RTA-TRACE

Configuration Guide

Contact Details

ETAS Group

www.etasgroup.com

Germany

ETAS GmbH
Borsigstraße 14
70469 Stuttgart

Tel.:+49 (711) 8 96 61-102
Fax:+49 (711) 8 96 61-106

www.etas.de

USA

ETAS Inc.
3021 Miller Road
Ann Arbor, MI 48103

Tel.: +1 (888) ETAS INC
Fax: +1 (734) 997-94 49

www.etasinc.com

Japan

ETAS K.K.
Queen's Tower C-17F,
2-3-5, Minatomirai, Nishi-ku,
Yokohama, Kanagawa
220-6217 Japan

Tel.: +81 (45) 222-0900
Fax: +81 (45) 222-0956

www.etas.co.jp

France

ETAS S.A.S.
1, place des États-Unis
SILIC 307
94588 Rungis Cedex

Tel.: +33 (1) 56 70 00 50
Fax: +33 (1) 56 70 00 51

www.etas.fr

Korea

ETAS Korea Co. Ltd.
3F, Samseung Bldg. 61-1
Yangjae-dong, Seocho-gu
Seoul

Tel.: +82 (2) 57 47-016
Fax: +82 (2) 57 47-120

www.etas.co.kr

Great Britain

ETAS UK Ltd.
Studio 3, Waterside Court
Third Avenue, Centrum 100
Burton-upon-Trent
Staffordshire DE14 2WQ

Tel.: +44 (0) 1283 - 54 65 12
Fax: +44 (0) 1283 - 54 87 67

www.etas-uk.net

Copyright

Document TD00008-002

Contents

1 About this Manual

RTA-TRACE is a software logic analyzer for embedded systems. Coupled with a suitably enhanced application, it provides the embedded application developer with a unique set of services to assist in debugging and testing a system. Foremost amongst these is the ability to see exactly what is happening in a system at runtime with a production build of the application software.

This document provides describes the RTA-TRACE configuration options specific to RTA-OSEK and ERCOS^{EK}.

1.1 Who Should Read this Manual?

The RTA-TRACE Configuration Guide is aimed at the technical reader who wishes to use RTA-TRACE to examine an application under RTA-OSEK or ERCOS^{EK}. It should be read in conjunction with the *RTA-TRACE User Manual*, which explains the RTA-TRACE C API calls.

1.2 Conventions

Important: Notes that appear like this contain important information that you need to be aware of. Make sure that you read them carefully and that you follow any instructions that you are given.

Portability: Notes that appear like this describe things that you will need to know if you want to write code that will work on any target processor.

In this guide you'll see that program code, header file names, C type names, C functions and API call names all appear in the `courier` typeface. When the name of an object is made available to the programmer the name also appears in the `courier` typeface, so, for example, a task named Task1 appears as a task handle called `Task1`.

2 RTA-OSEK Configuration

Configuration of RTA-TRACE parameters for RTA-OSEK is carried out using the RTA-OSEK GUI. The GUI is largely self-explanatory, so this section will simply describe a set of tasks and how one might achieve them.

It is assumed that you have some knowledge of using the RTA-OSEK configuration tool, so creation/configuration of the application is not discussed here.

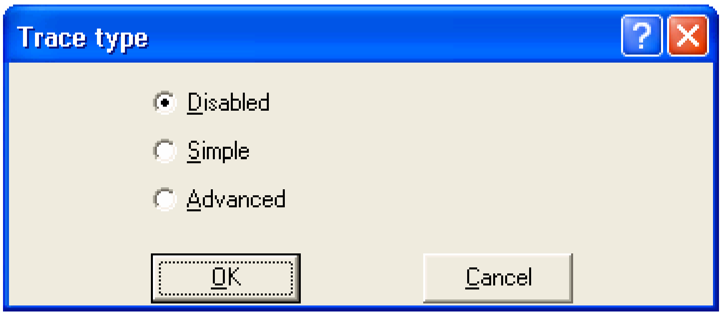
All of the configuration tasks relating to RTA-TRACE are accessed from the RTA-TRACE tab at the bottom-left hand side of the GUI:

2.1 General configuration

The following options can be set from this pane:

Trace Type Disables or enables (either simple or advanced) tracing.

Advanced tracing provides more detailed tracing at than simple tracing, with a corresponding increase in trace-records.



Compact IDs The compact trace format saves buffer space by only allowing 4 bits for task tracepoint ID values, and 8-bits for tracepoint and interval ID values. Other identifiers (Tasks, Resources etc.) use 8 bits.

If compact identifiers are not selected, 12 bits are used for tracepoint, task tracepoint, and interval ID values, and 16 bits are used for other identifiers.

Compact Time Select *compact* (16-bit) or *extended* (32-bit) time format. This option may not be available for every target.

Trace Stack Select whether to record stack usage or not

Target Triggering

Select whether runtime target triggering is available. If target triggering is not enabled then none of the `Trigger...()` API calls will function.

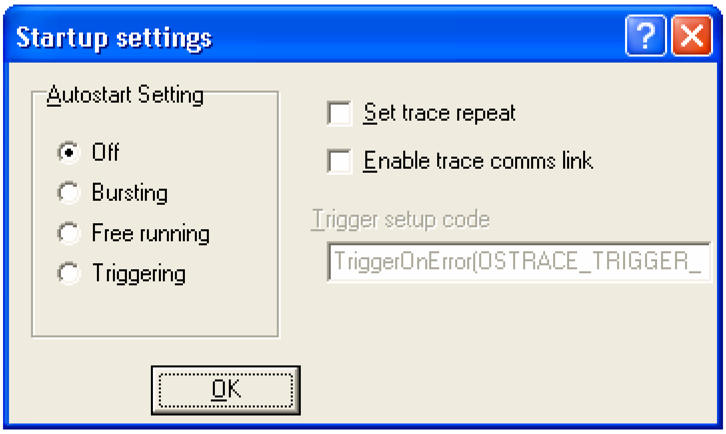
Buffer Size

This controls the size of the buffer reserved on the target for the tracing information. Note that the number is in records, not bytes, so the actual buffer size in bytes depends upon sizes selected for time and identifier.

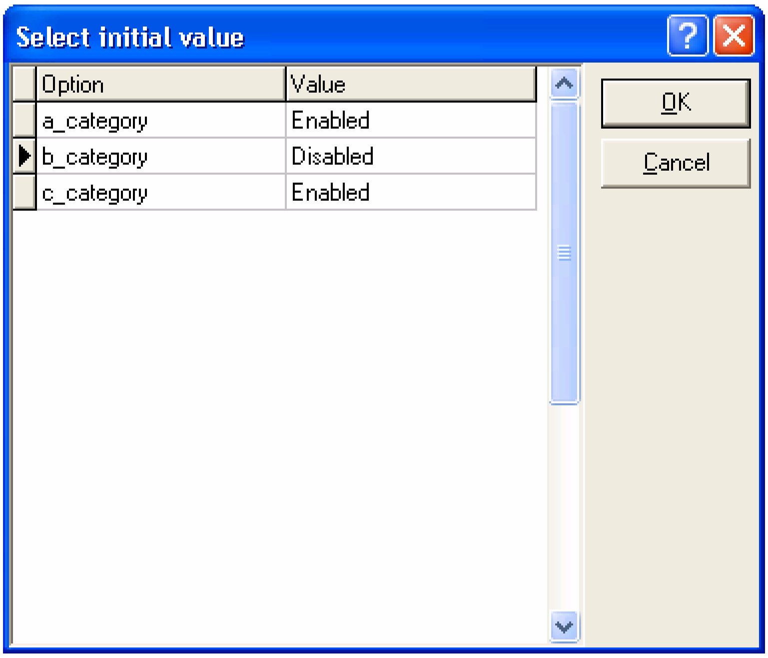
Autostart

Select whether tracing is started automatically, and which trace mode to start in.

For triggering operation, the trigger setup (`TriggerOn...`) code is entered here. Details of the triggering API can be found in the *RTA-TRACE User Manual*.

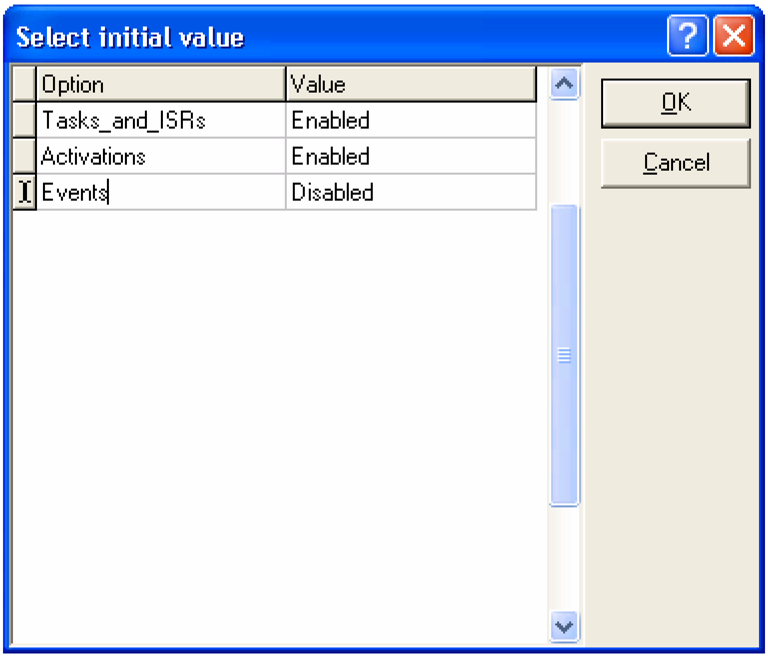


Initial Categories If run-time categories have been defined (See the *RTA-TRACE User Manual* for more details about categories), this dialog allows you to choose which run-time categories are enabled when tracing starts. Below, we can see three run-time categories, one of which is initially disabled.



Initial Classes

Choose which record classes are enabled when tracing starts. Below we can see that task and ISR, activation, and event tracing can be enabled and disabled at run-time and that event tracing is initially disabled.

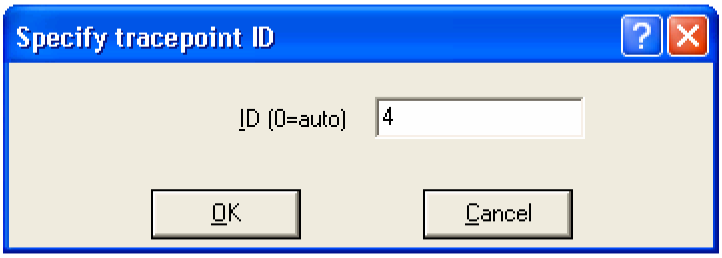


Stopwatch

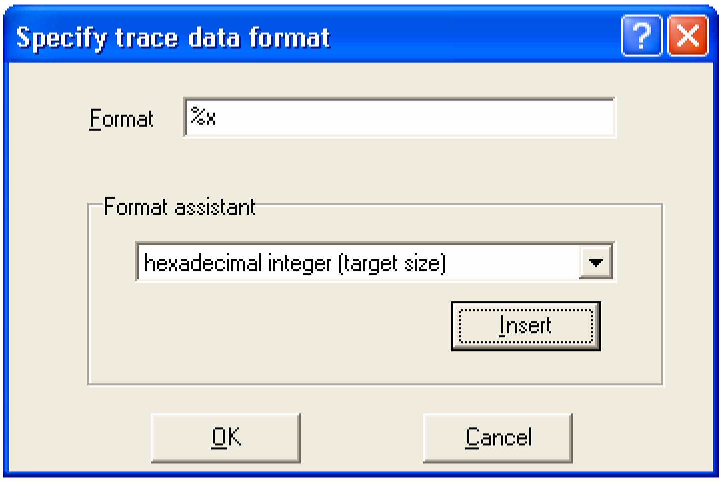
This dialog allows the user to specify which function is used to implement `GetStopwatch()`. In the dialog below, this is a user-supplied function called `now()`. The header file supporting this function is called `now.h`.

2.2 Tracepoints

This pane allows tracepoints to be defined. New tracepoints are initially given auto-generated identifiers, but this can be over-ridden using the ID button:



If the tracepoint has associated data, it is possible to supply a *format-string* (see section 4 for more information about format strings) to govern how the data will be displayed:



2.3 Task Tracepoints

This pane allows task-tracepoints to be defined. New task-tracepoints are initially given auto-generated identifiers, but this can be over-ridden using the ID button as for tracepoints.

Format strings are entered in the same way as for tracepoints.

2.4 Intervals

This pane allows intervals to be defined. New intervals are initially given auto-generated identifiers, but this can be over-ridden using the ID button as for tracepoints.

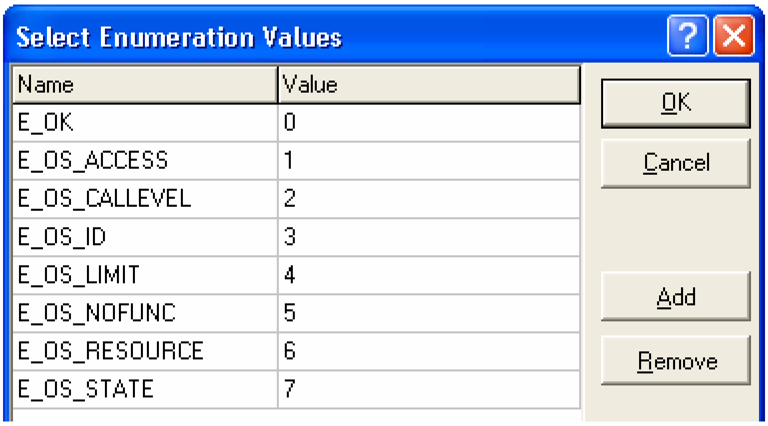
Format strings are entered in the same way as for tracepoints.

2.5 Categories

This pane allows trace categories to be defined, along with their mask-value. See the *RTA-TRACE User Manual* for more details about categories. Categories can be always enabled, always disabled, or enabled/disabled at run-time by using the *filter* pane (see 2.7 below).

2.6 Enumerations

This pane allows enumerated identifiers to be specified, along with numeric values. The example shown illustrates how the first few OSEK error codes might be enumerated.



2.7 Filters

This pane configures the filtering of event classes and categories. Events can be either always enabled; always disabled; or enabled/disabled at run-time. Run-time classes can be initially enabled or disabled based upon the 'Initial Classes' button on the configuration pane (See section 2.1).

2.8 osTraceStopwatch

RTA-TRACE requires a function to return the current system time. This function is required to have the following prototype:

`OS_NONREENTRANT(StopwatchTickType) osTraceStopwatch(void);`

When `GetStopwatch` is defined, it uses the same timer hardware as `osTraceStopwatch`. It is necessary on certain targets (where the timer cannot be read in a single instruction) to ensure that `GetStopwatch` reads the timer without interruption – this can be done using the `OS_ATOMIC()` macro as follows:

```
OS_NONREENTRANT(StopwatchTickType)
osTraceStopwatch(void)
{
        /* GET_TIMER_VAL() is a user-defined
         * macro that reads the appropriate
         * timer hardware */

        return GET_TIMER_VAL();
}

…

OS_NONREENTRANT(StopwatchTickType)
GetStopwatch(void)
{
        StopwatchTickType temp;

        /* GET_TIMER_VAL() is a user-defined
         * macro that reads the appropriate
         * timer hardware */

        OS_ATOMIC(temp = GET_TIMER_VAL());
        return temp;
}
```

Note: If `GetStopwatch()` has been defined in the stopwatch dialog (See Section2.1) then `osTraceStopwatch()` will be automatically defined.

3 ERCOS[EK] Configuration File

Configuration of RTA-TRACE for ERCOSEK is carried out using the file `RTAtrace.cfg` (located in the same directory as `project_settings.mk`), containing directives to control the tracing subsystem. A maximum of one directive can be present on each line. Comments are preceded with a '#' character. Anything after the first '#' character on a line is a comment.

No directive is mandatory.

3.1 Target Configuration

3.1.1 BUFFER_SIZE

Usage	`BUFFER_SIZE = ` *size*
Default	200
Description	This controls the size of the buffer reserved on the target for the tracing information. Note that the number is in records, not bytes, so the actual buffer size in bytes depends upon sizes selected for time and identifier (see 3.1.2 and3.1.3).
Legal values	32..65535

3.1.2 TIME_SIZE

Usage	`TIME_SIZE` = *size*
Default	32
Description	This setting defines the number of bits necessary to record a timestamp.
Legal values	16 or 32

3.1.3 COMPACT

Usage	`COMPACT = TRUE` or `FALSE`
Default	TRUE
Description	The `COMPACT` trace format saves buffer space by only allowing 4 bits for task tracepoint ID values, and 8-bits for tracepoint and interval ID values. Other identifiers (Tasks, Resources etc.) use 8 bits. If you need to use larger values for identifiers, you must set `COMPACT=FALSE`. In this case, 12 bits are used for tracepoint, task tracepoint, and interval ID values, and 16 bits are used for other identifiers.
Legal values	TRUE or FALSE

3.2 Tracing Configuration

These directives select objects or classes of object for tracing. Use them to achieve a suitable level of detail in the trace data for analysis, while omitting superfluous data.

Parameters are described using the following convention:

true-false	TRUE or FALSE
true-false-runtime	TRUE or FALSE or RUNTIME RUNTIME allows a class to be enabled or disabled within the code using the `EnableTraceClass()` and `DisableTraceClass()` API calls.
Maskbit	An integer power of 2, in other words a value whose representation in 32-bit binary has exactly one bit set.
Identifier	The C identifier that identifies a specific object, e.g. task.

Note: Classes marked as `RUNTIME` are initially disabled.

3.2.1 TASKS_AND_ISRS

Usage: `TASKS_AND_ISRS = ` *true-false-runtime*

Default: TRUE

Description: Enables logging of task and ISRs entering and leaving the running state, subject to the `EXCLUDE_TASK_OR_ISR` directive below.

3.2.2 EXCLUDE_TASK_OR_ISR

Usage: `EXCLUDE_TASK_OR_ISR = ` *identifier*

Default: This directive has no default.

Description: This directive prevents the logging of start, stop, and process information for task or ISR *name*.

Activation events for the excluded object will still be logged.

3.2.3 PROCESSES

Usage: `PROCESSES = ` *true-false-runtime*

Default: FALSE

Description: Enables logging of ERCOS^{EK} process objects. To fully log processes, `SWITCHING_OVERHEADS` (see 3.2.15) needs to be enabled as well; without `SWITCHING_OVERHEADS`, only process starts are logged.

3.2.4 STARTUP_AND_SHUTDOWN

Usage: `STARTUP_AND_SHUTDOWN = ` *true-false-runtime*

Default: `FALSE`

Description: Enables logging of `StartOS()` and `ShutdownOS()` in ERCOS^{EK}

3.2.5 ACTIVATIONS

Usage: `ACTIVATIONS = ` *true-false-runtime*

Default: `FALSE`

Description: Enables logging of task activation attempts, whether or not successful.

3.2.6 RESOURCES

Usage: `RESOURCES = ` *true-false-runtime*

Default: `FALSE`

Description: Enables logging of resource locking and unlocking.

3.2.7 INTERRUPT_LOCKS

Usage: `INTERRUPT_LOCKS = ` true-false-runtime

Default: `FALSE`

Description: Enables logging of interrupt enable/disable attempts made using the OSEK API.

3.2.8 ERRORS

Usage: `ERRORS = ` *true-false-runtime*

Default: `TRUE`

Description: Enables logging of operating-system error conditions

3.2.9 EXPLICIT_STATE_MESSAGES

Usage: `EXPLICIT_STATE_MESSAGES = ` *true-false-runtime*

Default: `FALSE`

Description: Enables logging of ERCOS^{EK} explicit state messages. See also `IMPLICIT_STATE_MESSAGES` below

3.2.10 IMPLICIT_STATE_MESSAGES

Usage: `IMPLICIT_STATE_MESSAGES = ` *true-false*

Default: `FALSE`

Description: Enables logging of ERCOS[EK] implicit state messages. See the *ESCAPE reference manual* for details of the ERCOS[EK] messaging system.

3.2.11 OSEK_MESSAGES

Usage: `OSEK_MESSAGES = ` *true-false-runtime*

Default: `FALSE`

Description: Enables logging of OSEK COM MESSAGE objects.

Note: Applies to ERCOS[EK] 4.3 and above.

3.2.12 MESSAGE_DATA

Usage: `MESSAGE_DATA = ` *true-false-runtime*

Default: `FALSE`

Description: Reports the data content of ERCOS[EK] state messages.

3.2.13 ALARMS

Usage: `ALARMS = ` *true-false-runtime*

Default: `FALSE`

Description: Enables logging of OSEK alarms.

3.2.14 TIMETABLES

Usage: `TIMETABLES = ` *true-false-runtime*

Default: `FALSE`

Description: Enables logging of ERCOS^{EK} timetables.

3.2.15 SWITCHING_OVERHEADS

Usage: `SWITCHING_OVERHEADS = ` *true-false-runtime*

Default: `FALSE`

Description: Increases the detail of task and process logging to include switching overheads; for example SystemISR (timer: may be more than one system ISR on some targets), preemptive scheduling overheads, and inter-process time.

> **Note:** Use of `EXCLUDE_TASK_OR_ISR` is not recommended in conjunction with this directive because not all overheads will be correctly recorded (e.g. task exit overheads are not recorded for excluded tasks).

3.2.16 TRACEPOINTS

Usage: `TRACEPOINTS = ` *true-false-runtime*

Default: `TRUE`

Description: Enables logging of tracepoints, recorded by calling `LogTracePoint…()` API functions.

3.2.17 TASK_TRACEPOINTS

Usage: `TASK_TRACEPOINTS = ` *true-false-runtime*

Default: `TRUE`

Description: Enables logging of task tracepoints, recorded by calling `LogTaskTracePoint…()` API functions.

3.2.18 INTERVALS

Usage: `INTERVALS = ` *true-false-runtime*

Default: `TRUE`

Description: Enables logging of intervals of elapsed time by calling `LogInterval…()` API functions.

3.2.19 STACK

Usage: `STACK = ` *true-false*

Default: `FALSE`

Description: Enables logging of stack usage within an application.

3.2.20 CATEGORY

Usage: `CATEGORY` *identifier* `=` *true-false*

or:

`CATEGORY` *identifier* `= RUNTIME MASK AUTO`

or:

`CATEGORY` *identifier* `= RUNTIME MASK` *maskbit*

Description: Defines a category. Category identifiers are made visible in the application. They are used in conjunction with the `Log…()` calls to enable or disable particular groups of user tracepoints.

Categories set to `RUNTIME` have their status held in a 4-byte bitmap: there can be up to 31 `RUNTIME` categories, but an unlimited number of configuration-time categories.

Designating a category as `'RUNTIME MASK AUTO'` causes the system to allocate a unique bit for that category's status.

If categories are referenced in pre-compiled libraries, then values must be explicitly given in *maskbit* since category values will be compiled into the library.

3.3 Tracing Display Control

These directives do not affect the target code. They influence the interpretation of the trace data before it is displayed in the visualizer.

Parameters are in the following convention:

id	The number passed to the API (e.g. in `LogTracePoint()`) to identify that tracing object.
identifier	The C identifier that identifies a specific object, e.g. task.
name	A name that will be associated in the visualizer display with the corresponding id for that object type. This should conform to C identifier syntax too, though it does not affect the target code.
string	A sequence of characters enclosed in double quotes (")
format-string	A string that describes the interpretation and representation of trace data. See Section 4.
index	For enum classes, the value within that class that corresponds to a name.

3.3.1 MESSAGE

Usage:	`MESSAGE` = *identifier* [`AS` *format-string*]
Description:	This directive governs how message *identifier* is displayed in the visualizer.

3.3.2 TRACEPOINT

Usage:	`TRACEPOINT` = *id* : *name* [`AS` *format-string*]
Description:	This directive assigns name *name* to tracepoint *id*. If the tracepoint has associated data, it is displayed according to *format-string*.

3.3.3 TASK_TRACEPOINT

Usage: `TASK_TRACEPOINT` = *id* [`,` *identifier*]`:` *name* [`AS` *format-string*]

Description: This directive assigns the name *name* to task tracepoint *id* and optionally limits the scope of the assignment to task tracepoints dropped by task *identifier*.

If the task tracepoint has associated data, it is displayed according to *format-string*.

3.3.4 INTERVAL

Usage: `INTERVAL` = *id* `:` *name* [`AS` *format-string*]

Description: This directive assigns the name *name* to the interval identified by *id*.

If the interval has associated data, it is displayed according to *format-string*.

3.3.5 COUNTER

Usage: `COUNTER` = *identifier* [`AS` *format-string*]

Description: This directive governs how a counter is displayed in the visualizer.

3.3.6 ENUM

Usage: `ENUM = ` *id* **:** *index* [`CALLED` *string*]

Description: This directive governs how an enumerated type is displayed in the visualizer.

Example there follows a C enum and a reflection of it in RTA-TRACE directives:

```
/* C fragment */

enum e_Rainbow {
    E_RED,
    E_ORANGE,
    E_YELLOW,
    E_GREEN,
    E_BLUE,
    E_INDIGO,
    E_VIOLET
};

/* End C fragment */
```

```
# RTAtrace.cfg fragment

enum = 1 : 0 as "Red";
enum = 1 : 1 as "Orange";
enum = 1 : 2 as "Yellow";
enum = 1 : 3 as "Green";
enum = 1 : 4 as "Blue";
enum = 1 : 5 as "Indigo";
enum = 1 : 6 as "Violet";

# End RTAtrace.cfg fragment
```

4 Format Strings

Format strings specify how a tracing item's data should be displayed. Simple numeric data can be displayed using a single format specifier. More complex data, e.g. a C `struct`, can be displayed by repeatedly moving a cursor around the data block and emitting data according to more complex format specifiers.

If a format string is not supplied, data is displayed in the following manner:

- if the data size is no greater than the size of the target's `int` type, data is decoded as if "`%d`" had been specified.
- Otherwise the data is displayed in a hex dump, e.g.

  ```
  0000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
  0010 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
  ```
- A maximum of 256 bytes is shown.

Note: when format specifiers are given, the target's endian-ness is taken into account. When a hex dump is shown, the target's memory is dumped byte-for-byte. In particular, you may not get the same output from a hex dump as from the `%x` format specifier.

4.1 Rules

Format strings are similar to the first parameter to the C function `printf()`:

- Format strings are surrounded by double-quote (`"`) symbols.
- A format string may contain two types of object: ordinary characters, which are copied to the output stream, and format elements, each of which causes conversion and printing of data supplied with the event.
- A format element comprises a percent sign, zero or more digits and a single non-digit character, with the exception of the `%E` element – see below.
- The format element is decoded according to the rules in the table below, and the resulting text is added to the output string.
- The special format element `%%` emits a `%`.
- In addition to ordinary characters and conversion specifications, certain characters may be emitted by using a 'backslash-escape-sequence'. To emit a double-quote (`"`) character, `\"` is used, and to emit a `\` character, `\\` is used.
- The optional size parameter to integer format specifiers defines the field's width in bytes. Valid values are 1, 2, 4 or 8.

Note: An important difference from `printf()` is that the cursor does not automatically move on from the current field when a field is emitted. This is to facilitate multi-format output of a single field.

Format Element	Meaning
`%offset@`	Moves the cursor *offset* bytes into the data. This can be used to extract values from multiple fields in a structure.
`%[size]d`	Interpret the current item as a signed integer. Output the value as signed decimal.
`%[size]u`	Interpret the current item as an unsigned integer. Output the value as unsigned decimal.
`%[size]x`	Interpret the current item as unsigned integer. Output the value as unsigned hexadecimal.
`%[size]b`	Interpret the current item as an unsigned integer. Output the value as unsigned binary.
`%enum[:size]E`	Interpret the current item as an index into the enumeration class who's ID is *enum*. Emit the text in that enumeration class that corresponds with the item's value. The enumeration class should be defined using `ENUM` directives. An exception is implicitly defined enum classes 98 and 99, which are startup and error codes respectively.
`%F`	Treat the current item as an IEEE 'double'. Output the value as a double, in exponent format if necessary.
`%?`	Emit in the form of a hex dump.
`%%`	No conversion is carried out; emit a `%`

4.2 Examples

Description	Format String	Example	Notes
A native integer displayed in decimal and hexadecimal.	`"%d 0x%x"`	`10 0xA`	The `"0x"` is not emitted by the `%x` format specifier but is specified in literal characters in the string. Absence of size specifier means the target's `int` size is assumed.
A single unsigned byte representing a percentage.	`"%1u%%"`	`73%`	Use of size specifier of 1 byte. Use of `%%` to emit `%`.

`struct{`
`int x;`
`int y;`
`};`
… On a 32-bit processor.
	`"(%d,%4@%d)"`	`(20,-15)`	Use of `%offset@` to move to byte-offset within the structure.

A value of type `enum e_Rainbow` (see Section 3.3.6), using the enum class shown in that section.	`"%1E"`	`Yellow`	The number `1` refers the ID of the enum class in the `ENUM` directives, not to the width of the field.

Index

R

S

T

Support

For product support, please contact your local ETAS representative.

Office locations and contact details can be found on the ETAS Group website www.etasgroup.com.