end of `S`, in other words the last `N` characters of `S`. If `size(S)` happens to be less than `N`, the entire string `S` is returned. Example:

```
tsp.dbn.tdc.setstatus(tail(12,$file) == string(HlrLocup  
d.cc))
```

This filter matches trace points in scope where the file in which the trace point resides is `HlrLocupd.cc`. The `$file` system parameter evaluates to the whole directory path for the file. The tail function is used here to strip off the directory path.

#### 4.5.2.6 String Seek Function

The function `seek(S, T)` scans the string `T`, starting from the front of `T`, for the first match of `S`. It returns the length from the **end** of `T` where the first match of `S` was found. The return value is designed to work with `tail`. If no match is found, 0 is returned. This function can be used either as a boolean indicator of whether `T` contains `S`, or as a basis to extract the substring of `T` that starts with the first match of `S`. Example:

```
tsp.dbn.tdc.setstatus(seek(string(HlrLocupd_OU), $file))
```

This filter matches trace points in scope where the file in which the trace point resides has a pathname that includes the string `HlrLocupd_OU`. The string starting with `HlrLocupd_OU` can be obtained by the expression:

```
tail(seek(string(HlrLocupd_OU), $file), $file)
```

### 4.5.3 ASCII String Variables

ASCII string variables are expressed as: `$_xyz`, where `xyz` is the name. This is used as opposed to parameters and numeric variables which use the syntax: `$xyz`; and address variables which use the syntax `$_xyz`.

An ASCII string variable is typically used either to communicate a string between trace point evaluations, or to avoid the need to evaluate a complex ASCII string expression more than once. A complex expression can be evaluated and assigned to a variable. When the value is needed again, it is fetched from the variable. Example:

```
app.op1 =>  
$_x = tail(seek(string(start), $function), $function),  
size($_x) => L($_x)
```

The previous expression has no filter, so all such trace points that are in scope match and the actions execute.

There are two actions. The first action is an **assignment** to an ASCII string variable. The ASCII string expression assigned is the substring of `$function` that starts with the string “start”. If there is no match with “start”, the empty string is assigned.

The second action is a **conditional action**, because the operator `=>` is used. This operator is used to express the root conditional of the trace expression, but it can also be used as an action in itself. The root conditional (the left-most conditional) is special in that if it fails, the evaluation of the trace point continues with the next trace expression in the trace program. A non-root conditional that fails simply fails as an action. It does not alter the fact that the root conditional has succeeded (the expression has matched).

In the above case, the condition of the conditional is that the size of the ASCII string held by the variable `$_x` is not zero. When that is the case, the action is to print the value of the ASCII string variable. If the size is zero, nothing is printed.

Almost the same effect is achieved by the following expression:

```
app.op1(seek(string(start), $function)) =>  
L(tail(seek(string(start), $function), $function))
```

An important difference in this second variant is that it has a condition in the root conditional. Thus if the condition fails, the next trace expression is evaluated. When there is a match, this second variant does the string seek operation twice. The first variant does a string copy (to the variable). Which one is more efficient depends on the length of the seek and size of the copy.

A third version of the expression is actually the most efficient variant. This third variant uses a numeric variable to store the result of `seek` instead of the result from `tail`:

```
app.op1 => $x = seek(string(start), $function)  
$x => L(tail($x, $function))
```

These examples illustrate the following:

- How ASCII string variables can be used.



- The difference about the numeric variables.
- How to obtain the same result using a numeric variable, which is more efficient.

Remember that the scope of a variable is the process (capsule). This means that variables such as “\$<sub>x</sub>” and “\$x” retain their values for possible access in the evaluation of other trace points in the same process. This larger scope was not exploited in the previous examples.

#### 4.5.4 Binary Strings

Expressions of type ASCII string (literals, variables, function results) are a special case of **data expression**. The user can create ASCII string literals by typing in the literal ASCII characters. When the user prints an ASCII string expression (using the logging action “L”), the value is printed as an ASCII string. This is convenient as long as the data itself **is** ASCII and consists only of printable characters.

When the data to be filtered or output is not ASCII, or contains non-printable characters, then the trace user can handle the data as a binary string. The syntax for binary strings is a sequence of consecutive hexadecimal digits of even numbers, in square brackets.

For example, the ASCII string literal `string(ab0)` could be expressed as a binary string literal `[61626f]`. If the binary string literal is printed, it is displayed as:

```
[61626f]
```

In other words, the value of each byte is printed as two hex digits, and not as an ASCII character.

Binary string variables are expressed as `$_[xyz]` where `xyz` is the name. An ASCII string value can be assigned to a binary string variable. This is useful for the case when the ASCII string contains non-printable characters. It is impossible to express an ASCII string literal that contains non-printable characters. To create a filter matching a string that contains non-printable characters, convert it to a binary string first. The reverse, assigning a binary string value to an ASCII string is possible (it passes compilation), but not recommended. Currently, AppTrace does not support the expression of characters as octal `\0nn` or hex `\0xnn` in ASCII string literals.





## 5 Examples

The first example walks through the minimal trace case, from start to finish. The purpose is to illustrate the complete sequence of steps needed to perform a trace. Examples after the first example focus on a feature available in a particular command. The overall sequence to produce a trace is not shown since it is otherwise the same as in the first example.

### 5.1 Example1: The Simplest Possible Trace Session

In this example a trace session is created that traces one trace domain, `hlr.use_case.location_update` in one process type, `HlrDistributor` on one processor, `Processor1`.

#### 1. Connect

Begin by connecting to one of the target system processors (nodes) using SSH:

```
> ssh root@sc1.moose.cba
```

The IP address and username are only examples. Obtain a username and the IP address or hostname of one of the processors (nodes) in the system.

#### 2. Enter the AppTrace-CLU directory:

Change directory to `/opt/lpmsv/bin/apptrace`:

```
> cd /opt/lpmsv/bin/apptrace
```

#### 3. Collect and Verify Domains:

If you have not executed `collect_domains.sh` and `verify_domains.sh` after the latest system upgrade, do so.

- Collect domains:

```
> ./collect_domains.sh
Starting to collect domains at all processors -\
will take a few seconds
====CollectDomains====
Domains collected with no problems for \
processor: Processor1
Domains collected with no problems for \
processor: Processor2
....
AppTrace collect_domains.sh done
>
```



- Check

Check which domains have been collected by running:

```
> ./ls_domains.sh
```

- Verify

Verify the domains:

```
> ./verify_domains.sh
ROOT
hlr[abstract|OK]
hlr.use_case[abstract|OK]
hlr.use_case.location_update[concrete|OK]
>
```

#### 4. Begin a Session:

```
> ./begin_session.sh
AppTrace begin_session.sh done sessionId:
1078314921
>
```

#### 5. Add a Processor to the Scope of the Session

Include `Processor1` in the scope of the session:

```
> ./include_processors.sh Processor1
AppTrace include_processors.sh done
>
```

#### 6. Add a Process Type to the Scope of the Session

Add the `HlrDistributor` process type to the scope of the session:

```
> ./add_process_type.sh HlrDistributor.10001
AppTrace add_process_type done
>
```

**Note:** Process types can be listed by `./ls_processtypes.sh`

#### 7. Add a Trace Domain to the Scope of the Session

Insert an expression with the `hlr.use_case.location_update` domain. This adds the domain to the scope of the session:

```
> ./insert_expression.sh hlr.use_case.location_update
===Trace Program(simplified)===
[0]:hlr.use_case.location_update
AppTrace insert_expression done
>
```



**Note:** Domains can be listed by `./ls_domains.sh`

## 8. Upload the Session

The session now has a complete scope. You can upload the session. It is uploaded only to Processor1 because that is the only processor included in the scope.

```
> ./upload_session.sh
====Upload Session====
====Enabling Processes====
AppTrace upload_session.sh done
>
```

## 9. Start the Trace

Finally the trace session can be started:

```
> ./start_trace.sh
Trace level is 32 [trace events in ranges: \
major and exceptional]
====Starting Trace====
AppTrace start_trace.sh done
>
```

You now see output from the trace, if there are any `hlr.use_case.location_update` events occurring in some `HlrDistributor` process on `Processor1`. The output ends up in the console logs by default.

## 10. Stop the Trace:

```
> ./stop_trace.sh
====Stopping Trace====
AppTrace stop_trace.sh done
>
```

The trace can now be started again, possibly with a different desired level, or the session can be unloaded.

## 11. Unload the Session

Remove the session from all included processors. This frees up resources allocated for the trace at these processors. Perform the following:

```
> ./unload_session.sh
====Deactivate Session====
New sessionID: 1136896256
AppTrace unload_session.sh done
>
```



The session is now open for editing again. Alternatively, the session can be ended.

#### 12. End the Session:

```
> ./end_session.sh
AppTrace end_session.sh done
>
```

## 5.2 Example 2: Add Several Processors to the Scope

To add several processors to the scope of the session, simply provide them as arguments to the `include_processors` command. This can be done either by several commands, or several arguments to one command.

```
> ./include_processors.sh Processor1 Processor2 Processor3
AppTrace include_processors.sh done
>
```

When all processors of the cluster are to be in the scope of the trace, it is possible to use the `-a` flag:

```
> ./include_processors.sh -a
AppTrace include_processors.sh done
>
```

On live systems, it is not recommended to start a new trace session with all processors in the scope. If possible, first test the behavior of the session by starting the trace with only one processor in the scope.

If the majority of the processors of a large cluster are to be in the scope, it is convenient to include all processors using the `-a` flag, and then to exclude some processors:

```
> ./include_processors.sh -a
AppTrace include_processors.sh done
> ./exclude_processors.sh Processor1 Processor2
AppTrace exclude_processors.sh done
>
```

To see which processors are currently in the scope of the session, use the `display_session` command.

## 5.3 Example 3: Add Several Process Types to the Scope

To add several process types to the scope of the session, simply provide them as arguments to the `add_process_type` command. This can be done either by several commands, or several arguments to one command.

```
> ./add_process_type.sh MyTestProcess HlrDistributor
AppTrace add_process_type.sh done
```





>

The flag `-l` works in the corresponding way for load modules.

To see which process types are currently in the scope of the session, use the `display_session` command.

## 5.4 Example 4: Use Wildcard to Add Several Similar Domains to the Scope

The `insert_expression` command supports “wildcard” matching on names for subdomains. The support is limited to complete subdomain names.

The wildcard in the domain name is expanded to all matching names and the result is the same as starting the `insert_expression` command for each matched name.

```
> ./insert_expression.sh hlr.use_case.*
===Trace Program(simplified)===
[0]:hlr.use_case.location_update
[1]:hlr.use_case.sri
[2]: ...
AppTrace insert_expression.sh done
>
```

See Section 6.11.1.1 on page 54 for more details on the use of wildcards.





## 6 Reference Manual for AppTrace Commands

The following commands are available under the `/opt/lpmsv/bin/apptrace/` directory on a target processor which has vDicos installed on it (that is, a node). The commands are listed more or less in order of use.

All commands listed in the following sections have a wrapper command version available. All of these wrapper commands start with the “`vdicos-apptrace-`” prefix. This means that, for example, the wrapper command of `ls_domains.sh` is `vdicos-apptrace-ls_domains`. Unlike the commands without the wrapper prefix, these commands are available on all target processors which have LemSCPool installed on them (these are usually the controller processors).

### 6.1 `ls_domains.sh`

The `ls_domains.sh` command is used to display the collected domains.

### 6.2 `cat_domain.sh`

The `cat_domain.sh` command is used to show detailed information on the specified domain given as a parameter. For more information, see Section 4.3.4 on page 25.

### 6.3 `collect_domains.sh`

The `collect_domains.sh` command gathers the trace domains registered at all processors in the cluster. Collected domains can be fetched with the `ls_domains.sh` command.

If you have executed `collect_domains.sh` successfully, it is normally not necessary to execute again until after the next software upgrade. For more information, see Section 3.1.4.1 on page 10.

#### 6.3.1 Use

```
collect_domains.sh [ -h]
```

`-h`                      Print a help summary



### 6.3.2 Common Errors for `collect_domains.sh`

If the `collect_domains.sh` command completes without errors, yet domains that you expect to fetch with `ls_domains.sh` are missing, this could be for a number of reasons, see Section 3.1.4.1 on page 10.

The command can take several seconds to complete, particularly on a large cluster, or when there are many trace domains to collect.

This command generates concurrent processing over the processors. There is a slight risk of interference between the processors in this processing. This can cause the execution of the command to take time and at worst the command fails. Failure or time-out of the command is shown in the command output. The consequences of command failure are benign. The only effect is that some domains cannot have been collected. Simply try the command again.

The timeout for the `collect_domains.sh` is 120 seconds. On large systems, where this default timeout is not long enough, it is possible to increase it by using the `clurun.sh` command:

```
clurun.sh -c tutil -t <seconds> -p apptrace/collect_domains
```

## 6.4 `verify_domains.sh`

The `verify_domains.sh` command verifies global consistency for all collected domains. Only domains that have been verified can be used for tracing.

On completion, the command output is a compact printout of the domain tree. The `verify_domains.sh` command can be re-executed, without harm, simply to get this domain tree printout.

If you have executed `verify_domains.sh` successfully, it is not necessary to execute it again until additional domains have been collected, which is normally after the next software upgrade.

More detailed information on specific domains can be obtained with the `cat_domain.sh <domain_name>` command, where `<domain name>` is the name of the domain you wish to check.

The “Condition” attribute in the printout of the domain has a value: “[.....|OK]”. If the value is displayed as “[.....|NOT-VERIFIED]”, the domain has not been, or could not be, verified. If verification has failed, there is either an error message at the end of the printout, or it has a subdomain that is the cause of the failure.

### 6.4.1 Use

```
verify_domains.sh [ -h]
```



-h                      Print a help summary

## 6.4.2 Common Errors for `verify_domains.sh`

The `collect_domains.sh` command must be executed for the current software version before `verify_domains.sh` can be executed.

The successful completion of the `verify_domains.sh` command does not mean that all, or indeed any, domains have been verified as “OK”.

## 6.5 `begin_session.sh`

Creates a trace session. Only one session at a time is allowed.

A trace session is “semi-persistent” that means, any serious cluster disturbance, such as a processor crash, causes a reset of AppTrace and the trace session is ended and discarded.

### 6.5.1 Use

`begin_session.sh [ -h]`

-h                      Print a help summary

### 6.5.2 Common Errors for `begin_session.sh`

The `collect_domains.sh` and the `verify_domains.sh` commands must be executed for the current software version before `begin_session.sh` can be executed.

## 6.6 `display_session.sh`

Prints the trace control state of AppTrace, the current execution state of the session, and the contents of the session. Only one session at a time is allowed to exist.

The output consists of six parts:

- `---Session <session id> is [FROZEN | OPEN for editing]--- general state: <trace control state>`

Displays the session Id, the execution state of the session, and the trace control state of AppTrace. The execution state of the session is either “FROZEN” or “OPEN for editing”. A session is “FROZEN” if it is uploaded.



- ---Included processors: <processor name>, <processor name> ...

Displays the names of the processors that are currently in the scope of the session. When `upload_session.sh` is started, the trace session is installed at these processors only.

- ---Enabled process types: <process type> located on:<processor name> ...

Displays the name and location of the process types currently in the scope of the session. When `upload_session.sh` is started, all process instances of these process types (on included processors) are enabled for trace.

- ---Enabled process instances: Pid:<pid> of type <process type>/<rtid> located on:<processor name> ...

Displays the pid, process type, and location of the process instances that are explicitly part of the session scope. When `upload_session.sh` is started, these process instances on included processors are enabled for trace.

- ---Trace program:

```
[0] <trace expression>
[1] <trace expression>
[2] ...
```

Displays the current trace program. When `upload_session.sh` is started, the trace program is installed in shared memory on included processors.

- ---Routing of output:

To <destination>.

Displays the current routing of output. When the trace is executing, the output is routed to the destination, see Section 6.12 on page 54.

A trace session is “semi-persistent”, which means, any serious cluster disturbance, such as a processor crash, causes a reset of AppTrace and the trace session is ended and discarded.

### 6.6.1 Use

`display_session.sh [ -h ] [<session id>]`

-h                      Print a help summary



`<session id>` is the identity of the current session. If the `<session id>` argument is omitted, the command assumes that the current session is intended.

### 6.6.2 Common Errors for `display_session.sh`

The command displays the content of the session. If the session data exceeds the maximum message size limit (64 KB), the command output is truncated and it contains the text `Session message truncated!` in the last line. Truncating the output does not affect session state.

## 6.7 `include_processors.sh` (edit session)

Includes one or more processors in the trace session.

### 6.7.1 Use

```
include_processors.sh [ -h ] { -a | <processor name> ... }
```

`-h` Print a help summary

`-a` Include all processors currently attached to the cluster

`<processor name>` is the name of a processor in the cluster, for example, SC-1, SC-2, PL-3, PL-4.

### 6.7.2 Common Errors for `include_processors.sh`

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the state using `display_session.sh`.

The `<processor name>` argument is case-sensitive.

## 6.8 `ls_processtypes.sh`

This command is used to list all the process types available in the target system.

## 6.9 `add_process_type.sh` (edit session)

Add process types to the trace session. All instances of the selected process types, when `upload_session.sh` is later called, are enabled on the processors included in the session. It is not only existing processes of the type that are enabled. New processes, or restarted processes, created when the session is uploaded, are also enabled.



### 6.9.1 Use

```
add_process_type.sh [ -h ] { <process type> [ <process type>
... ] | -l [<LM>] }
```

-h                                      Print a help summary  
-l                                      List LMs, or add process types that belong to the LM

<process type> is the name of a Delos-specified process type and <LM> is the name of a load module.

There are two variants of this command.

- The basic variant takes a list of process type names to be added.
- The -l variant takes the name of a load module and adds all process types that belong to the load module. If no load module name is provided, the list of available load modules is returned.

### 6.9.2 Common Errors for add\_process\_type.sh

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the state using `display_session.sh`.

On live systems, the -l variant must be used with care, since it is powerful (can enable many process types) which could consume much capacity when traced.

The <process type> argument is case-sensitive.

## 6.10 add\_process\_instance.sh (edit session)

Add process instances to the trace session. The selected instances are later enabled for trace (if the processor is included in the session).

### 6.10.1 Use

```
add_process_instance.sh [ -h] <processor name> <pid> <process
type> [ <pid> <process type> ... ]
```

-h                                      Print a help summary

<processor name> is the name of the processor on which the process executes. <pid> is the process identity of the process. <process type> is the name of the Delos-specified process type for the instance.





You can provide several processes in the same command line, but only for the same processor, the processor designated by the first argument. Processes on different processors can be added to the session by separate invocations of `add_process_instance.sh`.

### 6.10.2 Common Errors for `add_process_instance.sh`

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the state using `display_session.sh`.

If the process no longer exists when the session is uploaded to the processor, the upload fails.

The `<processor name>` and `<process type>` arguments are case-sensitive.

## 6.11 `insert_expression.sh` (edit session)

Inserts a trace expression in the trace program of the trace session. The trace program is an ordered list of trace expressions, where each element has a line number. A line number can be provided to the `insert_expression.sh` command. The insert is then done just above that line, pushing down all lines below it.

### 6.11.1 Use

```
insert_expression.sh [ -h ] [ -l <line-no> ] <trace
expression>
```

-h                                      Print a help summary

-l <line-no>                          Insert above the indicated line

*<trace expression>* takes one of the forms:

*<trace domain>*

*<trace domain>(<filter>)*

*<trace domain> ⇒ <action> [, <action> ... ]*

*<trace domain>(<filter>) ⇒ <action> [, <action> ... ]*

In the first two forms, the implicit action is to output all parameters of the trace domain. See Section 4.3 on page 21 for details on trace expressions.

If no line number argument is provided, the default insertion point is at the end of the trace program.



Use `display_session.sh` to view the current trace program.

#### 6.11.1.1 Use of Wildcard in `insert_expression.sh`

The `insert_expression.sh` command supports “wildcard” matching on names for subdomains. The support is limited to complete subdomain names. In other words, it can interpret expressions such as:

```
hlr.use_case.location_update.*
```

and sometimes even:

```
hlr.use_case.*.*
```

This works if all the names that match the first (left-most) wildcard have the same set of subdomains that match the second wildcard. However, the command rejects expressions such as:

```
hlr.use*
```

and:

```
*.use_case
```

The wildcard in the domain name is expanded to all matching names and the result is the same as starting the `insert_expression.sh` command for each matched name. For an example, see Section 5.4 on page 45.

**Note:** If the original expression includes filters or actions, these are copied to the expression for each expanded name. This can be powerful, but it also means that the filter and action are parsed in the context of each such name. If the parse fails for any matched name, the entire command fails. What is worse, the parse can succeed but with a different semantic than you intended. For example, a token such as `$tmp` could mean either a parameter, or a variable, see Section 4.3.2 on page 22. It is interpreted as a parameter if the domain has a matching parameter, otherwise it is interpreted as a variable.

#### 6.11.2 Common Errors for `insert_expression.sh`

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the state using `display_session.sh`.

The trace domain of the trace expression must have been verified “OK”. See Section 6.4 on page 48.

### 6.12 `route_output.sh` (edit session)

Alters the destination and capacity of output.



### 6.12.1 Use

```
route_output.sh [ -h] <destination>
```

-h                                      Print a help summary

<destination> takes one of the following forms:

console

applog

rawconsole

The default destination is `console` which means that the output is sent directly (from the application process) to the console log of each processor. The result is then not one but many streams of trace output. The console logs typically contain many other output, not related to AppTrace, but interleaved with AppTrace.

The destination `applog` is sent to the Applog stream under the AppTrace log label. This produces one merged stream of output. The output log file is free from any other output than that produced by Applog.

**Note:** Applog is often configured to “echo” output to the console log, which means that the AppTrace output can also appear in the console log.

The destination `rawconsole` in vDicos is the same as the destination `console`. This is because there is no internal buffer in apptrace itself. The traces are sent to the logging system in CDCLSV, and buffering is done there. Therefore, overload cannot be caused by using the `rawconsole` destination.

Use `display_session.sh` to view the current routing.

### 6.12.2 Common Errors for route\_output.sh

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the state using `display_session.sh`.

## 6.13 upload\_session.sh

Freezes the trace session and installs it on all included processors. Enables relevant processes and process types for trace. Starts a gateway process at each processor. If the command succeeds, the trace control state changes from “Clean” to “Ready”. AppTrace is then ready for `start_trace.sh`.



### 6.13.1 Use

```
upload_session.sh [ -h]
```

-h                      Print a help summary

### 6.13.2 Common Errors for upload\_session.sh

If the `add_process_instance.sh` command has been used, there is a possibility that the specified process does not exist at the processor. In this case, the upload command fails and you get a clear error message.

The upload command can fail because of changes in the configuration of the system, such as a system upgrade. A clear error message is then produced. If the upload command repeatedly fails, it can be necessary to execute a `reset.sh -f`. The session is then discarded, so you have to start over with a `begin_session.sh`.

The upload command also fails when the session data exceeds maximum message size limit of 64 KB, displaying “Failed to pack session for agent at processor: *<processor name>*, Reason:” followed by the fail reason: “Session is oversized” or “Cannot find session”. In this case, execute `reset.sh -f` and start over with a new session.

A trace session can be ineffective in the sense that it can never produce any output. An example is a trace session that has no processors included, or no processes added, or an empty trace program. Some (but not all) of these ineffective cases are recognized by `upload_session.sh` and rejected.

## 6.14 start\_trace.sh

Starts tracing. If the command succeeds, the trace control state changes from “Ready” to “Running”.

The command allows the desired level of the trace session to be set. Only trace points with a level value that is lower than the desired level are in the scope of the trace. The level value is an indicator of how detailed the trace is. The higher the value, the more detailed the trace is.

If no value for the desired level is provided, a default value of 32 is used. The highest allowed desired level on live systems is 48. The highest possible level is 64, but this includes debug trace points, not verified or intended for live site tracing. The lowest allowed level for a trace session is 0. This excludes all trace points and makes no sense for a normal session. A level of 0 can be used as a way of checking that the trace control machinery itself works properly, without generating any output.

For more details on level, see Section 3.3.2.1 on page 14 and the trace documentation of the application.



### 6.14.1 Use

```
start_trace.sh [ -h ] [<level>]
```

-h                                      Print a help summary

<level> is the desired level. Default value is 32 (major trace).

### 6.14.2 Common Errors for start\_trace.sh

If the trace session is uploaded successfully, the trace control state is “Ready”. You can check this by a `display_session.sh`. A `start_trace.sh` then succeeds, and the trace control state transitions to “Running”.

If the trace control state is “Running”, yet you do not get the output that you expect, then something is wrong with the way you specified the session. You can `stop_trace.sh` and start it again with a different level. Or you can `stop_trace.sh`, `unload_session.sh` and edit the session.

## 6.15 stop\_trace.sh

Stops tracing. Changes the state of the session from “Running” to “Ready”. The session is still frozen and trace can be restarted with `start_trace.sh` or unloaded with `unload_session.sh`.

### 6.15.1 Use

```
stop_trace.sh [ -h ]
```

-h                                      Print a help summary

## 6.16 unload\_session.sh

Uninstalls the trace session from processors and unfreezes it. The state changes from “Ready” to “Clean”. The trace session can then be ended, or it can be edited and uploaded again.

### 6.16.1 Use

```
unload_session.sh [ -h ]
```

-h                                      Print a help summary



### 6.16.2 Common Errors for `unload_session.sh`

Unload can only be performed when there is an existing trace session that is uploaded, but not started. The trace control state must be “Ready”. You can use `display_session.sh` to check the trace control state.

## 6.17 `exclude_processors.sh` (edit session)

Removes processors from the trace session. This command is the inverse of `include_processors.sh`.

### 6.17.1 Use

```
exclude_processors.sh [ -h ] { -a | <processor name> ... }
```

<code>-h</code>	Print a help summary
<code>-a</code>	Exclude all processors currently included in the session

`<processor name>` is the name of a processor included in the session.

### 6.17.2 Common Errors for `exclude_processors.sh`

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the trace control state using `display_session.sh`.

## 6.18 `remove_process_type.sh` (edit session)

Removes process types from the trace session. This command is the inverse of `add_process_type.sh`.

### 6.18.1 Use

```
remove_process_type.sh [ -h ] { <process type> [ <process type>... ] | -l [<LM>] }
```

<code>-h</code>	Print a help summary
<code>-l</code>	Remove process types that belong to the LM

`<process type>` is the name of a Delos-specified process type and `<LM>` is the name of a load module.

There are two variants of this command.



- The basic variant takes a list of process type names to be removed.
- The `-l` variant takes the name of a load module and removes all process types that belong to the load module.

### 6.18.2 Common Errors for `remove_process_type.sh`

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the trace control state using `display_session.sh`.

The `<process type>` argument is case-sensitive.

## 6.19 `remove_process_instance.sh` (edit session)

Removes process instances from the trace session. This command is the inverse of `add_process_instance.sh`.

### 6.19.1 Use

```
remove_process_instance.sh [ -h ] <pid> [ <pid> ... ]
```

`-h` Print a help summary

`<pid>` is the process identity of the process.

**Note:** This command does not take any processor name argument. If more than one process instance with the same `<pid>`, but on different processors, have been added to the session, all of them are removed by this command.

### 6.19.2 Common Errors for `remove_process_instance.sh`

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the trace control state using `display_session.sh`.

## 6.20 `delete_expression.sh` (edit session)

Removes one or more lines or expressions from the trace program. This command is the inverse of `insert_expression.sh`.

### 6.20.1 Use

```
delete_expression.sh [ -h ] <line-no> [<to-line-no>]
```



-h                      Print a help summary

<line-no> is the first line to remove and <to-line-no> is the last line to remove. All lines between these are also removed. If there is no <to-line-no>, only the first line is removed.

Use `display_session.sh` to view the current trace program.

### 6.20.2            **Common Errors for delete\_expression.sh**

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the trace control state using `display_session.sh`.

## 6.21            **end\_session.sh**

Deletes the trace session.

### 6.21.1           **Use**

`end_session.sh [ -h]`

-h                      Print a help summary

### 6.21.2           **Common Errors for end\_session**

The trace session could have been automatically removed by the system.

This command alters an existing trace session. The trace control state must be “Clean”, otherwise the command fails. You can check the trace control state using `display_session.sh`.

## 6.22            **reset.sh -f**

This command is only needed for emergencies, or when the trace control mechanism is “stuck” in some ways.

The command immediately restarts the trace control processes. This has the effect of immediately performing `stop_trace.sh`, `unload_session.sh`, and `end_session.sh`.

### 6.22.1           **Use**

`reset.sh [ -h ] -f`





- h                      Print a help summary
- f                      Force the command

**Note:** The -f flag is mandatory.