Typeset in \LaTeX{} from SGML source using the DOCBUILDER 3.4.1 Document System.
Contents
Chapter 1

Inviso User’s Guide

Inviso, an Erlang trace tool.

1.1 Inviso

1.1.1 Introduction

Inviso (Latin) to go to see, visit, inspect, look at.

The Inviso trace system consists of one or several runtime components supposed to run on each Erlang node doing tracing and one control component which can run on any node with available processor power. Inviso may also be part of a higher layer trace tool. See the inviso-tool as an example. The implementation is spread out over the Runtime tools and the Inviso Erlang/OTP applications. Erlang modules necessary to run the runtime component are located in Runtime tools and therefore assumed to be available on any node. Even though Inviso is introduced with Erlang/OTP R11B the runtime component implementation is done with backward compatibility in mind. Meaning that it is possible to compile and run it on older Erlang/OTP releases.
This document describes the control and runtime components of the Inviso trace system.

Underlying Mechanisms

Inviso is built on Erlang trace BIFs and standard linked in trace-port drivers for efficient trace message logging. This means that Inviso can not co-exist in runtime with any other trace tool using the trace BIFs.
1. Start the Inviso control component at any node. Preferably a node that is not participating in the “work” done by your “system”. The control component runs independently and normally not linked to any other process. (Prompt 2 in the example below.)

2. Add all Erlang nodes to the inviso control component where you want to trace. This is starting runtime components on all involved Erlang nodes. Can include the node where the control component runs as well. Note that the Runtime_Tools application must be running on the nodes where runtime components shall be started. (Prompt 1 and 3 in the example below.)

3. Initiate tracing on the added nodes. Initiating tracing means “opening” the output to where trace-messages will be written. Most commonly this is a file. Note that it is not actually necessary to initiate the same tracing on all nodes. It might for instance be wise to use different filenames on different nodes. In the example below tracing is initiated on two nodes. The same node as where the shell is running (node()) and at node2@hurin. Further both “regular” tracing (trace) as well as trace information (ti) are specified for both nodes. (Prompt 4 in the example below).

4. If needing pid-to-alias translations, activate meta tracing on the necessary functions. This requires that trace information was specified when initiating tracing. (Prompt 5 in the example below illustrates using pid to locally registered name translations).

5. Set trace-patterns on the functions that shall be traced. (Prompt 6 in the example below).

6. Set process trace flags on necessary processes. Do not forget to use the timestamp flag in order to be able to merge log files together in chronological order. (Prompt 7 in the example below).

7. Run your code. (Prompt 8 in the example below).

8. Stop tracing (opposite of initiate tracing) and clear trace-patterns on the nodes. It is actually not necessary to stop tracing on all nodes at once. Nodes no longer of interest can be made to stop tracing before others. (Prompt 9 in the example below stops tracing. Prompt 13 removes all trace flags and trace patterns. Removing trace flags are really not necessary since those will be removed when the runtime components are stopped. Removing trace patterns may many times be necessary to “return” the node to a “clean” state from a trace perspective. Trace patterns are never automatically cleared by the runtime system unless the Erlang module in question is reloaded.)

9. If necessary fetch the log files from the various nodes. (Prompt 10 in the example blow).

10. Merge and format the log files. (Prompt 12 in the example below).

11. Stop the runtime components. This is important if the Erlang nodes are real “live” systems, and will not necessarily be stopped just because the tracing is completed. (Prompt 14 in the example below).

This “recipe” is valid also when tracing in a non-distributed environment. The only difference is that function calls not taking a node-name as argument are used. The runtime component will then of course run on the same node as the control component.

Simple example illustrating the above listed recipe. It traces on two nodes, node1 where the control component also runs. And node2 which is a remote node from the control components perspective. The example uses a mixture of API-calls specifying what nodes to trace on and API functions working on all added nodes. This is in this example interchangeable since all to the control component known nodes are participating in the same way.

```
Eshell V5.5  (abort with ^C)
(node1@hurin)1> application:start(runtime_tools).
ok
(node1@hurin)2> inviso:start().
```

This is a short step-by-step description of how tracing using Inviso can be done.
1.1.2 Incarnation runtime tags

Incarnation runtime tags are used to identify an incarnation of a runtime component. An incarnation is one “start-up” of a runtime component on a specific Erlang node. The reason why it can sometimes be necessary to examine the incarnation runtime tag is that a user wants to connect, adopt, an already running runtime component. This may be the case if the runtime component has autostarted or because the control component terminated without killing the runtime component. While the user has been out of control of the runtime component it may very well have terminated and been restarted. If it was restarted without the user’s knowledge, its incarnation runtime tag has most likely changed. The user can therefore, if the current incarnation runtime tag is not what it is supposed to be, conclude that the runtime component is not “doing” what is expected.

The runtime tag is set at runtime component start-up. This is either done when it is started manually by a call to inviso:add_nodes/X, or according to a specification in one of the autostart configuration files.
1.1.3 Runtime component state and status

A runtime component has a state and a status. The possible states are: new, tracing and idle. A runtime component that is tracing has (possibly) open log files. A new runtime component has no current tracer-data. That is it lacks any history of what it has done just recently. An idle runtime component is no longer tracing. It does therefore have current tracer-data that describes what it did do when it was tracing.

The status describes if the runtime component is running or suspended. A suspended runtime component may very well be in state tracing. However the point is that it shall not generate any processor load. It will therefore refrain from generating any trace messages.

1.1.4 The Runtime Meta Tracer

Meta tracing is a trace mechanism separate from the regular tracing. It is normally used by a trace-tool to learn about function calls made anywhere in an Erlang node. A typical example is that there is a possibility in Inviso to get pids translated to registered name in the final formatted trace-log (for processes having registered names). This is done by meta-tracing on the BIF register/2 to learn about all name/pid associations made.

Meta tracing in Inviso is done by the inviso.rt.meta process, which is part of the runtime component if trace-information, ti, is initiated. See inviso:init_tracing/1 [page ??] for details. The runtime meta tracer opens and controls the so called trace information file. Translations can then be done off-line using the associations logged in the trace information file. Currently the only type of trace information file available is a straight binary file. A wrap-file makes no sense since pid-to-name associations made in the beginning will most likely be lost.

The runtime meta tracer can also be used to translate pids to own identifiers. The only thing needed is one or several association points in the form of function calls which will only be made if an association is done in the system. The pid and own-identifier must be arguments and/or return values from the same function call.

The runtime meta tracer can further more be used to achieve side-effects during tracing, like turning tracing on or off.

Matching function calls with return values

It may sometimes be necessary to wait for a meta traced function to return before it can be decided what to do. This may be due to that one piece of information to make the decision is in the arguments to the function, the other in the return value. This kind of logic can be programmed to be executed by the inviso meta tracer. In order for the inviso meta tracer to “remember” function-call arguments until the function return trace message arrives, a public loop data structure is implemented. The public loop data structure is first created when tracing is initiated (of course only when trace information is specified in the init_tracing call). The public loop data can then later be further initiated each time meta tracing (tpm and tpm_ms) is activated for a certain function.

The default public loop data structure is a tuple of size two. The first element in that tuple is used by the predefined meta tracing for capturing locally registered names. The second element is free to use for any other purpose. The elements of the tuple must in the default implementation be lists of tuples. Where each sub-tuple shall represent one waiting call. The last element of that tuple must be a now-stamp (as returned by the BIF now/0). See below for an explanation of the now-stamp. The size of the outer most tuple may be increased as long as the term residing in the first element is left unchanged, and all other elements follow the above described rules.

The inviso meta tracer “cleans” the public loop data structure approximately once every minute. The reason for this is that entries in the public loop data structure may become abandoned. If for instance a
process crashes while executing the body of a meta traced function, no return value will be generated. Or in other words, receiving the call meta trace-message can have caused information to have been written into the public loop data structure. That entry will be used and removed when the return_trace meta trace-message arrives. But if the meta traced function causes an exception, no return_trace message will come. The function which normally removes the entry is then therefore never called. The default clean-function assumes that every item in the public loop data tuple is a list. Where each list contains tuples where the last element of those tuples are "now-stamps". The default clean-function considers an entry older than 30 seconds to be abandoned.

Making pid/alias entries in the ti-file

When activating meta tracing for a function for the purpose of writing pid-alias associations in the trace information file, a call-func and possibly also a return-func is specified. These functions will be called when a meta trace message arrives to the inviso meta tracer as a result of function calls or returns for this meta traced function. What exactly to write in the trace information file is dictated by the merge mechanism. This since pid-alias translations are done off line when merging log-files. See the chapter on merging and formatting log files for more details.

Simple example where the call to the function `connection:assoc_id/2` will associated `Pid` with the id `Ref`. We will then in a merged log-file see a translation between `Pid` and `Ref`. Actually for all future since there is no unalias function meta traced in this example. The inviso meta tracer will receive a meta trace message every time `connection:assoc_id/2` is called. When that message arrives the meta tracer will call `mytrace:call_assoc_id/3` which must return `{ok,NewPublicLoopData,OutPutBinary}`.

```erlang
-module(mytrace).

call_assoc_id(_CallingPid,[Pid,Ref],PublLoopData) ->
    {ok,PublLoopData,term_to_binary([{Pid,Ref,alias,now()}]).
```

Extending the public loop data structure.

It is of course very likely that the public loop data structure must be extended to host all functions where the meta tracer must delay its action until the function in question returns. What is necessary is to create your own public loop data structure at trace initialization. This is done by using the TiSpec. TiSpec=`[InitMFA,RemoveMF,CleanMF]`, where InitMFA creates the structure, RemoveMF removes it (must often not necessary unless a database, file or similar is used as storage instead of a tuple). CleanMF is the function which will be called each every 60 seconds to go over the public loop data structure. Following the below rules, not much programming will be needed, apart from the InitMFA:

- Make the public loop data structure a tuple of lists, where each list is a list of tuples where the tuples represents one entry.
- Make the CallFunc (the function called each time a call meta trace message arrives for the function in question) add a tuple to the correct list where the last element of that tuple is a now-stamp.
- Make sure that the first element in the loop data structure tuple is left alone for the default implementation of the handling of registered names.
• Use `inviso_rt_meta:clean_std_publid/1` (which is exported for this purpose) as `CleanMF`. This function is normally the default clean function, if not using the possibility to in detail initiate the inner workings of the inviso meta tracer.

Simple example where tracing is initiated with a public loop data structure having 10 places for nine (the locally registered names is mandatory) different functions to be meta traced. Note that the BIF `list_to_tuple/1` is used as initialization function. And that the Stdlib function `lists:duplicate/2` is used to create something for the initialization function to work on.

```
(node1@hurin)4> inviso:init_tracing([[node2@hurin,[{trace,[file,"tracefile_node2.log"]},[ti,[{file,"ok",[{'node2@hurin'},{ok,[{trace_log,ok},{ti_log,ok}]}]}]]]
```

Using the inviso meta tracer to achieve side effects

Since meta tracing is independent of regular tracing and catches any function call to a particular function made in any process, it is well suited to be used to turn things on or off during execution. That trick is done by letting the `CallFunc` and (if used) `ReturnFunc` do these sideeffects. One must of course remember that the inviso meta tracer is a process amongst all other processes in the system. Meaning that the side effect is not necessarily done exactly when the meta traced function is called. Unless the side effect can be achieved using a match specification action.

### 1.1.5 Runtime Component Autostart

In order to trace before any user interaction is possible, an autostart mechanism is implemented. The runtime component is started by the top supervisor of the Runtime_Tools application top supervisor. Hence the Runtime_Tools application must be part of the boot script for autostart tracing to work. The Runtime_Tools applications must of course be started before any application that is to be traced. Do note that application startup is not entirely synchronous. Meaning that just because the application controller has begun starting the next application, Runtime_Tools is not necessarily fully up and running.

The autostart mechanism is configurable. The runtime component comes with a standard autostart configuration, only missing two text-files to be completely operational.

#### Autostart Configuration

The autostart is controlled by the Runtime_Tools application configuration parameter `inviso:autostart.mod`. It must be the name of a module exporting an `autostart/1` function. The default value is `inviso:autostart`, a module which is provided with Runtime_Tools. See below [page ??] for details.

Autostart/1 function must offer the following:

```
autostart(RuntimeToolsArg) = {MFA,Options,Tag} | any()
```

`RuntimeToolsArg` is the argument `Arg` provided to the Runtime_Tools application through the application resource file `{mod,Module,Arg}` parameter.

`MFA = {AutoMod,AutoFunc,AutoArgs}` | any() controls how tracing will be initiated. Note that initiating tracing is not necessarily the same as starting a runtime component. It is possible to have a runtime component without doing any tracing. The runtime component is started as long as `autostart/1` returns the proper tuple, and `Options` does not for instance require a certain non-existing
control component. If it is not a proper tuple or there are other faults in the tuple items, the autostart will terminate. Typically will this happen if there is no autostart/1 function.

If MFA does not properly point out a function possible to call with spawn(AutoMod,AutoFunc,AutoArg), there will simply be no initialization. (Initialization is done by a separate process spawned during runtime component autostart.) It may be worth reminding that AutoMod must be present at the node where the runtime component is supposed to run. Not necessarily the node where the control component usually runs.

Options is the list of options given to the runtime component. See Options in inviso:add_nodes/2 [page ??].

Tag is the runtime component incarnation tag. See Tag in inviso:add_nodes/2 [page ??].

The Standard Autostart Implementation

As mentioned above, Inviso comes with a complete implementation of autostart sufficient for most situations.

The default autostart module is inviso:autostart, provided as part of the Runtime_Tools application. Since that name is the default module name, it is not really necessary to set the inviso:autostart mod configuration parameter for the Runtime_Tools application.

Its autostart/1 function reads a configuration file pointed out by the Runtime_Tools application configuration parameter inviso:autostart_conf. If the parameter is not present, a default file, inviso:autostart.config in the current working directory, will be consulted.

The config file must be an ascii text file with one or more tuples ended with a dot. The following parameters are recognized:

{repeat,N} Optional parameter where N specifies the maximum remaining autostarts. The autostart functionality will rewrite the configuration decreasing N if present. If N=0 the autostart will be terminated.

{mfa,[M,F,Args]} Optional parameter controlling how initialization shall be done. The control component will spawn a separate process to do the initializations by doing spawn(M,F,Args).

{options,Options} Optional parameter specifying the options for the runtime component itself. See Options in inviso:add_nodes/2 [page ??].

{tag,Tag} Optional parameter specifying the runtime component tag. If missing the default tag will be default_tag.

Example:

{repeat,1}.
{mfa,

{inv iso:autostart_server,
 init,
   [[{tracerdata,{file,"mylogfile"}},
      {cmdfiles,"_a_trace_case.txt"}],
    {bindings,[[{'M',mymod},{'F','_'},{'Arity','_'}]},
     {translations,[]]]}.}
The example file results in the start of a runtime component given no specific options. There will only be one autostart since the repeat parameter is set to 1. Tracing will be initiated by the standard initiator (inviso.autostart_server [page ??]). The initiator will initiate tracing opening a plain trace-port logfile ("mylogfile"). It will further read the "a_trace_case.txt" file to get instructions on what patterns and flags to set. If there are variables mentioned in the trace-case file "a_trace_case.txt", it is parameterized, the variables M, F and Arity will get the values according to bindings. There will be no translations done, hence the trace-case file must be written using inviso_rt function calls directly.

inviso.autostart.server  To further facilitate the standard autostart implementation a default initiator is implemented. To use it, simply specify it as mfa in the config file read by the standard autostart module.

Its init/1 function takes one argument on the form of a list of tuples. The following tuple-parameters are recognized:

{tracerdata,TracerData} Specifies how tracing is initiated. See inviso:init_tracing/1 [page ??] for details on TracerData.
{cmdfiles,ListOfFileNames} Specifies trace-case files which shall be executed to set the patterns and flags of the trace. See the Trace Cases [page ??] chapter for more details. The files will be executed in the order specified.
{translations,Translations} Optional parameter specifying how functions in trace-case files shall be translated. This is useful since trace-cases can be written for higher-layer Inviso tools, but must during an autostart execute using inviso.rt function calls only.

TranslMod:TranslFunc(ListOfOrigArgs)-> ListOfTransformedArgs

Mod1:Func1/Arity specifies the function that shall be translated into Mod2:Func2/Arity. The actual arguments will be translated with TranslMod:TranslFunc/1. The translation function shall take a list of the actual arguments, and return a list of new arguments. The return-value list may for instance have certain arguments removed, if such are not relevant to the Func2 function. (Such arguments must actually be removed since the return-value list from the translation function must have the correct amount of elements corresponding to the arity of Func2.)

{bindings,Bindings} Bindings=[[Var,Val]]
Var=atom(), the name of the variable
Optional parameter specifying the actual values of variables used in the trace-cases. Bindings is a bindings structure as used by functions in the erl_eval module.

The Standard Autostart Utility Library  To facilitate creating the configuration file described above, there are functions in a module named inviso_as_lib which can both create new files according to supplied arguments and update existing configuration files.

The node(s) in question must be running since the functionality in the utility library uses distributed Erlang to access the file system.
Chapter 1: Inviso User’s Guide

1.1.6 The Dependency Property

In order to protect real “live” systems from getting a runtime component lingering around without a control component, a dependency property can be specified at runtime component start-up. The property specifies a dependency in milliseconds. Meaning that if the property is set to 0 (zero), the runtime component will terminate immediately if its current control component terminates.

If a control component tries to start a runtime component at an Erlang node where there already is a runtime component, the control component will adopt the already existing runtime component if it has no current control component. Otherwise the control component will experience an error, not being able to start a runtime component at that node.

It must also be noted that an autostart runtime component is running without control component, at least before any control component adopts it.

1.1.7 Overload Protection

Since Inviso is intended to be used on real “live” systems, it is possible to protect the system against overload, having Inviso suspend tracing should an overload situation occur.

What indicates an overload situation must be programmed and configured outside of Inviso. Inviso can initiate an overload protection, call an overload function periodically and clean-up an overload mechanism should it decide to terminate.

Internally inside the runtime component, suspending tracing means removing all process trace flags and meta patterns. Reactivating tracing is outside the scope of Inviso, but can be implemented in a tool using Inviso.

Simple example adding a runtime component and making it protect its Erlang node from overload.

```erlang
inviso:add_node(my_rt_tag, [{overload, [{my_ovl, check}, 15000, {my_ovl, start, [my_port_pgm]}, {my_ovl, stop, [my_port_pgm]}]}]).
```

Immediately when the runtime component is started, it will initiate overload protection by calling `my_ovl:start(my_port_pgm)`. When tracing (not when in state idle or new), the runtime component will every 15000 milliseconds call `my_ovl:check/1`. Depending on its return value, the runtime component will either do nothing or suspend tracing. When the runtime component is stopped, `my_ovl:stop(my_port_pgm)` will be called.

1.1.8 Merging and Formatting Logfiles

If logging trace messages to a logfile has been used (decided when tracing is initiated) the various log files will be located on the different Erlang nodes participating in the trace. The log files must be merged and formatted for the following reasons:

- The various log files from the different nodes must be merged into one logfile in chronological order (where trace messages from different nodes will be mixed). If only one Erlang node participated in the trace, this step is obviously not necessary.
- Trace-port log files are on binary format and must in most cases be transformed in some way. This can for instance be to a text-file format or inserted into a database for analysis.
Use trace information data to translate process identifiers to aliases, both standard Erlang ones (as registered names) as well as own invented.

The first step before any merging can take place is of course to get all log files, including any trace information files to a location where the logfile merger can access them. This can either be done by simply copying the files. However if the file systems on the Erlang nodes are not that easily accessed, there is a fetch_log function implemented in the runtime component. It will transfer log files using distributed Erlang.

Inviso comes with two Erlang modules, inviso_lfm and inviso_lfm_tpfreader, implementing a standard log file merger and formatter. The log file merger (inviso_lfm) uses a file reader process (implemented in inviso_lfm_tpfreader) to access log entries in parallel. It is possible to write your own logfile reader. This is necessary since you may have your own trace-log format and/or own trace information log format. The logfile merger can further more be configured to use your own formatter, customizing what to do with a trace message.

Trace messages in the log files must of course be time-stamped for the logfile merger to be capable of correctly merging them. This means using the timestamp process trace flag.

The standard inviso log-file reader understands the following trace information file entries:

- `{Pid,Alias,alias,NowStamp}`
- `{Pid,Alias,unalias,NowStamp}`

The Pid in an alias entry must always be a proper pid. In an unalias entry it may also be the atom `undefined`. The latter means that all associations involving Alias shall stop to be valid. The standard inviso log file reader uses the now-stamp to make sure that associations are only used during time periods in the log-file when such are valid.

### 1.1.9 Trace Cases

The idea behind trace cases is that someone knowledgeable of a certain system component can write a file specifying the trace-patterns and process trace flags necessary to trace on certain items once and for all. Hence a trace case will most likely be a series of calls to functions setting trace patterns and process trace flags.

However, the actual Erlang nodes and values of arguments given in the trace function calls can not be static in order for the trace cases to become useful and reusable. A trace case file must therefore be possible to parameterize. Introducing variables that will get their values at the time of trace case execution. It may also be the case that Inviso is used as a component in a higher layer trace tool. Trace cases may therefore be written calling more complex functions than the low level `inviso_rt` functions which are available to autostart mechanisms. In a matter of fact, the `inviso` API itself can be considered a higher layer. It addresses multiple nodes at once where the `inviso_rt` API can only address the local node.

This results in that for trace cases to be useful there must be a function call translation mechanism and an execution environment capable of handling variable bindings.

A trace-case is a text ascii file consisting of function calls written as they could have been done in the Erlang shell:

```
modulename:functionname(arg1,arg3,...).
```

A trace-case may contain any valid function call, including binding new variables which are used later in the trace-case, but:

- No spawn, send or receive.
- No apply or similar (including \texttt{mod:F(Arg1,Arg2)}). This because the variable environment is not available during the translation. Only during execution.

Example: Trace cases are expected to be written to be executed directly in an Erlang shell (by some utility reading a text file on trace case format) calling \texttt{inviso} functions. The translations must then translate \texttt{inviso} function calls to \texttt{inviso_rt} function calls, since \texttt{inviso} is not available in the Runtime Tools applications. It can not be assumed that any trace tools outside the Runtime Tools application is available on the nodes. Luckily (!) the \texttt{inviso_rt} API resembles the \texttt{inviso} API very much, apart from that the \texttt{inviso_rt} API does not take a list of nodes as an argument. Therefore in most situations the only transformation necessary is to change from \texttt{inviso} to \texttt{inviso_rt} and remove the first argument to the function call.

Assume that we have the following trace-case file:

\begin{verbatim}
inviso:tpl(Nodes,mymod,'_','_',MS).
inviso:tf(Nodes,all,[call,timestamp]).
\end{verbatim}

For this to work in an autostart the following translation is needed:

\begin{verbatim}
[{{inviso,tpl,5},{inviso_rt,tpl,{erlang,tl}}},
 {{inviso,tf,3},{inviso_rt,tf,{erlang,tl}}}]\end{verbatim}

Since transforming the arguments from \texttt{inviso} calls to \texttt{inviso_rt} calls is simply removing the first argument, there is no need to program any function to do this. The BIF \texttt{tl/1} can be used directly. Further there must be a variable binding for MS when executing the trace-case. It is not necessary to have one for Nodes since that argument is removed from all function calls by the translation.
Inviso Reference Manual

Short Summaries

- Erlang Module **inviso** [page ??] – Main API Module to the Inviso Tracer
- Erlang Module **inviso_as_lib** [page ??] – The Inviso Autostart Utility Library
- Erlang Module **inviso_lfm** [page ??] – An Inviso Off-Line Logfile Merger
- Erlang Module **inviso_lfm_tpfreader** [page ??] – Inviso Standard Reader Process to
  Standard Logfile Merger
- Erlang Module **inviso_rt** [page ??] – Direct API to the Inviso Runtime Component
- Erlang Module **inviso_rt_meta** [page ??] – Direct API to the Inviso Runtime
  Component’s meta tracer

**inviso**

The following functions are exported:

- **start()** -> {ok,pid()} | {error,Reason} [page ??] Start a control component at the local node
- **start(Options)** -> {ok,pid()} | {error,Reason} [page ??] Start a control component at the local node
- **stop()** -> shutdown [page ??] Stop the control component
- **add_node(RTtag)** -> NodeResult | {error,Reason} [page ??] Starts or adopts a runtime component at the local node
- **add_node(RTtag,Options)** -> NodeResult | {error,Reason} [page ??] Starts or adopts a runtime component at the local node
- **add_node_if_ref(RTtag)** -> NodeResult | {error,WrongReference,OtherTag} | {error,Reason} [page ??] Start or adopt a runtime component at the local node, provided it has a
  certain rtag
- **add_node_if_ref(RTtag,Options)** -> NodeResult | {error,WrongReference,OtherTag} | {error,Reason} [page ??] Start or adopt a runtime component at the local node, provided it has a
  certain rtag
- **add_nodes(Nodes,RTtag)** -> {ok,NodeResults} | {error,Reason} [page ??] Start or adopt runtime components at some nodes
- **add_nodes(Nodes,RTtag,Options)** -> {ok,NodeResults} | {error,Reason} [page ??] Start or adopt runtime components at some nodes
- `add_nodes_if_ref(Nodes, RTtag) -> NodeResult | {error, Reason}`
  
  Start or adopt runtime components at some nodes, provided they have a certain `rttag`

- `add_nodes_if_ref(Nodes, RTtag, Options) -> NodeResult | {error, Reason}`

  Start or adopt runtime components at some nodes, provided they have a certain `rttag`

- `stop_nodes() -> {ok, NodeResults} | NodeResult`
  
  Stop runtime components

- `stop_nodes(Nodes) -> {ok, NodeResults} | {error, Reason}`

  Stop runtime components

- `stop_all() = {ok, NodeResults} | NodeResult`

  Stop both control and runtime components

- `change_options(Options) -> NodeResult | {ok, NodeResults} | {error, Reason}`

  Change options for runtime components

- `change_options(Nodes, Options) -> {ok, NodeResults} | {error, Reason}`

  Change options for runtime components

- `init_tracing(TracerData) -> {ok, NodeResults} | NodeResult | {error, Reason}`

  Initiate tracing

- `init_tracing(TracerList) -> {ok, NodeResults} | {error, Reason}`

  Initiate tracing

- `init_tracing(Nodes, TracerData) -> {ok, NodeResults} | {error, Reason}`

  Initiate tracing

- `stop_tracing(Nodes) -> {ok, NodeResults} | {error, Reason}`

  Stop tracing

- `stop_tracing() -> {ok, NodeResults} | NodeResult`

  Stop tracing

- `clear() -> {ok, NodeResults} | NodeResult`

  Stop tracing and remove meta trace patterns

- `clear(Nodes, Options) -> {ok, NodeResults} | {error, Reason}`

  Stop tracing and remove meta trace patterns

- `clear(Options) -> {ok, NodeResults} | NodeResult | {error, Reason}`

  Stop tracing and remove meta trace patterns

- `tp(Nodes, Mod, Func, Arity, MatchSpec, Opts) ->`  

  Set global trace patterns

- `tp(Nodes, Mod, Func, Arity, MatchSpec) -> {ok, NodeResults} | {error, Reason}`

  Set global trace patterns

- `tp(Mod, Func, Arity, MatchSpec, Opts) ->`  

  Set global trace patterns

- `tp(Mod, Func, Arity, MatchSpec) -> {ok, NodeResults} | NodeResult | {error, Reason}`

  Set global trace patterns

- `tp(Nodes, PatternList) -> {ok, NodeResults} | {error, Reason}`

  Set global trace patterns
Inviso Reference Manual

- `tp(PatternList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Set global trace patterns
- `tpl(Nodes,Mod,Func,Arity,MatchSpec) -> {ok,NodeResults} | {error,Reason}`  [page ??] Set local trace patterns
- `tpl(Nodes,Mod,Func,Arity,MatchSpec,Opts) -> {ok,NodeResults} | {error,Reason}`  [page ??] Set local trace patterns
- `tpl(Mod,Func,Arity,MatchSpec) -> {ok,NodeResults} | {error,Reason}`  [page ??] Set local trace patterns
- `tpl(Mod,Func,Arity,MatchSpec,Opts) -> {ok,NodeResults} | {error,Reason}`  [page ??] Set local trace patterns
- `ctp(Nodes,Mod,Func,Arity) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear global trace patterns
- `ctp(Mod,Func,Arity) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear local trace patterns
- `ctpl(Nodes,Mod,Func,Arity) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear local trace patterns
- `ctpl(Mod,Funct,Arity) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear local trace patterns
- `tf(Nodes,PidSpec,FlagList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Set process trace flags
- `tf(PidSpec,FlagList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Set process trace flags
- `tf(TraceConfList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Set process trace flags
- `ctf(Nodes,PidSpec,FlagList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear process trace flags
- `ctf(PidSpec,FlagList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear process trace flags
- `ctf(TraceConfList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear process trace flags
- `ctf(Nodes,TraceConfList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear process trace flags
- `ctf(TraceConfList) -> {ok,NodeResults} | {error,Reason}`  [page ??] Clear process trace flags
- `ctf_all(Nodes) -> {ok,NodeResults} | {error,Reason}
  [page ??] Clear all process trace flags
- `ctf_all() -> {ok,NodeResults} | NodeResult | {error,Reason}
  [page ??] Clear all process trace flags
- `init_tpm(Mod,Func,Arity,CallFunc) -> {ok,NodeResults} | NodeResult | {error,Reason}
  [page ??] Initialize meta tracing
- `init_tpm(Nodes,Mod,Func,Arity,CallFunc) -> {ok,NodeResults} | NodeResult | {error,Reason}
  [page ??] Initialize meta tracing
- `init_tpm(Mod,Func,Arity,InitFunc,CallFunc,ReturnFunc,RemoveFunc) ->
  {ok,NodeResults} | NodeResult | {error,Reason}
  [page ??] Initialize meta tracing
- `init_tpm(Nodes,Mod,Func,Arity,InitFunc,CallFunc,ReturnFunc,RemoveFunc) ->
  {ok,NodeResults} | NodeResult | {error,Reason}
  [page ??] Initialize meta tracing
- `tpm(Mod,Func,Arity,MS) -> {ok,NodeResults} | NodeResult | {error,Reason}
  [page ??] Activate meta tracing
- `tpm(Nodes,Mod,Func,Arity,MS) -> {ok,NodeResults} | {error,Reason}
  [page ??] Activate meta tracing
- `tpm(Mod,Func,Arity,MS,CallFunc) -> {ok,NodeResults} | NodeResults | {error,Reason}
  [page ??] Activate meta tracing
- `tpm(Nodes,Mod,Func,Arity,MS,CallFunc) -> {ok,NodeResults} | {error,Reason}
  [page ??] Activate meta tracing
- `tpm(Mod,Func,Arity,MS,InitFunc,CallFunc,ReturnFunc,RemoveFunc) ->
  {ok,NodeResults} | NodeResults | {error,Reason}
  [page ??] Activate meta tracing
- `tpm(Nodes,Mod,Func,Arity,MS,InitFunc,CallFunc,ReturnFunc,RemoveFunc) ->
  {ok,NodeResults} | {error,Reason}
  [page ??] Activate meta tracing
- `tpm_tracer(Mod,Func,Arity,MS) -> {ok,NodeResults} | NodeResult | {error,Reason}
  [page ??] Activate meta tracing and at the same time append a \{tracer,Tracer\}
  process trace flag to the enable list in a match specification trace action term
- `tpm_tracer(Nodes,Mod,Func,Arity,MS) -> {ok,NodeResults} | {error,Reason}
  [page ??] Activate meta tracing and at the same time append a \{tracer,Tracer\}
  process trace flag to the enable list in a match specification trace action term
- `tpm_tracer(Mod,Func,Arity,MS,CallFunc) -> {ok,NodeResults} | NodeResults | {error,Reason}
  [page ??] Activate meta tracing and at the same time append a \{tracer,Tracer\}
  process trace flag to the enable list in a match specification trace action term
• **tpm_tracer(Nodes,Mod,Func,Arity,MS,CallFunc) — {ok,NodeResults} | {error,Reason}**
  [page ??] A ctivate meta tracing and at the same time append a {tracer,Tracer}
  process trace flag to the enable list in a match specification trace action term

• **tpm_tracer(Mod,Func,Arity,MS,InitFunc,CallFunc,ReturnFunc,RemoveFunc) — {ok,NodeResults} | NodeResult | {error,Reason}**
  [page ??] A ctivate meta tracing and at the same time append a {tracer,Tracer}
  process trace flag to the enable list in a match specification trace action term

• **tpm_ms(Mod,Func,Arity,MSname,MS) — {ok,NodeResults} | NodeResult | {error,Reason}**
  [page ??] A dd match specifications

• **tpm_ms(Nodes,Mod,Func,Arity,MSname,MS) — {ok,NodeResults} | {error,Reason}**
  [page ??] A dd match specifications

• **tpm_ms_tracer(Nodes,Mod,Func,Arity,MSname,MS) — {ok,NodeResults} | NodeResult | {error,Reason}**
  [page ??] A dd match specifications and at the same time append a {tracer,Tracer}
  process trace flag to the enable list in a match specification trace action term

• **tpm_ms_tracer(Nodes,Mod,Func,Arity,MSname,MS) — {ok,NodeResults} | {error,Reason}**
  [page ??] A dd match specifications and at the same time append a {tracer,Tracer}
  process trace flag to the enable list in a match specification trace action term

• **ctpm_ms(Mod,Func,Arity,MSname) — {ok,NodeResults} | NodeResult | {error,Reason}**
  [page ??] Remove a match specification

• **ctpm_ms(Nodes,Mod,Func,Arity,MSname) — {ok,NodeResults} | {error,Reason}**
  [page ??] Remove a match specification

• **ctpm(Mod,Func,Arity) — {ok,NodeResults} | NodeResult | {error,Reason}**
  [page ??] Remove a meta trace pattern

• **ctpm(Nodes,Mod,Func,Arity) — {ok,NodeResults} | {error,Reason}**
  [page ??] Remove a meta trace pattern

• **tpm_localnames() — {ok,NodeResults} | NodeResult | {error,Reason}**
  [page ??] Set meta trace pattern on register/2

• **tpm_localnames(Nodes) — {ok,NodeResults} | {error,Reason}**
  [page ??] Set meta trace pattern on register/2

• **ctpm_localnames() — {ok,NodeResults} | NodeResult | {error,Reason}**
  [page ??] Clear meta trace pattern on register/2

• **ctpm_localnames(Nodes) — {ok,NodeResults} | {error,Reason}**
  [page ??] Clear meta trace pattern on register/2

• **tpm_globalnames() — {ok,NodeResults} | NodeResult | {error,Reason}**
  [page ??] Set meta trace pattern on global:register_name/2
- `tpm_globalnames(Nodes)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Set meta trace pattern on `global:register name/2`
- `ctpm_globalnames()` -> `{ok,NodeResults} | NodeResult | `{error,Reason}`
  - [page ??] Clear meta trace pattern on `global:register name/2`
- `ctpm_globalnames(Nodes)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Clear meta trace pattern on `global:register name/2`
- `ctp_all()` -> `{ok,NodeResults} | NodeResult | `{error,Reason}`
  - [page ??] Clear all (global and local) trace patterns
- `ctp_all(Nodes)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Clear all (global and local) trace patterns
- `suspend(SReason)` -> `{ok,NodeResults} | NodeResult | `{error,Reason}`
  - [page ??] Suspend runtime components
- `suspend(Nodes,SReason)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Suspend runtime components
- `cancel_suspension()` -> `{ok,NodeResults} | NodeResult | `{error,Reason}`
  - [page ??] Reactivate suspended runtime components
- `cancel_suspend(Nodes)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Reactivate suspended runtime components
- `get_status()` -> `{ok,NodeResults} | NodeResult | `{error,Reason}`
  - [page ??] Get status of runtime components
- `get_status(Nodes)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Get status of runtime components
- `get_tracerdata()` -> `{ok,NodeResults} | NodeResult | `{error,Reason}`
  - [page ??] Get tracerdata of runtime components
- `get_tracerdata(Nodes)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Get tracerdata of runtime components
- `list_logs()` -> `{ok,NodeResults} | NodeResult | `{error,Reason}`
  - [page ??] Get log file names associated with tracerdata
- `list_logs(Nodes)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Get log file names associated with tracerdata
- `list_logs(NodeTracerData)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Get log file names associated with tracerdata
- `list_logs(TracerData)` -> `{ok,NodeResults} | NodeResult | `{error,Reason}`
  - [page ??] Get log file names associated with tracerdata
- `fetch_log(LogSpecList,DestDir,Prefix)` -> `{ok,NodeResults} | `{error,not_distributed} | `{error,Reason}`
  - [page ??] Fetch log files to control component node
- `fetch_log(DestDir,Prefix)` -> `{ok,NodeResults} | `{error,not_distributed} | `{error,Reason}`
  - [page ??] Fetch log files to control component node
- `delete_log(Nodes,TracerData)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Delete log files associated with tracerdata
- `delete_log(NodeSpecList)` -> `{ok,NodeResults} | `{error,Reason}`
  - [page ??] Delete log files associated with tracerdata
**delete** log(Spec) -> {ok,NodeResults} | NodeResult | {error,Reason}  
Delete log files associated with tracerdata

**delete** log(TracerData) -> {ok,NodeResults} | NodeResult | {error,Reason}  
Delete log files associated with tracerdata

**delete** log() -> {ok,NodeResults} | NodeResult | {error,Reason}  
Delete log files associated with tracerdata

**subscribe**() -> ok | {error,Reason}  
Subscribe to Inviso events

**subscribe**(Pid) -> ok | {error,Reason}  
Subscribe to Inviso events

**unsubscribe**() -> ok  
Unsubscribe to Inviso events

**unsubscribe**(Pid) -> ok  
Unsubscribe to Inviso events

---

**inviso_as_lib**

The following functions are exported:

- **setup_autostart**(Node, R, Opts, TracerData, CmdFiles, Bindings, Transl, RTtag) -> ok | {error,Reason}  
Create an autostart configuration file

- **set_repeat**(Node, R) -> ok | {error,Reason}  
Set the repeat parameter in the autostart file

- **inhibit_autostart**(Node) -> ok | {error,Reason}  
Set the repeat parameter in the autostart file to 0

---

**inviso_lfm**

The following functions are exported:

- **merge**(Files, OutFile) ->  
Merge logfiles into one file in chronological order

- **merge**(Files, WorkHFun, InitHandlerData) ->  
Merge logfiles into one file in chronological order

- **merge**(Files, BeginHFun, WorkHFun, EndHFun, InitHandlerData) -> {ok, Count} | {error,Reason}  
Merge logfiles into one file in chronological order

---

**inviso_lfm_tpfreader**

The following functions are exported:

- **handle_logfile_sort_wrapset**(LogFiles) -> FileList2  
Sort logfiles in chronological order
inviso

The following functions are exported:

- `init_tracing(TracerData) -> NodeResult | {error,Reason}`
  [page ??] Initiate tracing
- `tp(Mod,Func,Arity,MatchSpec,Opts) -> NodeResult | {error,Reason}`
  [page ??] Set global trace patterns
- `tp(Mod,Func,Arity,MatchSpec) -> NodeResult | {error,Reason}`
  [page ??] Set global trace patterns
- `tp(PatternList) -> NodeResult | {error,Reason}`
  [page ??] Set global trace patterns
- `tpl(Mod,Func,Arity,MatchSpec) -> NodeResult | {error,Reason}`
  [page ??] Set local trace patterns
- `tpl(Mod,Func,Arity,MatchSpec,Opts) -> NodeResult | {error,Reason}`
  [page ??] Set local trace patterns
- `ctp(Mod,Func,Arity) -> NodeResult | {error,Reason}`
  [page ??] Clear global trace patterns
- `ctpl(Mod,Func,Arity) -> NodeResult | {error,Reason}`
  [page ??] Clear local trace patterns
- `tf(PidSpec,FlagList) -> NodeResult | {error,Reason}`
  [page ??] Set process trace flags
- `tf(TraceConfList) -> NodeResult | {error,Reason}`
  [page ??] Set process trace flags
- `ctf(PidSpec,FlagList) -> NodeResult | {error,Reason}`
  [page ??] Clear process trace flags
- `ctf(TraceConfList) -> NodeResult | {error,Reason}`
  [page ??] Clear process trace flags
- `init_tpm(Mod,Func,Arity,CallFunc) -> NodeResult | {error,Reason}`
  [page ??] Initialize meta tracing
- `init_tpm(Mod,Func,Arity,InitFunc,CallFunc,ReturnFunc,RemoveFunc) -> NodeResult | {error,Reason}`
  [page ??] Initialize meta tracing
- `tpm(Mod,Func,Arity,MS) -> NodeResult | {error,Reason}`
  [page ??] Activate meta tracing
- `tpm(Mod,Func,Arity,MS,CallFunc) -> NodeResults | {error,Reason}`
  [page ??] Activate meta tracing
- `tpm(Mod,Func,Arity,MS,InitFunc,CallFunc,ReturnFunc,RemoveFunc) -> NodeResults | {error,Reason}`
  [page ??] Activate meta tracing
- `tpm_tracer(Mod,Func,Arity,MS) -> NodeResult | {error,Reason}`
  [page ??] Activate meta tracing
- `tpm_tracer(Mod,Func,Arity,MS,CallFunc) -> NodeResults | {error,Reason}`
  [page ??] Activate meta tracing
The following functions are exported:

- tpm_ms(Mod, Func, Arity, MSname, MS) -> NodeResult | {error, Reason}  
  [page ??] Adds a list of match-specs, associated with the name MSname, to Mod:Func/Arity.

- tpm_ms_tracer(Mod, Func, Arity, MSname, MS) -> NodeResult | {error, Reason}  
  [page ??] Adds a list of match-specs, associated with the name MSname, to Mod:Func/Arity.

- ctpm_ms(Mod, Func, Arity, MSname) -> NodeResult | {error, Reason}  
  [page ??] Removes the list of match-specs associated with the MSname from the meta trace-pattern of Mod:Func/Arity.

- get_tracer() -> Tracer  
  [page ??] Returns the pid or port acting as regular tracer.
**inviso**

**Erlang Module**

With the *inviso* API runtime components can be started and tracing managed across a network of distributed Erlang nodes, using a control component also started with *inviso* API functions.

*Inviso* can be used both in a distributed environment and in a non-distributed. API functions not taking a list of nodes as argument works on all started runtime components. If it is the non-distributed case, that is the local runtime component. The API functions taking a list of nodes as argument, or as part of one of the arguments, can not be used in a non-distributed environment. Return values named NodeResult refers to return values from a single Erlang node, and will therefore be the return in the non-distributed environment.

**Exports**

\[
\text{start()} \rightarrow \{\text{ok}, \text{pid()}\} \mid \{\text{error}, \text{Reason}\}
\]

\[
\text{start(Options)} \rightarrow \{\text{ok}, \text{pid()}\} \mid \{\text{error}, \text{Reason}\}
\]

Types:
- Options = [Option]

Options may contain both options which will be default options to a runtime component when started, and options to the control component. See add_nodes/3 [page ??] for details on runtime component options. The control component recognizes the following options:

- \{subscribe, Pid\} Making the process Pid receive Inviso events from the control component.
  
  Starts a control component process on the local node. A control component must be started before runtime components can be started manually or otherwise accessed through the *inviso* API.

\[
\text{stop()} \rightarrow \text{shutdown}
\]

Stops the control component. Runtime components are left as is. They will behave according to their dependency values.

\[
\text{add_node(RTtag)} \rightarrow \text{NodeResult} \mid \{\text{error}, \text{Reason}\}
\]

\[
\text{add_node(RTtag, Options)} \rightarrow \text{NodeResult} \mid \{\text{error}, \text{Reason}\}
\]

Types:
- RTtag = PreviousRTtag = term()
- Options = [Option]
- Option - see below
- Option = {dependency,Dep}
- Dep = int() | infinity
  The timeout, in milliseconds, before the runtime component will terminate if abandoned by this control component.
- Option = {overload,Overload} | overload
  Controls how and how often overload checks shall be performed. Just overload specifies that no loadcheck shall be performed.
- Overload = Interval | {LoadMF,Interval,InitMFA,RemoveMFA}
- LoadMF = {Mod,Func} | function()/1
- Interval = int() | infinity
  Interval is the time in milliseconds between overload checks.
- InitMFA = RemoveMFA = {Mod,Func,ArgList} | void
  When starting up the runtime component or when changing options (see change_options/2) the overload mechanism is initialized with a call to the InitMFA function. It shall return LoadCheckData. Every time a load check is performed, LoadMF is called with LoadCheckData as its only argument. LoadMF shall return ok or {suspend,Reason}. When the runtime component is stopped or made to change options involving changing overload-check, the RemoveMFA function is called. Its return value is discarded.
- NodeResult = {ok,NAns} | {error,Reason}
- NAns = new | {adopted,State,Status,PreviousRTtag} | already_added
- State = new | tracing | idle
- Status = running | {suspended,SReason}

Starts or tries to connect to an existing runtime component at the local node, regardless if the system is distributed or not. Options will override any default options specified at start-up of the control component.

The PreviousRTtag can indicate if the incarnation of the runtime component at the node in question was started by "us" and then can be expected to do tracing according to "our" instructions or not.

```
add_node_if_ref(RTtag) -> NodeResult | {error,{wrong_reference,OtherTag}} | {error,Reason}
add_node_if_ref(RTtag,Options) -> NodeResult | {error,{wrong_reference,OtherRef}} | {error,Reason}
```

Types:
- OtherRef = term()
  rttag of the running incarnation

As add_node/1,2 [page ??] but will only adopt the runtime component if its rttag is RTtag.

```
add_nodes(Nodes,RTtag) -> {ok,NodeResults} | {error,Reason}
add_nodes(Nodes,RTtag,Options) -> {ok,NodeResults} | {error,Reason}
```

Types:
- Nodes = [Node]
- NodeResults = [{Node,NodeResult}]
As `add_node/1,2 [page ??]` but for a distributed environment.

```
add_node(Ref,RTtag) -> NodeResult | {error,Reason}
```

Types:
- `Nodes` = [Node]
- `NodeResult` = [Node,NodeResult]

As `add_node_if_ref/1,2 [page ??]` but for a distributed environment.

```
stop_nodes() -> {ok,NodeResults} | NodeResult
stop_nodes(Nodes) -> {ok,NodeResults} | {error,Reason}
```

Types:
- `NodeResults` = [Node,NodeResult]
- `NodeResult` = ok | {error,Reason}

Stops runtime component on `Nodes`. `stop_nodes/0` will if the control component is running on a distributed node stop all runtime components. And if running on a non-distributed node, stop the local and only runtime component.

```
change_options(Options) -> NodeResult | {ok,NodeResults} | {error,Reason}
change_options(Nodes,Options) -> {ok,NodeResults} | {error,Reason}
```

Types:
- `Nodes` = [Node]
- `NodeResults` = [Node,NodeResult]
- `NodeResult` = ok | {error,Reason}

Changes the options for one or several runtime components. If for instance overload is redefined, the previous overload will be stopped and the new started. See `add_node/1 [page ??]` for details on `Options`.

```
init_tracing(TracerData) -> {ok,NodeResults} | NodeResult | {error,Reason}
init_tracing(TracerList) -> {ok,NodeResults} | {error,Reason}
init_tracing(Nodes,TracerData) -> {ok,NodeResults} | {error,Reason}
```

Types:
- `TracerData` = [{trace,LogTD,[?,T,T,TD]],[?,T,T,TD]] | LogTD
- `LogTD` = [HandlerFun,Data1] | collector | [relay,CollectingNode] | [ip,IPPortParameters] | [file,FilePortParameters]
- `T,T,TD` = [file,FileName] | [file,FileName,TiSpec] | [relay,Node]
- `TiSpec` = [InitMFA,RemoveM,F,CleanM,F]
- `InitMFA` = [Mi,Fi,Argsi]
RemoveMF = \(\{Mr,Fr\} \rightarrow \text{void}\)
CleanMF = \(\{Mc,Fc\}\)
Mi = Fi = Mr = Fr = Mc = Fd = \text{atom}\()
Argsi = \{\text{term}\}\)
TracerList = \{\text{Node},\text{TracerData}\}\)
IPPortParameters = \{Portno \rightarrow \{\text{Portno},Q\text{size}\}\}
Portno = tcp\(\text{portno}\()\)
Q\text{size} = int\()
FilePortParameters = \{\text{Filename},\text{wrap},T\text{ail},\text{WrapTime},\text{WrapCn}t\} \rightarrow \{\text{Filename},\text{wrap},T\text{ail},\text{WrapSize},\text{WrapCnt}\} \rightarrow \{\text{FileName},\text{wrap},T\text{ail}\} \rightarrow \text{FileName}\)
FileName = string\()
Tail = string\(\neq "\)"\)
WrapTime = WrapCnt = WrapSize = int\(\neq 0\)
TracerList = \{\text{Node},\text{TracerData}\}\)
Nodes = \{\text{Node}\}\)
HandlerFun = function()/2;
HandlerFun(TraceMsg,Data1) \rightarrow \text{NewData}\)
CollectingNode = pid\()\) \rightarrow \text{node}\()
NodeResults = \{\text{Node},\text{NodeResult}\}\)
NodeResult = \{\text{ok},\text{LogResults}\} \rightarrow \{\text{error},\text{N\text{Reason}}\}\)
LogResults = \{\text{LogResult}\}\)
LogResult = \{\text{traceLog},\text{LogRes}\} \rightarrow \{\text{tiLog},\text{LogRes}\}\)
LogRes = \{\text{ok}\} \rightarrow \{\text{error},\text{Reason}\}\)

Starts the tracing at the specified nodes, meaning that the runtime component transits from the state new or idle to tracing. For trace messages to be generated, there must of course also be trace pattern and/or trace flags set. Such can not be set before tracing has been initiated with init_tracing/1,2.

TracerData controls how the runtime component will handle generated trace messages. The trace tag controls how regular trace messages are handled. The ti tag controls if and how trace information will be stored and the meta tracer will be activated. That is if ti is omitted, no meta tracer will be started as part of the runtime component. It is possible to have ti without trace, but most likely not useful.

The ip and file trace tracerdata instructions results in using the built in trace ip-port and file-port respectively. relay will result in that all regular trace messages are forwarded to a runtime component at the specified node. Using a HandlerFun will result in that every incoming regular trace message is applied to the HandlerFun. collector can be used to use this runtime component to receive relayed trace messages and print them to the shell.

The trace information can be configured to either write trace information to a plain trace information file or to relay it to another inviso meta tracer on another node. The inviso meta tracer is capable of matching function calls with their function returns (only if return_trace is activated in the meta trace match specification for the function in question). This is necessary since it may not be possible to decide what to do, if anything shall be done at all, until the return value of the function call is examined.

To be able to match calls with returns a state can be saved when detecting a function call in a public loop data structure kept by the inviso meta tracer. The public loop data structure is given as argument to a handler-function called whenever a meta trace
message arrives to the inviso meta tracer (both function calls and function returns). The public loop data structure is first initiated by the \texttt{Mi:Fi} function which takes the items in \texttt{Argsi} as arguments. \texttt{Fi} shall return the initial public loop data structure. When meta tracing is stopped, either because tracing is stopped or because tracing is suspended, the \texttt{Mr:Fr(PublicLoopData)} is called to offer a possibility to clean-up. Note that for every function meta-tracing is activated, a public loop data modification function can be specified. That function will prepare the current loop data structure for this particular function.

Further there is a risk that function call states becomes abandoned inside the public loop data structure. This will happen if a function call is entered into the public loop data structure, but no function return occurs. To prevent the public loop data structure from growing infinitely the clean function \texttt{Fc} will periodically be called with the public loop data structure as argument. Elements entered into the public loop data structure as a result of a function call must contain a timestamp for the \texttt{Fc} to be able to conclude if it is abandoned or not. \texttt{Fc} shall return a new public loop data structure.

When initiating tracing involving trace information without a TiSpec, a default public loop data structure will be initiated to handle locally registered process aliases. The default public loop data structure is a two-tuple where the first element is used by the meta tracing on the BIF \texttt{register/2}. The second element is left for user usage.

The default public loop data structure may be extended with more element positions. The first position must be left to the implementation of registered-name translations. If the public loop data structure is changed no longer meeting this requirement, the \texttt{tpm_localnames/0,1} and \texttt{tpm_globalnames/0,1} can no longer be used.

A wrap files specification is used to limit the disk space consumed by the trace. The trace is written to a limited number of files each with a limited size. The actual filenames are \texttt{Filename ++ SeqCnt ++ Tail}, where \texttt{SeqCnt} counts as a decimal string from 0 to \texttt{WrapCnt} and then around again from 0. When a trace message written to the current file makes it longer than \texttt{WrapSize}, that file is closed, if the number of files in this wrap trace is as many as \texttt{WrapCnt} the oldest file is deleted then a new file is opened to become the current. Thus, when a wrap trace has been stopped, there are at most \texttt{WrapCnt} trace files saved with a size of at least \texttt{WrapSize} (but not much bigger), except for the last file that might even be empty. The default values are \texttt{WrapSize == 128*1024} and \texttt{WrapCnt == 8}.

The \texttt{SeqCnt} values in the filenames are all in the range 0 through \texttt{WrapCnt} with a gap in the circular sequence. The gap is needed to find the end of the trace.

If the \texttt{WrapSize} is specified as \{time,WrapTime\}, the current file is closed when it has been open more than \texttt{WrapTime} milliseconds, regardless of it being empty or not.

The ip trace driver has a queue of \texttt{QSize} messages waiting to be delivered. If the driver cannot deliver messages as fast as they are produced by the runtime system, they are dropped. The number of dropped messages are indicated in the trace log as separate trace message.

\begin{verbatim}
stop_tracing(Nodes) -> {ok,NodeResults} | {error,Reason}
stop_tracing() -> {ok,NodeResults} | NodeResult

Types:
- Nodes = [Node]
- NodeResults = [{Node,NodeResult}]
- NodeResult = {ok,State} | {error,Reason}
\end{verbatim}
• State = new | idle

   Stops tracing on all or specified Nodes. Flushes the trace buffer if a trace-port is used, closes the trace-port and removes all trace flags and meta-patterns. The nodes are called in parallel.

   Stopping tracing means going to state idle<new>. If the runtime component was already in state <new>, it will of course remain in state new (then there was no tracing to stop).

   clear() -> {ok,NodeResults} | NodeResult
   clear(Nodes,Options) -> {ok,NodeResults} | {error,Reason}
   clear(Options) -> {ok,NodeResults} | NodeResult | {error,Reason}

   Types:
   • Nodes = [Node]
   • Options = [Option]
   • Option = keep_trace_patterns | keep_log_files
   • NodeResults = [{Node,NodeResult}]
   • NodeResult = {ok, {new,Status}} | {error,Reason}
   • Status = running | suspended,SReason

   Stops all tracing including removing meta-trace patterns. Removes all trace patterns. If the node is tracing or idle, trace-logs belonging to the current tracerdata are removed. Hence the node is returned to state new. Note that the node can still be suspended.

   Various options can make the node keep set trace patterns and log-files. The node still enters the new state.

   tp(Nodes,Mod,Func,Arity,MatchSpec,Opts) ->
   tp(Nodes,Mod,Func,Arity,MatchSpec) -> {ok,NodeResults} | {error,Reason}
   tp(Mod,Func,Arity,MatchSpec,Opts) ->
   tp(Mod,Func,Arity,MatchSpec) -> {ok,NodeResults} | NodeResult | {error,Reason}
   tp(Nodes,PatternList) -> {ok,NodeResults} | {error,Reason}
   tp(PatternList) -> {ok,NodeResults} | NodeResult | {error,Reason}

   Types:
   • Nodes = [Node]
   • Mod = Func = atom() | '.'
   • Arity = int() | '.'
   • MatchSpec = true | false | [] | matchspec()
   • PatternList = [Pattern],
   • Pattern = {Mod,Func,Arity,MatchSpec,Opts}
   • Opts = [Opt]
   • Opts = only_loaded
   • NodeResults = [NodeResult]
   • NodeResult = {ok,[Ans]} | {error,Reason}
   • Ans = int() | {error,Reason}
Set trace pattern (global) on specified or all nodes. The integer replied if the call was successfully describes the number of matched functions. The functions without a Nodes argument means all nodes, in a non-distributed environment it means the local node. Using wildcards follows the rules for wildcards of erlang:trace_pattern/3. It is for instance illegal to specify \( M = \_ \) while \( F \) is not \( \_ \).

When calling several nodes, the nodes are called in parallel.

The option only \_loaded will prevent modules not loaded (yet) into the runtime system to become loaded just as a result of that a trace pattern is requested to be set on it. Otherwise modules are automatically loaded if not already loaded (since the module must be present for a trace pattern to be set on it). The latter does not apply if the wildcard \( \_ \) is used as module specification.

```erlang
tpl(Nodes, Mod, Func, Arity, MatchSpec) ->
    tpl(Nodes, Mod, Func, Arity, MatchSpec, Opts) ->
    {ok, NodeResults} | {error, Reason}
    tpl(Mod, Func, Arity, MatchSpec) ->
    tpl(Mod, Func, Arity, MatchSpec, Opts) ->
    {ok, NodeResults} | NodeResult | {error, Reason}
    tpl(Nodes, PatternList) ->
    {ok, NodeResults} | {error, Reason}
    tpl(PatternList) ->
    {ok, NodeResults} | NodeResult | {error, Reason}

See \( \text{tp/N} \) [page ??] function above for details on arguments and return values.
```

Set local trace pattern on specified functions. When calling several nodes, the nodes are called in parallel.

```erlang
ctp(Nodes, Mod, Func, Arity) ->
ctp(Mod, Func, Arity) ->
    {ok, NodeResults} | NodeResult | {error, Reason}

See \( \text{tp/N} \) [page ??] for argument descriptions.
```

Clear global trace patterns. When calling several nodes, the nodes are called in parallel.

```erlang
ctpl(Nodes, Mod, Func, Arity) ->
ctpl(Mod, Funct, Arity) ->
    {ok, NodeResults} | NodeResult | {error, Reason}

See \( \text{tp/N} \) [page ??] for argument description.
```

Clear local trace patterns. When calling several nodes, the nodes are called in parallel.

```erlang
tf(Nodes, PidSpec, FlagList) ->
tf(PidSpec, FlagList) ->
    {ok, NodeResults} | NodeResult | {error, Reason}
    tf(Nodes, TraceConfList) ->
tf(NodeTraceConfList) ->
tf(TraceConfList) ->
    {ok, NodeResults} | NodeResult | {error, Reason}
```

Types:

- \( \text{Nodes} = \left[ \text{Node} \right] \)
- \( \text{NodeTraceConfList} = \left[ \left[ \text{Node}, \text{TraceConfList} \right] \right] \)
- \( \text{TraceConfList} = \left[ \left[ \text{PidSpec}, \text{FlagList} \right] \right] \)
- \( \text{FlagList} = \left[ \text{Flag} \right] \)
- \( \text{PidSpec} = \text{all} | \text{new} | \text{existing} | \text{pid()} | \text{locally_registered_name()} \)
- \( \text{Flag} \) - see erlang:trace/3
- \( \text{NodeResult} = \left[ \text{ok}, \left[ \text{Ans} \right] \right] | \text{error}, \text{Reason} \)
Set process trace flags on processes on all or specified nodes. The integer returned if the call was successful describes the matched number of processes. The functions without a Nodes argument means all nodes, in a non-distributed environment it means the local node.

There are many combinations which does not make much sense. For instance specifying a certain process identifier at all nodes. Or an empty TraceConfList for all nodes. When calling several nodes, the nodes are called in parallel.

ctf(Nodes, PidSpec, FlagList) -> {ok, NodeResults} | {error, Reason}
ctf(PidSpec, FlagList) -> {ok, NodeResults} | NodeResult | {error, Reason}
ctf(Nodes, TraceConfList) -> {ok, NodeResults} | {error, Reason}
ctf(TraceConfList) -> {ok, NodeResults} | NodeResult | {error, Reason}

See tf/N [page ??] for arguments and return value description.

Clears all trace flags on all or specified nodes. Just for convenience.

init_tpm(Mod, Func, Arity, CallFunc) -> {ok, NodeResults} | NodeResult | {error, Reason}
init_tpm(Nodes, Mod, Func, Arity, CallFunc) -> {ok, NodeResults} | {error, Reason}
init_tpm(Mod, Func, Arity, InitFunc, CallFunc, ReturnFunc, RemoveFunc) -> {ok, NodeResults} | NodeResult | {error, Reason}
init_tpm(Nodes, Mod, Func, Arity, InitFunc, CallFunc, ReturnFunc, RemoveFunc) -> {ok, NodeResults} | {error, Reason}

Types:
- Mod = Func = atom()
- Arity = int()
- NodeResults = [{Node, NodeResult}]
- NodeResult = ok | {error, Reason}
- InitFunc, RemoveFunc = {Module, Function} | function()/4 | void
- CallFunc = ReturnFunc = {Module, Function} | function()/3 | void
Initializes Mod:Func/Arity for meta tracing without setting any meta trace patterns. This is necessary if the named match specs will be used (see tpm/ms/5,6 [page ??]). Otherwise initialization of public loop data can be done at the same time as setting meta trace patterns using tpm/8,9 [page ??].

Note that we can not use wildcards here (even if it is perfectly legal in Erlang). It also sets the CallFunc and ReturnFunc for the meta traced function. That is the functions which will be called when a function call and a return_trace meta trace message respectively arrives to the inviso meta tracer for Mod:Func/Arity.

This function is also available without InitFunc and RemoveFunc. That means that no initialization of the public loop data structure will be done and that CallFunc and ReturnFunc must either use already existing parts of public loop data structure or not use it at all.

The InitFunc initializes the already existing public loop data structure for use with Mod:Func/Arity. InitFunc(Mod,Func,Arity,PublLD) -> {ok,NewPublLD,Output} where Output can be a binary which will then be written to the trace information file. If it is not a binary, no output will be done. RemoveFunc will be called when the meta tracing is cleared with ctpm/3,4 [page ??].

RemoveFunc(Mod,Func,Arity,PublLD) -> {ok,NewPublLD}.

See tpm/N [page ??] for details on CallFunc and ReturnFunc.

tpm(Mod,Func,Arity,MS) -> {ok,NodeResults} | NodeResult | {error,Reason}
tpm(Nodes,Mod,Func,Arity,MS) -> {ok,NodeResults} | {error,Reason}
tpm(Mod,Func,Arity,MS,CallFunc) -> {ok,NodeResults} | NodeResults | {error,Reason}
tpm(Nodes,Mod,Func,Arity,MS,CallFunc) -> {ok,NodeResults} | NodeResults | {error,Reason}
tpm(Mod,Func,Arity,MS,InitFunc,CallFunc,ReturnFunc,RemoveFunc) -> {ok,NodeResults} | NodeResults | {error,Reason}
tpm(Nodes,Mod,Func,Arity,MS,InitFunc,CallFunc,ReturnFunc,RemoveFunc) -> {ok,NodeResults} | {error,Reason}

Types:
- Mod = Func = atom()
- Arity = int()
- MS = [match_spec()]
- Nodes = [Node]
- InitFunc = RemoveFunc = {Module,Function} | function()/4 | void
- CallFunc = ReturnFunc = {Module,Function} | function()/3 | void
- NodeResults = [{Node,NodeResult}]
- NodeResult = {ok,1} | {ok,0} | {error,Reason} 1

Activates meta-tracing in the inviso rt meta tracer. Except when using tpm/6, tpm/8 and tpm/9 the Mod:Func/Arity must first have been initiated using init_tpm/N [page ??]. When calling several nodes, the nodes are called in parallel.

CallFunc will be called every time a meta trace message arrives to the inviso meta tracer because of a call to Func. CallFunc(CallingPid,ActualArgList,PublLD) -> {ok,NewPrivLD,Output} where Output can be a binary or void. If it is a binary it will be written to the trace information file.

ReturnFunc will be called every time a meta return_trace message arrives to the inviso meta tracer because of a return_trace of a call to Func. ReturnFunc(CallingPid,ReturnValue,PublLD) -> {ok,NewPrivLD,Output}.

Further the ReturnFunc must handle the fact that a return_trace message arrives for a
call which was never noticed. This because the message queue of the meta tracer may have been emptied.

```erlang
tpm_tracer(Mod, Func, Arity, MS) -> {ok, NodeResults} | NodeResult | {error, Reason}
tpm_tracer(Nodes, Mod, Func, Arity, MS) -> {ok, NodeResults} | {error, Reason}
tpm_tracer(Mod, Func, Arity, MS, CallFunc) -> {ok, NodeResults} | NodeResults |
                                 | {error, Reason}
tpm_tracer(Nodes, Mod, Func, Arity, MS, CallFunc) -> {ok, NodeResults} | {error, Reason}
tpm_tracer(Mod, Func, Arity, MS, InitFunc, CallFunc, ReturnFunc, RemoveFunc) -> 
                                 {ok, NodeResults} | {error, Reason}
tpm_tracer(Nodes, Mod, Func, Arity, MS, InitFunc, CallFunc, ReturnFunc, RemoveFunc) ->
                                 {ok, NodeResults} | {error, Reason}
```

See `tpm/X` for details on arguments and return values.

Same as `tpm/X` but all match specs in `MS` containing a `trace action term` will have a `tracer` appended to its enable-list. `Tracer` will be the current output for regular trace messages as specified when tracing was initiated. This function is useful when setting a meta trace pattern on a function with the intent that its execution shall turn tracing on for the process executing the match-spec in the meta trace pattern. The reason the `tracer` process trace flag can not be explicitly written in the action term by the user is that it may be difficult to learn its exact value for a remote node. Further more inviso functions are made to work on several nodes at the same time, requiering different match specs to be set for different nodes.

Simple example: We want any process executing the function `mymod:init(1234)` (with the argument, exactly the integer 1234) to begin function-call tracing. In the example, if the process is found to be one that shall start call tracing, we also first disable all process trace flags to ensure that we have full control over what the process traces. 

```erlang
inviso:tpm_tracer(mymod,init,1,[[{1234},[],[{trace,[all],[call]},{tracer,T}]],void).
```

Where `T` is the tracer for regular trace messages (most often a trace-port, but can be the runtime component `inviso:rt` process), and `P` is the meta tracer (the `inviso:rt` process).

```erlang
tpm_ms(Mod, Func, Arity, MSname, MS) -> {ok, NodeResults} | NodeResult | {error, Reason}
tpm_ms(Nodes, Mod, Func, Arity, MSname, MS) -> {ok, NodeResults} | {error, Reason}
```

Types:

- `Nodes` = `[Node]<v> M od = Func = atom()<v> A rity = int()<v> <v> M S name = term()<v> M S = [match spec()]<v> NodeResults = [{Node,NodeResult}]<v> NodeResult = {ok,1} | {ok,0} | {error,Reason}<v>`
This function adds a list of match-specs to the already existing ones. It uses an internal database to keep track of existing match-specs. This set of match specs can hereafter be referred to with the name \texttt{MSname}. If the match-spec does not result in any meta traced functions (for whatever reason), the \texttt{MS} is not saved in the database. The previously known match-specs are not removed. If \texttt{MSname} is already in use as a name referring to a set of match-specs for this particular meta-traced function, the previous set of match-specs are replaced with \texttt{MS}.

\texttt{Mod:Func/Arity} must previously have been initiated in order for this function to add a match-spec.

When calling several nodes, the nodes are called in parallel. \{\texttt{ok,1}\} indicates success.

\begin{verbatim}
tpm\_ms\_tracer(Mod,Func,Arity,MSname,MS) \rightarrow \{\texttt{ok,NodeResults} | \texttt{NodeResult} | \texttt{error,Reason}\}
tpm\_ms\_tracer(Nodes,Mod,Func,Arity,MSname,MS) \rightarrow \{\texttt{ok,NodeResults} | \{\texttt{error,Reason}\}

See \texttt{tpm\_ms\_X} for details on arguments and return values, and \texttt{tpm\_tracer\_X} for explanations about the appending of \{tracer,Tracer\} process trace flag.
\end{verbatim}

\begin{verbatim}
ctpm\_ms(Mod,Func,Arity,MSname) \rightarrow \{\texttt{ok,NodeResults} | \texttt{NodeResult} | \{\texttt{error,Reason}\}
ctpm\_ms(Nodes,Mod,Func,Arity,MSname) \rightarrow \{\texttt{ok,NodeResults} | \{\texttt{error,Reason}\}

Types:
\begin{itemize}
  \item \texttt{NodeResults} = \{\texttt{Node,NodeResult}\}
  \item \texttt{NodeResult} = \texttt{ok} | \{\texttt{error,Reason}\}
\end{itemize}

Removes a named match-spec from the meta traced function. Note that it never is a fault to remove a match spec. Not even from a function which is non existent.

When calling several nodes, the nodes are called in parallel.
\end{verbatim}

\begin{verbatim}
ctpm(Mod,Func,Arity) \rightarrow \{\texttt{ok,NodeResults} | \texttt{NodeResult} | \{\texttt{error,Reason}\}
ctpm(Nodes,Mod,Func,Arity) \rightarrow \{\texttt{ok,NodeResults} | \{\texttt{error,Reason}\}

Types:
\begin{itemize}
  \item \texttt{NodeResults} = \{\texttt{Node,NodeResult}\}
  \item \texttt{NodeResult} = \texttt{ok} | \{\texttt{error,Reason}\}
\end{itemize}

Removes the meta trace pattern for the function, means stops generating output for this function. The public loop data structure may be cleared by the previously entered \texttt{RemoveFunc}.

When calling several nodes, the nodes are called in parallel.
\end{verbatim}

\begin{verbatim}
ctpm\_localnames() \rightarrow \{\texttt{ok,NodeResults} | \texttt{NodeResult} | \{\texttt{error,Reason}\}
ctpm\_localnames(Nodes) \rightarrow \{\texttt{ok,NodeResults} | \{\texttt{error,Reason}\}

Types:
\begin{itemize}
  \item \texttt{NodeResults} = \{\texttt{Node,NodeResult}\}
  \item \texttt{NodeResult} = \{\texttt{R1,R2}\}
  \item \texttt{R1 = R2 = ok,0} | \{\texttt{ok,1} | \{\texttt{error,Reason}\}
\end{itemize}
\end{verbatim}
Quick version for setting meta-trace patterns on `erlang:register/2`. It uses a default CallFunc and ReturnFunc in the meta-tracer server. The main purpose of this function is to create ti-log entries for associations between pids and registered name aliases. The implementation uses return_trace to see if the registration was successful or not, before actually making the ti-log alias entry. Further the implementation also meta traces the BIF `unregister/1`.

If both \( N_1 \) and \( N_2 \) is 1, function call was successful. \( N_1 \) and \( N_2 \) represent setting meta trace pattern on `register/2` and `unregister/1`.

ctpm_localnames() -> {ok,NodeResults} | NodeResult | {error,Reason}
ctpm_localnames(Nodes) -> {ok,NodeResults} | {error,Reason}

Types:
- NodeResults = [{Node,NodeResult}]
- NodeResult = {R1,R2}
- \( R_1 = R_2 = ok \) | {error,Reason}

Function for removing previously set patterns by `tpm_localnames/0` [page ??]. The two results \( R_1 \) and \( R_2 \) represents that meta pattern is removed from both `register/2` and `unregister/1`.

ctpm_globalnames() -> {ok,NodeResults} | NodeResult | {error,Reason}
ctpm_globalnames(Nodes) -> {ok,NodeResults} | {error,Reason}

Types:
- NodeResults = [{Node,NodeResult}]
- NodeResult = {R1,R2}
- \( R_1 = R_2 = \{ok,0\} \) | \( \{ok,1\} \) | {error,Reason}

Quick version for setting meta-trace patterns capable of learning the association of a pid with a globally registered name (registered using `global:register_name`). The implementation meta-traces on `global:handle_call` and `global:delete_global_name/2`. The \( N_1 \) and \( N_2 \) represents the success of the two sub-tmp calls.

ctpm_globalnames() -> {ok,NodeResults} | NodeResult | {error,Reason}
ctpm_globalnames(Nodes) -> {ok,NodeResults} | {error,Reason}

Types:
- NodeResults = [{Node,NodeResult}]
- NodeResult = {R1,R2} | {error,Reason}
- \( R_1 = R_2 = ok \) | {error,Reason}

Function for removing previously set meta patters by `tpm_globalnames/0,1` [page ??]. The two results \( R_1 \) and \( R_2 \) represents that meta pattern are removed from both `global:handle_call/3` and `global:delete_global_name/1`.

ctp_all() -> {ok,NodeResults} | NodeResult | {error,Reason}
ctp_all(Nodes) -> {ok,NodeResults} | {error,Reason}

Types:
- NodeResults = [{Node,NodeResult}]

Inviso Application
Clears all, both global and local trace patterns. Does not clear meta trace patterns. Equivalent to a call to ctp/3,4 [page ??] and to ctpl/3,4 [page ??] with wildcards ‘_’ for all modules, functions and arities.

\[ \text{suspend}(SReason) \rightarrow \{ \text{ok}, \text{NodeResults} \} | \text{NodeResult} | \{ \text{error}, \text{Reason} \} \]
\[ \text{suspend}(\text{Nodes}, SReason) \rightarrow \{ \text{ok}, \text{NodeResults} \} | \{ \text{error}, \text{Reason} \} \]

Types:
- \( SReason = \text{term}() \)
- \( \text{NodeResults} = \{ \{ \text{Node}, \text{NodeResult} \} \} \)
- \( \text{NodeResult} = \text{ok} \lor \{ \text{error}, \text{Reason} \} \)

Suspends the runtime components. \( SReason \) will become the suspend-reason replied in for instance a \text{get_status}/0,1 [page ??] call. A runtime component that becomes suspended removes all trace flags and all meta trace patterns. In that way trace output is no longer generated. The task of reactivating a suspended runtime component is outside the scope of inviso. It can for instance be implemented by a higher layer trace-tool “remembering” all trace flags and meta patterns set.

\[ \text{cancel_suspension}() \rightarrow \{ \text{ok}, \text{NodeResults} \} | \text{NodeResult} | \{ \text{error}, \text{Reason} \} \]
\[ \text{cancel_suspend}(\text{Nodes}) \rightarrow \{ \text{ok}, \text{NodeResults} \} | \{ \text{error}, \text{Reason} \} \]

Types:
- \( \text{NodeResults} = \{ \{ \text{Node}, \text{NodeResult} \} \} \)
- \( \text{NodeResult} = \text{ok} \lor \{ \text{error}, \text{Reason} \} \)

Makes the runtime components running again (as opposite to suspended). Since reactivating previous trace flags and meta trace patterns is outside the scope of inviso, cancelling suspension is simply making it possible to set trace flags and meta trace patterns again.

\[ \text{get_status}() \rightarrow \{ \text{ok}, \text{NodeResults} \} | \text{NodeResult} | \{ \text{error}, \text{Reason} \} \]
\[ \text{get_status}(\text{Nodes}) \rightarrow \{ \text{ok}, \text{NodeResults} \} | \{ \text{error}, \text{Reason} \} \]

Types:
- \( \text{NodeResults} = \{ \{ \text{Node}, \text{NodeResult} \} \} \)
- \( \text{NodeResult} = \{ \text{ok}, \{ \text{State}, \text{Status} \} \} \lor \{ \text{error}, \text{Reason} \} \)
- \( \text{State} = \text{new} \lor \text{idle} \lor \text{tracing} \)
- \( \text{Status} = \text{running} \lor \{ \text{suspended}, \text{SReason} \} \)
- \( \text{SReason} = \text{term}() \)

Finds out the state and status of a runtime component. A runtime component is in state \text{new} before it has been initiated to do any tracing the first time. There are clear-functions which can make a runtime component become \text{new} again without having to restart. A runtime component becomes \text{idle} after tracing is stopped.

\[ \text{get_tracerdata}() \rightarrow \{ \text{ok}, \text{NodeResults} \} | \text{NodeResult} | \{ \text{error}, \text{Reason} \} \]
\[ \text{get_tracerdata}(\text{Nodes}) \rightarrow \{ \text{ok}, \text{NodeResults} \} | \{ \text{error}, \text{Reason} \} \]

Types:
- \( \text{NodeResults} = \{ \{ \text{Node}, \text{NodeResult} \} \} \)
Returns the current tracerdata of a runtime component. A runtime component in state `new` can not have tracerdata. An idle runtime component does have tracerdata, the last active tracerdata. `TracerData` will be a term as specified to `init_tracing` when tracing was initiated for the runtime component.

`list_logs()` -> \{ok, NodeResults\} | NodeResult | \{error, Reason\}
`list_logs(Nodes)` -> \{ok, NodeResults\} | \{error, Reason\}
`list_logs(NodeTracerData)` -> \{ok, NodeResults\} | \{error, Reason\}
`list_logs(TracerData)` -> \{ok, NodeResults\} | NodeResult | \{error, Reason\}

Types:
- `TracerData` - see `init_tracing\1,2`
- `NodeResults` = [{Node, NodeResult}]
- `NodeResult` = \{ok, FileList\} | \{ok, no_Jog\} | \{error, Reason\}
- `FileList` = [FileType]
- `FileType` = \{trace_log, Dir, FileList\} | \{ti_log, Dir, Files\}
- `Files` = [FileNameWithOutPath]

Returns the actually existing log files associated with `TracerData`. If a tracerdata is not specified, current tracerdata is used for that particular runtime component. `Files` will be a list of one or more files should it be a wrap-set. Otherwise the it is a list of only one filename.

This function is useful to learn the name and path of all files belonging to a trace. This information can later be used to move those files for merging. Note that since it is possible to ask on other tracerdata than the current, it is possible to learn filenames of previously done traces, under the circumstances that they have not been removed.

`fetch_log(LogSpecList, DestDir, Prefix)` -> \{ok, NodeResults\} | \{error, not_distributed\} | \{error, Reason\}
`fetch_log(DestDir, Prefix)` -> \{ok, NodeResults\} | \{error, not_distributed\} | \{error, Reason\}

Types:
- `DestDir` = string()
- `Prefix` = string()
- `LogSpecList` = [LogSpec]
- `LogSpec` = \{Node, FileSpecList\} | Node | \{Node, TracerData\}
- `TracerData` = see `init_tracing\1,2`
- `FileSpecList` = [{trace_log, Dir, FileList}, {ti_log, Dir, FileList}] | [{trace_log, Dir, FileList}]
- `FileList` = [FileNameWithOutPath]
- `NodeResult` = \{Conclusion, ResultFileSpec\} | no_Jog | \{error, NReason\}
- `NReason` = own_node | Reason
- `Conclusion` = complete | incomplete
- `ResultFileSpec` = [{trace_log, FileResults}, {ti_log, FileResults}]
- `FileResults` = [FileResult]
- `FileResult` = \{ok, Filename\} | \{error, FReason\}
CFS: Reason = \{file_open, posix(), FileName\} | \{file_open, posix(), DestDir, Prefix, RemoteFileName\} | \
\{file_write, posix(), FileName\} | \{truncated, Reason, FileName\}

posix() = atom()

Copies log files over distributed erlang to the control component node. This function can only be used in a distributed system.

The resulting transferred files will have the prefix Prefix and will be located in DestDir. The source files can either be pointed out using a FileListSpec or tracerdata. If no files are explicitly specified, current tracerdata for that node will be used. Note that if source files have the same name (on several nodes) they will overwrite each other at DestDir.

```
define_log(Nodes, TracerData) -> {ok, NodeResults} | {error, Reason}
define_log(NodeSpecList) -> {ok, NodeResults} | {error, Reason}
define_log(Spec) -> {ok, NodeResults} | NodeResult | {error, Reason}
define_log(TracerData) -> {ok, NodeResults} | NodeResult | {error, Reason}
define_log() -> {ok, NodeResults} | NodeResult | {error, Reason}
```

Types:
- Nodes = \[Node\]
- NodeSpecList = \[\{Node, Spec\}\]
- Spec = \{AbsPathFileName\} | LogSpecs
- LogSpecs = \[LogSpec\]
- LogSpec = \{trace_log, Dir, FileNameWithoutPath\} | \
\{ti_log, Dir, FileNameWithoutPath\}
- TracerData – see init_tracing/1, 2
- NodeResults = \[\{Node, NodeResult\}\]
- NodeResult = \{ok, no_log\} | \{ok, LogInfos\} | \{ok, FileInfo\}
- LogInfos = \[LogInfo\]
- LogInfo = \{trace_log, FileInfos\} | \{ti_log, FileInfos\}
- FileInfo = \{ok, FileName\} | \{error, Reason\}

Deletes listed files or files corresponding to tracerdata. If no tracerdata or list of files are specified in the call, current tracerdata at the runtime components will be used to identify files to delete. All filenames shall be strings.

FileName can either be an absolute path or just a filename depending on if AbsPathFileName or a LogSpec was used to identify the file.

```
subscribe() -> ok | \{error, Reason\}
subscribe(Pid) -> ok | \{error, Reason\}
```

Types:
- Pid = pid()

Adds Pid or self() if using subscribe/0 to the inviso-event sending list. Note that it is possible to add a pid several times and that the Pid then will receive multiple copies of inviso-event messages.

All events will be sent to all subscribers in the event sending list.
Event = {inviso_event,ControllerPid,erlang:localtime(),Msg}
    Msg = {connected, Node, {RTtag, {State,Status}}}
        | {disconnected, Node, NA}
        | {state_change,Node,{State,Status}}
        | {port_down,Node,Reason}
    Node = node() | local_runtime

Subscribing to inviso-event may be necessary for a higher layer trace-tool using inviso to follow the runtime components. `local_runtime` will be used for a runtime component running in a non-distributed environment.

unsubscribe() -> ok
unsubscribe(Pid) -> ok

Removes Pid (once) from the subscription list.
The purpose of the Inviso autostart utility library is to facilitate the creation and modification of autostart configuration files used by the standard autostart.

Exports

setup_autostart(Node, R, Opts, TracerData, CmdFiles, Bindings, Transl, RTtag) -> ok | {error, Reason}

Types:
- Node = atom()
- R = int()
- Opts - see inviso:add_nodes/2,3
- TracerData - see inviso:init_tracing/1,2
- CmdFiles = [CmdFile]
- CmdFile = string()
- Bindings = [{Var,Val}]
- Var = atom()
- Val = term()
- Transl = [{{M1,F1,Arity}, {M2,F2, {Mt,Ft}}}]  
- M1 = F1 = M2 = F2 = Mt = Ft = atom()
- Arity = int()
- RTtag = term()
- Reason = term()

Creates an autostart configuration file on Node. The name of the file is automatically deduced from consulting the Runtime Tools configuration parameters at Node. 

R is the number of allowed autostarts remaining.

Opts is the options which shall be given to the runtime component. See inviso:add_nodes/2,3 [page ??].

TracerData is used when initiating tracing on this node. See inviso:init_tracing/1,2 [page ??].

CmdFiles points out files containing instructions understood by the inviso_autostart_server implementation of an autostart initiator.

Bindings is a list of {Var, Val} tuples, where Var is the name of a variable and Val the actual value of the variable.

Transl means that M1:F1/Arity shall be translated into M2:F2.
RTtag is the incarnation tag of the runtime component. See See `inviso:add_nodes/2,3` [page ??].

```erlang
set_repeat(Node, R) -> ok | {error, Reason}
```

**Types:**
- Node = atom()
- R = int()
- Reason = term()

Sets the repeat parameter in the autostart file at Node without changing any of its other contents. The autostart configuration file must exist.

R is the number of allowed autostarts remaining.

```erlang
inhibit_autostart(Node) -> ok | {error, Reason}
```

**Types:**
- Node = atom()
- Reason = term()

Sets the repeat parameter in the autostart file at Node to 0. Equivalent to `set_repeat(Node, 0)`.
**inviso_lfm**

**Erlang Module**

Implements an off-line logfile merger, merging binary trace-log files from several nodes together in chronological order. The logfile merger can also do pid-to-alias translations.

The logfile merger is supposed to be called from the Erlang shell or a higher layer trace tool. For it to work, all logfiles and trace information files (containing the pid-alias associations) must be located in the file system accessible from this node and organized according to the API description.

The logfile merger starts a process, the output process, which in its turn starts one reader process for every node it shall merge logfiles from. Note that the reason for a process for each node is not remote communication, the logfile merger is an off-line utility, it is to sort the logfile entries in chronological order.

The logfile merger can be customized both when it comes to the implementation of the reader processes and the output the output process shall generate for every logfile entry.

**Exports**

merge(Files, OutFile) ->
merge(Files, WorkHFun, InitHandlerData) ->
merge(Files, BeginHFun, WorkHFun, EndHFun, InitHandlerData) -> {ok, Count} | {error, Reason}

**Types:**
- Files = [FileDescription]
- FileDescription = FileSet | {reader,RMod,RFunc,FileSet}
- FileSet = [Node, LogFiles]
- Node = atom()
- LogFiles = [{trace_log, FileName} | {ti_log, FileName}]
- FileName = string()
- RMod = RFunc = atom()
- OutFile = string()
- BeginHFun = fun(InitHandlerData) -> {ok, NewHandlerData} | {error, Reason}
- WorkHFun = fun(Node, LogEntry, PidMappings, HandlerData) -> {ok, NewHandlerData}
- LogEntry = tuple()
- PidMappings = term()
- EndHFun = fun(HandlerData) -> ok | {error, Reason}
- Count = int()
- Reason = term()
Merges the logfiles in Files together into one file in chronological order. The logfile merger consists of an output process and one or several reader processes.

Returns \{ok, Count\} where Count is the total number of log entries process, if successful.

When specifying LogFiles, currently the standard reader-process only supports:

- one single file
- a list of wraplog files, following the naming convention
  \texttt{\langle Prefix\rangle\langle Nr\rangle\langle Suffix\rangle}.

FileDescription == \{reader, RMod, RFunc, FileSet\} indicates that \texttt{spawn(RMod, RFunc, [OutputPid, LogFiles])} shall create a reader process.

The output process is customized with \texttt{BeginHFun}, \texttt{WorkHFun} and \texttt{EndHFun}. If using \texttt{merge/2} a default output process configuration is used, basically creating a text file and writing the output line by line. \texttt{BeginHFun} is called once before requesting log entries from the reader processes. \texttt{WorkHFun} is called for every log entry (trace message) \texttt{LogEntry}. Here the log entry typically gets written to the output. \texttt{PidMappings} is the translations produced by the reader process. \texttt{EndHFun} is called when all reader processes have terminated.

The standard reader process is implemented in the module \texttt{inviso_lfmttpreader} (trace port reader). It understands Erlang linked in trace-port driver generated trace-logs and \texttt{inviso_rtm} generated trace information files.

Writing Your Own Reader Process

Writing a reader process is not that difficult. It must:

- Export an init-like function accepting two arguments, pid of the output process and the LogFiles component. LogFiles is actually only used by the reader processes, making it possible to redefine LogFiles if implementing an own reader process.
- Respond to \{get\_next\_entry, OutputPid\} messages with \{next\_entry, self(), PidMappings, NowTimeStamp, Term\} or \{next\_entry, self(), \{error, Reason\}\}.
- Terminate normally when no more log entries are available.
- Terminate on an incoming EXIT-signal from OutputPid.

The reader process must of course understand the format of a logfile written by the runtime component.
inviso_lfm_tpfreader

Erlang Module

Implements the standard reader process to the standard logfile merger inviso_lfm. The reader process reads logfiles belonging to the same set (normally one node) in chronological order and delivers logged trace messages one by one to the output process. Before any trace messages are delivered, the inviso_lfm_tpfreader implementation reads the entire trace information file (if in use) and builds a database over pid-to-alias associations.

The inviso_lfm_tpfreader implementation is capable of considering that an alias may have been used for several processes during different times. An alias may also be in use for several pids at the same time, on purpose. If a process has generated a trace message, all associations between that pid and aliases will be presented as the list PidMappings in the message sent to the output process.

Exports

handle_logfile_sort_wrapset(LogFiles) -> FileList2

Types:
- LogFiles = [{trace_log, FileList}]
- FileList = FileList2 = [FileName]
- FileName = string()

Only one {trace_log, FileList} tuple is expected in LogFiles, all other tuples are ignored. FileList must:

- contain one single file name, or
- a list of wraplog files, following the naming convention <Prefix><Nr><Suffix>.

Sorts the files in FileList in chronological order beginning with the oldest. Sorting is only relevant if FileList is a list of wraplogs. The sorting is done on finding the modulo-counter in the filename and not on filesystem timestamps.

This function is exported for convenience should an own reader process be implemented.

Inviso Application
The Trace Information File Protocol

The format of a trace information file is dictated by the meta tracer process. The inviso_lfm_tpfreader implementation of a reader process understands the following trace information entries. Note that the inviso_rt_meta trace information file is on binary format prefixing every entry with a 4 byte length indicator.

{Pid, Alias, alias, NowStamp} Pid = pid()
  Alias = term()
  NowStamp = term(), but in current implementation as returned from erlang:now/0
  This message indicates that from now on shall Pid be associated with Alias.

{MaybePid, Alias, unalias, NowStamp} MaybePid = pid() | undefined
  Alias = term()
  NowStamp = term(), see above
  This message indicates that, if MaybePid is a pid, this pid shall no longer be associated with Alias. If it is undefined, all associations with Alias from now shall be considered invalid.
  Also note that there are many situations where unalias entries will be missing. For instance if a process terminates without making explicit function calls removing its associations first. This is seldom a problem unless the pid is reused.
Erlang Module

The inviso_rt API is normally only used when programming autostart scripts or similar mechanisms. The reason is that the runtime component is part of the Runtime tools application and will therefore always be available. But the regular inviso API is part of the Inviso application not necessarily available on the node doing an autostart. It is of course possible to run a “lean” tracer only using the runtime component manually (i.e. not through autostart). The runtime component shall otherwise be controlled through the control component, which is accessed with the inviso API.

Exports

init_tracing(TracerData) -> NodeResult | {error,Reason}
See inviso:init_tracing/2 [page ??] for details.

tp(Mod,Func,Arity,MatchSpec,Opts) ->
tp(Mod,Func,Arity,MatchSpec) -> NodeResult | {error,Reason}
tp(PatternList) -> NodeResult | {error,Reason}

Types:
- Mod,Func = atom() | '_' | ModRegExp | {DirRegExp,ModRegExp}
- ModRegExp = regexp_string()
- DirRegExp = regexp_string()
- Arity = int() | '_'
- MatchSpec = true | false | [] | matchspec()
- PatternList = [Pattern],
- Pattern = {Mod,Func,Arity,MatchSpec,Opts}
- Opts = [Opt]
- Opts = only_loaded
- NodeResult = {ok,[Ans]} | {error,Reason}
- Ans = int() | {error,Reason}

Set global trace patterns. The integer replied if the call was successfull describes the number of matched functions. Using wildcards follows the rules for wildcards of erlang:trace_pattern. It is for instance illegal to specify M=='_' while F is not '_'.

Modules can also be specified using Erlang regular expressions as described in the regexp module. If {DirRegExp,ModRegExp} is used, module selection will further be restricted by that the module must be loaded from a location containing DirRegExp somewhere in the path. This can be used to for instance trace on all modules belonging to a certain application.
tpl(Mod, Func, Arity, MatchSpec) -> NodeResult | {error, Reason}

tpl(Mod, Func, Arity, MatchSpec, Optx) -> NodeResult | {error, Reason}

See tp/N [page ??] function above for details on arguments and return values.
Set local trace pattern on specified functions.

ctp(Mod, Func, Arity) -> NodeResult | {error, Reason}

See tp/N [page ??] for argument descriptions.
Clear global trace patterns.

ctpl(Mod, Func, Arity) -> NodeResult | {error, Reason}

See tp/N [page ??] for argument description.
Clear local trace patterns.

tf(PidSpec, FlagList) -> NodeResult | {error, Reason}
tf(TraceConfList) -> NodeResult | {error, Reason}

Types:
- TraceConfList = [{PidSpec, FlagList}]
- FlagList = [Flag]
- PidSpec = all | new | existing | pid() | locally_registered_name()
- Flag = all process trace flags allowed.
- NodeResult = {ok, [Ans]} | {error, Reason}
- Ans = int() | {error, Reason}

Set process trace flags. The integer returned if the call was successful describes the matched number of processes.

ctf(PidSpec, FlagList) -> NodeResult | {error, Reason}
ctf(TraceConfList) -> NodeResult | {error, Reason}

See tf/1,2 [page ??] for arguments and return value description.
Clear process trace flags.

init_tpm(Mod, Func, Arity, CallFunc) -> NodeResult | {error, Reason}

init_tpm(Mod, Func, Arity, InitFunc, CallFunc, ReturnFunc, RemoveFunc) -> NodeResult | {error, Reason}

Types:
- Mod = Func = atom()
- Arity = int()
- NodeResult = ok | {error, Reason}
- InitFunc = ReturnFunc = {Module, Function} | function()/4 | void

See inviso:init_tpm/5,7 [page ??] for details.

tpm(Mod, Func, Arity, MS) -> NodeResult | {error, Reason}
tpm(Mod, Func, Arity, MS, CallFunc) -> NodeResults | {error, Reason}
inviso_rt

Inviso Reference Manual

\texttt{tpm(Mod,Func,Arity,MS,InitFunc,CallFunc,ReturnFunc,RemoveFunc) -> NodeResults | \{error,Reason\}}

Types:
- \texttt{Mod = Func = atom() \neq \_}
- \texttt{Arity = int()}
- \texttt{MS = match_spec()}
- \texttt{InitFunc = CallFunc = ReturnFunc = RemoveFunc = \{Module,Function\} | function()}
- \texttt{NodeResult = \{ok,1\} | \{ok,0\} | \{error,Reason\}}

See inviso:tpm/4,5,8 [page ??] for details.

\texttt{tpm tracer(Mod,Func,Arity,MS) -> NodeResult | \{error,Reason\}}

\texttt{tpm tracer(Mod,Func,Arity,MS,CallFunc) -> NodeResults | \{error,Reason\}}

\texttt{tpm tracer(Mod,Func,Arity,MS,InitFunc,CallFunc,ReturnFunc,RemoveFunc) -> NodeResults | \{error,Reason\}}

See inviso:tpm tracer/4,5,8 for details.

\texttt{tpm_ms(Mod,Func,Arity,MSname,MS) -> NodeResult | \{error,Reason\}}

Types:
- \texttt{Mod = Func = atom()}
- \texttt{Arity = int()}
- \texttt{MSname = term()}
- \texttt{MatchSpec = \{match_spec()\}}
- \texttt{NodeResult = \{ok,1\} | \{ok,0\} | \{error,Reason\}}

See inviso:tpm_ms/5 [page ??] for details.

\texttt{tpm_ms tracer(Mod,Func,Arity,MSname,MS) -> NodeResult | \{error,Reason\}}

See inviso:tpm_ms tracer/5 for details.

\texttt{ctpm_ms(Mod,Func,Arity,MSname) -> NodeResult | \{error,Reason\}}

Types:
- \texttt{NodeResult = ok | \{error,Reason\}}

See inviso:ctpm_ms/4 [page ??] for details.

\texttt{ctpm(Mod,Func,Arity) -> \{ok,NodeResults\} | NodeResult | \{error,Reason\}}

Types:
- \texttt{NodeResults = \{Node,NodeResult\}}
- \texttt{NodeResult = ok | \{error,Reason\}}

See inviso:ctpm/3 [page ??] for details.

\texttt{local_register() -> NodeResult | \{error,Reason\}}

Types:
- \texttt{NodeResult = \{R1,R2\}}
- \texttt{R1 = R2 = \{ok,0\} | \{ok,1\} | \{error,Reason\}}
See inviso:tpm_localnames/0 [page ??] for details.

remove_local_register() -> NodeResult | {error,Reason}

Types:
- NodeResult = {R1,R2} | {error,Reason}
- R1 = R2 = ok | {error,Reason}

See inviso:ctpm_localnames/0 [page ??] for details.

global_register() -> NodeResult | {error,Reason}

Types:
- NodeResult = {R1,R2} | {error,Reason}
- R1 = R2 = {ok,0} | {ok,1} | {error,Reason}

See inviso:tpm_globalnames/0 [page ??] for details.

remove_global_register() -> NodeResult | {error,Reason}

Types:
- NodeResult = {R1,R2} | {error,Reason}
- R1 = R2 = ok | {error,Reason}

See inviso:ctpm_globalnames/0 [page ??] for details.