RTA-TRACE

User Manual

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

Contents

1 About this Manual

RTA-TRACE is a software logic analyzer for embedded systems. Coupled with a suitably-enhanced application, it provides the 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 manual describes the target API for RTA-TRACE.

1.1 Who Should Read this Manual?

It is assumed that you are a developer. You should read this guide if you want to find out about the target API for RTA-TRACE. This document also contains a top-level overview of the RTA-TRACE architecture for reference.

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.

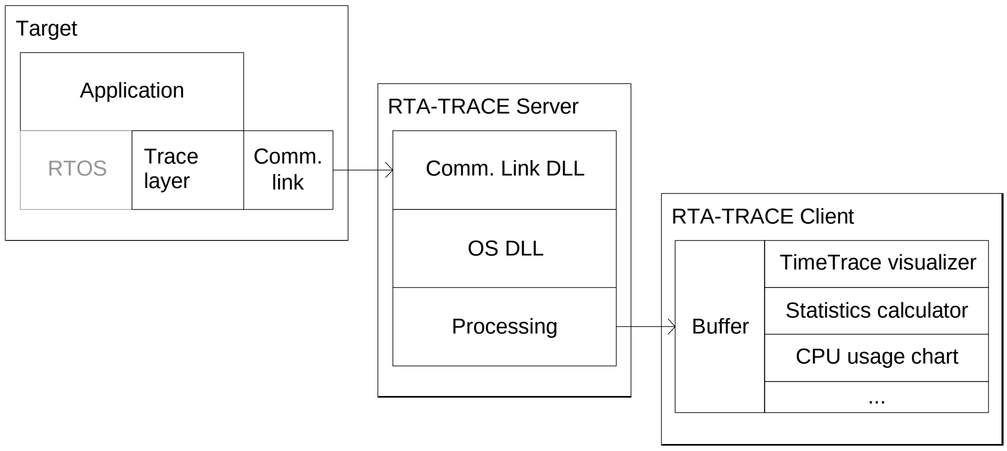
In this guide you'll see that program code, header file names, C type names, C functions and API call names all appear in the `Courier` typeface. When the name of an object is made available to the programmer the name also appears in the `Courier` typeface, so, for example, a task named Task1 appears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

2 RTA-TRACE Overview

RTA-TRACE is a software logic analyzer for embedded systems. Coupled with a suitably-enhanced application, it provides the 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.

The RTA-TRACE product consists of three layers – target, RTA-TRACE Server, and RTA-TRACE Client (each described in more detail below).

End-to-end, RTA-TRACE looks like this:



2.1 Target

The target layer consists of an application, a supported OSEK OS (optional if using the RTA-TRACE Instrumenting Kit), and the RTA-TRACE target software. The RTA-TRACE target software consists of a library and header files allowing OS calls to be intercepted, as well as providing an API for the logging of user-generated tracing information (using Tracepoints, Task-Tracepoints, and intervals – see individual sections in this document).

During the system build process, it is possible to enable and disable classes of log-data (both OS and user trace information); the mechanism for this is described in the *RTA-TRACE Configuration Reference Manual*.

Trace data generated by the target is transferred via an ECU Link (described in the *RTA-TRACE ECU Link Guide*) to the RTA-TRACE Server.

2.2 RTA-TRACE Server

The RTA-TRACE Server is responsible for collecting trace data generated by the target (a single Server is capable of connecting to multiple targets simultaneously with appropriate licensing) and processing the trace data into a form suitable for the RTA-Trace client. Appropriate DLLs for the ECU Link and OS in use are used to convert/unpack data into a form suitable for display.

Although the Server application is separate from the Client, it will most often be executed automatically when the Client is started; the only evidence that the Server is running is the icon in the system tray of the host PC. See the *RTA-TRACE ECU Link Guide* to determine how to modify parameters of the ECU Link (if necessary or possible).

The RTA-TRACE Server can support connections from multiple RTA-TRACE Clients (as well as target applications) – either locally (on the same PC) or remotely (across a network). These features are subject to licensing.

2.3 RTA-TRACE Client

The RTA-TRACE Client is the most visible portion of RTA-TRACE since it is responsible for displaying the recorded data, calculating statistics etc.. The various visualization options are supplied as *plug-ins*, which may be subject to additional licensing.

Processing is carried out which converts the raw trace data generated by the target into a more readable form, as described by the user during the configuration process. See the *RTA-TRACE Configuration Reference Manual* for details about how this is done.

Examples of plug-ins are the *TimeTrace visualizer* (allowing the task activity of an application to be examined), and the CPU usage pie and bar graphs (allowing a view of the CPU usage by system element).

Plug-in features are more completely described by online help files – available from the 'Help' menu of the RTA-TRACE Client.

3 API Functions

3.1 Trace control

The following API calls control the mode in which RTA-TRACE operates.

3.1.1 StartFreeRunningTrace

Syntax:	`void StartFreeRunningTrace(void)`
Parameters:	None
Description:	Starts tracing in *Free Running* mode.
In Particular:	*Free running* trace mode captures event information continuously. Data is uploaded to the host as soon as it is available, concurrently with capture. If the trace buffer becomes full, logging of trace data is suspended until the buffer has been emptied. When the buffer is available again, tracing resumes. This situation might occur if the ECU Link is too slow for the desired volume of trace data. If this call is made whilst tracing, the trace buffer is cleared and tracing begins again.
Applies to:	All OS.

3.1.2 StartBurstingTrace

Syntax:	`void StartBurstingTrace(void)`
Parameters:	None
Description:	Starts tracing in *Bursting* mode.
In Particular:	*Bursting* trace mode captures event information into the trace buffer in a 'one-shot' manner. When the buffer is full, tracing stops and data transfer begins. No attempt is made to upload data to the Server until the trace buffer has filled. If `SetTraceRepeat()` (see 3.1.4) has been used to enable repeated bursting traces, tracing resumes when data transfer is complete. If this call is made whilst tracing, the trace buffer is cleared and tracing begins again.
Applies to:	All OS.

3.1.3 StartTriggeringTrace

Syntax:	`void StartTriggeringTrace(void)`
Parameters:	None
Description:	Starts tracing in *Triggering* mode.
In Particular:	*Triggering* trace mode waits for specific events before transferring trace data to the host. Trigger events are set using the `TriggerOn…()` API calls (see 3.7.4 onward). A pre- and post-trigger buffer size can be specified using `SetTriggerWindow()` (see 3.7.1) so that only the set of events before and after the trigger event can be seen. `SetTraceRepeat()` (see 3.1.4) can be used to enable repeated triggered traces. In this case, when data-transfer has completed, tracing will resume the next time the trigger event occurs. If this call is made whilst tracing, the trace buffer is cleared and tracing begins again.
Applies to:	All OS.

3.1.4 SetTraceRepeat

Syntax:	`void SetTraceRepeat(osTraceBool <mode>)`
Parameters:	
`mode`	Non-zero value enables repeat mode, zero disables repeat mode.
Description:	Enables/disables repeat feature of *Bursting* and *Triggering* trace modes.
In Particular:	See 3.1.2 and 3.1.3.
Applies to:	All OS.

3.1.5 StopTrace

Syntax:	`void StopTrace(void)`
Parameters:	None
Description:	Stops recording of trace data to the trace buffer.
In Particular:	This does not abort the data link: any data remaining in the trace buffer is uploaded to the host.
Applies to:	All OS.

3.2 General

RTA-TRACE allows three different visualization aids to be defined to aid in program debugging – these are *Tracepoints*, *Task-Tracepoints*, and *Intervals*:

- *Tracepoints* (Section 3.3) are displayed on the TimeTrace visualizer on individual tapes. They can be logged from any point in a program – perhaps to indicate some global program state;
- *Task-Tracepoints* (Section 3.4) are displayed on the tape of the task which logs them – so these are ideal for showing the activity of a particular task;
- *Intervals* (Section 3.5) can be logged from any place in a program – they have both a start and an end, and might be used to show a time-related feature of a program (i.e. stimulus -> response).

Logging any of the above markers can also have some data attached (which will be displayed in the *TimeTrace visualizer*) either as a single 16- or 32-bit value (using the …`Value()` version of the `Log…()` calls) or a larger data-block (using the …`Data()` version of the `Log…()` calls). The attached data/value will be displayed according to the format-string supplied at configuration time (See the *RTA-TRACE Configuration Guide* for details).

Each marker described above has an identifier, and can belong to one or more categories. These terms are described below.

3.2.1 Identifers

The identifiers (IDs) used for *Tracepoints*, *Task-Tracepoints*, and *Intervals* have a limited range. Two ranges of identifier exist for each of these (selected at configuration-time): *compact* and *extended* IDs. The ranges for each of these are shown below:

ID	Compact[1]	Extended (default)[1]
`Tracepoint`	8-bit (1..255)	13-bit (1..8191)
`TaskTracePoint`	4-bit (1..15)	13-bit (1..8191)
`Interval`	8-bit (1..255)	13-bit (1..8191)

Identifiers can be generated either in the configuration process (recommended for ERCOS^{EK} and RTA-OSEK) or literals can be used. For ERCOS^{EK} and RTA-OSEK, generated identifiers are `#defined` in the `rtatrace.h` header file. Use of identifiers generated in the configuration process means that visualization of the trace data will be more readily understood (for example, the *TimeTrace Visualizer* will display "`Tracepoint <name>`" instead of "`Tracepoint 5`").

If a *Tracepoint* has been configured with a name and number, subsequent use of the literal number as an identifier will cause the name to be seen in the RTA-TRACE Client – it is therefore recommended that named identifiers be used.

When using the OS Instrumenting kit this mechanism is still available, except that the description file (with a `.rta` extension) may need to be hand-generated.

Note: Using a `Log...()` call with an identifier that is out of range, will silently fail.

3.2.2 Categories

Every user *(Task)Tracepoint* belongs to one or more user-defined categories. This allows sets of *(Task)Tracepoints* to be to enabled or disabled either at runtime or at configuration time. There are 31 runtime categories available, and an unlimited number of configuration-time categories.

[1] Although it is possible to generate an ID of 0, it is not possible to trigger on an interval/tracepoint/task-tracepoint with an ID of 0.

Categories are specified as bitmasks during the configuration process[2] hence it is possible to combine multiple trace categories using the bitwise OR operator (‘|’).

Note: In a `Log...()` call, a *Tracepoint* is logged if **any** of the categories are active (i.e. for a particular *Tracepoint*, if a runtime category is combined with a category which has been enabled at configuration time, disabling the runtime category will have no effect – the *Tracepoint* will still be logged).

Note: If all of the categories attached to a particular `Log...()` call are marked as `FALSE` at configuration-time, the `Log...()` call will not be inserted into the code stream.

Note: Categories must only be enabled or disabled (using `EnableTraceCategories` and `DisableTraceCategories` respectively) when tracing is stopped. If this is not done, the RTA-TRACE visualizer may display incorrect and/or misleading data.

3.2.3 Classes

Logged events belong to predefined classes, allowing the user to enable or disable classes of events either at runtime or configuration-time. A list of classes is given in the following table.

Since classes are bitmasks, it is possible to enable or disable a set of classes by using the bitwise OR operator (|).

[2] See the RTA-TRACE configuration reference for your particular operating system for details of the configuration process.

osTraceClassesType	**Applies to:**
`TRACE_ACTIVATIONS_CLASS`	All OS
`TRACE_OSEK_MESSAGES_CLASS`	All OS
`TRACE_RESOURCES_CLASS`	All OS
`TRACE_INTERRUPT_LOCKS_CLASS`	All OS
`TRACE_SWITCHING_OVERHEADS_CLASS`	All OS
`TRACE_TASKS_AND_ISRS_CLASS`	All OS
`TRACE_PROCESSES_CLASS`	ERCOS^{EK}/Custom
`TRACE_EXPLICIT_STATE_MESSAGES_CLASS`	ERCOS^{EK}
`TRACE_ERRORS_CLASS`	All OS
`TRACE_TASK_TRACEPOINT_CLASS`	All OS
`TRACE_TRACEPOINT_CLASS`	All OS
`TRACE_INTERVALS_CLASS`	All OS
`TRACE_MESSAGE_DATA_CLASS`	All OS
`TRACE_STARTUP_AND_SHUTDOWN_CLASS`	All OS
`TRACE_ALARMS_CLASS`	All OS
`TRACE_TIMETABLES_CLASS`	ERCOS^{EK}
`TRACE_SCHEDULES_CLASS`	RTA-OSEK
`TRACE_OSEK_EVENTS_CLASS`	RTA-OSEK/Custom
`TRACE_NO_CLASSES`	All OS
`TRACE_ALL_CLASSES`	All OS

Note: Classes must only be enabled or disabled (using `EnableTraceClasses` and `DisableTraceClasses` respectively) when tracing is stopped. If this is not done, the RTA-TRACE visualizer may display incorrect and/or misleading data.

3.2.4 EnableTraceClasses

Syntax:

```
void EnableTraceClasses(
            osTraceClassesType <mask>)
```

Parameters:

`mask`	Mask of one or more event classes to enable.

Description: This call enables one or more classes of trace events at run time.

The dual to 3.2.5.

In Particular: Classes not listed in the call will be left in their current state.

Applies to: All OS.

3.2.5 DisableTraceClasses

Syntax:

```
void DisableTraceClasses(
            osTraceClassesType <mask>)
```

Parameters:

`mask`	Mask of one or more event families to disable.

Description: This call disables one or more families of trace events at run time.

The dual to 3.2.4.

In Particular: Classes not listed in the call will be left in their current state.

Applies to: All OS.

3.2.6 EnableTraceCategories

Syntax:

```
void EnableTraceCategories(
     osTraceCategoriesType <mask>)
```

Parameters:

`mask` An event-category mask.

Description: Causes a user-specified subset of user Tracepoints to be enabled.

The dual to 3.2.7.

In Particular: Every user *Tracepoint* belongs to one or more categories.

Categories not listed in the call will be left in their current state.

Applies to: All OS.

3.2.7 DisableTraceCategories

Syntax:

```
void DisableTraceCategories(
     osTraceCategoriesType <mask>)
```

Parameters:

`mask` An event-category mask.

Description: Causes a user-specified subset of user Tracepoints to be disabled.

The dual to 3.2.6.

In Particular: Every user *Tracepoint* belongs to one or more categories.

Categories not listed in the call will be left in their current state.

Applies to: All OS.

3.3 Tracepoints

Tracepoints are displayed in the *TimeTrace visualizer* on their own tapes. Tracepoint identifiers can be named during the configuration process (and *TimeTrace* will display the name) or literal numerical values can be used instead.

3.3.1 LogTracepoint

Syntax:

```
void LogTracepoint(
     <TpointId>,
     osTraceCategoriesType <catmask>)
```

Parameters:

`TpointId`	Identifies a *Tracepoint* (size varies – see 3.2.1)
`catmask`	The category or categories that this *Tracepoint* belongs to.

Description: Logs a *Tracepoint* against its designated task.

In Particular: Logging is subject to the `TRACE_TRACEPOINT_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.3.2 LogTracepointValue

Syntax:

```
void LogTracepointValue(
      <TpointId>,
      uint <val>,
      osTraceCategoriesType <catmask>)
```

Parameters:

`TpointId`	Identifies a *Tracepoint* (size varies – see 3.2.1)
`val`	Numerical value to be sent with the *Tracepoint*. The size of val (16 or 32 bit) depends upon the size of the system time selected during configuration – see the *RTA-TRACE Configuration Reference Manual* for further information.
`catmask`	The category or categories that this *Tracepoint* belongs to.

Description: Logs a *Tracepoint* against its designated task, along with an unsigned 16 or 32 bit integer value.

In Particular: Logging is subject to the `TRACE_TRACEPOINT_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.3.3 LogTracepointData

Syntax:

```
void LogTracepointData(
      <TpointID>,
      PTR(ByteType) <dataPtr>,
      osUIntType <length>,
      osTraceCategoriesType <catmask>)
```

Parameters:

`TpointID`	Identifies a *Tracepoint* (size varies – see 3.2.1)
`dataPtr`	Points to a data block to be sent with the *Tracepoint*.
`length`	Length of the data block pointed to by `<dataPtr>` in bytes.
`catmask`	The category or categories that this call belongs to.

Description: Logs a *Tracepoint* against its designated task, along with arbitrary binary data.

In Particular: Logging is subject to the `TRACE_TRACEPOINT_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.4 Task-Tracepoints

Task-Tracepoints are displayed in the *TimeTrace visualizer* on the tape of the task that issued the `LogTaskTracepoint…()` call. Task-Tracepoint identifiers can be named during the configuration process (and *TimeTrace* will display the name) or literal numerical values can be used instead.

3.4.1 LogTaskTracepoint

Syntax:

```
void LogTaskTracepoint(
    <TTpointId>,
    osTraceCategoriesType <catmask>)
```

Parameters:

`TTpointId`	Identifies a *Task-Tracepoint* (size varies – see 3.2.1)
`catmask`	The category or categories that this *Task-Tracepoint* belongs to.

Description: Logs *Task-Tracepoint* `<TTPointID>` against its designated task.

In Particular: Logging is subject to the `TRACE_TASK_TRACEPOINT_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.4.2 LogTaskTracepointValue

Syntax:

```
void LogTaskTracepointValue(
    <TTpointId>,
    uint <val>,
    osTraceCategoriesType <catmask>)
```

Parameters:

`TTpointId`	Identifies a Task-Tracepoint (size varies – see 3.2.1)
`val`	Numerical value to be sent with the *Task-Tracepoint*. The size of val (16 or 32 bit) depends upon the size of the system time selected during configuration – see the *RTA-TRACE Configuration Reference Manual* for further information.
`catmask`	The category or categories that this *Task-Tracepoint* belongs to.

Description: Logs *Task-Tracepoint* `<TTPointID>` against its designated task, along with an unsigned 16 or 32 bit integer value.

In Particular: Logging is subject to the `TRACE_TASK_TRACEPOINT_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.4.3 LogTaskTracepointData

Syntax:

```
void LogTaskTracepointData(
    <TTPointID>,
    PTR(ByteType) <dataPtr>,
    osUIntType <length>,
    osTraceCategoriesType <catmask>)
```

Parameters:

`TTPointID`	Identifies a *Task-Tracepoint* (size varies – see 3.2.1)
`dataPtr`	Points to a data block to be sent with the *Task-Tracepoint*.
`length`	Length of the data block pointed to by `<dataPtr>` in bytes.
`catmask`	The category or categories that this *Tracepoint* belongs to.

Description: Logs *Task-Tracepoint* `<TTPointID>` against its designated task, along with arbitrary binary data.

In Particular: Logging is subject to the `TRACE_TASK_TRACEPOINT_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.5 Intervals

Intervals provide a mechanism for users to measure elapsed time between two events. Interval indications appear on their own tapes in the *TimeTrace visualizer*. Interval identifiers can be named during the configuration process (and *TimeTrace* will display the name) or literal numerical values can be used instead.

3.5.1 LogIntervalStart

Syntax:

```
void LogIntervalStart(
     <intervalId>,
     osTraceCategoriesType <catmask>)
```

Parameters:

`intervalId`	An *Interval* identifier (size varies – see 3.2.1)
`catmask`	The category or categories that this call belongs to.

Description: This call starts the *interval* referenced by `<intervalId>`.

In Particular: Logging is subject to the `TRACE_INTERVALS_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.5.2 LogIntervalStartValue

Syntax:

```
void LogIntervalStartValue(
     <intervalId>,
     uint <val>,
     osTraceCategoriesType <catmask>)
```

Parameters:

`intervalId`	An *Interval* identifier (size varies – see 3.2.1)
`val`	Numerical value to be sent with the *Interval* start. The size of val (16 or 32 bit) depends upon the size of the system time selected during configuration – see the *RTA-TRACE Configuration Reference Manual* for further information.
`catmask`	The category or categories that this call belongs to.

Description: This call starts the *interval* referenced by `<intervalId>`, along with an unsigned 16 or 32 bit integer value.

In Particular: Logging is subject to the `TRACE_INTERVALS_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.5.3 LogIntervalStartData

Syntax:

```
void LogIntervalStartData(
     <intervalId>,
     PTR(ByteType) <dataPtr>,
     osUIntType <length>,
     osTraceCategoriesType <catmask>)
```

Parameters:

`intervalId`	An *Interval* identifier (size varies – see 3.2.1)
`dataPtr`	Points to a data block to be sent with the *interval* start.
`length`	Length of the data block pointed to by `<dataPtr>` in bytes.
`catmask`	The category or categories that this call belongs to.

Description: This call starts the *interval* referenced by `<intervalId>`, along with arbitrary binary data.

In Particular: Logging is subject to the `TRACE_INTERVALS_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.5.4 LogIntervalEnd

Syntax:

```
void LogIntervalEnd(
     <intervalId>,
     osTraceCategoriesType <catmask>)
```

Parameters:

`intervalId`	An *Interval* identifier (size varies – see 3.2.1)
`catmask`	The category or categories that this call belongs to.

Description: This call ends the *interval* referenced by `<intervalId>`.

In Particular: Logging is subject to the `TRACE_INTERVALS_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.5.5 LogIntervalEndValue

Syntax:	`void LogIntervalEndValue(`
	`     <intervalId>,`
	`     `*`uint`*` <val>,`
	`     osTraceCategoriesType <catmask>)`

Parameters:	
`intervalId`	An *Interval* identifier (size varies – see 3.2.1)
`val`	Numerical value to be sent with the *Interval* end. The size of val (16 or 32 bit) depends upon the size of the system time selected during configuration – see the *RTA-TRACE Configuration Reference Manual* for further information.
`catmask`	The category or categories that this call belongs to.
Description:	This call ends the *interval* referenced by `<intervalId>`, along with an unsigned 16 or 32 bit integer value.
In Particular:	Logging is subject to the `TRACE_INTERVALS_CLASS` class and one or more of the categories in `<catmask>` being enabled.
Applies to:	All OS.

3.5.6 LogIntervalEndData

Syntax:

```
void LogIntervalEndData(
     <intervalId>,
     PTR(ByteType) <dataPtr>,
     osUIntType <length>,
     osTraceCategoriesType <catmask>)
```

Parameters:

`intervalId`	An *Interval* identifier (size varies – see 3.2.1)
`dataPtr`	Address of a data block to be sent with the *interval* end.
`length`	Length of the data block pointed to by `<dataPtr>` in bytes.
`catmask`	The category or categories that this call belongs to.

Description: This call ends the *interval* referenced by `<intervalId>`, along with arbitrary binary data

In Particular: Logging is subject to the `TRACE_INTERVALS_CLASS` class and one or more of the categories in `<catmask>` being enabled.

Applies to: All OS.

3.6 Miscellaneous Logging

3.6.1 LogProfileStart

Syntax:

```
void LogProfileStart(
     osTraceInfoType <profileId>)
```

Parameters:

`profileId`	A *Profile* identifier.

Description: This call marks the switch over to a new runtime profile.

In Particular: Logging is subject to the `TRACE_TASKS_AND_ISRS_CLASS` class.

Applies to: RTA-OSEK, Generic OS

3.6.2 LogCriticalExecutionEnd

Syntax:

```
void LogCriticalExecutionEnd(
     osTraceInfoType <critExecId>)
```

Parameters:

CritExecId	A Critical Execution Point identifier.

Description: This call marks the end of a critical execution section. This will most often be used to observe response time to some stimulus.

In Particular: Logging is subject to the `TRACE_TASKS_AND_ISRS_CLASS` class.

Applies to: RTA-OSEK, Generic OS

3.6.3 LogCat1ISRStart

Syntax:

```
void LogCat1ISRStart(
     osTraceInfoType <IsrId>)
```

Parameters:

IsrId	A Category 1 ISR identifier.

Description: This call marks the start of a category 1 ISR. This type of ISR is not controlled by the operating system, and so requires manual logging.

In Particular: Logging is subject to the `TRACE_TASKS_AND_ISRS_CLASS` class.

Applies to: RTA-OSEK

3.6.4 LogCat1ISREnd

Syntax:	`void LogCat1ISREnd( osTraceInfoType <IsrId>)`
Parameters:	
IsrId	A Category 1 ISR identifier.
Description:	This call marks the end of a category 1 ISR. This type of ISR is not controlled by the operating system, and so requires manual logging.
In Particular:	Logging is subject to the `TRACE_TASKS_AND_ISRS_CLASS` class.
Applies to:	RTA-OSEK

3.6.5 LogOverrunHook

Syntax:	`void LogOverrunHook()`
Parameters:	None
Description:	This call marks that the overrun-hook was called. This is called if a task has run for too long at its termination.
In Particular:	Logging is subject to the `TRACE_ERRORS_CLASS` class.
Applies to:	RTA-OSEK

3.7 Triggering

3.7.1 ClearTrigger

Syntax:	`void ClearTrigger(void)`
Parameters:	None
Description:	Clears the trigger condition so that no trace record can cause triggering.
In Particular:	If the trigger condition has occurred, events will continue to be logged.
Applies to:	All OS.

3.7.2 TriggerNow

Syntax:	`void TriggerNow(void)`
Parameters:	None
Description:	Trigger now regardless of trigger conditions.
In Particular:	If trigger conditions have been set, this call will not clear them.
Applies to:	All OS.

3.7.3 SetTriggerWindow

Syntax:

```
void SetTriggerWindow(
     UIntType <before>,
     UIntType <after>)
```

Parameters:

`Before`	Number of records to be uploaded from before the trigger event.
`After`	Number of records to be uploaded after the trigger event.

Description: Sets the number of records pre- and post-trigger that will be uploaded.

In Particular: The total number of records uploaded (`<before> + <after>`) must be less than the total buffer size available.

If the total is greater than the buffer size available, `<after>` is truncated so that `<before>` will fit – if `<before>` is greater than the available buffer, then `<before>` is set to the available buffer size, and `<after>` is set to zero.

Note: since each trace *event* may be carried in multiple trace *records*, the number of events before and after the trigger point will not match the number of records specified in this call. Truncation of trace events may occur (for example with `Log…Data()` calls), causing incomplete data or 'missed' events.

Small values of after should be avoided - consider using `StopTrace()` when triggering on terminal events such as system shutdown.

Applies to: All OS.

3.7.4 TriggerOnActivation

Syntax:

```
void TriggerOnActivation(
                         TaskType <taskId>)
```

Parameters:

`taskId` Identifier of the task to trigger on.

Description: Sets the trigger to be an attempt to activate `<taskId>` (i.e. an `ActivateTask(<TaskId>)` call has been made).

In Particular: If `<taskId>` is set to `OSTRACE_TRIGGER_ANY`, any call to `ActivateTask()` will trigger tracing.

The trigger will occur on explicit task activation, and on task activation via a timetable. It will occur on task activation via an alarm unless logging of alarms is enabled. It will not occur as a result of `ChainTask()`. (See `TriggerOnChain()`), or activation of a taskset containing task `<taskId>` (applies to RTA-OSEK only).

Applies to: All OS.

3.7.5 TriggerOnChain

Syntax: `void TriggerOnChain(TaskType <taskId>)`

Parameters:

`taskId` Identifier of the task to trigger on.

Description: Sets the trigger to be an attempt to chain task `<taskId>` (i.e. a `ChainTask(<TaskId>)` call has been made).

In Particular: If `<taskId>` is set to `OSTRACE_TRIGGER_ANY`, any call to `ChainTask()` will trigger tracing.

Applies to: RTA-OSEK, ERCOS^{EK}

3.7.6 TriggerOnTaskStart

Syntax: `void TriggerOnTaskStart(TaskType <taskId>)`

Parameters:

`taskId` Identifier of the task to trigger on.

Description: Sets the trigger to be the start of `<taskId>`.

In Particular: If `<taskId>` is set to `OSTRACE_TRIGGER_ANY`, any task start will trigger tracing.

Applies to: All OS.

3.7.7 TriggerOnTaskStop

Syntax: `void TriggerOnTaskStop(TaskType <taskId>)`

Parameters:

`taskId` Identifier of the task to trigger on.

Description: Sets the trigger to be the end of `<taskId>`.

In Particular: If `<taskId>` is set to `OSTRACE_TRIGGER_ANY`, any task stop will trigger tracing.

Applies to: All OS.

3.7.8 TriggerOnISRStart

Syntax:	`void TriggerOnISRStart(ISRType <ISRId>)`
Parameters:	
`ISRId`	The identifier of the ISR.
Description:	Sets the trigger to be the start of `<ISRId>`.
In Particular:	If `<ISRId>` is set to `OSTRACE_TRIGGER_ANY`, any ISR start will trigger tracing. `<ISRId>` must be the identifier used during system generation.
Applies to:	ERCOS^{EK}.

3.7.9 TriggerOnISRStop

Syntax:	`void TriggerOnISRStop(ISRType <ISRId>)`
Parameters:	
`ISRId`	The identifier of the ISR.
Description:	Sets the trigger to be the end of `<ISRId>`.
In Particular:	If `<ISRId>` is set to `OSTRACE_TRIGGER_ANY`, any ISR stop will trigger tracing. `<ISRId>` must be the identifier used during system generation.
Applies to:	ERCOS^{EK}.

3.7.10 TriggerOnCat1ISRStart

Syntax:

```
void TriggerOnCat1ISRStart(
                    ISRType <ISRId>)
```

Parameters:

`ISRId`	The identifier of the ISR.

Description: Sets the trigger to be the start of `<ISRId>`.

In Particular: If `<ISRId>` is set to `OSTRACE_TRIGGER_ANY`, the start of any category 1 ISR will trigger tracing.

`<ISRId>` must be the identifier used during system generation.

Applies to: RTA-OSEK.

3.7.11 TriggerOnCat1ISRStop

Syntax:

```
void TriggerOnCat1ISRStop(
                    ISRType <ISRId>)
```

Parameters:

`ISRId`	The identifier of the ISR.

Description: Sets the trigger to be the end of `<ISRId>`.

In Particular: If `<ISRId>` is set to `OSTRACE_TRIGGER_ANY`, the stop of any category 1 ISR will trigger tracing.

`<ISRId>` must be the identifier used during system generation.

Applies to: RTA-OSEK.

3.7.12 TriggerOnCat2ISRStart

Syntax:

```
void TriggerOnCat2ISRStart(
                    ISRType <ISRId>)
```

Parameters:

ISRId	The identifier of the ISR.

Description: Sets the trigger to be the start of `<ISRId>`.

In Particular: If `<ISRId>` is set to `OSTRACE_TRIGGER_ANY`, the start of any category 2 ISR will trigger tracing.

`<ISRId>` must be the identifier used during system generation.

Applies to: RTA-OSEK.

3.7.13 TriggerOnCat2ISRStop

Syntax:

```
void TriggerOnCat2ISRStop(
                    ISRType <ISRId>)
```

Parameters:

ISRId	The identifier of the ISR.

Description: Sets the trigger to be the end of `<ISRId>`.

In Particular: If `<ISRId>` is set to `OSTRACE_TRIGGER_ANY`, the stop of any category 2 ISR will trigger tracing.

`<ISRId>` must be the identifier used during system generation.

Applies to: RTA-OSEK.

3.7.14 TriggerOnInitTaskStart

Syntax:

```
void TriggerOnInitTaskStart(
                    AppModeType <mode>)
```

Parameters:

mode	An application mode.

Description: Sets the trigger to be the start of the init task in application mode `<mode>`.

In Particular: The init-task is run when application mode `<mode>` is entered.

Applies to: ERCOS^{EK}

3.7.15 TriggerOnInitTaskStop

Syntax:

```
void TriggerOnInitTaskStop(
                    AppModeType <mode>)
```

Parameters:

mode	An application mode.

Description: Sets the trigger to be the end of the init task in application mode `<mode>`.

In Particular: The init-task is run when application mode `<mode>` is entered.

Applies to: ERCOS^{EK}

3.7.16 TriggerOnGetResource

Syntax:

```
void TriggerOnGetResource(
            ResourceType <ResourceId>)
```

Parameters:

`ResourceId`	Identifier of resource.

Description: Sets the trigger to be an attempt to get the resource `<ResourceId>` (i.e. a `GetResource( <ResourceId> )` call has been made).

In Particular: If `<ResourceId>` is set to `OSTRACE_TRIGGER_ANY`, any call to `GetResource()` will trigger tracing.

Applies to: All OS.

3.7.17 TriggerOnReleaseResource

Syntax:

```
void TriggerOnReleaseResource(
            ResourceType <ResourceId>)
```

Parameters:

`ResourceId`	Identifier of resource.

Description: Sets the trigger to be an attempt to release the resource `<ResourceId>` (i.e. a `ReleaseResource( <ResourceId>)` call has been made).

In Particular: If `<ResourceId>` is set to `OSTRACE_TRIGGER_ANY`, any call to `ReleaseResource()` will trigger tracing.

Applies to: All OS.

3.7.18 TriggerOnSetEvent

Syntax:
```
void TriggerOnSetEvent (
                      TaskType <TaskId>)
```

Parameters:

`TaskId`	Identifier of task.

Description: Sets the trigger to be a `SetEvent()` call targeting the specified task.

In Particular: If `<TaskId>` is set to `OSTRACE_TRIGGER_ANY`, any call to `SetEvent()` will trigger tracing.

Applies to: RTA-OSEK, Custom OS

Note: remember that this call requires a Task ID, not an Event ID.

3.7.19 TriggerOnTracepoint

Syntax:
```
void TriggerOnTracepoint(
           osTraceTracepointType <point>)
```

Parameters:

`point`	A Tracepoint ID.

Description: Sets the trigger to be the logging of *Tracepoint* `<point>`.

In Particular: If `<point>` is set to `OSTRACE_TRIGGER_ANY`, any call to `LogTracepoint…()` will trigger tracing.

Applies to: All OS.

3.7.20 TriggerOnTaskTracepoint

Syntax:

```
void TriggerOnTaskTracepoint(
     osTraceTaskTracepointType <point>,
     TaskType <task>)
```

Parameters:

`point`	A Task-Tracepoint ID.
`task`	A task ID.

Description: Sets the trigger to be the logging of *Task-Tracepoint* `<point>` from task <task>.

In Particular: If `<task>` is set to `OSTRACE_TRIGGER_ANY`, a call of `LogTaskTracepoint...()` from any Task will trigger tracing.

Applies to: All OS.

3.7.21 TriggerOnIntervalStart

Syntax:

```
void TriggerOnIntervalStart(
     osTraceIntervalType <intervalId>)
```

Parameters:

`intervalId`	An *Interval* identifier.

Description: Sets the trigger to be the start of the *interval* referenced by `<intervalId>`.

In Particular: If `<intervalId>` is set to `OSTRACE_TRIGGER_ANY`, any call to `LogIntervalStart...()` will trigger tracing.

Applies to: All OS.

3.7.22 TriggerOnIntervalEnd (TriggerOnIntervalStop)

Syntax:

```
void TriggerOnIntervalEnd(
     osTraceIntervalType <intervalId>)

void TriggerOnIntervalStop(
     osTraceIntervalType <intervalId>)
```

Parameters:

`intervalId`	An *Interval* identifier.

Description: Sets the trigger to be the end of the *interval* referenced by `<intervalId>`.

In Particular: If `<intervalId>` is set to `OSTRACE_TRIGGER_ANY`, any call to `LogIntervalEnd...()` will trigger tracing.

Applies to: All OS.

3.7.23 TriggerOnTimetableExpiry

Syntax:

```
void TriggerOnTimetableExpiry(
                      TimeTableType <tt>)
```

Parameters:

`tt`	Identifier of the timetable to trigger on.

Description: Sets the trigger to be the expiry of any point in timetable `<tt>`.

In Particular: If `<tt>` is set to `OSTRACE_TRIGGER_ANY`, the expiry of any point in any timetable will trigger tracing.

Applies to: ERCOS^{EK}

3.7.24 TriggerOnTickSchedule

Syntax:

```
void TriggerOnTickSchedule (
                ScheduleType <sched>)
```

Parameters:

`sched`	Identifier of the schedule to trigger on.

Description: Sets the trigger to be the expiry of any point in schedule `<sched>`.

In Particular: If `<sched>` is set to `OSTRACE_TRIGGER_ANY`, the expiry of any point in any schedule will trigger tracing.

Applies to: RTA-OSEK

3.7.25 TriggerOnAdvanceSchedule

Syntax:

```
void TriggerOnAdvanceSchedule (
                ScheduleType <sched>)
```

Parameters:

`sched`	Identifier of the schedule to trigger on.

Description: Sets the trigger to be the expiry of any point in schedule `<sched>`.

In Particular: If `<sched>` is set to `OSTRACE_TRIGGER_ANY`, the expiry of any point in any advanced schedule will trigger tracing.

Applies to: RTA-OSEK

3.7.26 TriggerOnAlarmExpiry

Syntax:

```
void TriggerOnAlarmExpiry(
                      AlarmType <alarm>)
```

Parameters:

`alarm` Identifier of the alarm to trigger on.

Description: Sets the trigger to be the expiry of `<alarm>`.

In Particular: If `<alarm>` is set to `OSTRACE_TRIGGER_ANY`, the expiry of any alarm will trigger tracing.

Applies to: All OS.

3.7.27 TriggerOnExplicitSendStateMessage

Syntax:

```
void TriggerOnExplicitSendStateMessage(
           STATEMESSAGE <messageId>)
```

Parameters:

`messageId` Identifier of the state message to trigger on.

Description: Sets the trigger to be the sending of state message `<messageId>`.

In Particular: If `<messageId>` is set to `OSTRACE_TRIGGER_ANY`, sending any state message will trigger tracing.

`<messageId>` must be the identifier used during system generation.

Applies to: ERCOS^{EK}

3.7.28 TriggerOnExplicitReceiveStateMessage

Syntax:

```
void
TriggerOnExplicitReceiveStateMessage(
            STATEMESSAGE <messageId>)
```

Parameters:

`messageId`	Identifier of the state message to trigger on.

Description: Sets the trigger to be the reception of state message `<messageId>`.

In Particular: If `<messageId>` is set to `OSTRACE_TRIGGER_ANY`, reception of any state message will trigger tracing.

`<messageId>` must be the identifier used during system generation.

Applies to: ERCOS^{EK}

3.7.29 TriggerOnSendMessage

Syntax:

```
void TriggerOnSendMessage(
                   SymbolicName <messName>)
```

Parameters:

`messName`	Symbolic name of the OSEK COM message.

Description: Sets the trigger to be the sending of OSEK COM message `<messName>`.

In Particular: If `<messName>` is set to `OSTRACE_TRIGGER_ANY`, sending of any OSEK COM message will trigger tracing.

Applies to: All OS supporting COM.

This includes messages logged by the OS instrumenting kit.

3.7.30 TriggerOnReceiveMessage

Syntax:
```
void TriggerOnReceiveMessage(
                SymbolicName <messName>)
```

Parameters:

`messName` Symbolic name of the OSEK COM message.

Description: Sets the trigger to be the reception of OSEK COM message `<messName>`.

In Particular: If `<messName>` is set to `OSTRACE_TRIGGER_ANY`, reception of any OSEK COM message will trigger tracing.

Applies to: All OS supporting COM.

This includes messages logged by the instrumenting kit.

3.7.31 TriggerOnError

Syntax:
```
void TriggerOnError(StatusType <err>)
```

Parameters:

`err` Error code.

Description: Sets the trigger to be the error `<err>`.

In Particular: If `<err>` is set to `OSTRACE_TRIGGER_ANY`, generation of any error will trigger tracing.

Error codes are listed in the documentation for the particular OS in use.

Applies to: All OS.

3.7.32 TriggerOnShutdown

Syntax:
```
void TriggerOnShutdown(StatusType <stat>)
```

Parameters:

`stat` Exit code.

Description: Sets the trigger to be the shutdown of the application.

In Particular: -

Applies to: All OS.

4 API Restrictions

4.1 Introduction

RTA-TRACE redefines many RTA-OSEK and ERCOS^{EK} API symbols as macros in order to instrument the operating system. In the great majority of cases, this is completely transparent to the user. However it is possible in rare cases to be caught out by a subtle issue outlined here.

4.2 General Problem

In a `#defined` macro with parameters, if a formal parameter appears more than once then there are a few possible wrong or sub-optimal outcomes:

- If the supplied parameter has side effects and is evaluated more than once then the side effect will occur multiple times and may constitute a bug.
- If the supplied parameter is a call to a function of significant run-time and is evaluated more than once then the overhead is multiplied.
- If the parameter is itself a macro then even if it is not evaluated more than once (*e.g.* the macro expands to an `if` or `switch` statement) then there is a multiplying of code-size overhead.

For these reasons it is undesirable for a macro's arguments to appear more than once in the macro body. In some cases however it is not sensibly avoidable.

4.3 Applicable Macros (ERCOS^{EK})

The following identifiers expand one or more of their arguments more than once. In each case, the replicated argument is shown in bold.

```
ActivateTask(task)
GetResource(res)
ReleaseResource(res)
ChainTask(task)
TriggerOnActivation(task)
TriggerOnChain(task)
TriggerOnTaskStart(task)
TriggerOnTaskStop(task)
TriggerOnGetResource(res)
TriggerOnReleaseResource(res)
TriggerOnTaskTracepoint(point, task)
SendMessage(msg, pointer)
ReceiveMessage(msg, pointer)
```

4.4 Applicable Macros (RTA-OSEK)

The following identifiers expand one or more of their arguments more than once. In each case, the replicated argument is shown in bold.

ActivateTask(**task**)
ActivateTaskset(**taskset**)
ChainTask(**task**)
ChainTaskset(**taskset**)
TickSchedule(**schedule**)
AdvanceSchedule(**schedule**, status)
SendMessage(**msg**, **pointer**)
ReceiveMessage(msg, **pointer**)
GetResource(**res**)
ReleaseResource(**res**)
SetEvent(task, mask)
WaitEvent(**mask**)
ClearEvent(**mask**)
ShutdownOS(**mode**)

Index

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.