RTA-TRACE

OS Instrumenting Kit 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

Copyright

Document TD00007-004

Contents

1 About this Manual

RTA-TRACE is a software logic analyzer for embedded systems. Coupled with a suitably enhanced operating system, it provides the embedded application developer with a unique set of services to assist in debugging and testing a system. Foremost among 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 explains how to use RTA-TRACE with other embedded systems – whether using a preemptive operating system, a cyclic executive, or no scheduler at all.

1.1 Who Should Read this Manual?

It is assumed that you are a developer. You should read this guide if you want to customize RTA-TRACE to support a previously unsupported embedded system.

The reader should be familiar with C programming concepts for embedded systems, and the operation of RTA-TRACE.

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`.

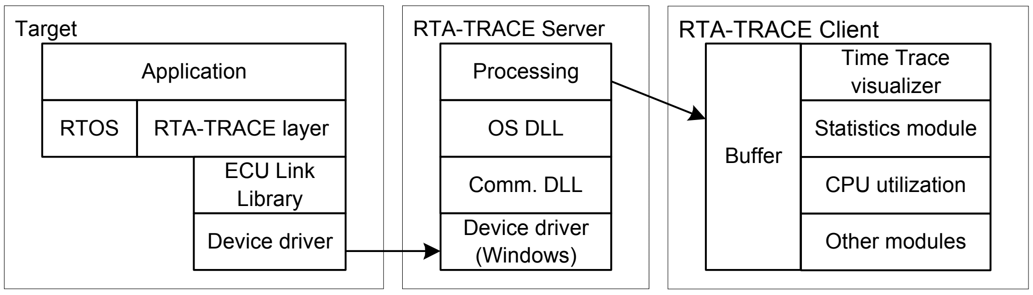
Sessions at the command prompt are shown with output shown in `courier` and user input shown in **`courier bold`**.

2 Introduction

RTA-TRACE records the activity of a running application and displays both instantaneous and summary information at runtime. It gives the engineer a valuable insight into the actual behavior of a system and allows detailed analysis of behavior via runtime visualization, reporting, and measurement tools.

RTA-TRACE can record OS activity (such as task activations, resource-locks, and alarms) as well as user-specified occurrences. The OS and application must be *instrumented* in order to record these events.

The diagram below illustrates the separate components involved:



RTA-TRACE supports the following OS implementations:

- RTA-OSEK;
- ERCOS^{EK};
- Other OS

This document describes the mechanisms used in instrumenting a system based on an 'Other OS'. This also applies to applications not using an OS at all.

Usage with RTA-OSEK and ERCOS^{EK} are described in separate documents.

2.1 Mechanisms

The target RTA-TRACE layer is responsible for placing trace data in a trace buffer. The trace data may originate from the OS or from application code. The ECU Link library is responsible for transmitting the trace buffer contents to the RTA-TRACE Server via any communications mechanism appropriate.

The *OS Instrumenting Kit* contains the code that is used on the target, as well as a PC-based plugin for the RTA-TRACE Server application.

The target code is split into 'instrumenting' and 'support' parts: the instrumenting API is the set of C functions (or macros) that are used to place instrumentation points in the target code; whilst the support API is the set of C functions and data that are required to support the instrumenting API.

The API is described in Sections 4 and 5.

2.2 Use

The OS Instrumenting Kit contains C source code that can be compiled and linked alongside an application/OS. It is your job to place instrumenting calls at appropriate places in the application or OS code (see Section 6 for hints about this) in order for meaningful trace data to be extracted. The resultant application will also need to be linked with an appropriate ECU Link technology – See the *RTA-TRACE ECU Link Guide* for more details about this.

2.3 Kit contents

The OS Instrumenting Kit consists of a number of source files in the `OS Instrumenting Kit\RTLib\` directory. These are broadly split into macro definitions (contained in `RTapi.h`) and support code (in the `.c` files).

The ECU Link source code is contained in the directory `OS Instrumenting Kit\RTComm\`.

Also supplied is an example application (in the directory `OS Instrumenting Kit\Example\)` to illustrate the way in which instrumenting calls may be integrated.

The example application does not use any OS services and is single-threaded. This allows is to run on any platform, including a Microsoft Windows PC (a Windows executable is provided to allow the installation to be tested before needing to derive an embedded application). The application simulates the behavior of a multi-threaded application by making commented-out calls to a purely illustrative OS API and then imitating the behavior that would result. The illustrative OS API calls all take the form `OSxxx().`

As the application is single-threaded, explicit calls to upload trace information have been inserted throughout. This would not be required in a multi-threaded environment. Guidance on the insertion of code into real applications to upload trace information is provided in the *RTA-TRACE ECU Link Guide*.

3 Trace Object Description

RTA-TRACE deals with traceable objects, and the trace events that occur for each of them. These objects include thing like tasks, messages, resources, and tracepoints. Each object has a trace ID by which it can be identified in the trace data stream.

The trace objects, their characteristics and their trace ID have to be described to RTA-TRACE in order for it to be able to correctly decode the target trace data. This declaration is done via a Run-Time Interface file (having a `.rta` suffix)

The format of the `.rta` file is based around the ORTI language[1] used for supporting OSEK-aware debuggers. This is intended to make it easy to adapt existing OSEK code generation tools to support RTA-TRACE. Non-OSEK implementations will generally have little difficulty in generating an equivalent file however.

3.1 Describing the Target System

The purpose of a `.rta` file is to describe the target's traceable objects to RTA-TRACE.

The target application is composed of a collection of trace objects. Objects include:

- Tasks
- ISRs
- Resources
- Counters
- Alarms
- Messages
- Processes
- Profiles
- Tracepoints (i.e. arbitrary trace events)
- Task tracepoints (i.e. arbitrary trace events that are associated with a particular task)
- Intervals (a mechanism used to measure how long some activity takes)

The `.rta` file describes each such object found in the target. The remainder of this section introduces the objects that RTA-TRACE understands. RTA-TRACE has been designed to work primarily with operating systems that are similar to OSEK/VDX and this is reflected in the way that RTA-TRACE models target objects. Readers who are already familiar with OSEK should also find much of the following description familiar. Although RTA-TRACE focuses on OSEK it can also be used with other targets such as cyclic executives – simply

[1] "OSEK/VDX OSEK Run Time Interface (ORTI) Part A: Language Specification", Version 2.1.1, 4 March 2002, http://www.osek-vdx.org/orti_documents.htm

ensure that the description in the `.rta` file is consistent with the way that the target is instrumented. Instrumentation hints for non-OSEK systems are given in section 6.

3.2 Outline of .rta File Format

This section describes the format of `.rta` – files. The description is not exhaustive, but is sufficient to allow `.rta` files to be written.

A `.rta` file consists of two major parts:

Declaration section	This declares the types of objects that can be present in the system.
Information section	This contains information about all of the objects present in the system. Each object has a type declared in the declaration section.

3.2.1 Declaration Section

The declaration section declares the types of objects that can be present in the system. The declaration section has the form:

```
IMPLEMENTATION implementation_name {
  object type declarations
};
```

`implementation_name` is the name of the system chosen by the file author. It can be any valid identifier. Identifiers follow the same rules as C language identifiers. All identifiers and keywords (e.g. `IMPLEMENTATION`) are case sensitive.

An object type declaration has the form:

```
object_type {
  attribute declarations
};
```

`object_type` is the name of the object type (e.g. `TASK`, `ISR1` or `RESOURCE`) and must be a valid identifier.

Attribute declarations have several forms. Those relevant to `.rta` files are shown in the table below. In all cases `attribute_name` is the name of the attribute and must be a valid identifier. `label` is a human friendly short description of the attribute.

`STRING` *attribute_name*, "*label*";	This declares an attribute that has a string value.
`UINT16` *attribute_name*, "*label*";	This declares an attribute that has a 16-bit unsigned integer value.

The name of an attribute is local to the object type that contains the attribute. That is, multiple object types may have attributes with the same name.

Consider a simple example of a declaration section:

```
IMPLEMENTATION rta_trace {
  OBJECT_A {
    STRING attr_a, "String attribute of OBJECT_A";
    UINT16 attr_b, "uint16 attribute of OBJECT_A";
  };
  OBJECT_B {
    STRING attr_a, "String attribute of OBJECT_B";
    UINT16 attr_c, "uint16 attribute of OBJECT_B";
    };
  OBJECT_C {
    STRING attr_a, "String attribute of OBJECT_C";
    UINT16 attr_d, "uint16 attribute of OBJECT_C";
  };
};
```

This declaration section declares three objects types – called `OBJECT_A`, `OBJECT_B` and `OBJECT_C`. `OBJECT_A` has a string attribute called `attr_a` and a 16 bit unsigned integer attribute called `attr_b`. `OBJECT_B` has a string attribute called `attr_a` and a 16 bit unsigned integer attribute called `attr_c`. `OBJECT_C` has a string attribute called `attr_a` and a 16 bit unsigned integer attribute called `attr_d`.

3.2.2 Sample .rta Implementation

Apart from trace enumerations (described later), the required contents of the `.rta` Implementation section tends not to have to change between different applications, so the following example could be used as the basis for a template.

```
IMPLEMENTATION rta_trace  {
 OS {
       ENUM UINT8 [
             "E_OK" = 0,
             "E_OS_ACCESS" = 1,
             "E_OS_CALLEVEL" = 2,
             "E_OS_ID" = 3,
             "E_OS_LIMIT" = 4,
             "E_OS_NOFUNC" = 5,
             "E_OS_RESOURCE" = 6,
             "E_OS_STATE" = 7,
             "E_OS_VALUE" = 8,
             "E_OS_SYS_IDLE" = 16,
             "E_OS_SYS_AP_INVALID" = 17,
             "E_OS_SYS_AP_NULL" = 18,
             "E_OS_SYS_AP_READONLY" = 19,
             "E_OS_SYS_TS_INVALID" = 20,
             "E_OS_SYS_TS_READONLY" = 21,
             "E_OS_SYS_S_MODULO" = 22,
             "E_OS_SYS_S_INVALID" = 23,
                "E_OS_SYS_S_MISMATCH" = 24,
                "E_OS_SYS_STACK_FAULT" = 25,
                "E_OS_SYS_T_INVALID" = 26,
                "E_OS_SYS_R_PERMISSION" = 28,
                "E_OS_SYS_COUNTER_INVALID" = 29,
                "E_OS_SYS_CONFIG_ERROR" = 30,
                "E_OS_SYS_CALLEVEL" = 31,
                "E_COM_ID" = 32,
                "E_COM_BUSY" = 33,
                "E_COM_NOMSG" = 34,
                "E_COM_LIMIT" = 35,
                "E_COM_LOCKED" = 36,
                "E_COM_SYS_STOPPED" = 48,
                "Budget Overrun" = 255
          ] LASTERROR, "Last OSEK error";
          ENUM UINT8 [
                "NO_APPMODE" = 0,
                "OSDEFAULTAPPMODE" = 1
          ] CURRENTAPPMODE, "Current AppMode";
          ENUM UINT32 [
                "a" = 1,
                "b" = 2,
                "c" = 3,
                "d" = 4
          ] a, "OS_1";
          ENUM UINT32 [
                "t" = 44,
                "s" = 55,
                "r" = 66,
                "q" = 77,
                "p" = 88
          ] b, "OS_2";
          STRING vs_p_Fmt1, "StartOS Data format";
          STRING vs_p_Fmt2, "ShutdownOS Data format";
  };
  TASK {
          STRING vs_ID, "Trace ID";
          UINT16 vs_ACTIVATIONS, "Max activations";
          STRING vs_TYPE, "Conformance type";
          STRING vs_p_Pri, "Base priority";
          STRING vs_p_Disp, "Dispatch priority";
          STRING vs_InternalRes, "Internal resource
ID";
          STRING vs_p_StackCeiling, "Stack limit";
          STRING vs_p_StackRange, "Stack range";
          STRING vs_p_Budget, "Budget";
          STRING vs_p_Excl, "Excluded?";
          STRING vs_p_OSEvents, "Events";
  };
  ISR2 {
          STRING vs_ID, "Trace ID";
          STRING vs_RESOURCES, "Resources";
          STRING vs_BUFFERING, "Buffering";
```

```
        STRING vs_p_Pri, "Base priority";
        STRING vs_p_Disp, "Dispatch priority";
        STRING vs_p_StackCeiling, "Stack limit";
        STRING vs_p_StackRange, "Stack range";
        STRING vs_p_Budget, "Budget";
        STRING vs_p_Excl, "Excluded?";
        STRING vs_p_Arb, "Arbitration";
  };
  ISR1 {
        STRING vs_ID, "Trace ID";
        STRING vs_BUFFERING, "Buffering";
        STRING vs_p_Pri, "Base priority";
        STRING vs_p_Disp, "Dispatch priority";
        STRING vs_p_StackCeiling, "Stack limit";
        STRING vs_p_StackRange, "Stack range";
        STRING vs_p_Budget, "Budget";
        STRING vs_p_Excl, "Excluded?";
        STRING vs_p_Arb, "Arbitration";
  };
  ISR0 {
        STRING vs_ID, "Trace ID";
        STRING vs_BUFFERING, "Buffering";
        STRING vs_p_Pri, "Base priority";
        STRING vs_p_Disp, "Dispatch priority";
        STRING vs_p_StackCeiling, "Stack limit";
        STRING vs_p_StackRange, "Stack range";
        STRING vs_p_Budget, "Budget";
        STRING vs_p_Excl, "Excluded?";
        STRING vs_p_Arb, "Arbitration";
  };
  ALARM {
        STRING vs_ID, "Trace ID";
        STRING ACTION, "Action";
        STRING vs_Owner, "Owning counter ID";
        STRING vs_Activates, "Activates";
        STRING vs_SetEvent, "Sets Event";
  };
  COUNTER {
        STRING vs_ID, "Trace ID";
        STRING vs_p_Fmt, "Data format";
  };
  MESSAGECONTAINER {
        STRING vs_ID, "Trace ID";
        STRING MSGNAME, "Message Name";
        STRING vs_CDATATYPE, "C type";
        STRING vs_p_Fmt, "Data format";
        STRING vs_Activates, "Activates";
        STRING vs_SetEvent, "Sets Event";
  };
  Trace {
        STRING vs_VERSION, "Trace version";
```

```
       STRING vs_p_TickDuration, "Stopwatch tick
duration";
       STRING vs_p_MaxAbsTime, "Max Stopwatch
value";
       STRING vs_p_BigEndian, "BigEndian";
       STRING vs_p_IntSize, "IntSize";
       STRING vs_ErrorFmt, "ErrorFmt";
       STRING vs_KindSize, "KindSize";
       STRING vs_InfoSize, "InfoSize";
       STRING vs_TimeSize, "TimeSize";
       STRING vs_TASKS_AND_ISRS, "Filter
TASKS_AND_ISRS";
       STRING vs_STARTUP_SHUTDOWN, "Filter
STARTUP_SHUTDOWN";
       STRING vs_ACTIVATIONS, "Filter ACTIVATIONS";
       STRING vs_COUNTERS_ALARMS, "Filter
COUNTERS_ALARMS";
       STRING vs_SCHEDULES, "Filter SCHEDULES";
       STRING vs_RESOURCES, "Filter RESOURCES";
       STRING vs_INTERRUPT_LOCKS, "Filter
INTERRUPT_LOCKS";
       STRING vs_ERRORS, "Filter ERRORS";
       STRING vs_OSEK_MESSAGES, "Filter
OSEK_MESSAGES";
       STRING vs_MESSAGE_DATA, "Filter
MESSAGE_DATA";
       STRING vs_SWITCHING_OVERHEADS, "Filter
SWITCHING_OVERHEADS";
       STRING vs_OSEK_EVENTS, "Filter OSEK_EVENTS";
       STRING vs_TRACEPOINTS, "Filter TRACEPOINTS";
       STRING vs_TASK_TRACEPOINTS, "Filter
TASK_TRACEPOINTS";
       STRING vs_INTERVALS, "Filter INTERVALS";
       STRING vs_STACK, "Filter STACK";
  };
  Profile {
       STRING vs_ID, "Trace ID";
       STRING vs_Owner, "Owning Task/ISR";
  };
  TaskTracepoint {
       STRING vs_ID, "Trace ID";
       STRING vs_Owner, "Owning Task/ISR";
       STRING vs_p_Fmt, "Data Format";
  };
  Tracepoint {
       STRING vs_ID, "Trace ID";
       STRING vs_p_Fmt, "Data Format";
  };
  CritExec {
       STRING vs_ID, "Trace ID";
       STRING vs_Owner, "Owning Task/ISR/Profile";
       STRING vs_p_Budget, "Budget";
  };
  Interval {
       STRING vs_ID, "Trace ID";
       STRING vs_p_Fmt, "Data Format";
  };
  Resource {
       STRING vs_ID, "Trace ID";
       STRING vs_p_Pri, "Task priority";
       STRING vs_p_Isr, "ISR priority";
       STRING vs_Owner, "Owning resource";
       STRING vs_Internal, "Internal?";
  };
  Schedule {
       STRING vs_ID, "Trace ID";
  };
};
```

3.2.3 Information Section

The information section contains definitions of each of the objects that exist in the system. Each object is described in turn, following the implementation section. An object definition has the form:

```
object_type object_name {
  attribute_name0 = "value0";
  …
  attribute_nameN = "valueN";
};
```

`object_type` is the name of an object type declared in the declaration section. `object_name` is the name of the object and must be a valid identifier. `attribute_name0` … `attribute_nameN` are the names of attributes declared in the declaration of `object_type`. `value0` … `valueN` are values assigned to the attributes. Not all of the attributes declared in an object type declaration need to be assigned values when an object of that type is defined. The attribute description (Section 3.3) will indicate which attributes are optional. The information section does not need to define an object of every object type in the declaration section.

Consider an example using the reference implementation section shown above:

```
TASK myTask {
 vs_ID = "7";
 vs_ACTIVATIONS = "1";
 vs_TYPE = "ECC2";
 vs_p_Pri = "2";
 vs_p_Disp = "2";
 vs_p_OSEvents = "ea2.1 ";
 vs_p_StackCeiling = "169";
 vs_p_StackRange = "37";
};
Resource RES_SCHEDULER {
  vs_ID = "1";
  vs_p_Pri = "5";
};
Trace Trace {
  vs_VERSION = "2.0.0";
  vs_p_TickDuration = "250";
  vs_p_MaxAbsTime = "65535";
  vs_p_BigEndian = "1";
  vs_p_IntSize = "32";
  vs_ErrorFmt = "%99E";
  vs_KindSize = "1";
  vs_InfoSize = "1";
  vs_TimeSize = "2";
  vs_TASKS_AND_ISRS = "false";
  vs_STARTUP_SHUTDOWN = "runtime";
  vs_ACTIVATIONS = "true";
  vs_COUNTERS_ALARMS = "true";
  vs_SCHEDULES = "true";
  vs_RESOURCES = "true";
  vs_INTERRUPT_LOCKS = "true";
  vs_ERRORS = "true";
  vs_OSEK_MESSAGES = "true";
  vs_MESSAGE_DATA = "true";
  vs_SWITCHING_OVERHEADS = "true";
  vs_OSEK_EVENTS = "true";
  vs_TRACEPOINTS = "true";
  vs_TASK_TRACEPOINTS = "true";
  vs_INTERVALS = "true";
  vs_STACK = "true";
};
```

3.3 Trace Objects

This section describes the trace object information that must be present in a `.rta` file. The description consists of a list of object types and attributes understood by RTA-TRACE. Some of the object types are specific to RTA-TRACE, but many are also used by the OSEK ORTI language. If you are starting from an ORTI file generated by an OSEK OS tool then you will find that the file already contains objects of type `TASK` for example.

The `.rta` file can contain objects and attributes not used by RTA-TRACE since RTA-TRACE only looks for the objects and attributes that it uses. So if you start from an automatically generated file there is no need to remove objects and attributes not used by RTA-TRACE.

Note: All numeric values should be specified as decimal numbers.

An explanation of *format strings* (referenced in the following descriptions) can be found in section 8

3.3.1 Object type OS

A single object of type `OS` is used to define general characteristics of the operating system, and enumeration values (below).

Attribute	Type	Description
`vs_p_Fmt1`	STRING (optional)	A format string that describes how to display the value associated with the trace API `osTraceOSStart()`. A typical value is "%98E" which maps the startup value to the `CURRENTAPPMODE` enumeration in the OS object.
`vs_p_Fmt2`	STRING (optional)	A format string that describes how to display the value associated with the trace API `osTraceOSExit()`. A typical value is "%99E" which maps the startup value to the `LASTERROR` enumeration in the OS object.

Enumerations provide a way of mapping numeric values in the target application onto textual descriptions that can be used by RTA-TRACE to visualize the target behavior better. They are often used to decode error codes for example.

Enumerations are defined in the OS implementation clause and take the following form:

```
ENUM <type> [
    "<name>" = <value>,
    …
] <enum_name>, "<enum_reference>";
```

For example – `RAINBOW` may be defined as follows:

```
ENUM UINT8 [
    "RED" = 0,
    "ORANGE" = 1,
    "YELLOW" = 2,
    "GREEN" = 3,
    "BLUE" = 4,
    "INDIGO" = 5,
    "VIOLET" = 6
] RAINBOW, "OS_1";
```

Enumerations are referenced in trace data format specifications using the form "%*n*E" where *n* is given in the enum_reference field (i.e."OS_*n*").

The two 'special' enumerations are `CURRENTAPPMODE` which is taken to be equivalent to "OS_98" and `LASTERROR` which is taken to be equivalent to "OS_99".

3.3.2 Object type Trace

An object of type `Trace` is used to provide RTA-TRACE with general information about the system being traced. A single object of this type must exist.

Attribute	**Type**	**Description**
`vs_VERSION`	STRING (mandatory)	The version of the `.rta` file. Typically set to "2.0.0".
`vs_p_TickDuration`	STRING (mandatory)	The number of nanoseconds in one tick of the traced system's timestamp clock (may contain a decimal point).
`vs_p_MaxAbsTime`	STRING (mandatory)	The maximum absolute value of the traced system's clock. E.g. if the traced system's clock is 16 bit then this value would be "65535".
`vs_p_BigEndian`	UINT8 (mandatory)	"0" if the traced system is little-endian or "1" if the traced system is big-endian.
`vs_p_IntSize`	UINT8 (mandatory)	The number of bits in a C language `int` type on the traced system.
`vs_ErrorFmt`	STRING (mandatory)	A format string that describes how error information should be displayed. Typically `%99E`.
`vs_KindSize`	UINT8 (mandatory)	The number of bytes in the traced system's "kind" field – either 1 or 2. The "kind" field contains a descriptor for the kind of an event (i.e. Task start, get-resource, etc.)

`vs_InfoSize`	UINT8 (mandatory)	The number of bytes in the traced system's "info" field – either 1 or 2. The "info" field contains the object to which the event relates (i.e. identifier of 'Task1', 'Resource7' etc.)
`vs_TimeSize`	UINT8 (mandatory)	The number of bytes used to record time in a trace record – either 2 or 4 (16 or 32 bits).
`vs_TASKS_AND_ISRS` `vs_ERRORS` `vs_ACTIVATIONS` `vs_SCHEDULES` `vs_RESOURCES` `vs_OSEK_EVENTS` `vs_TRACEPOINTS` `vs_INTERVALS` `vs_MESSAGE_DATA` `vs_COUNTERS_ALARMS` `vs_STARTUP_SHUTDOWN` `vs_OSEK_MESSAGES` `vs_INTERRUPT_LOCKS` `vs_SWITCHING_OVERHEADS` `vs_TASK_TRACEPOINTS`	STRING (optional)	Each trace class (See *RTA-TRACE User Manual*) can be filtered by build-time options. Each class can be filtered out at build-time (string value `"false"`), always traced at build-time (string value `"true"`) or filtered at runtime. (string value `"runtime"`).

Note: In the current implementation, `vs_KindSize` and `vs_InfoSize` must both have the same value. The size of these fields correlates to the use of standard/compact identifiers (standard: 16-bit; compact: 8-bit). Also see Section 5.1

3.3.3 Tasks, ISRs, Processes and Profiles - overview

A **Task** is an element of program execution – sometimes also called a thread. A task executes some specific program code to perform a particular function. For RTA-TRACE, tasks can exist in a number of states – unknown, activated, running, preempted and waiting. Each task has a priority allocated to it.

Since there is only one processor, only one task can be executing code at once – that is only one task can be in the *running* state. The OS manages which task is running. Usually the OS will ensure that the highest priority runnable task executes in preference to other tasks. If a task that has a higher priority than the currently running task becomes ready then the OS can suspend the currently running task and start running the higher priority task. The higher priority task is said to have *preempted* the lower priority task.

In the `.rta` file there are two descriptions of priority. A task's base priority describes how it is prioritized in relation to other tasks when considering which task to start. A task's dispatch priority is used to indicate that the priority that the task takes when it starts running. This can be used as a way of ensuring that only one of a group of tasks may run at once. For example, consider task A with a base priority of 1 and a dispatch priority of 2 and task B with a base priority of 2 and a dispatch priority of 2. When the OS has to choose which of A and B to run first it will choose B as it has a higher base priority. However, if A is running when B is activated, B will not preempt A as they have the same dispatch priority. A task's base and dispatch priorities can be the same.

An **ISR** is the element of target code that handles an interrupt. Generally ISRs have a higher priority than tasks. When an interrupt signal occurs the OS or hardware checks the priority of the task or ISR that is running, and if the new interrupt has a higher priority then it preempts the currently running task/ISR. If the new interrupt has a lower priority, then its ISR is not run until it is the highest priority ISR that can be run. The priority of an ISR may be determined by the target hardware – specification of ISR priority is implementation-specific.

RTA-TRACE supports OSEK-like Category 1 and Category 2 interrupts. A Category 1 interrupt is intended to be 'fast', and not require access to the OS API functions or services. A Category 2 interrupt is managed by the OS and its ISR may use a subset of OS API functions. Category 0 interrupts are also provided. These are non OSEK, but can be used just like any other interrupt.

For the purposes of RTA-TRACE, ISRs are treated exactly the same as tasks.

A **process** is a typically small piece of code that runs in a task or ISR to perform a specific sub-function. A process is run by a task/ISR from beginning to end. A task or ISR can contain several processes that run one after the other. The process start and end points can be easily seen in RTA-TRACE. A process can only be owned by a single Task/ISR.

(Note that the process concept here is not the same as the process concept in an OS like UNIX or Windows.)

A **profile** is a means of describing an execution path through a program. For example, for an ISR that executes one of many alternative branches based on some runtime condition, the ISR can be instrumented to report which execution profile it is executing at each invocation – e.g. 'CAN_Rx_Interrupt', 'Timer3_Expiry', 'ADC_complete'.

Note: Tasks, processes and ISRs share the same identifier numberspace – the `vs_ID` values for these objects must not overlap.

3.3.4 Object Type Task

A `TASK` object is used to describe an OS task. A `TASK` object must exist for every task in the system.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the task object.
`vs_p_Pri`	STRING (mandatory)	The base priority of the task. The lower the number the lower the priority.
`vs_p_Disp`	STRING (mandatory)	The dispatch priority of the task. It is the same or higher than the base priority. See the description of tasks in section 3.2.
`vs_ACTIVATIONS`	UINT16 (optional)	The maximum number of outstanding task activations that can be recognized. In a simple OSEK 'BCC1' task, activation requests for tasks are not queued, so this value would be '1'. In other cases, the OS may be able to queue up to 4 activation requests for the task, so the value can be set accordingly.
`vs_TYPE`	STRING (mandatory)	A short description of the task type. For OSEK systems this could be "`BCC1`", "`BCC2`", "`ECC1`" and "`ECC2`", but other values can be specified.
`vs_InternalRes`	STRING (optional)	The `vs_ID` of an internal resource that is implicitly locked when the task is running. An internal resource may be shared by multiple tasks to ensure that only one of the tasks can run at once.
`vs_p_StackCeiling`	STRING (optional)	The maximum stack value that is expected from any event logged during the execution of this task. This value is in bytes.
`vs_p_StackRange`	STRING (optional)	Gives the maximum amount of stack that the task is expected to use during the execution of this task.

`vs_p_Budget`	STRING (optional)	The execution budget declared for the task. i.e. the maximum number of stopwatch ticks that the task is allowed to run for.
`vs_p_Excl`	STRING (optional)	"`1`" if the task should not be traced. Omitted if the task should be traced.
`vs_p_OSEvents`	STRING (optional)	The names of OSEK events on which the task may wait. Only present for ECC tasks. The format of this is a space separated list of <eventname>:mask pairs. e.g. `"ea1.1 ea2:8"` represents an event named `ea1` with mask value `1`, and an event named `ea2` with a mask value `8`.

3.3.5 Object Type ISR2/ISR1/ISR0

ISR objects are used to describe an Interrupt Service Routines.

Note: Tasks, profiles and ISRs share the same identifier numberspace – the `vs_ID` values for these objects must not overlap.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the ISR object.
`vs_p_Pri`	STRING (mandatory)	The base priority of the ISR. The lower the number the lower the priority.
`vs_p_Disp`	STRING (mandatory)	The dispatch priority of the ISR. It is the same or higher than base priority.
`vs_p_Arb`	STRING (mandatory)	The arbitration order of the ISR. Where two ISRs that share the same priority become ready at the same instant, the target hardware will service the ISR with the highest arbitration value first. This order is often determined by the target hardware.

`vs_p_StackCeiling`	STRING (optional)	The maximum size of the traced system's stack.
`vs_p_StackRange`	STRING (optional)	Gives the maximum amount of stack that the task is expected to use during the execution of this task.
`vs_p_Budget`	STRING (optional)	The execution budget declared for the ISR. i.e. the maximum number of stopwatch ticks that the ISR is allowed to run for.
`vs_p_Excl`	STRING (optional)	"1" if the ISR should not be traced. Omitted if the ISR should be traced.

3.3.6 Object Type Process

A `Process` object is used to describe a small piece of code – generally run as part of a task. This is a means for partitioning functional blocks.

Attribute	**Type**	**Description**
`vs_ID`	UINT16 (mandatory)	The trace identifier of the process object. Each Process object must be numbered uniquely, starting at 1.
`vs_Owner`	UINT16 (mandatory)	The vs_ID of the task or ISR that owns the process

3.3.7 Object Type Profile

A `Profile` object is simply a means of indicating to RTA-TRACE which piece of code is actually being executed in the case of a Task/ISR which may perform different functions at runtime.

Note: Tasks, profiles and ISRs share the same identifier numberspace – the `vs_ID` values for these objects must not overlap.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the profile object.
`vs_Owner`	STRING (mandatory)	The vs_ID of the task or ISR that owns the profile.

3.3.8 Object Type Resource

A resource is an entity that allows an application to serialize execution. After task/ISR X has locked a resource, no other task/ISR may lock that resource until task/ISR X has unlocked the resource (resources may also be called "mutexes"). OSEK operating systems use the "priority ceiling protocol" to implement resources. That is, each resource has a priority that is equal to the maximum dispatch priority of any task/ISR that may lock the resource. When a task/ISR locks a resource its effective dispatch priority is increased to the priority of the resource. This prohibits any other task/ISR that is allowed to lock the resource from being run by the OS. When the task/ISR unlocks the resource its effective dispatch priority is returned to its previous value. In OSEK, it may be possible to share resources between tasks and ISRs, although this is implementation specific. (Resources may also be called semaphores (either binary or counted) for other OSs)

Resource objects must exist for all resource types (standard, internal, and linked).

The `Resource` object is used to describe a resource.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the resource object. Each Resource object must be numbered uniquely, starting at 1.
`vs_p_Pri`	STRING (see description)	The ceiling priority of the resource if the resource can only be locked by tasks. Omitted if the resource can be locked by an ISR.

`vs_p_Isr`	STRING (see description)	The ceiling priority of the resource if the resource can be locked by an ISR. Omitted if the resource cannot be locked by an ISR.
`vs_Owner`	STRING (optional)	The trace ID of this resource's owner. Only present for a linked resource.
`vs_Internal`	STRING (optional)	If this is set to "1", the resource is taken to be of 'internal' or 'automatic' type. Any task with a `vs_InternalRes` value that matches the trace ID of this resource is deemed to lock/unlock this resource automatically when it starts/stops.

3.3.9 Object Type COUNTER

Counters are tightly coupled with Alarms (see next section). An OSEK Alarm is an OS resource that allows activity to occur some time in the future (Alarms may also be used to implement periodic behaviour). Alarms perform certain actions when they expire (activating a task, invoking a callback function, or setting an event for an ECC task); expiry times being set either at configuration time or at run time. Alarms are attached to Counters which, in turn, are ticked from an appropriate source (i.e. a periodic timer, an external stimuli, a task etc.).

A `COUNTER` object is used to describe a counter. Counters are the mechanism used to drive ALARMs. They cause alarms to execute when the alarm expiry time matches the counter's current value.

Attribute	**Type**	**Description**
`vs_ID`	STRING (mandatory)	The trace identifier of the counter object. Each COUNTER object must be numbered uniquely, starting at 1.
`vs_p_Fmt`	STRING (optional)	The format string describing how to display any count value.

3.3.10 Object Type ALARM

An `ALARM` object is used to describe an alarm.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the alarm object. Each ALARM object must be numbered uniquely, starting at 1.
`vs_Owner`	STRING (mandatory)	The trace ID of the counter that owns this alarm.
`vs_p_Action`	STRING (optional)	A description of what happens when the alarm expires. This appears in text form in RTA-TRACE floating hints.
`vs_Activates`	STRING (optional)	The trace ID of a task that this alarm is deemed to activate. Where present, the 'activate' indication will be inserted automatically by RTA-TRACE.
`vs_SetEvent`	STRING (optional)	A string in the form "<num1>:<num2>" is taken to mean that the alarm sets an event belonging to task ID <num1>, with event mask <num2>. Where present, the 'set event' indication will be inserted automatically by RTA-TRACE.

3.3.11 Object Type MESSAGECONTAINER

Alongside the OSEK OS, there is a communications standard known as OSEK COM. OSEK COM provides a means of sending messages between tasks. These messages are sent via message containers. A message container has the following information in its definition: the C language data type of the message, the number of messages that may be queued and optionally notification (activate a task, set an event, or invoke a callback function) when a message is sent or received.

A `MESSAGECONTAINER` object is used to describe OSEK-style messages. The content of the message can be displayed by RTA-TRACE if configured to do so.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the message object. Each MESSAGECONTAINER object must be numbered uniquely, starting at 1.
`vs_p_CType`	STRING (optional)	The C language data type used for the message.
`vs_p_Fmt`	STRING (optional)	The format string describing how to display the message data.
`vs_Activates`	STRING (optional)	The trace ID of a task that this message is deemed to activate. Where present, the 'activate' indication will be inserted automatically by RTA-TRACE and does not have to be recorded on the target.
`vs_SetEvent`	STRING (optional)	A string in the form "<num1>:<num2>" is taken to mean that the message sets an event belonging to task ID <num1>, with event mask <num2>. Where present, the 'set event' indication will be inserted automatically by RTA-TRACE and does not have to be recorded on the target.

3.3.12 Object Type Tracepoint

A tracepoint is used to log the occurrence of an arbitrary event. The target can contain tracepoints at any point in the program.

The `Tracepoint` object is used to describe a tracepoint. `Tracepoint` objects do not need to be declared before use unless you want to assign a particular name or format string to them.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the tracepoint object. Each Tracepoint object must be numbered uniquely, starting at 1.
`vs_p_Fmt`	STRING (optional)	A format string used for displaying data associated with the tracepoint.

3.3.13 Task Tracepoints

A task tracepoint is a special form of a tracepoint that gets associated with the task/ISR that logs it. In RTA-TRACE, they get drawn alongside the task/ISR that was running when they were logged.

The `TaskTracepoint` object is used to describe a task tracepoint. `TaskTracepoint` objects do not need to be declared before use unless you want to assign a particular name or format string to them.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the task tracepoint object. Each TaskTracepoint object must be numbered uniquely, starting at 1.
`vs_Owner`	STRING (optional)	The trace ID of the task/ISR that owns this task tracepoint. If omitted then any task tracepoint in the traced system that specifies this object's trace ID will use this description.
`vs_p_Fmt`	STRING (optional)	A format string used for displaying data associated with the tracepoint.

3.3.14 Object Type Interval

An interval is used to measure the amount of time that some activity takes. The target program code contains interval start and end instrumentation around the activity.

An `Interval` object is used to describe an interval. `Interval` objects do not need to be declared before use unless you want to assign a particular name or format string to them.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the interval object. Each Interval object must be numbered uniquely, starting at 1.
`vs_p_Fmt`	STRING (optional)	A format string used for displaying data associated with the interval.

3.3.15 Object Type CritExec

A `CritExec` object is used to represent a critical execution point in a task/ISR/profile. It is similar to a task tracepoint, and is typically used to mark the completion of a particular section of code. RTA-TRACE can monitor the min/max execution time from the start of the task/ISR to each critical execution point.

Attribute	Type	Description
`vs_ID`	STRING (mandatory)	The trace identifier of the CritExec object. Each CritExec object must be numbered uniquely, starting at 1.
`vs_Owner`	STRING (optional)	The trace ID of the task/ISR that owns this critical execution point.
`vs_p_Budget`	STRING (optional)	The execution time declared for the CritExec object. i.e. the maximum number of stopwatch ticks that are expected before it occurs.

4 Instrumentation Part 1 – macro API

The macros described in this section are defined in the header file `RTapi.h`.

Trace events are placed in the trace buffer using macros defined in the supplied header file. Each trace event has a particular mapping to the behavior of an Operating System object, and therefore a particular representation in the Time-Trace visualizer. The available macros are described here, along with a description of typical usage and visualizer representation.

Further information can be gathered from examination of the example application supplied on the CD.

Note: for the visualizer to display meaningful trace data, each object reference used must be defined in the `.rta` file (Section 3).

4.1 OS

The API macros defined in this section deal with functions within the OS itself. If instrumenting a non-OS based system, not all of these APIs will be relevant.

4.1.1 osTraceOSStart

Usage: `osTraceOSStart(<value>)`

Description: This indicates that the OS/Scheduler/system has started. For an OSEK system, this might be placed within the startup-hook for example.

`<value>` is OS dependent. It could indicate which mode of operation the system is running in. The value can be displayed by RTA-TRACE if the OS object attribute `vs_p_Fmt1` is specified correctly.

4.1.2 osTraceOSExit

Usage: `osTraceOSExit(<value>)`

Description: This indicates that the OS/Scheduler/system has shutdown. For an OSEK system, this might be placed within the shutdown-hook for example.

`<value>` is OS dependent. It could indicate why the OS has stopped. The value can be displayed by RTA-TRACE if the OS object attribute `vs_p_Fmt2` is specified correctly.

4.1.3 osTraceSchedulerEntry

Usage:	`osTraceSchedulerEntry()`
Description:	This is used to indicate that the system being instrumented is just about to enter the operating system scheduler. Typically this will be just before some preemption takes place (either because of a new higher-priority task becoming ready, or because of an interrupt). In the case of an interrupt handler, this event would be placed as early as possible in the interrupt handler.

4.1.4 osTraceSchedulerExit

Usage:	`osTraceSchedulerExit()`
Description:	This is used to indicate that the system being instrumented is just about to exit the operating system scheduler.

4.2 Tasks/ISRs

For the Instrumenting Kit, Tasks and interrupt handlers (ISRs) are given the generic group name of *Tasks* since they share many characteristics. For this reason, it is important that the identifiers for tasks and ISRs do not overlap (e.g. a single system cannot contain both a task with an ID of 3 and an ISR with an ID of 3).

Profiles are logged using standard RTA-TRACE instrumenting calls – described in the *RTA-TRACE User Manual*.

4.2.1 osTraceTaskActivate

Usage:	`osTraceTaskActivate(<task_id>)`
Description:	This is used to indicate that task activation has been requested. In an OSEK system, this API will be placed as early as possible in (or just prior to) the call to `ActivateTask()`.
Note	`<task_id>` must refer to a task.

4.2.2 osTraceTaskStart

Usage:	`osTraceTaskStart(<task_id>)`
Description:	This indicates that the indicated task has started. For a non-OS based system, this might indicate the start of a significant code block for example.

4.2.3 osTraceTaskEnd

Usage:	`osTraceTaskEnd(<task_id>)`
Description:	This indicates that the indicated task has finished.

4.2.4 osTraceCat1Start

Usage:	`osTraceCat1Start(<isr1_id>)`
Description:	This indicates that the indicated category 1 ISR has started. This call should be inserted as early as possible in the interrupt handler.

4.2.5 osTraceCat1End

Usage:	`osTraceCat1End(<isr1_id>)`
Description:	This indicates that the indicated category 1 ISR has finished. This call should be inserted as late as possible in the interrupt handler.

4.2.6 osTraceCat2Start

Usage:	`osTraceCat2Start(<isr2_id>)`
Description:	This indicates that the indicated category 2 ISR has started. This call should be inserted as early as possible in the interrupt handler.

4.2.7 osTraceCat2End

Usage:	`osTraceCat2End(<isr2_id>)`
Description:	This indicates that the indicated category 2 ISR has finished. This call should be inserted as late as possible in the interrupt handler.

4.2.8 osTraceProcessStart

Usage: `osTraceProcessStart(<process_id>)`

`osTraceProcessStart(<task_id>)`

Description: This is used to indicate that a process has been started. A process is a sub-function contained within a task.

If `<process_id>` is not known, then the id of the task owning the process can be used and the visualizer will infer which process should be indicated, assuming that processes run in order.

For a non-OS system, this might be used to mark the start of a sub-function.

4.2.9 osTraceProcessEnd

Usage: `osTraceProcessEnd(<process_id>)`

`osTraceProcessEnd(<task_id>)`

Description: This is used to indicate that a process has ended.

It is permissible to omit process ends – RTA-TRACE will infer end of a process when the next process starts or the task ends.

4.3 Task/ISR Switching API

4.3.1 osTraceTaskSchedulerEntry

Usage: `osTraceTaskSchedulerEntry()`

Description: This is used to indicate that the task has offered up a re-scheduling point. For an OSEK system, this is a call to `Schedule()`.

In a co-operative scheduling system, this would be the point at which a task yields.

4.3.2 osTraceTaskSchedulerExit

Usage:	`osTraceTaskSchedulerExit()`
Description:	This is used to indicate that the re-scheduling point has returned to the calling task.

4.3.3 osTraceInterruptHandlerEntry

Usage:	`osTraceInterruptHandlerEntry()`
Description:	This is used to indicate that an interrupt has been recognized. There may be some time between this point and the start of the interrupt handler proper (logged with an `osTraceCat`*`n`*`Start()` call). This call should be placed as early as possible in the interrupt-recognition process, offering the opportunity to measure operating-system overheads.

4.3.4 osTraceInterruptHandlerExit

Usage:	`osTraceInterruptHandlerExit()`
Description:	This is used to indicate that an interrupt has been recognized. There may be some time between the end of the interrupt handler proper (logged with an `osTraceCat`*`n`*`End()` call) and this point. This call should be placed as late as possible in the interrupt-handler.

4.3.5 osTraceTaskSleep

Usage:	`osTraceTaskSleep(<task_id>)`
Description:	This is used to indicate that the task is being put to sleep – for example at the end of a timeslice. This event will be generated from within the operating system since a task should be unaware of any time-slicing that takes place. In a time-sliced OS, a Task would typically be started by the OS (logging a task-start event) and for each time-slice, a sleep/wake pair would be used.

4.3.6 osTraceTaskWake

Usage:	`osTraceTaskWake(<task_id>)`
Description:	This is used to indicate that a task is being woken after a sleep, i.e. a new time-slice has been allocated to the task.

4.4 Events

The following API calls refer to OSEK-style event behavior. Events are always defined as masks.

4.4.1 osTraceEventWaitEntry

Usage:	`osTraceEventWaitEntry(<task_id>, EventMaskType <event_mask>)`
Description:	This is used to indicate that the task is waiting for an OSEK event (`<event_mask>`) to be set. Once the event has been set, task execution will resume.

4.4.2 osTraceEventWaitExit

Usage:	`osTraceEventWaitExit(<task_id>)`
Description:	This is used to indicate that the task previously waiting for an OSEK event has been resumed.

4.4.3 osTraceEventSet

Usage:	`osTraceEventSet(<task_id>, EventMaskType<event_mask>)`
Description:	This is used to indicate that the event given by `<event_mask>` is being set for the task referenced by `<task_id>`.
Note	Any task can set an event for another task, but `<task_id>` must refer to the task for which the event is being set.

4.4.4 osTraceEventClear

Usage:

```
osTraceEventClear(<task_id>,
                  EventMaskType<event_mask>)
```

Description: This is used to indicate that the event given by `<event_mask>` is being cleared for the task referenced by `<task_id>`.

4.5 Resources

The following API calls refer to resources.

4.5.1 osTraceResourceGet

Usage:

```
osTraceResourceGet(<resource_id>)
```

Description: This is used to indicate that the resource `<resource_id>` is being locked.

4.5.2 osTraceResourceRelease

Usage:

```
osTraceResourceRelease(<resource_id>)
```

Description: This is used to indicate that the resource indicated is being unlocked.

4.6 Alarms and Counters

The following calls allow counters and alarms to be instrumented. These mechanisms are also used when instrumenting a schedule or timetable of programmed task activations.

4.6.1 osTraceCounterTick

Usage:

```
osTraceCounterTick(<counter_id>,
                TickType <counter_value>)
```

Description: This is used to indicate that the counter indicated is being ticked, along with its new value.

4.6.2 osTraceAlarmExpire

Usage:

```
osTraceAlarmExpire(<alarm_id>)
```

Description: This is used to indicate that the alarm indicated has expired.

4.7 Messages

These API calls allow messages to be tracked.

4.7.1 osTraceMessageSend

Usage:

```
osTraceMessageSend(<msg_id>)
osTraceMessageSendData(<msg_id>,
                       <data_ptr>,
                       <data_len>)
```

Description: This is used to indicate that the message indicated has been sent.

The second form of this call allows the actual message content to be logged as well as the message identifier.

4.7.2 osTraceMessageReceive

Usage:

```
osTraceMessageReceive(<msg_id>)
osTraceMessageReceiveData(<msg_id>,
                          <data_ptr>,
                          <data_len>)
```

Description: This is used to indicate that a receive-message call has been made for the indicated message.

The second form of this call allows the actual message content to be logged as well as the message identifier.

4.8 Interrupt manipulation

4.8.1 osTraceInterruptAllDisable

Usage: `osTraceInterruptAllDisable()`

Description: This is used to indicate that all interrupts are being disabled.

4.8.2 osTraceInterruptAllEnable

Usage: `osTraceInterruptAllEnable()`

Description: This is used to indicate that all interrupts are being enabled.

4.8.3 osTraceInterruptAllSuspend

Usage:

```
osTraceInterruptAllSuspend(
                  <nesting_count>)
```

Description: This indicates that all interrupts are being suspended. `<nesting_count>` will be zero when the interrupt level has been raised and a positive number otherwise. `<nesting_count>` increases by one for every 'SuspendAll' call.

4.8.4 osTraceInterruptAllResume

Usage:

```
osTraceInterruptAllResume(
                  <nesting_count>)
```

Description: This indicates that the interrupt level in place at the matching 'Suspend All' call is being resumed. `<nesting_count>` will be zero when the interrupt level has been lowered and a positive number otherwise. `<nesting_count>` decreases by one for every 'Resume All' call.

4.8.5 osTraceInterruptOSSuspend

Usage:	`osTraceInterruptOSSuspend(<nest_count>)`
Description:	This indicates that interrupts up to OS level are being suspended. `<nest_count>` will be zero when the interrupt level has been raised and a positive number otherwise. `<nest_count>` increases by one for every 'Suspend OS' call.

4.8.6 osTraceInterruptOSResume

Usage:	`osTraceInterruptOSResume(<nest_count>)`
Description:	This indicates that the interrupt level in place at the matching 'Suspend OS' call is being resumed. `<nest_count>` will be zero when the interrupt level has been lowered and a positive number otherwise. `<nest_count>` decreases by one for every 'Resume OS' call.

4.9 Error reporting

This API is used to instrument errors. It is suggested that an enumerated variable is used for error codes, allowing the visualizer to display meaningful text instead of a simple error code.

4.9.1 osTraceError

Usage:	`osTraceError(<error_code>)`
Description:	This indicates that an error has been reported. The display format for the error code depends on the `vs_ErrorFmt` attribute of the OS object within the .rta.file.

5 Instrumentation Part 2 – support API

In addition to the instrumentation API defined in section 4, there are some functions that are supplied as C code. These functions deal with the insertion of trace records into the trace-buffer, and trace-buffer management. The communication-elements of the RTA-TRACE target code are described separately in the *RTA-TRACE ECU Link Guide*.

The code should not require any modification since it is written in 'vanilla' C, although certain target specific code enhancements may be required (i.e. if placing data elements in 'near' RAM). Such modifications are beyond the scope of this document. Please contact LiveDevices if assistance is required.

In addition to the supplied support API, there is a requirement on the user to supply a number of target-specific macros/functions in order for the supplied support API to work correctly. These are described below.

5.1 Types used

Basic types are defined by the user in the file `RTLib\RTtarget.h`. This file is required to define the following:

Name	Description
`Int8Type`	A signed 8-bit integer.
`UInt8Type`	An unsigned 8-bit integer.
`Int16Type`	A signed 16-bit integer.
`UInt16Type`	An unsigned 16 bit integer.
`Int32Type`	A signed 32-bit integer.
`UInt32Type`	An unsigned 32 bit integer.
`BooleanType`	A Boolean.
`IntType`	A 'natural' integer for the platform
`UIntType`	An unsigned 'natural' integer for the platform.
`osTraceEventMaskType`	A type used for event bitmasks. If instrumenting an ECC OSEK OS, this will need to be defined as `EventMaskType`.
`osTraceTickType`	A type used for capturing counter-values. If instrumenting an OSEK OS, this will need to be defined as `TickType`.

Three basic types are used by the tracing library – `osTraceTimeType`, `osTraceInfoType` and `osTraceKindType`. The size of these types depends upon whether standard (16-bit) or compact (8-bit) identifiers, or standard (32-bit) or compact (16-bit) times are in use. The default operation is to use standard identifiers (16-bit) and times (32-bit).

Compact times will be used if the pre-processor symbol `COMPACT_TIME` is defined; compact identifiers will be used (for '`kind`' and '`info`' types) if the pre-processor symbol `COMPACT_ID` is defined. These definitions are contained within the file `RTconfig.h` (see Section 5.2).

This is shown in the following table:

Type Name	**Maps to**	
	Standard	**Compact**
`osTraceTimeType`	`UInt32Type`	`UInt16Type`
`osTraceKindType`	`UInt16Type`	`UInt8Type`
`osTraceInfoType`	`UInt16Type`	`UInt8Type`
`osTraceCategoriesType`	`UInt32Type`	
`osTraceClassesType`	`UInt16Type`	

Note that the standard/compact setting must match the info/kind sizes defined in the *Trace object* defined in the `.rta` file.

5.2 Target Configuration

Target configuration details are contained within the file `RTconfig.h`. This file is specific to your application – and hence should be in the application directory. This file contains configuration parameters which have an equivalent in the `.rta` file.

Note 1: Any changes made to this file will require that the library code be rebuilt.

Note 2: It is your responsibility to keep `RTconfig.h` and your application's `.rta` file consistent with each other (buffer size, compact time, and compact identifiers).

`OSTRACE_ENABLED`	Defining this symbol enables tracing on the target. If tracing is not required, then this symbol should not be defined.
`OSTRACEBUFFERSIZE`	The size of the trace-buffer (in records). Typically a buffer size of 200-600 records is required to provide sufficient contiguous data for proper system analysis.
`COMPACT_TIME`	If this symbol is defined, the compact representation of time is used (see 5.1), otherwise standard representation is used.
`COMPACT_ID`	If this symbol is defined, the compact representation of 'kind' and 'info' fields are used (see 5.1), otherwise standard sizes are used.
`OSTRACE_TRIGGERING_ENABLED`	If runtime triggering is required on the target, this symbol should be defined. Leaving this symbol undefined will result in a reduction in size of the tracing code, since runtime trigger checks will not be done.
`OSTRACE_USING_COMMLINK`	Define this symbol if an ECU Link is being used for transferring trace data from the target. Defining this symbol means that the files in the ...`\RTcomm\` directory will also need to be incorporated into your application. If it is not defined, the debugger interface will be used.

```
OSTRACE_ACTIVATIONS_FLTR
OSTRACE_TASKS_AND_ISRS_FLTR
OSTRACE_RESOURCES_FLTR
OSTRACE_PROCESSES_FLTR
OSTRACE_OSEKEVENTS_FLTR
OSTRACE_ERRORS_FLTR
OSTRACE_ALARMS_FLTR
OSTRACE_OSEK_MESSAGES_FLTR
OSTRACE_INTERRUPT_LOCKS_FLTR
OSTRACE_SWITCHING_OVERHEADS_FLTR
OSTRACE_STARTUP_AND_SHUTDOWN_FLTR
OSTRACE_TRACEPOINTS_FLTR
OSTRACE_TASK_TRACEPOINTS_FLTR
OSTRACE_INTERVALS_FLTR
```

Define tracing support for the various classes of events. All of these symbols must be defined to either:

`OSTRACE_ALWAYS`,

`OSTRACE_NEVER`,

`OSTRACE_RUNTIME`.

Runtime enabling/disabling of classes is demonstrated in the `demo_preemption()` routine of the example application (in `main.c`).

Note: In order to keep code size small, use `OSTRACE_ALWAYS` or `OSTRACE_NEVER` rather than `OSTRACE_RUNTIME` when tracing is not going to be altered at runtime.

5.3 osTraceGetSystemTime

Prototype:	`osTraceTimeType osTraceGetSystemTime(void);`
Description:	This function is responsible for supplying time values for all of the event-insertion code. Note that `osTraceTimeType` will be 16- or 32-bits in size depending upon the setting of the `COMPACT_TIME` symbol.

5.4 osTraceRunningTaskID

Prototype:

```
osTraceInfoType osTraceRunningTaskID (
                                void)
```

Description:

This function is responsible for supplying the trace identifier of the currently running task. This will reflect the task which currently occupies the CPU.

The function is only used by the LogTaskTracepoint(…) set of API calls to implement an aspect of run-time triggering. If run-time triggering is not being used, then this function can simply return 0. If task-tracepoints are not being used at all, the function can be omitted.

5.5 Interrupt manipulation macros

The OS Instrumenting kit uses a number of macros in order to manipulate interrupt levels as follows
NOTE re: recursion

Macro name:

`RESERVE_PREV()`

`SAVE_PREV()`

`DISABLE_INTRPTS()`

`RESTORE_PREV()`

Description:

`RESERVE_PREV()` declares a variable for a subsequent save of interrupt level

`SAVE_PREV()` saves the current interrupt level in the variable declared in the `RESERVE_PREV()` macro

`DISABLE_INTRPTS()` disables interrupts

`RESTORE_PREV()` restores interrupt level to that stored in `SAVE_PREV()`.

5.6 Other API calls

Other API calls (starting/stopping the trace, logging tracepoints, triggering, etc.) function as described in the *RTA-TRACE User Manual*.

6 Instrumenting hints

Many different embedded scenarios may be enhanced with the instrumenting kit – both with and without an operating system.

Some broad guidelines about instrumenting are given below. Please contact technical support if more assistance is required.

6.1 General hints

It is very important that 'paired' API calls (i.e. `osTraceTaskStart()` and `osTraceTaskEnd()`) are actually nested correctly. If the API call closing a section (i.e. `osTraceTaskEnd()`) is omitted, then RTA-TRACE will display increasing levels of pre-emption rather than execution of a background task (for example).

This applies to all entry/exit and start/end calls. i.e.:

```
void task_a(void)
{
  osTraceTaskStart(TASKA)
  if (on_button_pressed()) {
        switch_system_on();
  } else {
        osTraceTaskEnd(TASKA);
        return;
  }
  check_system_status();

  osTraceTaskEnd(TASKA);
  return;
}
```

Note that in the above example, both task-exit paths need to have the `osTraceTaskEnd()` function present.

6.2 Functional blocks

Pieces of code which form significant functional blocks (i.e. OSEK(time) tasks) should be modeled as task objects, with a `osTaskStart` (`osTaskEnd`) call placed as early (late) as possible in the block. If there is a particular activity that causes the block to be executed, it may be appropriate to mark it with a `osTaskActivate` call. A suitable task type (`vs_TYPE`) should be chosen in the `.rta` file – BCC1 is often the most suitable.

Processes can be used to represent functional blocks, with `osProcessStart` and `osProcessEnd` calls placed appropriately.

6.3 Periodic events

Periodic events may be modeled using counters and alarms. Typically, there is some ‘tick’ event which periodically causes some code to be executed. The ‘tick’ event may be modeled as a counter, whilst an alarm may be used to indicate the time at which a piece of code (or task) was activated.

For a time-sliced system, ‘Sleep’ and ‘Wake’ trace calls should be inserted as appropriate.

7 OS Instrumenting Kit Files

This section describes the files contained in the instrumenting kit library directory (`\RTLib\`), and the function(s) contained within each file.

7.1 RTapevnt.c

`osTraceAppendEventMask`	Append trace record plus Event-Mask to buffer (calls `osTraceAppendData`)

7.2 RTapnd.c

`osTraceAppend`	Append trace record to buffer (calls `osTraceWriteTraceRecord`)

7.3 RTapnddt.c

`osTraceAppendData`	Append trace record plus arbitrary data bytes to buffer (calls `osTraceWriteTraceRecord`)

7.4 RTapndvl.c

`osTraceAppendVal`	Append trace record plus integer data to buffer (calls `osTraceWriteTraceRecord`)

7.5 RTbef.c

`osTraceBufferEmptyFunction`	Called on buffer empty

7.6 RTbreak.c

`osTraceBreakLabel`	Debugger integration label: Breakpoint on this to detect buffer full when using debugger link rather than ECU Link.

7.7 RTbwcf.c

`osTraceBufferWrCheckFunction`	Called to check writes to buffer, typically on buffer full

7.8 RTctick.c

`osTraceCounterTick1`	Append trace record plus Counter Value to buffer (calls `osTraceAppendData`)

7.9 RTcto.c

`CheckTraceOutput`	See *RTA TRACE ECU Link Guide*

7.10 RTdata.c

This file contains OS Instrumenting Kit implementation variables. These are required for the kit to function correctly.

7.11 RTfin.c

`osTraceFinished`	Ensures finished trace data is correctly terminated

7.12 RTsetrep.c

`SetTraceRepeat`	See *RTA-TRACE User Manual*

7.13 RTsettrg.c

`SetTriggerConditions`	Support function for triggering

7.14 RTsetwin.c

`SetTriggerWindow`	Support function for triggering

7.15 RTstbt.c

`StartBurstingTrace`	See *RTA-TRACE User Manual*

7.16 RTstfr.c

`StartFreeRunningTrace`	See *RTA-TRACE User Manual*

7.17 RTstop.c

`StopTrace`	See *RTA-TRACE User Manual*

7.18 RTsttt.c

`StartTriggeringTrace`	See *RTA TRACE User Manual*

7.19 RTtrgsup.c

`osTraceCheckForTrigger`	Support functions for triggering
`TriggerNow`	

7.20 RTwrrec.c

`osTraceWriteTraceRecord`	Write a trace record to the buffer (not re-entrant)

8 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.

8.1 Rules

- 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.

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 whose 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.
`%%`	No conversion is carried out; emit a `%`
`%?`	Emit in the form of a hex dump.

8.2 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 RAINBOW`, using an enum class (see 3.3.1)	`"%1E"`	`Yellow`	The number `1` refers the ID of the enum class in the `ENUM` directives, not to the width of the field.

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.