RTA-TRACE

ECU Link Guide

Contact Details

ETAS Group

www.etasgroup.com

Germany

ETAS GmbH
Borsigstraße 14
70469 Stuttgart

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

www.etas.de

USA

ETAS Inc.
3021 Miller Road
Ann Arbor, MI 48103

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

www.etasinc.com

Japan

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

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

www.etas.co.jp

France

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

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

www.etas.fr

Korea

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

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

www.etas.co.kr

Great Britain

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

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

www.etas-uk.net

Copyright

Document TD00009-003

Contents

1 About this Manual

RTA-TRACE is a software logic analyzer for embedded systems. Coupled with a suitably enhanced application or operating system, it provides the embedded application developer with a unique set of services to assist in debugging and testing. 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 explains how to get trace data from the target hardware to the PC running RTA-TRACE. It covers the supplied byte- and block- serial communication libraries, implementing a custom communication layer, and running RTA-TRACE without a communication layer, using a debugger to retrieve the trace data.

This manual does not explain how to instrument application code or build the target application with tracing enabled. For this you should consult either the *RTA-TRACE Getting Started Guide* or the *RTA-TRACE User Manual.*

1.1 Who Should Read this Manual?

The *RTA-TRACE ECU Link Guide* is for the software engineer who has an application running on target hardware with tracing enabled, and who:

- wishes to transfer trace data to the host PC using one of the supplied Server Plug-ins;
- wishes to implement a custom ECU Link

It does not cover enabling tracing within the target application. For this, refer to the *RTA-TRACE Getting Started Guide*.

The reader should be familiar with C programming concepts for embedded systems and their chosen target, build environment and debugger.

1.2 Document 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.

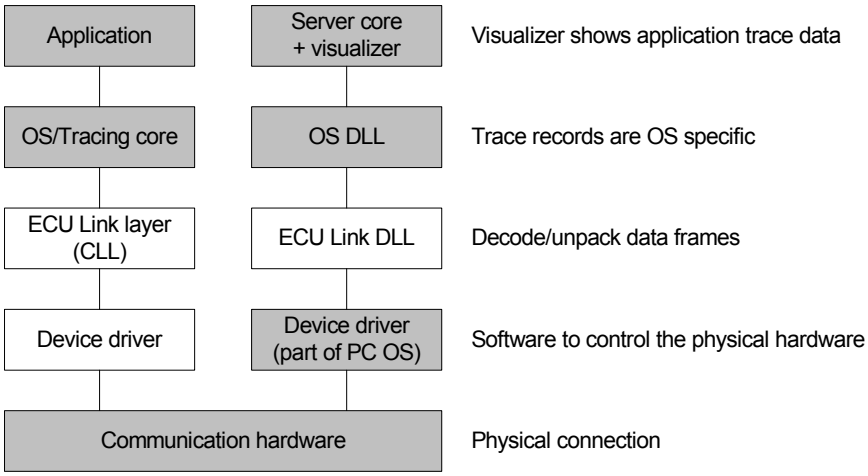
Program code, header file names, C type names, C functions and API call names appear in the `courier` typeface. When the name of an object is made available to the programmer the name also appears in the `courier` typeface, so, for example, a task named Task1 appears as a task handle called `Task1`.

Courier oblique is used for placeholders where the user should substitute relevant text, e.g. RTser*xxx* refers to both RTserbyt and RTserblk libraries.

When interaction with GUI elements is described, the elements' captions are shown in **bold**. Navigation of a hierarchy, such as a menu structure, is shown by separating the levels with a chevron, e.g. "Choose the **Edit > Select All** Menu item." Or "Choose **Edit > Select All**".

2 What is an ECU Link?

RTA-TRACE requires an *ECU Link* to transfer data from the target to the RTA-TRACE Server. The ECU Link can be implemented by any suitable hardware available on the application hardware. The general layering of the communications architecture is illustrated below:



In this reference model, the target code consists of two elements and the server code consists of one. The target's ECU Link Layer (ELL) and the server's ECU Link DLL encapsulate a common encoding/decoding scheme (for example inserting and removing frame delimiters and escaping certain characters). The ELL should be written in a target-independent manner and the Driver layer beneath it would then tie it to some specific hardware.

In practice, it may be that both link layer and device driver get written together as a single module of code. Whilst this may make it easier to write the first edition of the combined ELL & Driver, subsequent migration of the code to other platforms may be made much more difficult.

Supplied with RTA-TRACE are two example ECU Link technologies – *Serial (RS232)*, and *Debugger*.

2.1 What is the Serial (RS232) ECU Link?

The Serial ECU Link provided with RTA-TRACE provides the quickest way to get running for many applications. It comprises an ECU Link DLL (RS232) for the RTA-TRACE Server and a choice of ELL for the target hardware. The target libraries and Server DLL encapsulate a framing and escaping protocol suitable for transmitting trace data over a serial line.

The Serial ECU Link DLL connects to any hardware driver presenting itself as a COM port on the host PC. The RTA-TRACE server plug-in allows the baud rate and COM port to be configured to support the best data rate for the target hardware and connection medium. The COM port does not necessarily have to be a conventional RS232 line: for example, a USB Bluetooth transceiver that presents itself in Windows as a COM port would also be usable with the DLL.

The ELL libraries for the target hardware implement various functions which must be invoked by the application. The libraries also expect a device-driver to have been implemented which corresponds to the interface described in 4.6.

RTA-TRACE includes Driver source code for a reference target which may need to be tailored to meet specific target requirements.

Further details can be found in Section 4.

2.2 What is the Debugger ECU Link?

The user may select an In-Circuit Debugger (ICD) to extract trace data. The ELL and device driver functionality for the debugger ECU Link is provided by a set of debugger macros, rather than target code; this saves compiling and linking code to drive the communications hardware, and may be the only option if the target board has no native ECU Link.

Using the debugger can have advantages in certain applications: there is a reduced impact on the CPU time available to the application because trace data is transmitted while the CPU is stopped. However, stopping the CPU alters its behavior with respect to the outside world: interactions with peripheral devices, timers, or other devices connected via I/O may be adversely affected.

The Server plug-in reads trace data from the hard drive for display by the Client. A debugger macro must be written to extract the contents of the trace-buffer and write the data to a suitably named file. For certain combinations of operating system and debugger, these macros are provided by LiveDevices.

Further details can be found in section 3.

2.3 Choosing a Configuration

The most suitable ECU Link for a particular application will vary based on responsiveness, the volume of trace-data, and various other factors. In addition to the two ECU Links described above, it is also possible to create a custom ECU Link, with a corresponding increase in workload for the developers involved.

The available components are tabulated below:

DLL	**ELL**	**Driver**
DebugDump	None (Use Debugger)	Reference
RS232	Byte-serial/Block-serial	Reference/Custom
Custom	Byte-serial/Block-serial	Custom
Custom	Custom	Custom

Additional supported connections may be available separately for RTA-TRACE – contact LiveDevices for details of availability. This guide provides useful

background information for users of other ECU Links, but complete information for other links will be found in the respective documentation.

Information relevant to implementing a custom ELL/DLL can be found in section 5.

2.4 Configuring the RTA-TRACE Server

ECU Link DLLs appear as '*OS-ECU Link*' choices in the RTA-TRACE Client connection setup dialogue – the Serial ECU Link appears as '*OS*-RS232', e.g. "ERCOSek-RS232".

3 The Debugger ECU Link

The Debugger ECU Link uses disk files to transfer data between target and Server.

Note: For optimal performance RTA-TRACE Server should be running on the same computer as the debugger, since both the debugger and Server require access to the same directory in order to transfer data.

3.1 Using a Supported Debugger

LiveDevices supplies scripts for supported debuggers to automate the process of extracting trace data and transferring it ultimately to the Client. This section outlines their use.

Currently, Lauterbach and Crossview debuggers are supported.

3.1.1 Retrieving Trace Data

The debugger scripts convert the trace data captured on the target board into text files. These are written into the directory containing the Run-time Interface (`.rtp`, `.rta`, `.ort`) file describing the target application to RTA-TRACE.

Once the macros have been installed in the debugger session, and the target is running, text files will be written to the `cgen` directory which the RTA-TRACE Server will read and then delete.

Upon starting a new connection in the RTA-TRACE Client, choose the '*xxx*-Debugger' Interface. Locate the run-time interface file that was built with your application and the client will start.

RTA-TRACE Server then monitors the directory containing the run-time interface file for files dropped by the supplied debugger scripts and passes the contents to the Client for display.

3.2 Using an Unsupported Debugger

When using '*xxx*-Debugger' Interface as the ECU Link Interface, RTA-TRACE Server monitors the directory containing the run-time interface file for files dropped by the debugger, and passes the contents to the Client for display.

The transformation from the application's trace buffer to files suitable for RTA-TRACE Server is described below.

3.2.1 Extracting the Trace Buffer

At the target, the trace data is stored in the array `osTraceBuffer[]`. This is a circular buffer and the start position is pointed to by `osTraceBufferDataStart` (the end of the data being given by

osTraceBufferWrPtr). In order to stop the target application when the trace buffer fills, it is necessary to place a breakpoint at the location indicated by the label osTraceBreakLabel – it is necessary to then execute a debugger macro to generate a text-file containing the trace buffer data.

3.2.2 Buffer style 1 (CrossView)

One form of the text file to be generated is as follows:

```
osTraceBufferDataStart = <ds_addr>
osTraceBufferWrPtr = <wp_address>
<ds_addr>  = <dump of data as 4 32bit values>
<ds_addr+4>= <dump of data as 4 32bit values>
…
…
```

The generated text file should be given the name tracebuffer*n*.txt where *n* indicates the sequence number of the file.

For example, filename tracebuffer0.txt:

```
osTraceBufferDataStart = 0xd00002fc
osTraceBufferWrPtr = 0xd00002fc
0xd00002fc = 0           1704591476     3692928        1501587013
0xd000030c = 1740244481  100882816      1501587619     1789986308
…
…
```

3.2.3 Buffer Style 2 (Lauterbach)

The other supported text-file format is as follows:

```
B::v.v %symbol osTraceBufferDataStart osTraceBufferWrPtr
&osTraceBuffer[0] %tree.open osTraceBuffer
**osTraceBufferDataStart = <ds_addr> * osTraceBuffer[0]
**osTraceBufferWrPtr = <wp_address> * osTraceBuffer[100]+0x2
**&osTraceBuffer[0] = <buff_addr> * osTraceBuffer[0]
**osTraceBuffer = (
    1,
    1946157056,
    26857216,
    1,
    1946157056,
    56348416,
    2,
    1946157056,
    67489536,
    2,
…
)
```

The generated text file should be given the name tracebuffer0x*n*.txt where *n* indicates the sequence number of the file in hexadecimal.

For example, filename `tracebuffer0x0.txt`:

```
B::v.v %symbol osTraceBufferDataStart osTraceBufferWrPtr
&osTraceBuffer[0] %tree.open osTraceBuffer
**osTraceBufferDataStart = 0x00FF7350 * osTraceBuffer[0]
**osTraceBufferWrPtr = 0x00FF74E2 * osTraceBuffer[100]+0x2
**&osTraceBuffer[0] = 0x00FF7350 * osTraceBuffer[0]
**osTraceBuffer = (
    1,
    1946157056,
    26857216,
    1,
    1946157056,
    56348416,
    2,
    1946157056,
    67489536,
    2,
…
)
```

3.3 Other uses

The Debugger ECU Link may also be used in the case where a custom ELL is being implemented on the target and the burden of writing a DLL wants to be avoided. As an example, an ECU Link that transmits Ethernet frames may have been written. Ordinarily, a DLL would need to be written that will decode the Ethernet frames and pass them on to the OS DLL. Instead, a standalone executable could be written that receives Ethernet frames and writes files in the prescribed debugger format (see 3.2) – the Debugger DLL will then be able to decode the trace data.

3.4 Configuring the RTA-TRACE Server

The Debugger ECU Link requires no server configuration.

4 The Serial ECU Link

In order to use the Serial ECU Link, target code needs to be modified. The basic steps are as follows:

- Insert calls to `TraceCommInit()`, `UploadTraceData()` , and `CheckTraceOutput()` in the application
- Review the driver code (supplied as source)
- Link against the `RTserbyt` (or `RTserblk`) library

4.1.1 Insert calls to TraceCommInit() and UploadTraceData()

`TraceCommInit()` should be called as part of target initialization (i.e. before the operating system is started). In turn, it will call `osTraceCommInitTarget()` (see below).

This function is provided in the ELL. The prototype for the function is found in the header file `RTserial.h`.

Note: In RTA-OSEK, if the ECU Link is being autostarted there is no need to make a call to `TraceCommInit()`.

`UploadTraceData()` is responsible for transferring the records in the trace buffer to the communications hardware via the device-driver layer. Each call to `UploadTraceData()` may cause data to be emitted from the communications device.

`UploadTraceData()` should be called regularly from the lowest-priority task that does not get starved of CPU time; if it is called from too high a priority, it may affect system behavior.

4.1.2 Review Driver Code

The target-specific portion of the RS232 target code is supplied as source code. It is likely that you will want to make changes to some parts of the code. You should review the code paying special attention to the following:

- `osTraceCommInitTarget()` sets up the desired serial port with a suitable baud rate.
- `osTraceCommTxByte()` puts a single byte into the port's transmit register.
- `osTraceCommTxReady()` returns a `BooleanType` to indicate whether the hardware is ready to receive another byte for transmission

4.1.3 Link against the appropriate ELL

As supplied, the RS232 ECU Link is used with the byte-serial library (`RTserbyt`) which contains the `UploadTraceData()` function responsible for framing the trace data and calling `osTraceCommTxByte()` to transmit

the resulting sequence of bytes. Note that this should be linked *in addition* to the RTA-TRACE core library.

4.2 Generalized Use

The target software is flexible in its application. It supports interrupt-based handling of the I/O device, supports block-serial devices such as a buffered serial module, and is not limited to driving the serial port. Any medium is suitable if it can cause the host computer to receive data on a COM port (e.g. a Bluetooth device with a suitable driver).

Once your transmission medium is selected, the first question is whether you would prefer to work a byte at a time or in blocks of data. Working in byte-serial mode can mean a simpler development process on unfamiliar hardware, and may often result in good-enough performance. Block-serial mode will allow you to exploit any built-in buffering in your I/O peripheral for efficient implementation of the ECU Link.

Once this decision is made, you must implement the following functions:

- `osTraceCommInitTarget()`, to set up the I/O peripheral for transmitting, and set `osTracePacketMax` to the maximum packet size that can be carried by the ECU Link.
- `osTraceCommTxByte()` for byte-serial mode or `osTraceCommTxBlock()` for block-serial mode, to place a byte or block of data into the I/O peripheral's transmit buffer,
- `osTraceCommTxReady()` to test whether the I/O peripheral is ready to accept another transmission unit.

Optionally the following functions may be required:

- `osTraceCommTxStart()` to perform any actions required at the start of a transmission - such as enabling a transmit interrupt,
- `osTraceTxEnd()` to perform any actions required at the end of a transmission - such as disabling a transmit interrupt

More detail on all the library functions and user-supplied functions can be found in Sections 4.5 and 4.6.

4.3 Efficiency Issues

When instrumenting code to measure timing characteristics, it must be borne in mind that the measuring activity takes up time that would otherwise be available to the application. In a high-utilization situation care may be necessary to successfully trace an application without missing additional deadlines.

The RTA-TRACE Serial ECU Link can operate in either *interrupt* or *polling* mode. Interrupt mode prioritizes communication at the expense of the application's timing characteristics, whilst polling mode prioritizes the application's timing behavior at the possible risk of some loss of trace data. In general it is recommended to use polling mode and if necessary set target-side triggers and filters to generate a smaller volume of data.

4.3.1 Common Issues

Except when using the Debugger ECU Link, target code must periodically call `UploadTraceData()`. This function results in at most one unit (byte or block) of data being transmitted, so should be called sufficiently often to prevent trace buffer overflow. There is a balance to be struck between this and the requirement that it should not interfere with the normal operation of the software. It may be best to place a single call to `UploadTraceData()` in the lowest-priority task that gets sufficient share of the CPU, or it may be better to distribute calls between a background task and a higher-priority task that is intermittent or "bursty".

If RTA-TRACE is in free-running mode, the buffer will be uploaded when full – in order to upload data ahead of this (in an attempt to prevent the buffer becoming full, and hence losing data), the application may call `CheckTraceOutput()` regularly. If this is not called in a timely fashion, `UploadTraceData()` will not transmit any data, and RTA-TRACE will fill up the trace buffer then suspend tracing until `CheckTraceOutput()` is called when the buffer becomes full. `CheckTraceOutput()` has a short execution time so there is no significant overhead on the application if it gets called more frequently than strictly necessary – especially if it is being called from within an idle loop.

4.3.2 Polling mode

Polling mode is suitable for use with all trace modes and the byte-serial or block-serial communications library.

Note: With the core in Free Running mode, tracing continues while trace data is being transmitted to the host, so care must be taken to avoid affecting the runtime characteristics of the application.

In polling mode, calls to `UploadTraceData()` (and, if applicable, `CheckTraceOutput()`) are needed to cause each byte to be transmitted.

If there is data waiting, `UploadTraceData()` attempts to queue the byte for sending:

- The byte-serial library calls the user-supplied function `osTraceCommTxReady()` to check whether the serial interface can accept a character. If so, then it calls `osTraceTxByte()` with the byte to transmit passed as the parameter.
- The block-serial version places the byte into the block buffer and calls `osTraceCommTxReady()` when it is full. If the hardware is ready to accept a buffer, `osTraceTxBlock()` is called with the number of valid bytes in the buffer.

A typical background loop might look like this:

```
TASK(tskIdle)
{
  for(;;) {
    /* Check whether trace data is ready.  Only
needed if core is in free-running mode */
    CheckTraceOutput();

    /* perhaps output a character */
    UploadTraceData();
  }
}
```

Note: in order for data to be sent, it is important that the task responsible for calling `CheckTraceOutput()` and `UploadTraceData()` is not starved of CPU time – it may be necessary to call this function from a task which is not actually the lowest priority.

4.3.3 Interrupt mode

Using the byte-serial library, trace-data throughput can be optimized by using a user-supplied function and the serial module's 'Transmit Complete' interrupt to trigger transmission of a byte of data. This way, no call to `UploadTraceData()` is needed in the idle loop, and the transmission hardware is occupied as efficiently as possible.

Note: Since this may significantly affect the timing behavior of the system, it is not recommended to use interrupt transmission in free-running mode. In bursting and triggered modes, transmission takes place after trace recording has stopped. In free-running mode, RTA-TRACE may be recording the behavior of the system during transmission of trace data.

When trace data is ready for transmission in Bursting or Triggered mode, RTA-TRACE calls the user-supplied function `osTraceCommDataReady()` (see 4.6.2). This must call `UploadTraceData()` to start the transmission of the trace data.

When transmission begins and ends, RTA-TRACE calls `osTraceCommTxStart()`, and `osTraceCommTxEnd()` respectively (see 4.6.5 and 4.6.7 for details). Implement these to enable and disable the serial port's transmission-complete interrupt.

Finally, implement the interrupt handler. It should call `UploadTraceData()` to transmit the next data byte.

4.4 Configuring the RTA-TRACE Server

ECU Link DLLs appear as '*OS-ECU Link*' choices in the RTA-TRACE Client connection setup dialogue – the Serial ECU Link appears as '*OS*-RS232', e.g. "ERCOSek-RS232".

There are two ways to set up the baud rate and COM port for the Serial ECU Link: directly in the server, or "remotely" through the client. For either method you must have a Client-to-Server RS232 connection active.

- Server Method: Right-click the RTA-TRACE Server icon in the System Tray. Click the **Properties** menu item and choose the relevant *operatingsystem*-RS232 menu item. Changes to the settings take effect when you click **OK** to close the dialog box.
- Client Method: Choose **File** > **Configure Connection…** . A dialog appears showing the configurable options. Changes take effect when the **OK** button is clicked.

4.5 Reference 1: Library functions

4.5.1 TraceCommInit

```
osTraceStatusType TraceCommInit(void)
```

This function needs to be called at initialization time in order to initialize the trace library. It, in turn, makes a call to the user-supplied function `osTraceCommInitTarget()` defined below. The return value indicates the any error that occurred during the initialization process, including any errors from the call to `osTraceCommInitTarget()`.

4.5.2 CheckTraceOutput

```
void CheckTraceOutput(void)
```

This function checks the availability of trace data; it is not necessary to call this function when the core is in Bursting or Triggered mode, although it is not harmful to do so.

4.5.3 UploadTraceData

```
void UploadTraceData(void)
```

This function is responsible for sending individual bytes of trace data over the Serial ECU Link. It interacts with the hardware via the user-supplied functions defined below.

In polled mode, it is necessary to call this function frequently enough to ensure data is transmitted in a timely manner. As a special case, in interrupt mode using the byte-serial library, this function should be called from the user-supplied function `osTraceCommDataReady()` and the transmit-interrupt handler.

4.6 Reference 2: User-supplied functions

The following functions must be written for your specific hardware and linked with the trace-enabled application. Their prototypes are in the C-header file `RTserial.h`.

When using interrupt mode, it is also necessary to write a transmit-interrupt handler which calls `UploadTraceData()`.

4.6.1 osTraceCommInitTarget

`osTraceStatusType osTraceCommInitTarget(void)` *(mandatory)*

In this function you must set up the ECU Link hardware – bit rate, stop bits, data bits, etc. – and must set the variable `osTracePacketMax`. The server RS232 plug-in is expecting serial data as follows: 8 data bits, no parity, and 1 stop bit ("8n1").

If you wish to use interrupt mode, you should initialize, but not enable, the UART's "transmit complete" interrupt here.

If an error occurs during this function, it should be returned – it will in turn be returned to the caller of `TraceCommInit()`.

4.6.2 osTraceCommDataReady

`void osTraceCommDataReady(void)` *(optional)*

If you implement this function, it will be called when trace data is ready to transmit.

It is especially useful in interrupt mode with the byte-serial library to make the first call to `UploadTraceData()`. Because data is ready to send, this results in a call to `osTraceCommTxStart()`, and the first byte of the trace data being passed to `osTraceTxByte()`. The transmit interrupt handler also contains a call to `UploadTraceData()`, causing subsequent bytes of data to be transmitted.

In polling mode, or when linked against the block-serial library, this function is less likely to be useful and need not be implemented.

4.6.3 osTraceCommTxBlock

`void osTraceCommTxBlock(UIntType nbytes)` *(mandated by block-serial library)*

This is called when the serial ECU Link is ready to transmit a block of data: either the block is full or the end of the waiting data has been reached. The number of valid bytes is given by `nbytes`, which will never exceed the value given in `osTraceBlockSize`. You must ensure that only the valid bytes are transmitted as the rest of the buffer may contain frame-start markers which would result in corrupted data on the server.

Note: The function must not block. The serial ECU Link libraries call `osTraceCommReady()` to check whether the hardware is ready to accept data so this function will not be called unless user-supplied code has indicated it is able to transmit a data block.

The data block to be transmitted will be placed in the array `ByteType osTraceBlockBuffer[]`.

4.6.4 osTraceCommTxByte

`void osTraceCommTxByte(const ByteType c)` *(mandated by byte-serial library)*

This function is responsible for transmitting the single byte `c` over the ECU Link.

Note: The function must not block. The serial ECU Link libraries call `osTraceCommReady()` to check whether the hardware is ready to accept data so this function will not be called unless user-supplied code has indicated it can take a byte.

4.6.5 osTraceCommTxEnd

`void osTraceCommTxEnd(void)` *(optional)*

This function is called at the end of a batch of trace data. In interrupt mode (see 4.3.3) this may be used to disable the serial transmit interrupt.

Note: The function must not block.

4.6.6 osTraceCommTxReady

`BooleanType osTraceCommTxReady(void)` *(mandatory)*

This function is required to check whether the serial transmit buffer is available. If the transmit buffer is able to accept a byte or block (depending on selected library), the function returns a non-zero (TRUE) value.

For interrupt operation, this function can have an empty body.

Note: The function must not block.

4.6.7 osTraceCommTxStart

`void osTraceCommTxStart(void)` *(optional)*

This function is called at the start of a batch of trace data. In interrupt mode (see 4.3.3) this may be used to enable the serial transmit interrupt.

Note: The function must not block.

4.7 The RTA-TRACE Serial Protocol

Trace record Data is transmitted between target and Server in frames.

A frame contains a single leading identifier byte followed by up to `osTracePacketMax`[1] bytes of data.

The identifier byte is split into two 4-bit nibbles:

- The high nibble is used to identify the frame type, and can take values 0 to 14.Value 0 represents trace data.
- The low nibble is used to indicate the size of the individual records which make up the frame. The size of each record (1 to 16 bytes with 0 representing 16) is stored in the nibble. This is of particular use with trace data, because the width of a trace record is determined at build-time by configuration options.

The following optional frame protocol (OFP) is specified to cope with such cases. Its use is not mandatory, but will simplify code re-use at both target and PC.

4.7.1 Optional frame protocol

The block- and byte- serial target libraries implement a simple framing protocol. In ideal situations (i.e. with a packet-oriented ECU Link such as raw Ethernet or UDP) the output driver can simply transmit one frame at a time and not have to worry about issues such as signaling the start and end of a frame.

This is not possible for byte-oriented media such as RS232, or media such as CAN where the packet size is small.

An OFP frame comprises:

1. Start of frame marker byte (`0xFE`).
2. Frame identifier byte as described earlier.
3. Frame data, taken to be a sequence of unsigned chars. The 'unescaped' size is limited to `osTracePacketMax`

   Any data values >= `0xFE` that occur in the data are transmitted with a postfixed `0xFF` escape character. (i.e. `0xFE` is transmitted as `0xFE 0xFF`; `0xFF` is transmitted as `0xFF 0xFF`, neither of which can be misinterpreted as a start or end of frame).
4. End of frame marker (`0xFE 0xFE`).

Note that even if a receiver starts listening part-way through the OFP frame, no false 'start of frames' are possible. No checksums, CRCs or length fields are

[1] The value of `osTracePacketMax` depends on the ECU Link characteristics. It is initialized during `TraceCommInit()` via a callout to media-specific user code (contained in `osTraceCommInitTarget()`).

included because it is assumed (required) that loss of data on reception will be identifiable by the receiver.

5 Custom ECU Links

If the Serial or Debugger ECU Links are not suitable, it is possible to write a custom ECU Link. LiveDevices intends to release new ECU Link products as part of the continual product improvement process, as well as offering consultancy services to aid in authoring custom ECU Links.

5.1 Target ELL

An ELL is only required to implement a single function: `UploadTraceData()`.

5.1.1 UploadTraceData

`void UploadTraceData(void)`

This function is responsible for translating the raw trace buffer data into a form suitable for transmission via a device driver. If interaction with an existing RTA-TRACE device driver is required, then the interface functions described in 4.6 should be followed; however this is not mandatory.

Information about the trace buffer is contained in objects called *trace descriptors.* Trace descriptors are held in an array (`osTraceOutputDescriptors`), with element 0 being used for describing data to upload. A trace descriptor is defined as the following C structure:

```
typedef struct {
  ByteType         *ptr;       /* pointer to data */
  UIntType         count;      /* bytes in data */
  ByteType         id_byte;    /* size of 1 record */
  osTraceODState   state;      /* descriptor state */
}osTraceOutputDescriptor;
```

Where `osTraceODState` is defined as the following enumerated type:

```
typedef enum {
  osTraceODEmpty,
  osTraceODComplete,
  osTraceODReadyToTransmit,
  osTraceODBeforeIDByte,
  osTraceODInContent,
  osTraceODNeedEscapeChar,
  osTraceODFinal
}osTraceODState;
```

When `UploadTraceData()` is called it needs to check the state of the output desriptors (in the array `osTraceOutputDescriptors[]`) to check

whether it contains data which is ready to transmit. This is indicated by the `state` field being equal to `osTraceODReadyToTransmit`. If there is data ready, then it is the responsibility of `UploadTraceData()` to run a state machine out that will result in the data being transmitted.

In the supplied libraries, each pass through `UploadTraceData()` has a short execution time – rather than simply transmitting all of the data in a single burst, data is trickled out so as to have minimal impact on the running system. Once all of the data has been transmitted, the state field must be set to `osTraceODFinal`. This will be detected by the tracing core at the next call to `CheckTraceOutput()`, and more data may be made available.

If the ELL is being implemented over a technology which supports large packets of data (such as Ethernet), the value of `osTracePacketMax` set in `osTraceCommInitTarget()` will need to reflect this (so, for Ethernet, this may be set to 1536 rather than 65535). The size of the data block passed to the ELL will not exceed this size.

5.2 Server DLL

An ECU Link DLL is a Win32 DLL; it is required to decode any framing scheme implemented by the target ELL and to conform to a certain API (described below) that the RTA-TRACE Server expects.

Note: The DLL file name must take the form `'rtc*.dll'`.

Once the DLL has been built, it should be copied into the same directory as the RTA-TRACE Server executable – the next time the RTA-TRACE Server starts, it will recognize the DLL and be able to use it for communication.

If required, LiveDevices offer consultancy services to assist with the creation of custom communications DLLs.

5.2.1 General

Other than via `CommConfigure()` and `CommStatus()` below, an ECU Link DLL has no visual component and may not generate dialogs or windows that require user intervention. ECU Link DLLs can be written in any suitable language, provided that they are able to conform to the calling convention specified here, and support the functionality of the required API calls.

API calls provided by the DLL are exported using the C calling convention.

e.g. In C++ Builder and Visual C++, an exported function 'foo' is declared using the following form:

```
extern "C" __declspec(dllexport) int foo(int bar)
```

This causes the function to be exported with the name `_foo`[2].

For the purposes of this document the word EXPORT is used to represent this behavior.

Data types, constants and function prototypes are supplied in the file `rtcDLLInterface.h`, which must be used when building an ECU Link DLL.

The RTA-TRACE Server detects the presence of an ECU Link DLL by following the steps below:

1. It looks for files name `rtc*.dll` in the installation directory.
2. For each such DLL, it looks for the existence of all the required API calls.
3. For each conformant DLL, the DLL is registered and available for use.

5.2.2 CommIdentify

Prototype `EXPORT const char * CommIdentify(void)`

Description This function returns an ASCIIZ string that identifies the support provided by the DLL e.g. "RS232". The string returned occupies statically assigned memory owned by the DLL.

5.2.3 CommBind

Prototype

```
EXPORT const void * CommBind(
                              const char *name)
```

Parameters `name` is the ASCIIZ name of the OS DLL.

Description As shown in the diagram in section 2, an OS DLL connects to a target via an ECU Link DLL. This call tells an ECU Link DLL about each OS DLL that is trying to bind to it.

The ECU Link DLL, if possible, creates an internal 'instance' of an object that binds to the specific OS DLL. A handle is returned to the caller that can be used to reference this specific OS/ECU Link instance. The OS/ECU Link instance handle is subsequently used by many other API calls, detailed below.

An ECU Link DLL can expect to create an OS/ECU Link instance for each OS type supported on the system.

If the DLL cannot bind to the OS DLL, it returns NULL.

[2] It may be necessary to call the function `_foo()` with VC++.

5.2.4 CommUnbind

Prototype

```
EXPORT void CommUnbind(
                              const void *h)
```

Parameters `h` is an instance of the OS/ECU Link pair, returned from the previous call to `CommBind()`.

Description This instructs the DLL to free up any resources related to the bound OS/ECU Link instance. It is called as the ECU Link DLL is being unloaded.

5.2.5 CommConnect

Prototype

```
EXPORT const void *CommConnect(
                              const void *h,
                              const char *f,
                              const rtcInfo *i)
```

Parameters `h` is the OS/ECU Link instance handle.

`f` is an ASCIIZ string that gives the address of the OS configuration file currently selected. Its only valid use is to derive a 'working directory' for links such as debugger interfaces or situations where the directory contains other link-specific files.

`i` is a pointer to a data structure that describes the low-level characteristics that apply to the data received from the target. It is used by the DLL to unpack trace data byte-streams into 'who', 'what' and 'when' fields. It contains the following fields:

`IntSize`	The size of an 'int' on the target, in bytes
`TimeSize;`	The number of bytes in the 'when' field of the trace data
`IDSize;`	The number of bytes in the 'who' field of the trace data
`InfoSize;`	The number of bytes in the 'what' field of the trace data
`MaxAbsTime;`	The maximum value of 'when' that can be recorded on the target
`BigEndian;`	Non-zero if the target data is big-endian.

Description An ECU Link DLL may be able to support several different independent physical links ('channels') at a time. e.g. COM1 and COM2. An OS DLL performs a 'connect' operation to open each new communications channel.

The DLL must if possible create a channel instance that will deal with the connection. A handle for the instance is returned, used by subsequent API calls that require a channel handle.

If the connection cannot be made, NULL is returned.

5.2.6 CommDisconnect

Prototype

```
EXPORT void CommDisconnect(
                            const void *h,
                            const void *chan)
```

Parameters `h` is the OS/ECU Link instance handle.

`chan` is the channel instance handle.

Description `CommDisconnect()` is called when a channel is no longer needed.

The DLL deletes the channel instance identified by `chan` and any references to it. Any communications resources belonging to the channel are released.

5.2.7 CommOpen

Prototype

```
EXPORT BOOL CommOpen(
                            const void *h,
                            const void *chan,
                            const char *conf)
```

Parameters `h` is the OS/ECU Link instance handle.

`chan` is the channel instance handle.

`conf` is an ASCIIZ string containing the configuration information to be applied to the connection. It is in the same format as that expected by `CommSetConfigOptions()`, being derived from previously set defaults or a previous `CommGetConfig()`. The string may be empty, in which case appropriate defaults should be taken.

Description CommOpen is called to instruct the DLL to open the physical link to the target, in preparation for receiving and decoding

trace data.

The call returns `TRUE` if the physical link was opened successfully.

5.2.8 CommClose

Prototype

```
EXPORT void CommClose(
                          const void *h,
                          const void *chan)
```

Parameters `h` is the OS/ECU Link instance handle.

`chan` is the channel instance handle.

Description `CommClose()` is called when the physical link can be closed because no further trace data collection is required.

The DLL releases any resources claimed during the corresponding `CommOpen()`.

5.2.9 CommListen

Prototype

```
EXPORT void CommListen(
                          const void *h,
                          const void *chan,
                          CommTraceCallback cb,
                          const void *inst,
                          const void *prj,
                          BOOL *finished)
```

Parameters `h` is the OS/ECU Link instance handle.

`chan` is the channel instance handle.

`inst` is a value private to the calling OS DLL that must be passed back to it as the first parameter of the callback function above.

`prj` is a value private to the calling OS DLL that must be passed back to it as the second parameter of the callback function above.

`finished` is a pointer to a flag that gets 'set' to indicate that tracing is no longer needed and that the call can terminate.

`cb` is a pointer to a callback function that is called to send the trace data records up to the OS DLL. The function is

described by:

```
typedef BOOL (*CommTraceCallback)( const
void *inst, const void *prj, const rtcData
*d, int num);
```

The trace data records are passed as an array with length num and type `rtcData`. This type contains the following fields:

`when`	A timestamp for this event.
`what`	The type of event.
`who`	The subject of the event.

Description When a trace client requests trace data from a selected target that is not already supplying trace data, RTA-TRACE Server creates a new thread and calls the OS DLL's `ActivateTrace()` function. This in turn calls the ECU Link DLL's `CommListen()` function.

The ECU Link DLL is now responsible for gathering trace records from the target, repackaging them into raw 'who' , 'what' and 'when' form and passing them via a callback function to the OS DLL for decoding.

The DLL remains in the `CommListen()` function feeding data up the line the 'finished' flag becomes non-zero, or an error occurs.

If the DLL detects that trace data sent from the target has been lost, it signals this to the OS DLL by passing a record through the callback with zero values for 'who', 'what' and 'when'. It must then stop uploading trace data until it has re-established reliable communication.

The raw data received from the target is decoded based on the field sizes and endianness indicated in the `rtcInfo` data specified during `CommConnect()`.

Note: swapping of byte order must be performed for all big-endian targets.

5.2.10 CommStatus

Prototype

```
EXPORT void CommStatus(
                              const void *h)
```

Parameters `h` is the OS/ECU Link instance handle.

Description ECU Link DLLs must provide a simple modeless dialog that shows the state of the specified OS/ECU Link instance. Each instance may be supporting zero, one or more active connections.

As a minimum, the number of active connections, the rate of data transfer and the tracing state is shown.

Diagnostic text should also be available to help resolve communication issues.

This data should be updated automatically.

Only one instance of the status dialog is allowed at a time, regardless of the number of times that `CommStatus()` is called.

The status dialog closes when the DLL closes, or when closed by the user.

5.2.11 CommConfigure

Prototype

```
EXPORT void CommConfigure(
                              const void *h)
```

Parameters `h` is the OS/ECU Link instance handle.

Description ECU Link DLLs may require some user-configuration to tailor their operation to a particular system. (e.g. baud-rate selection).

A DLL must implement a modal configuration dialog (even if it says 'nothing to configure'), that gets run when `CommConfigure()` is called. In the general case, there will be a 'default' configuration setting, plus the option to reconfigure 'live' any active connections.

The DLL must show its configuration dialog and use the registry to store the resulting configuration settings before returning from this call.

5.2.12 CommGetConfig

Prototype

```
EXPORT const char *CommGetConfig(
                          const void *h,
                          const void *chan)
```

Parameters

`h` is the OS/ECU Link instance handle.

`chan` is the channel instance handle.

Description

This call is used by RTA-TRACE Server to determine the active configuration information for a specific active OS/ECU Link instance. This information can be passed to a remote RTA-TRACE client.

The OS DLL instructs the ECU Link DLL to return the current configuration data by calling `CommGetConfig()`.

The returned value is an ASCIIZ string residing in memory allocated using the Win32 call `GlobalAlloc(GMEM_FIXED,-)`.

The allocated memory gets freed by the caller.

The string comprises zero or more 'lines' of data, separated by a carriage-return, linefeed pair (`CRLF`).

Each line has the form:

`<name>=<value>`

(e.g. `Port=COM1`)

Where `<name>` represents the name of the configuration option and `<value>` its value.

The set of data shipped via `CommGetConfig()` must encompass all of the elements that are required by `CommSetConfig()` for it to completely configure the ECU Link. That is, it provides a complete description of the configurable items.

5.2.13 CommGetConfigOptions

Prototype

```
EXPORT const char *CommGetConfigOptions(
                          const void *h,
                          const void *chan)
```

Parameters

`h` is the OS/ECU Link instance handle.

`chan` is the channel instance handle.

Description

This call is used by RTA-TRACE Server to determine the possible configuration options for a specific active OS/ECU Link instance. This information can be passed to a remote RTA-TRACE client.

The OS DLL instructs the ECU Link DLL to return the configuration options by calling `CommGetConfigOptions()`.

The returned value is an ASCIIZ string residing in memory allocated using the Win32 call `GlobalAlloc(GMEM_FIXED,-)`.

The allocated memory gets freed by the caller.

The string comprises zero or more 'lines' of data, separated by `CRLF`.

Each line has the form:

`<name>=<opt1>,<opt2>,<opt3>...`

(e.g. `Port=COM1,&COM2,COM3`)

Where `<name>` represents the name of the configuration option and the `<opt>` elements to the right of the '=' represent the possible values of the option. The current value of the option is identified by being prefixed with a '&' character.

The set of data shipped via `CommGetConfigOptions()` must encompass all of the elements that the user can configure.

5.2.14 CommSetConfigOptions

Prototype

```
EXPORT const char *CommSetConfigOptions(
                          const void *h,
                          const void *chan,
                          const char *opt)
```

Parameters

`h` is the OS/ECU Link instance handle.

`chan` is the channel instance handle.

`opt` points to an ASCIIZ string in the format specified above for `CommGetConfig()`.

Description

This call is used by the RTA-TRACE Server to change the configuration data for a specific active OS/ECU Link instance. This information can be passed from a remote RTA-TRACE client.

The OS DLL instructs the ECU Link DLL to set the configuration data by calling `CommSetConfigOptions()`.

The returned value is an ASCIIZ string residing in memory allocated using the Win32 call `GlobalAlloc(GMEM_FIXED,-)`.

The string contains any error information to be sent back to the caller, or an empty string if the call succeeded.

5.2.15 CommRegisterDiagCallback

Prototype

```
EXPORT void CommRegisterDiagCallback(
                            void *h,
                            DiagCallback cb,
                            const void *connID)
```

Parameters

`h` is the OS/ECU Link instance handle;

`connID` is an identifier passed in from the OS DLL; the meaning of which will be specific to the particular OS.

`cb` is a callback handle where `DiagCallback` is defined as:

```
typedef void (*DiagCallback)(const void
*connID, const char *txt);
```

Description

RTA-Server does not normally have a visible window. However there is a diagnostic mode in which text can be displayed. This mode can be set via the "Server Options" dialog which is invoked via Properties->Server on the RTA-TRACE Server popup menu. An ECU Link DLL can add text to this window via a callback function that is passed to it via the CommRegisterDiagCallback API.

Once `CommRegisterDiagCallback` has been called, the DLL can call through the callback, passing ASCIIZ text `txt` to be displayed. `connID` must match the value supplied in `CommRegisterDiagCallback`.

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.