RTEdbg data logging and tracing toolkit

for **R**eal **T**ime **E**mbedded System **D**e**b**u**g**ging and Testing

User Manual

For RTEdbg data logging library v1.00 and RTEmsg v1.00

Branko Premzel 2024-12-16

Table of contents

1 INTRODUCTION AND KEY FEATURES

Embedded systems are becoming more complex, interconnected, and exposed. Most real-time systems cannot be stopped because either the conditions would change after a restart (producing different results) or loss of control could cause damage. New data is typically generated much faster than anyone can observe, and the true cause of an individual event may be milliseconds or minutes before a critical event that leads to failure. Even if the system can be stopped to inspect the variables, the programmer has no history. System optimization is not possible without access to real-time data. Only what can be observed and measured can be improved.

The *RTEdbg* toolkit allows not only diagnostics, but also optimization of the firmware and the entire application. The toolkit presented here includes a data logging library for minimally intrusive code instrumentation and tools for flexible data decoding and sorting on the host side. Code instrumentation provides much more detailed insight into firmware execution than variable sampling with debug probes or similar asynchronous methods.

Notes:

1. In this manual, the term **message** is used to refer to a record (any type of data or data set) written to the circular buffer with a call to a data logging function.
2. Many links have been included in this document to quickly jump to a more detailed description of each feature. All of these links to other sections and external information are highlighted in **blue**.

This document assumes that you are familiar with

- The C and/or C++ programming language
- Embedded system firmware development
- Target processor core
- Windows command line

1.1 Foreword

This guide shows how to use the toolkit to test and debug software in real time. It is a description of the library and the toolkit, and a collection of ideas that came up during development. Browse through this manual to get a feel for what the toolkit can do. For a basic understanding and to get started, read the INTRODUCTION AND KEY FEATURES and the GETTING STARTED GUIDE.

The main design requirement for the new solution was that the data logging should have as little impact on system timing as possible. Data logging in a low-priority function should not interfere with or delay the execution of higher-priority tasks or interrupts. Real-time system execution should never be halted when the data logging buffer is full. For this reason, blocking functionality has not been implemented for the data logging functions, as this could be dangerous, especially when considering functional safety requirements. Data logging must also continue normally after a fatal exception, restart with watchdog reset, or other similar conditions. This solution is also suitable for systems with small amounts of RAM, as minimally 32 bits of data are logged per event. The footprint of the logging library is small because the data is recorded in binary format. There are no specific logging functions for different data types. They differ only in the amount of data they record.

Practical examples and some demo projects are included to get you started. The software community is also invited to contribute things like code testing and review, suggestions for improvements, integration with other tools, improvement or addition of new solutions, tools for streaming mode data transfer, contributions to the quality and understandability of the documentation, etc. The toolkit enables dynamic code analysis as well as analysis of the entire system in which the embedded system is integrated.

1.2 Disadvantages of Current Solutions

Many embedded system programmers use printf for firmware instrumentation. Data decoding is done inside the target system, which requires a lot of CPU time, stack space, and program memory for printf – including floating-point arithmetic library if the CPU core does not have hardware support for it. The printf strings also consume a lot of memory. The printf-style instrumentation is poorly suited for multitasking applications because it requires a semaphore to access a serial port or other channel to transfer data to the host.

A review of existing logging and data analysis tools has revealed many shortcomings, such as

1. Inflexible data format designed primarily to log events, not large amounts of other data of various types required for embedded system optimization and testing.
2. The printf-style strings are usually decoded offline, but the format string must be copied to the circular buffer. This consumes CPU time and circular buffer space. Such formatting typically has limited functionality compared to the standard library printf function implementation.
3. Relatively slow code execution and large stack usage
4. It is not possible to log and decode bitfields or packed structures to reduce circular buffer usage.
5. Some solutions are optimized for RTOS-based systems and not for real-time control systems.
6. Message filtering in the embedded firmware is either not supported or has a limited set of features.
7. Simple sorting of important data into separate output files is not possible.
8. Steep learning curve (number of functions about 250 in the most complex solution).
9. The firmware cannot directly influence the data logging – for example, to stop data sampling when a critical error is detected, or to start single shot data logging when a special event occurs.
10. Commercial solutions are expensive and/or tied to specific hardware (e.g., debug probes) or software.

1.3 Main Features of the RTEdbg Toolkit

The *RTEdbg* toolkit is designed to solve most of the problems of current solutions and to improve the debugging of embedded systems. The data logging library allows easy instrumentation of C and C++ firmware. It provides insight into the dynamic execution of a real-time application with fast and time-deterministic event and data logging capabilities. Data logging enables testing and debugging either in the lab or after systems are deployed in the field. This solution can be thought of as a user-configurable, time-stamped *fprintf* function that runs on the host rather than in the embedded system. The printf-style format strings reside on the host computer, not in the embedded system's memory. Embedded system programmers need not worry about the file system, *fprintf* functionality, or other details because they are not needed by the embedded system equipped with this library.

The new data logging/tracing solution is not a replacement for existing event-based solutions such as SystemView or Tracealyzer, as it is based on a different concept. Both of these solutions use data tagging so that their decoding tools can properly process and display the data. Large number of functions makes it complicated to use. The *RTEdbg* toolkit provides minimally intrusive data tagging and flexible data decoding. Typically only 35 CPU cycles and 4 bytes of stack are required to log an event on a device with a Cortex-M4 core. Nearly 200 CPU cycles and a maximum of 150 to 510 bytes of stack are required to log an event with SystemView.

To get an idea of where and how the data logging library and tools can be used, see the features listed below:

- **Fast execution:** The data logging functions are optimized for the fastest execution. Events can be logged in 27 CPU cycles on a chip with a Cortex-M7 core (assuming zero memory latency). The load for logging 10,000 events per second is about 0.05% on a Cortex-M7-based CPU running at 600 MHz or about 0.2% on a Cortex-M4 CPU at 200 MHz. It goes even faster – see Optimize Data Logging Code Size and Speed.

- **Low stack usage** virtually eliminates the possibility of stack overflows after the code is instrumented. This makes the solution suitable for integration into existing projects (e.g., to analyze complex, difficult-to-reproduce problems or for reverse engineering poorly documented code). Typically only four bytes of stack are required to log a simple event. See Execution Times, Circular Buffer, and Stack Usage.
- **Low RAM usage:** Only 32 bits of data memory are required to log an event. Format ID, timestamp, and 0 to 8 bits of data are packed into a 32-bit word. **Support for bitfields and packed data** structures enables efficient data logging in embedded systems with small amounts of RAM, or more logged messages in the same amount of RAM dedicated to data logging.
- **The timestamp is part of each log message.** Each message is also an event marker and can be used to measure the time between events and include them in the statistics to quickly identify extremes.
- **Non-blocking, time-deterministic and reentrant functions:** Interrupts are never disabled if the CPU core supports mutex instructions (exclusive Read-Modify-Write). Functions never block execution if, for example, the data logging buffer is full. **Note:** If the processor does not support mutex commands (atomic operations), and memory reservation is done by disabling interrupts, then the solution is not reentrant for non-maskable interrupt and exception handlers.
- **Powerful data and event logging:** Any atomic or complex data type can be logged (from bitfields to structures, buffers, strings, etc.). Data logging functions do not distinguish data type, only data size.
- **Powerful binary data decoding** using fprintf style functionality. The same value can be printed to one or more output files. The output data format can be anything from plain text (ASCII, UTF-8), CSV, JSON, binary, etc. This allows different types of data to be sorted into separate files in either a human-readable format or a machine-friendly format for data post-processing (e.g., for automated data check during software regression testing).
- **Data sorting:** Sorting the logged data into separate files during decoding is important. Continuous data logging creates a flood of data. It is difficult to find critical events in very large log files. It is much easier to notice the outstanding ones in a smaller file and then look at the whole picture in the main log file for causes and consequences. Data can be sorted into files containing errors, warnings, mode changes, RTOS events, interrupts, commands received over the communication interface or from operator, etc. Very important data can be written to multiple files to reduce the chance of it being missed by testers.
- **Powerful message filtering:** Because many data logging calls can be integrated into the firmware, this can create a flood of messages that exceeds the available data transfer bandwidth. Simple and fast message filtering for up to 32 message groups is implemented. Logging of different message groups can be enabled even after the system is deployed in the field. There is no need to recompile and reload the firmware to enable a smaller number of messages if the communication path has insufficient bandwidth or the logging buffer size is limited.
- **Custom triggers** can be easily implemented in the firmware. Data logging can be enabled by setting the message filter to a non-zero value, or disabled (paused) by setting it to zero. The logging can be limited to the most relevant data, and the circular buffer can be used in the most efficient way possible.
- **Low compile time overhead** due to fast pre-build format header file processing with the *RTEmsg* application. It automatically assigns format IDs and filter numbers, and checks the format definition syntax.
- **Simple integration into the project:** Only a C source and a few header files need to be added to the project. Only one initialization function needs to be called before data logging can begin.
- **Ease of use:** Small set of data logging macros and functions that are easy to learn. Data format definitions using a standard printf string syntax in C header files define how the logged data should be printed.
- **Suitable for all types of embedded systems:** *RTEdbg* toolkit is not only suitable for large RTOS-based applications, but also for small embedded systems due to its low program memory and RAM consumption.

- **Small overall code footprint:** No printf or similar functionality is required for the embedded system firmware. Only a small number of simple macros and functions are required because the data logging functions only distinguish between data sizes, not data types. See Resources Needed for Message Logging.
- **Designed with functional safety in mind:** statically allocated data logging structure, robust logging library design, code execution is never halted when the trace buffer is full, fast execution, predictable maximum execution time of data logging functions, and low stack usage.
- **No data format strings in firmware code and circular buffer:** All fprintf-style data format strings reside only on the host computer.
- **Take snapshots without stopping the application:** When the message filter value is set to zero, either by the firmware itself or by a data transfer tool, message logging is temporarily disabled without affecting firmware execution. This can be used to temporarily disable data logging and transfer the data from a circular buffer to the host computer (or write it to non-volatile memory such as an SD card) for analysis.
- **Powerful 'post-mortem' data logging** allows not only in-house streaming or single shot testing, but also in-field 'post-mortem' debugging of all types of code, from RTOS tasks to interrupt/fault handlers. Data logging has been optimized for cases where it is most important to have all data in the buffer for the time just before an error occurred that stopped the system or detected a serious problem. It is possible to transfer logged data from an unattended embedded system to a host computer when maintenance personnel connect via a wired connection or remotely (e.g., for IoT systems).
- **Streaming (continuous) data logging** without data loss is possible if the throughput of the data transfer interface is sufficient and the data transfer software on the host computer supports it.
- **Single-shot mode data logging** allows data from an event to be logged until the buffer is full. The firmware (or debugger) can enable logging by setting the message filter value to a non-zero value after the event is detected – e.g., a value exceeds the predefined limit. The logging mode ('port-mortem' or 'single shot') can be selected via debug probe or firmware.
- **Easy integration into RTOS –** data can be printed in a format suitable for analysis with event analysis tools (e.g., Best Trace Format – BTF).
- **Designed to remain in production code** due to its small memory footprint, deterministic execution, and low execution time penalty. This enables 'post-mortem' or on-demand data logging and analysis (e.g., for IoT systems).
- **Powerful statistics:** Minimum/maximum/average values of selected data and time values are calculated and reported without the need to log additional *EventStart* and *EventStop* markers.
- **Easy data retrieval from the embedded system:** All logged data is stored in a static data structure and can be easily retrieved using a debug probe application or firmware.
- **Fast decoding of binary data:** Decoding (printing) binary data with the decoding application is very fast because the format definitions are pre-compiled and not interpreted.
- **Can run in parallel with the debugger:** Data transfer via debug link (SWD, JTAG) can be performed in parallel with the debugger if the debug probe driver supports shared operation.
- **No special debugging/tracing hardware required.** Any communication interface can be used to transfer the logged data to the host.
- **Data logging library designed for portability.**
- **Ready to use for ARM Cortex-M devices and other little-endian 32-bit CPU devices.**
- **Open source solution:** The complete code for the data acquisition library and tools running on the host computer are open source and not tied to any software (IDE, compiler, libraries) or debug probes.

1.4 How do the RTEdbg Data Logging Functions and Tools Work?

Programmers of real-time embedded systems often implement custom data acquisition solutions. Such methods typically require custom software running on the host computer. Usually, only the most important real-time control data and variables are logged (no events, exception data, error information, etc.). This allows programmers to optimize the main control loops, but does not allow them to search for complex errors. The *RTEdbg* library and tools enable logging and decoding of all kinds of data and events. Programmers can focus on application design/test and not on how to build and integrate debugging support for insight into the embedded system or how to write their own host-based tools.

The schematics on the following page show how the *RTEdbg* library is instrumented with calls to the data logging functions. These functions log data (messages) to the circular buffer in RAM. They are reentrant and non-blocking and can be called from anywhere in the firmware, including the RTOS kernel and exception handlers. Data logging is fast because the data is logged as raw binary values. All messages contain timestamps that can be used to calculate and print the time between events (and detect extremes as well). In 'post-mortem' logging mode the logging buffer is used as a circular buffer. In single shot mode, the same buffer is used in a linear fashion.

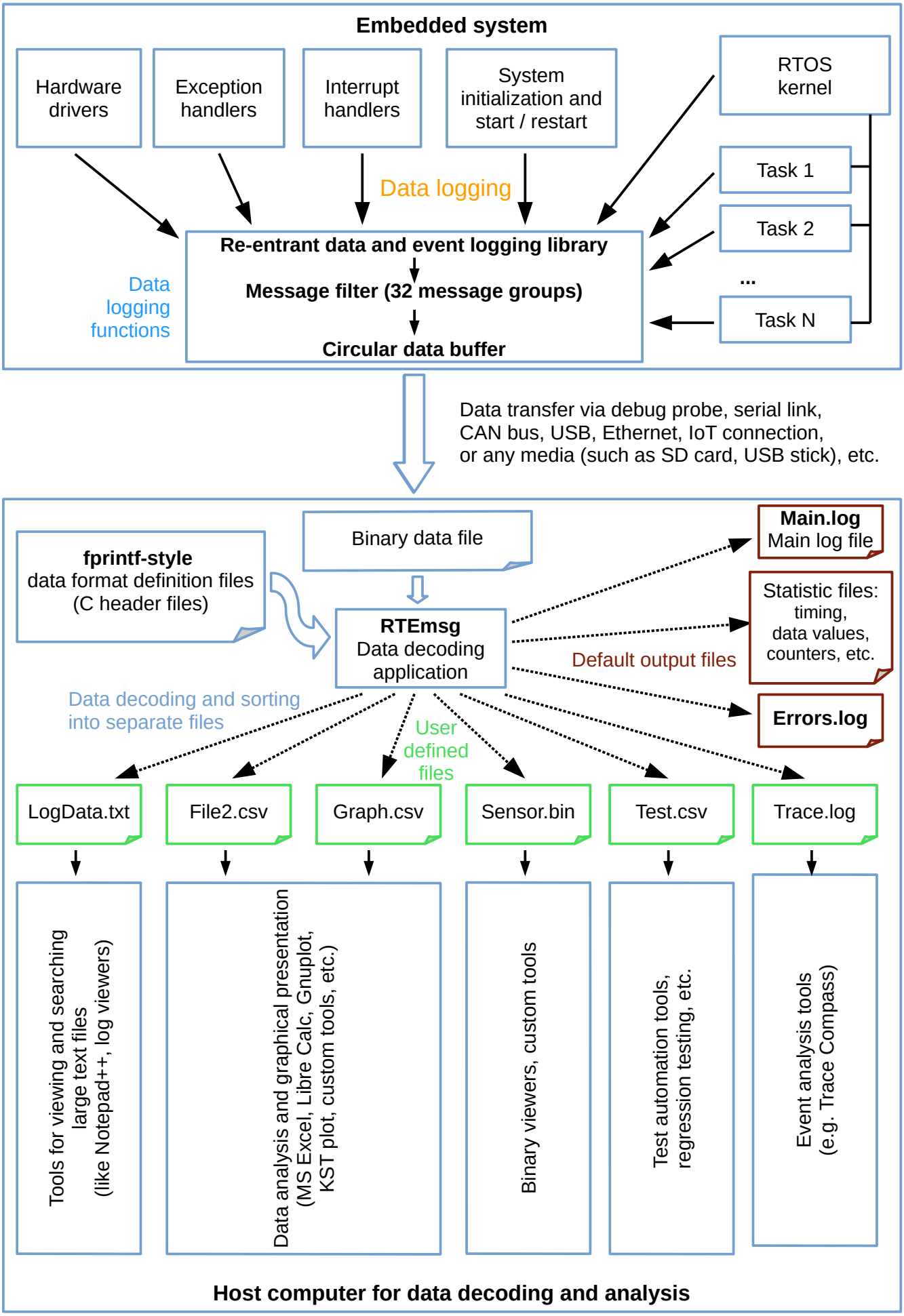
For each message type, a logging function parameter defines which bit of the 32-bit filter enables logging of that message. If a message with a particular filter number is enabled, it is written to the circular buffer, otherwise it is discarded. If the message filter variable is set to zero, logging can be temporarily halted – for example, to send snapshot data to the host computer or to preserve logged data until someone connects to the embedded system physically or via a wireless connection. Filtering allows the selection of relevant message groups for a particular test when the bandwidth to the host is too low for all messages or the circular buffer is small. The message filter value can be changed during data logging by the firmware or a debug probe.

Data can be transferred to the host computer via any communication interface or medium – from debugging probes to serial communication channels to SD cards. Because data is logged in raw binary format, bandwidth requirements are low.

The *RTEmsg* application performs binary data printing (decoding). This tool is currently available for Windows only. The logged messages are decoded using formatting definitions that include powerful data sorting capabilities. Real-time and non-real-time systems generate a flood of data. It is important to sort this data to quickly identify potential problems. The information is sorted into custom files using fprintf-style formatting definitions (*User defined files* in the schematic) and into default output files that include powerful statistics (*Default output files* in the schematic). The decoded information is printed to the default Main.log file, unless otherwise specified. This may be the only log file for small projects. If errors are found in the binary file or format definition file(s), they are reported in the Errors.log file. A message can contain multiple separate values. Each value from a single message can be written to one or more files in any format.

Data visualization and analysis is performed using existing tools – from log file viewers to tools for CSV file analysis or graphical display and event analysis. See the list of Data Visualization and Analysis Tools. Some basic data analysis support, such as value and timing statistics, is built into the RTEmsg message decoding tool. For all other purposes, external tools must be used.

The toolkit is not only useful for system optimization, but also for finding the root cause of complex errors that are difficult to reproduce. Good insight into system operation allows identification of bottlenecks, optimization of software functions and complete systems. It can be used for fast event data logging or analysis of failures that occur in the field ('post-mortem' data logging).



Schematic of the RTEdbg Toolkit

2 GETTING STARTED GUIDE

This chapter explains the basic features. More detailed descriptions of binary data logging functions, data decoding tools, format definitions, and more are provided in the following chapters. The short version is intended for quick learning and immediate practical use of the software package.

2.1 Where to Get the RTEdbg Package and Documentation

The data logging library source code for the library is available on Github. The complete *RTEdbg* toolkit is currently distributed as a ZIP file only.

Web link	**Description**
RTEdbg toolkit presentation	Brief introduction to the data logging library and toolkit.
RTEdbg manual	Check the link for the latest version of this document.
github.com/RTEdbg/RTEdbg/releases	Download the latest RTEdbg.zip file containing the library, software and documentation. See installation instructions below.
github.com/RTEdbg/RTEdbg	Data logging library source code. Use the Github Issues for bug reports or feature requests.
RTEdbg.freeforums.net	Feel free to ask questions, post comments, and report problems with the toolkit and documentation.

Note: The links are subject to change in the future. Changes will always be posted at github.com/RTEdbg/RTEdbg.

The distribution file **RTEdbg.zip** contains the following folders

Folder name	**Description**
Doc	Documentation, including this manual
Library	RTEdbg library source code, configuration files, drivers, etc.
RTEmsg	RTEmsg data decoding application
Demo	Demo projects – see Simple Demo Project and other Demo Projects
UTIL	Various utilities – e.g. for transferring logged data to the host

2.2 Installing the Library and Tools

The contents of **`RTEdbg.zip`** should be extracted to the **`c:\RTEdbg`** folder. Delete the old contents before unzipping the new version.

The batch files that support testing the demo code assume that the following software packages are installed in the folders listed below.

1. Notepad++ 64-bit version to the folder: `"c:\Program Files\Notepad++"`
2. STM32CubeProgrammer (if the ST-LINK is used) to `"c:\ST\STM32CubeProgrammer"`
3. Segger J-Link (if used) to: `"C:\Program Files\SEGGER\JLink"`
   Make the following settings during the installation. Select
   [● Install for all users] and [● Update existing installation]
   to enable installation to the folder defined above.

4. Install a CSV viewer or editor and set the application as the default for the CSV files if you prefer to automatically open the CSV files from a batch file after the data has been transferred and decoded. If you want to use the Flow CSV Viewer, install it from the Microsoft App Store.

The batch files in the demo project ***TEST*** folders and the IDE pre-build settings for the demo projects must be modified if a different folders are used.

When you install new versions of software tools, such as debug probe applications, they are often installed in a new folder. This makes it difficult to call these tools from batch files. Therefore, it is recommended that new versions always be installed in the same folder so that it is not necessary to modify the batch files used to automate testing.

2.3 RTEdbg Basics

In hardware, we do not add test points or test connectors during product development just to check them in production and have them in case of problems later. They are needed to verify that prototypes are working as they should and to optimize them. We can only improve what we detect and measure. A 32-channel software test connector is easy to create using the *RTEdbg* library. A 32-bit message filter allows us to select and log up to 32 groups of messages. Since both the impact on execution and the use of program memory are small, such a firmware test connector can (and should) remain in the production version of the firmware. With firmware support, the message filter can be set and data can be transferred from the embedded system even after the device with the embedded control or monitoring system has been shipped and installed – for example, when the service technician comes to the site or via an IoT connection.

The RTEdbg toolkit consists of the following main parts

1. **A set of data logging macros and functions.** They are used to instrument the project code. Data is logged in raw binary form, and code execution is fast because no data encoding or data tagging is done in the embedded system.
2. **RTEmsg – a command-line application** that runs on the host and performs the following functions:
   - check the syntax of format definition files,
   - automatically assign numbers to format definitions and filters before compiling project code, and
   - decode binary data from the embedded system according to the format definitions, do the statistics, prepare report, etc.
3. **Supporting software** for binary data transfer from the embedded system to the host computer.

> In this manual, the term **message** is used to refer to a record written to the circular buffer with a call to a data logging macro.

Additional documentation

If you want to know more about how the data logging library is implemented, what the current shortcomings are, and what the planned solutions to these deficiencies are, you can read the following documents (located in the c:\RTEdbg\DOC folder):

- *How the Data Logging is Implemented in the RTEdbg Library.pdf*
- *Current Shortcomings of the RTEdbg Toolkit.pdf*

It makes sense to read the above documents only after you are familiar with the basics of the *RTEdbg* toolkit. Read the document *RTEdbg Project History.pdf* if you want to know why the toolkit is the way it is and the document *RTEdbg Toolkit Portability Issues.pdf* if you are interested in porting the library to another CPU core.

2.4 Integrate RTEdbg Data Logging into Your Project

Note: It is recommended to first read this chapter quickly to become familiar with the *RTEdbg* concept. To understand it properly, it is necessary to know the individual components and how they work. In the second step, read in more detail.

We instrument the embedded system code with macros that capture data. A block of data captured by a logging macro is called a **message** in this document. Each message contains:

- Format ID – Index that defines which format definition will be used to decode the message on the host,
- Timestamp – tells when the message was logged, and
- Data provided by the programmer to the logging macro (this part of the message can contain a lot of data or be empty if we are just logging an event).

To enable logging, we need to add a library of logging functions to the project, which consists of a C source and some header files. Data is logged as 32-bit words in a circular buffer. One word of each message is used for the timestamp and format ID (and optionally up to 8 bits of user data). The rest of the message is user data, i.e. data specified by the programmer. Thus, the minimum size of a message is a 32-bit word when logging an event or an event with short data (up to 8 bits). The maximum message size is set by the programmer.

Follow these steps to add logging to a project

1. **Add the RTEdbg source code and header files to the project.** See the table in the Simple Demo Project section for a list of files to add to the project and check the demo code – e.g., compare the *Simple_STM32H743* and *STM32H743.CubeMX* projects in the **Demo** folder using a tool such as WinMerge to see what has been added or modified from the original code generated by the STM32CubeMX code generator. Only two files depend on the processor family you are using. These are a timestamp driver and a circular buffer reservation driver that must either use mutex instructions or disable interrupts during reservation. The CPU driver files are ready to use – see demo projects and folder `c:\RTEdbg\Library\Portable`. The folder also contains sample timestamp drivers.

2. **Add the *rtedbg_config.h* configuration file** and customize it according to the application requirements, CPU core, size of memory used for the circular buffer, etc. – see RTEdbg Configuration File. Create a new configuration file based on the template file *c:\RTEdbg\Library\Inc\rtedbg_config_template.h* or copy *rtedbg_config.h* from one of the demo projects and modify it. The compile time parameter RTE_BUFFER_SIZE defines the size of the circular buffer. The total RAM requirement in bytes is
   40 + 4 × RTE_BUFFER_SIZE.
   The parameter RTE_FMT_ID_BITS defines the number of format ID bits used to log a message. For smaller projects 10 is usually sufficient (9 is the minimum value). Set it to the smallest value possible to have more bits available for the timestamp (to increase timestamp resolution), unless you plan to use the same format definitions for future firmware versions also – see How to Avoid Problems When Adding or Changing Format Definitions.
   Other compile time definitions in the configuration file can usually be left unchanged for typical projects. Each definition has a description and setting instructions in its comments.

3. **Instrument your code with calls to the RTEdbg library functions and macros** – see Introduction to the RTEdbg Library. Data logging must be initialized by calling the *rte_init()* function. It is best to call this function as soon as possible after starting code execution, e.g., right after initializing the CPU clock generator after power-on. The value of the *rte_init()* message filter parameter defines whether logging should be enabled immediately (filter != 0) or later (filter = 0). Message filter defines which message groups are captured and which are discarded – see Message Filtering. Each data logging macro has a

filter number parameter that defines the message group. The message filter variable can be changed on the fly either by the debug probe or by the firmware using the *rte_set_filter()* function.

4. **Prepare format definition header files** for binary data decoding on the host computer – see RTEdbg Library and Format Definitions Basics and the examples in the **Demo** folders (check the **Fmt** folder). It is not necessary to put all format files in the *Fmt* folder as in the demo. They can be placed anywhere - for example, in the same folder as other headers. The *RTEmsg* program must be given the name of the folder containing the *rte_main_fmt.h* file (a list of all included format definition files) as a parameter.
5. **Add the pre-build call to the *RTEmsg* application** (using IDE settings or a Makefile)**.** The format definition parser built into application checks the syntax of format definitions and assigns numbers to format IDs and filter numbers (inserts the *#define's* in the format definition header files). It must be run before the project code is compiled. The format ID and filter number are parameter values for the data logging functions. See Verifying Format Definitions with RTEmsg.
6. **Set up the memory for the *RTEdbg* data structure *g_rtedbg*.** Place the RTEDBG section in a part of the memory that will not be erased after a reset or reboot. It must be located in a part of RAM that the entire instrumented firmware can access. Modify the linker settings to place the *RTEdbg* data logging structure at a defined start address (see Linking Requirements). This allows the debug probe to easily access the logged data. Place the data trace structure in uninitialized memory to avoid overwriting the entries on program reset or restart. The address of the *g_rtedbg* structure is not important if the data transfer to the host is done by the firmware itself and not by the debug probe.
7. **Add software triggers** (if appropriate)
   Programmable software triggers that can start and/or stop logging can be easily added to the firmware. The firmware enables logging by setting the message filter to nonzero and disables it by setting it to zero – see the rte_set_filter() function. This way we can limit the logging to the data we are most interested in and use the limited amount of circular buffer in the most useful way. If the trigger parameter(s) can be set through the debug probe or some other interface, then there is no need to recompile/reload the firmware to affect data collection.

Test and debug instrumented firmware

1. **Compile, download and run the code.** Data logging with the *RTEdbg* macros can be started immediately after initialization with the *rte_init()* function, or later with the debugger, data transfer application, or firmware – e.g., after a condition is triggered, the firmware sets the message filter to a non-zero value. **Note:** The *rte_init()* function is not needed if the *g_rtedbg* data structure is initialized by the debug probe at the beginning of the code test (e.g. after the breakpoint at the *main()* function is hit). It reduces memory usage for resource-constrained systems. See also the ***RTEgdbData*** utility *Readme* file.
2. **Transfer the logged binary data to the host.** It is not necessary to stop code execution to get snapshots of the logged data. Only the message filter value must be temporarily set to zero to stop message logging. Only the contents of the *g_rtedbg* data structure need be transferred to the host – see Using a Debug Probe to Transfer Data to a Host. Any interface or media can be used for data transfer.
3. **Decode the binary file** containing the embedded system log data using the RTEmsg MESSAGE DECODING APPLICATION on the host computer – see Binary Data Decoding.
4. **Analyze log files, visualize data** exported to CSV or other file types. See the list of some of the Data Visualization and Analysis Tools.

See the calculator in the spreadsheet file ***"DOC\RTEdbg calculator.ods"***. Its main purpose is to give the programmer an idea of how many different messages can be recorded if **N** (the number of bits for the format IDs) is equal to a certain value, and some other things like timestamp timer parameters, timestamp period, etc.

Introduction to Binary Data Logging

Many programmers use the printf test method, where the data is printed using the printf or sprintf function and stored in a circular buffer or sent directly to the host computer, e.g., via a serial port. Both the printf function and the format string reside in the microcontroller's program memory and require stack space to execute. Printf-style debugging is not suitable for most real-time systems and for systems with limited memory. It is also poorly suited for multi-threaded applications because it requires a semaphore to access a serial port or other channel to transfer data to the host.

With the RTEdbg toolkit, the embedded system only logs the data, and decoding/printing is done on the host computer, where the format definitions (format strings) are stored. Reentrant logging functions write data to the buffer in RAM. They are fast and require little program memory and stack space – see sections Resources Needed for Message Logging and Execution Times, Circular Buffer, and Stack Usage.

Note: In this manual the **symbol N** is used instead of the compile time parameter RTE_FMT_ID_BITS. It defines how many bits are used for the format definition ID (format identifier). The number of bits used for the format ID is limited to 16, which means that up to 2^16 = 65536 different format IDs are available to record up to 65535 different messages. The actual number of different messages is smaller, because most message types use more than one format ID – see the table Short Message Type Comparison. The **Format ID** is the message identifier. It defines which format definition is used to print each message type to be logged.

Data Logging Modes

Two main data logging modes are available:

1. **'Post-mortem' data logging mode**
   Writing data to the circular buffer does not stop when the buffer is full. The new data overwrites the old data as the circular buffer wraps around. If the data transfer to the host computer is too slow, the amount of logged data must be limited by setting the message filter accordingly, or the oldest data will be lost. This logging mode can be used to, for example:
   - Log data and send it to the host only when an error or exception is detected.
   - Log the data and temporarily pause logging by setting the message filter to zero, transferring the snapshot to the host, and setting the filter back to its previous value to continue logging data. This mode is called 'snapshot mode' in some logging solutions.
   - Transfer data periodically from the embedded system to the host computer and assemble it into a continuous data stream. This mode is called 'streaming mode' in some logging solutions. ***Note:*** *If the data transfer bandwidth is too low, some of the logged data will be lost.*

2. **Single shot data logging mode**
   Data is written as long as there is space in the buffer, which is used in a linear fashion (in contrast to "post-mortem" mode, where it is used as a circular buffer). The message filter is set to zero to disable logging and data can be transferred to the host. The *rte_init()* function must be called again to restart logging. Another way is to use a debug probe, as shown in the example batch files *Restart_single_shot*.bat* in the demo projects TEST folder. This mode allows programmers to log data after a specific event – for example, a signal (internal or external) to which we are interested in how our firmware responds. The trigger function that starts data logging by setting the message filter to a non-zero value must be implemented by the programmer. This data logging mode must be enabled by setting the compile time parameter RTE_SINGLE_SHOT_ENABLED to 1 in the *rtedbg_config.h* file. When this mode is enabled, the firmware can use the parameter of the *rte_init()* function to select whether to use single shot or 'post-mortem' mode.

RTEdbg Library and Format Definitions Basics

The following examples demonstrate the *RTEdbg* toolkit for logging and printing (decoding) data to the host computer. They show how to test and debug the firmware using the *printf* functionality and how to do the same using the functions of the library in a much less intrusive way.

Legend:

- The top part of each table is the firmware as it would be written for an embedded system where the data would be printed to a console, for example.
- The lower part of each table (highlighted in gray) shows how the same can be done with calls to the RTEdbg library, where the embedded system firmware simply writes the data to the circular buffer.
- Data format definitions are **highlighted in green**. They are defined for the host side only (as comments in header files) and are used to decode the binary data – print it according to the format definitions.

Example 1: Log an event and print the information to the Main.log file (this is the default log file).

```
  printf("System stopped – overtemperature detected\n");
```

```
// FILTER(F_EVENTS)
// MSG0_OVERTEMP "System stopped – overtemperature detected\n"
```

```
  RTE_MSG0(MSG0_OVERTEMP, F_EVENTS)
```

The first RTE_MSG0 macro parameter is the format ID (format definition index) and the second is the message filter number. The format ID defines which format definition is used to decode the binary data of the message. The message is written to the circular buffer if the corresponding bit of the 32-bit filter variable is set to one. The programmer does not have to worry about the assignment of format IDs and filter numbers. The *RTEmsg* application automatically assigns (defines) the format ID and filter numbers during the pre-build phase.

Example 2: Print two 32-bit float values to the *Main.log* file.

```
  float battery_voltage, battery_current;
  printf("Battery voltage %.2f, current: %.2f", battery_voltage, battery_current);
```

```
// FILTER(F_BATT_DATA, "Battery data")
// MSG2_BATT_DATA "Battery voltage %.2f, current: %.2f"
```

```
  RTE_MSG2(MSG2_BATT_DATA, F_BATT_DATA, battery_voltage, battery_current)
```

The simplest way to log and print data is shown above. In the format definition, the data must be written in the same order as parameters in the RTE_MSG logging macro or variables in the structure if the entire structure is logged with the *RTE_MSGN()* macro. If you do not have resource problems, you can stick to 32-bit value logging and simple format definitions. They are a direct replacement for printf strings. However, if you need as much history as possible in a given amount of RAM available for data logging, then you need to use the *RTEdbg* format extensions.

Example 3: Print data to a CSV file including header (first line) – uint16_t values used for battery data

```
  uint16_t batt_voltage_x100, batt_current_x100;      // Resolution = 10 mV / 10 mA
  FILE * batt = fopen("battery data.csv", "w");
  fprintf(batt, "Battery voltage [V];Current [A]\n");
  ...
  fprintf(batt, "%.2f;%.2f\n", batt_voltage_x100 / 100., batt_current_x100 / 100.);
```

```
// OUT_FILE(BATT_DATA, "battery data.csv", "w", "Battery Voltage [V];Current [A];\n")
// MSG1_BATT_DATA >BATT_DATA "%[16u](*.01).2f;%[16u](*.01).2f;\n"
```

```
   RTE_MSG1(MSG1_BATT_DATA, F_BATT_DATA, batt_voltage_x100 | (batt_current_x100 << 16u))
```

The example shows how to pack two 16-bit values into one 32-bit value to reduce circular buffer usage. The values are printed to a CSV file named "*battery data.csv*". A header line "Battery Voltage [V];Current [A];\n" is

inserted after the file is created. A more complex format definition ("%[16u](*.01).2f" for each of the values) is used above to show some of the printf style extensions. The "[16u]" indicates that a 16-bit unsigned value is logged and (*.01) that the value is multiplied by 0.01 before printing. This also eliminates the need for the embedded system to have float support to scale values when printing. See Syntax of the Printf Style Format Definition String for a complete description.

Note: Data logged with calls to the RTEdbg logging functions contains timestamp information, while time-stamps would have to be added to data printed with *printf()* or *fprintf()*.

The RTEdbg data logging functions are reentrant. The execution of a data-logging function started by a lower-priority piece of code can be interrupted at any time before the function has read the value of the timer counter that measures the time. The function that interrupted logging could therefore log a timestamp with a slightly smaller value. The possibility of such events must be taken into account when analyzing data in log files.

Introduction to Message Filters

Each message logging macro has a filter number parameter that defines which message group the particular message belongs to. The filter number (0 … 31) defines which bit of the 32-bit message filter variable will enable the particular message group. The filter variable can be manipulated by calling the *rte_set_filter()* function or by using a debug probe. Filtering reduces the amount of data sent from the target to the debugger. The programmer can enable more or fewer message groups to get either longer logging times with the same circular buffer size or shorter logging times with more detail. Data logging can be stopped completely by setting the filter to 0 (e.g., after a system error). See details and usage examples in Message Filtering. The initial filter value is set when the *rte_init()* function is called. If the initial value is zero, message logging is disabled until the firmware (or the programmer using the debug probe) enables the filter, e.g., when a trigger initiates data logging.

Programmers should plan for the use of filters

For messages or message groups that generate a large amount of data, it is useful to provide a separate enable. Filter F_SYSTEM (number 0) is reserved for system messages, but can be used for important user-defined messages (exception handler messages). It can only be disabled by the firmware if all 32 bits of the message filter variable are set to zero. The remaining 31 filter numbers from 1 to 31 can be used for any purpose.

When decoding binary files, statistics are kept on the most frequent messages and the messages that took up the most space in the file (in the circular buffer) – see Stat_main.log. This data can be useful during testing, for example, to decide which message group to disable in order to keep certain message groups in the buffer longer, or to prevent data loss due to limited bandwidth to the host. Good message filter planning allows the selection of desired data groups when the system is already in the field and the firmware cannot be easily changed.

Introduction to the RTEdbg Library

The library contains functions that initialize the data logging structure, log the binary data to the circular buffer, manipulate the message filter, etc. The following table lists all of the macros and functions. For smaller projects, only a subset of these will be needed. Click on the macro and function links for more detailed descriptions. All data-logging macros call only nine internal functions. There are also six additional functions that support filter manipulation, *RTEdbg* structure initialization, etc. Such a small set of functions is sufficient because the data is not treated differently by type (only size matters) and is logged in binary form without additional encoding or tagging. The whole set of macros and functions is small and easy to learn.

Data Logging Macros

void **RTE_MSG0** (fmt_id, filter_no) // Event logging (no data)
void **RTE_EXT_MSG0_y** (fmt_id, filter_no, short_data) // Event logging + up to 8 bit data (**y** = 1… 8)
void **RTE_MSG1** (fmt_id, filter_no, data1) // 1 × 32 bit data logging
void **RTE_EXT_MSG1_y** (fmt_id, filter_no, data1, short_data) // 1 × 32 bit + up to 7 bits
void **RTE_MSG2** (fmt_id, filter_no, data1, data2) // 2 × 32 bit data logging
void **RTE_EXT_MSG2_y** (fmt_id, filter_no, data1, data2, short_data) // 2 × 32 bit + up to 6 bits
void **RTE_MSG3** (fmt_id, filter_no, data1, data2, data3) // 3 × 32 bit data
void **RTE_EXT_MSG3_y** (fmt_id, filter_no, data1, data2, data3, short_data) // 3 × 32 bit + up to 5 bits
void **RTE_MSG4** (fmt_id, filter_no, data1, data2, data3, data4) // 4 × 32 bit data
void **RTE_EXT_MSG4_y** (fmt_id, filter_no, data1, data2, data3, data4, short_data) // 4 × 32 bit + up to 4
void **RTE_MSGN** (fmt_id, filter_no, data_address, data_length) // Log a complete data structure
void **RTE_MSGX** (fmt_id, filter_no, data_address, data_length) // See link
void **RTE_STRING** (fmt_id, filter_no, string_address) // Log a string
void **RTE_STRINGN** (fmt_id, filter_no, string_address, max_length) // Log limited-length string
void **RTE_RESTART_TIMING()** // Logs a message with a request to restart time decoding in *RTEmsg*

Parameters:

uint32_t **fmt_id** – Format ID defines which format definition is used for printing values
uint32_t **filter_no** – Number of the filter bit that enables the message (0 … 31)
rte_any32_t **data1 … data4** – Any 32-bit data (from char to float, pointer address, packed values, etc.)
uint32_t short_data – 1 to 8 bits of additional data possible with the same message size (**y** = number of bits).
void * **data_address** – Pointer to the data that is to be stored in the circular buffer
Complete or partial data structures, strings, or buffers can be logged.
uint32_t **data_length** – length of data in bytes (size of the variable, data structure, string, or buffer, etc.)

Various Functions

void **rte_init**(initial_filter_value, init_mode);
This function must be called before the first message is logged. It initializes the data logging structure.
void **rte_long_timestamp**(void);
Time synchronization with host – write the message with a long timestamp value to the circular buffer.
void **rte_timestamp_frequency**(uint32_t frequency);
Should be called after changing the timestamp timer frequency, e.g., after switching the clock source.

Message Filter Manipulation Functions

void **rte_set_filter**(uint32_t filter_value);
The firmware can start (filter != 0), or pause (filter = 0) data logging by setting the filter.
uint32_t **rte_get_filter**(void); // Get the current filter value
void **rte_restore_filter**(void); // Restore the value used before logging was disabled with *rte_set_filter(0);*

Note: The void type is specified before the macro name only to indicate that the function called by the macro will not return a value.

The union type ***rte_any32_t*** allows logging of arbitrary values with a length of 32 bits or less, packed data (multiple values packed into 32-bit words or 32-bit structures), or bitfields with a length of up to 32 bits. Some compilers do not allow this union type. See Workaround for Missing/Disabled Compiler Support for the rte_any32_t Union. Also see the description if you have a C++ or mixed C/C++ project.

The ***rte_init()*** function must be called before the first call to any of the data logging functions, including ***rte_long_timestamp()***. It is not necessary to disable interrupts before calling the data-logging functions because

they are reentrant. The exception is the default implementation of the ***rte_long_timestamp()*** function provided with the timestamp drivers, which is not reentrant.

The data logging macros *RTE_MSG0()* to *RTE_MSGN()* not only perform the data logging, but also have the following additional functions:

1. During compilation, the correctness of the parameters is checked with *static_assert*, since checking the parameters at runtime would slow down the code execution.
2. Two or three parameters (format ID, filter number, and optionally a few bits of additional information) are combined into one function parameter. This reduces the amount of machine code for a function call compared to a function that would have two or three separate parameters.
   Example: Both macros ***RTE_MSG0****(fmt_id, filter_no)* and ***RTE_EXT_MSG0_y****(fmt_id, filter_no, ext_data)* call the *__rte_msg0()* function with one parameter – the combined value of two or three macro parameters.

Logging of More Than Four Values at Once

The macros *RTE_MSG1()* to *RTE_MSG4()* allow to log one to four 32-bit words with a single function call. Logging more than four separate 32-bit words with a single function is not supported because the stack usage in the logging functions would increase. Due to the way data is written to the circular buffer, additional macros with more parameters would not reduce the circular buffer usage.

If you need to log a large number of values at the same time:

- Place the variables into structures during the firmware design, which can then be logged with a single call to the *RTE_MSGN()* macro.
- Copy the values into a temporary structure and then call the *RTE_MSGN()* macro.
- Use a combination of *RTE_MSG()* type macros – e.g., *RTE_MSG4()* and *RTE_MSG3()* to log 7 values.
- Pack the values as shown in Pack Multiple Short Variables Into a 32-bit Word or Structure. For example, if two 16-bit values are packed into each of the four 32-bit words of *RTE_MSG4()*, this macro can log eight values at once.

Comment of the Macro Type RTE_EXT_MSGx_y

These macros are provided because one to eight bits of additional information can be added to messages up to four words in length without using additional circular buffer space. Any extra (high) bits of the additional data value are truncated in the macro. If necessary, the programmer must limit the value to the maximum possible size based on the number of bits provided by the macro type. Incorrect information that appears in the log file may mislead the person reviewing the data.

One bit is enough to store data when something is true or false. Without this option, an additional 32-bit word would be required to log the true/false value, taking up valuable space in the circular buffer where the data is logged.

See also Extended Messages RTE_EXT_MSG0_y … RTE_EXT_MSG4_y.

Library Source Code and Hardware-Dependent Drivers

The complete data logging source code is in the file *rtedbg.c*. The code is optimized for 32-bit microcontrollers. Some header files and the configuration file must also be added to the project. See the complete list of files that have been added to the Simple Demo Project.

Each project must include two hardware dependent driver files to adapt the library to the used microcontroller:

1. **CPU driver for reentrant circular buffer space reservation:** Space must be reserved in the circular buffer before data is written to it. This should be done either by using mutex instructions, if supported by the CPU core, or by temporarily disabling interrupts.
   The following driver files are provided with the library:
   - **rtedbg_generic_irq_disable.h** – The driver can be used for all devices as it is the most universal solution. While it provides a robust and broadly compatible solution, it is the only possible driver for devices with cores that don't have hardware mutex (atomic) support.
   - **rtedbg_generic_atomic.h –** generic driver for devices with support for the atomic operations library (these are processors whose cores support mutex instructions).
   - **rtedbg_generic_atomic_smp.h** – generic driver for symmetric multi-core devices with support for the atomic operations library. Use with caution, as shared memory usage is not trivial on multi-core processors. See also Using the RTEdbg Library on Multi-Core Processors.
   - **rtedbg_cortex_m_mutex.h** – for ARM Cortex-M3, M4, M7, M33 / M85 (uses mutex instructions). This driver version allows smaller and faster code for the supported CPU core families than the version in *rtedbg_generic_atomic.h* or *rtedbg_generic_atomic_smp.h*.

   See additional descriptions in the **Readme.md** files (see *Library\Portable* folder and subfolders) and in the comments of the driver files.
   The interrupts are not disabled at all if the CPU core supports the mutex instructions (atomic operations). Otherwise, interrupts are disabled only for a few processor cycles – only while space is being reserved in the circular buffer.

2. **Timestamp driver:** A hardware timer provides timestamps that are logged along with messages. Driver examples are part of the *RTEdbg* library – see Timestamp Drivers. Two macro definitions are required to define the timer frequency and division factor – see the driver examples.
   Note: The timestamp driver is usually the only file that needs to be adapted to the timestamp timer and its clock frequency.

The RTE_CPU_DRIVER macro must be defined in the *rtedbg_config.h* configuration file – for example

```
#define RTE_CPU_DRIVER "rtedbg_cortex_m_mutex.h"
```

It defines which CPU driver file is included during the project compilation.

The RTE_TIMER_DRIVER macro defines which timer driver will be used for the project – for example

```
#define RTE_TIMER_DRIVER  "rtedbg_timer_cyccnt.h"
```

The driver files must be adapted to the particular device if the existing drivers do not support the CPU core or the timer used for the timestamp.

Note: The simple demo presented in this chapter uses the CYCCNT timer from the ARM Cortex DWT unit for timestamping. The timer counts how many clock cycles the processor took to execute the instructions. The number is only correct if the program is executed continuously. CYCCNT is incremented by more cycles than it took to execute the instructions when the code is executed step-by-step with the debugger.

Getting Started with Data Format Definitions

The format definition files define how the data from the logged messages are processed and printed to the Main.log file or to the specified output files. The format definitions in the demo projects are located in the *Fmt* project folder. See the excerpt from one of the files below.

```
/* This is a formatting text for the RTE_MSG1(MSG1_CPU_FREQUENCY, ...) */
// MSG1_CPU_FREQUENCY  "CPU core frequency = %[32](*1e-6)g MHz"
#define MSG1_CPU_FREQUENCY 330U                    <= Inserted by the RTEdbg application
```

The format definition header file(s) should contain only:

- `/* ... */` Single line comments are treated as comments by the *RTEmsg* application.
- `// ...` A double slash comment indicates a format definition for the *RTEmsg* application.
- `#define / #ifdef / #endif` Directives inserted by the *RTEmsg* application during the format header syntax checking and parsing (programmers should not modify the lines containing directives).

The *RTEmsg* application automatically enumerates the format IDs and filters and inserts #define directives with the format ID and filter names and numbers. Programmers should write only the directives (// …) with format definitions and comments (/* … */). No other directives or C code are allowed in the format definition files.

The format definitions are hidden from the compiler inside '//' comments and are either keywords (see RTEdbg Format Definition Keywords) or format strings (see Format ID Names and Format Strings).

RTEdbg Format Definition Keywords

Keywords allow you to define output file names to which the decoded message content can be written, filter names, etc. Each name must be defined before it is used in the format definitions.

`// INCLUDE("file name")` Tells the *RTEmsg* application to process the contents of another format definition file. Example: `// INCLUDE("rte_system_fmt.h")`

`// FILTER(F_FILTER_NAME, "Filter description – optional")`
Specifies the name and (optional) description of the message filter. See the Introduction to Message Filters.
Examples: `// FILTER(F_ERR, "Errors and warnings")`
`// FILTER(F_TEMPERATURES)`

`// OUT_FILE(NAME, "file_path",  "mode", "Initial contents – optional")`
Specifies the output file to which some of the data will be printed, its path, mode (e.g., "w" - write or "wb" – write binary), and (optional) initial text written to the output file – e.g., CSV file header.
Examples: `// OUT_FILE(RTOS_TASK, "rtos task.log", "w")`
`// OUT_FILE(MEASUREMENTS, "test.csv", "w", "Time[s];Voltage[V];Current[A]\n")`

`// IN_FILE(NAME, "file_path")` See IN_FILE() – Specify the input file for the indexed text (%Y format type).
`// MEMO(M_MEMO_NAME, Initial value – optional)` Allows you to store values from decoded messages. See MEMO() – Define a memory variable for a detailed description and Example: Combine fast- and slow-changing data in the same CSV file.

`// FMT_START(start_code)`
`// FMT_ALIGN(align_value)`
See FMT_ALIGN() / FMT_START() – Reserve space in the format ID area for a detailed description.

Note: Keyword parameters shown in green are optional (do not need to be specified).

Format ID Names and Format Strings

This section provides a brief introduction to the format definitions (format strings) that are used to specify how individual message values must be printed. Each message logged during firmware execution can contain anything from zero (*RTE_MSG0()* macro) to a large amount of data (e.g., *RTE_MSGN()* macro).

The format definition must be inside a C comment that starts with '//'. It must contain at least one name and at least one format string. See the example below for decoding the event message.

```
// MSG0_MOTOR_OVERTEMPERATURE "Motor temperature above the maximum value."
```

```
//MSG0_MOTOR_OVERTEMPERATURE                    <= Definition can be split over multiple lines
// "Motor temperature above the "
// "maximum value."
```

Unless otherwise specified, the message is printed only to the Main.log file, as in this example. First, the sequence number of the decoded message, the time of the message (timestamp), and the name of the message (in this case) are written to the *Main.log* file. Then the message text is printed as specified in the format string. See the example from *Main.log* below:

```
N01039 1598.817 MSG0_MOTOR_OVERTEMPERATURE: Motor temperature above the maximum value.
```

The message name must begin with a prefix that defines the message size. The prefix **MSG0_** indicates that the message contains no data other than the timestamp and format ID. Format ID names should follow the naming conventions – see Format ID and Filter Naming Conventions. The '\n' does not need to be at the end of the format string, as it is automatically added when the message is printed to the default Main.log output file.

Print to a Custom File

The following example shows how to print two float numbers that were logged with the macro RTE_**MSG2**(value1, value2) and redirect the printout to a custom output file.

```
// OUT_FILE(MOTOR_DATA, "Motor.txt", "w")
// MSG2_MOTOR_DATA >MOTOR_DATA "Motor power %5.2f, temperature %4.1f\n"
```

The output file must be defined before the first use and can be used in any number of format definitions. Output is redirected to the specified file with **>MOTOR_DATA**. The character '\n' must be at the end of the string if to continue printing on the next line, as it is not automatically added for custom files. The output files are created in the folder whose name is defined by the *RTEmsg* application command line argument **output_folder** (the same folder as for Main.log and Errors.log).

How to print to Main.log and user defined file at the same time: If the output file is defined with **>>MOTOR_DATA** (redirection with double **>**) to print the information to the user defined file and to *Main.log*.

The programmer must ensure that the type (and size) of the printed value is the same as the type of the logged value. The advantage of the *RTEdbg* toolkit is that we can change the format string after the data has been logged if we find an error in the format definition. It is not necessary to repeat the system test and data transfer, which is very annoying if the problem is difficult to repeat.

The **sequence of numbers in the format definition** must be the same as the sequence of parameters of the macro with which data is logged, or the sequence of values in the data structure if the entire (or partial) data structure is logged. The size of the value to be printed (number of bits used), its type, and its start bit must be defined for non-32-bit values, packed structures, bitfields, and so on (see the [Value…] definition. The same applies if we want to print the values in a different order than they were logged.

A format definition can be single-line or multi-line. There is no limit to the size of a format definition for any message type. It is not necessary for the format string to be on the same line as the format ID name. The above

example can also be written in the following ways, all of which are syntactically correct. Comment /* ... */ or a blank line within a definition is also fine.

```
// MSG2_MOTOR_DATA
// >MOTOR_DATA "Motor power %5.2f, temperature %4.1f\n"
```

```
// MSG2_MOTOR_DATA
//    >MOTOR_DATA "Motor power %5.2f,"
//    >MOTOR_DATA " temperature %4.1f\n"
/* Print the motor power value to the Main.log file also */
//    "Motor power %g"
```

Note: The redirection to the custom output file must be defined for each format string, as shown in the example above. If omitted, by default the output will go to the Main.log file.

Print values from one message to two output files with different formatting

```
// OUT_FILE(MOTOR_DATA1, "Motor.txt", "w")
// OUT_FILE(MOTOR_DATA2, "Motor.csv", "w", "Time [s];Power [W];Temperature [°C]\n")
// MSG2_MOTOR_DATA
// >MOTOR_DATA1 "Time %t s, Motor power %5.2f W, Temperature %4.1f °C\n"
// >MOTOR_DATA2 "%t;%5.2f;%4.1f\n"
```

The example shows how to print the values from the message logged with the *RTE_MSG2()* macro to the *Main.log* file with the format string in the second line of the definition and to the MOTOR_DATA with the second line. The data formatting in the third line contains the data as needed to process CSV files.
The example also introduces the format type extension '**%t**', which is an extension to the standard set of format types in C printf strings. It specifies that the timestamp of the message should be printed in the same format as the time in the *Main.log* file.

The table below contains examples of the most commonly used special format types. The complete list is in Additional Type Field Characters.

%N	Print the current message serial number.
%t	Print the timestamp of the current message.
%T	Print message period – the difference between the current message timestamp and the timestamp of the previous occurrence of the same message. Zero is printed if there is no previous message.
%nH	Print complete message data as hex numbers (hex dump of message contents) **n** = 1 – message printed as bytes, **n** = 2 – as 16-bit words and **n** = 4 – as 32-bit words This output method is useful, for example, for messages whose length is unknown at the time of compilation (time when the format definitions were created).

Notes:

1. The **%N** and **%t** types are useful when printing data to files other than Main.log. To this data we add either the message time (timestamp) or the message number. Since typically only part of the data is sorted into the user-defined output file, we can then, for example, look in the main *Main.log* file or other files for the remaining messages that were logged before or after the message whose contents we find interesting or suspicious.
2. Decimal values are printed with a decimal point or comma, depending on the user's operating system locale settings. This behavior can be overridden with a command line argument to the *RTEmsg* application (add the ***-locale=C*** command line argument to force the use of the decimal point).

Extensions to the Standard Printf Syntax

The previous examples have shown the normal syntax of printf strings and some of the additional type field characters. After the '%' character and before the format type, additional fields can be added to specify how many bits should be taken from the logged message for each piece of data printed, how the data should be scaled, what type the data is, etc.

A printf conversion specification (print string) consists of optional and required fields in the following form:

%[**f**lags][**w**idth][.**p**recision][**s**ize]**t**ype - **fwpt** in the example below

See the following link for a detailed explanation of the format syntax and definition of the **fwpt** fields:

docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions

The following is an example of a format string to print a floating-point number:

```
"Value: %8.3f"
```

Extensions have been added to the printf string syntax to provide additional functionality such as data size definition or value scaling. **Additional definitions must be placed immediately after the '%' character.**

The following example shows how to print a 16-bit unsigned value that is multiplied by 0.01 before being printed using the print string `"%8.3f"`.

```
"Value: %[16u](*0.01)8.3f"
```

The complete syntax of a **conversion specification with extensions** is shown below:

`"text %[value](scaling){text1|text2|…|textN}<memo>|statistics|fwpt text"`

- **text** – Plain text (any text, ASCII or UTF-8).
- [**value**] – Specifies either which part of the logged message to use when printing with the value, or what other information (e.g., timestamp, message number) to use instead of the logged message value – see [Value…].
- **(scaling)** – Defines value scaling (offset and/or multiplier by which the value is scaled before printing) – see (±Offset*Multiplier). It is not necessary to use offset and multiplier. Only the offset or multiplier can be used.
- **<memo>** – The name of the location where the value is stored for later use – see <M_NAME>.
  See also the Example: Combine fast- and slow-changing data in the same CSV file.
- **|statistics|** – Enable statistics for the value and define its name – see |Value Name|. It can be used for any printed value. Just add the name of the value between the '|' characters and statistics will be enabled for that value. See the full description in Statistical Features.
- **{text1|text2|…|textN}** – Define the text for the %Y format type – see {text1|text2|…|textN}.
  Example: Text can be printed instead of the error number or the current state of a state machine.
- **fwpt** – Fields that define **f**lag characters, **w**idth and **p**recision specifications, and **t**ype conversion specifications. All standard type, flag, width, and precision characters are allowed.

All fields between the percent sign '**%**' and the **fwpt** field are optional *RTEdbg* extensions, and each can occur only once. The **[value]** definition field must be the first (if used).

The most useful syntax extensions are briefly described below. A detailed description of all printf string syntax extensions can be found in the Syntax of the Printf Style Format Definition String.

Note: The *RTEmsg* application checks the syntax of format definitions during precompilation and reports an error. The error message can be customized to be recognized by the IDE. This allows the IDE to jump to the source of the error. See the description of the "**-e**" compile time argument.

Define Which Value to Print (Bit Address and Size)

A programmer needs to know how data is stored in memory if he wants to log and print packed data or a packed data structure. The following description refers to the little-endian byte order of the word (as for ARM Cortex-M cores). The following example shows data captured by a macro call

```
RTE_MSG2(MSG2_SOME_DATA, F_FILTER, 0x12345678, 0x01020304)
```

Hexadecimal values have been used to make the following descriptions easier to follow. Each of the 32-bit values can be a packed value of several shorter data types, as shown in Pack Multiple Short Variables Into a 32-bit Word or Structure.

Bit address	0	8	16	24	32	40	48	56
Byte value	0x78	0x56	0x34	0x12	0x04	0x03	0x02	0x01

The value address and size definition [**nn:mmF**] or [**mmF**] is used to define the bit address, number of bits, and data type to be used for the next printed value if the data is not printed in the same order as it was logged or its size is not 32 bits. The **nn** defines the bit address of the lowest bit and **mm** the size of the value (1 … 64 bits). The address does not need to be specified when printing consecutive values, as the address is automatically updated by the size of the previously printed value. The **[value]** value definition must be inserted after the '**%**' and before the format type specifier.

The type of the printed value can be (**F** in the **[value]** definition above):

u – unsigned integer (from 1 up to 64-bit) – **default if the type is omitted**
i – signed integer (from 2 up to 64-bits)
f – floating point – 64 (double), 32 (float) and 16-bit (half-float) values are allowed
s – null-terminated string or string of known maximum length

Reset the bit address: For each message type, the bit address for the first printed value in any format type definition is 0. If the message values are printed in multiple ways in multiple output files, then the bit address is reset to zero for each printed value that is printed in another output file as before. For example: If the content of the logged message is written first to **>FILE1** and then to **>FILE2** (or to Main.log if the output file is not specified), then the bit address is reset when we stop writing to **FILE1**. If two format commands for the same message write to the same output file once with **>FILE1** and the next line with **>>FILE1**, then it is the same output file in both cases and the bit address is not reset.

The bit address of the value is the address of the first bit of the message that will be taken during decoding for the value to be printed with %+type – e.g., with %**u**. Bit zero of the first 32-bit value has address zero, bit one has address one, and so on. The bit address of the second 32-bit word is 32, 64 for the third word, and so on. The programmer can use either an absolute address (e.g., [32:8u]) or a relative address (e.g., [-32:16i] or [+8:3u]). Offset can be a positive or negative value for relative addressing mode. A positive value is used when we want to skip some of the message data that is not relevant when printing to a custom file, for example. A negative value is used when, for example, we want to print the same value again with a different type (print the same value as hex and decimal). The resulting bit address must be within the size of the message – including the length of the data we want to print. See the full description in [nn:mmF].

Example: To print a portion of the message (byte values 0x12 0x04) from the example above in hex format, [24:16u] must be used. A format definition for such an value would be

```
// "0x%[24:16u]4x
```

and "0x0412" would be printed. After the value is printed, the bit address in this case is incremented by the size of the data (by 16) and becomes 40. See also format definition examples in demo code.

Note: If you are not sure that both the data packaging and the format definition match, fill the data structure with known data, log it, and check the output from the *RTEmsg* application.

Printing the Same Number Two Different Ways

```
// MSG2_APP_DATA "Value1: %u, Value2: %u (0x2%[-32:32u]8x, 0b%[-32:32u]B)"
```

Value1 is printed as unsigned number. Value2 is printed as an unsigned number and then in hexadecimal and binary format. The value bit address is automatically incremented by 32 after printing Value1 and by another 32 after printing Value2. To print Value2 again, the bit address must be decremented by 32. We do this with **[-32:32u]**, which means that first the address is decremented by 32 and the 32-bit value (Value2) is interpreted as a 32-bit unsigned number. An unsigned value is the default and we can only use **[-32:32]** instead of **[-32:32u]**. An absolute address can also be used for the Value2 instead of a relative one: **[32:32]** or **[32:32u].**

Value Scaling

Programmers often use fixed point arithmetic and scaled or offset values. To avoid the need to scale the value in the embedded system, the option to scale the value before printing has been added as a printf syntax extension. To scale, add an expression **(±offset*multiplier)** between '**%**' and format type, as shown in the example below. When scaling a value, the offset is added first and then the result is multiplied. The offset or multiplier can be omitted.

```
// FILTER(F_PACKED)
// MSG1_PACKED
//    "Power: %[16i](*.01).2f W, temp1: %[8](-100*.5).1f ºC, temp1: %[8](-100*.5).1f ºC"
```

The following is an example of program code that pack data into a single 32-bit value that is printed on the host computer using the format definition above. By doing this, we reduce the amount of data in the circular buffer, which means we can fit more messages into the same buffer and have a longer history.

```
int16_t power_x100;     // Power [W] multiplied by 100 (resolution 0.01 W)
uint8_t temp1;          // Temperatures scaled by 2 and offset by 100 which
uint8_t temp2;          //  allows a range from -50ºC to + 77.5ºC (for 0 … 255)
RTE_MSG1(MSG1_PACKED, F_PACKED, power_x100 | (temp1 << 16u) | (temp2 << 24u))
```

The expression can also contain only an offset or only a multiplier, or both. The '+' or '-' character must precede the offset value, and the '*' character must precede the multiplier. The decimal point must be used for the values, as for the C/C++ language constants. See the full description in Value Scaling Definition (±Offset*Multiplier).

Printing Indexed Text

Text can be printed instead of a numeric value (e.g., error message instead of error number). The '%Y' printf format string extension has been added for this purpose. Texts can be defined with the inline definition {text1|text2|...|textN} or read from a file – input file defined with the IN_FILE() directive. Below are two examples. The first one is for inline text definition and the second one is for reading text from an input file. The value taken as index must be an unsigned integer from 1 to a maximum of 64 bits. If the value of this number is 0, then it prints the first text from the inline definition or the text from the first line of the file. If the value is 1, it takes the second text, and so on. If the value is greater than or equal to the number of text fields, the last text field is used. For example, you can define a text like "Unknown error" as the last text field, and this text will be printed whenever the number is greater than the expected maximum.

The following example shows how to print a system status value first as a numeric value and then with a text description. The `[0:8u]` selects the value to be printed using the value address and size.

```
// EXT_MSG0_3_SYSTEM_STATE
// "State: %[8u]u - %[0:8u]{Off|Standby|Ready|Running|Error|Unknown}Y"
```

The following example shows how to log the state of the state machine using the *RTE_EXT_MSG0_y()* macro for cases where the number of states is less than or equal to 256. In this case, one 32-bit word is required to

record the information instead of two if the *RTE_MSG1()* macro is used to record the data. The string from the first line of the input file is printed if the value is 0, from line 2 if the value is 1, and so on.

```
// FILTER(F_MACHINE_STATES)
// IN_FILE(MACHINE_STATE, "Machine-states.txt")
// EXT_MSG0_4_MACHINE_STATE < MACHINE_STATE "Machine state changed to: %Y"
```

```
machine_state_t current_state;
RTE_EXE_MSG0_4(EXT_MSG0_4_MACHINE_STATE, F_MACHINE_STATES, current_state)
```

See the additional description in Indexed Text Definition {text1|text2|…|textN} and IN_FILE() – Specify the input file for the indexed text (%Y format type).

Other Features, Hints, and Examples

Below are some links to sections with additional examples and feature descriptions:

- Bitfield Logging
- **Measure the time between two messages (events)** using the [t-MSG_Name] value definition
- Minimum Circular Buffer Size and tips for reducing buffer usage
- Any value can be used for statistics to determine extreme values – see Statistical Features
- Example: Combine fast- and slow-changing data in the same CSV file
- Data logging in RTOS-based applications
- Optimize Data Logging Code Size and Speed for maximum data-logging speed or minimum flash memory footprint.

RTEdbg Configuration File

The *rtedbg_config.h* configuration file is used to customize data logging to the needs of the project, such as:

- Set the circular buffer size (size of the log history) and the maximum message size,
- Enable/disable message filtering and set filtering options,
- Set the number of bits used for the format IDs,
- Set the *RTEdbg* library optimization options,
- Enable the single shot data logging option (by default, only 'post-mortem' mode is available).

Copy the *Library/Inc/rtedbg_config_template.h* to the project folder and rename it to *rtedbg_config.h* or use a config file from one of the demo projects. Follow the instructions in the config file. The full description of each compile time parameter can be found in the manual. See also the example in the small demo project below and other parts of the demo code. Set RTE_TIMESTAMP_SHIFT to a higher value (> 1) if the timestamp timer clock frequency is high and the maximum time between consecutive messages in your application could be large.

The demo configuration files have a section called "Compiler code optimization parameters". It contains compiler optimization definitions for four different compilers to allow compilation with different toolchains. Remove the redundant definitions for clarity and modify them to suit your needs.

The RTE_BUFFER_SIZE command line parameter sets the size of the circular buffer. It is set to a small value for demo projects to allow compilation with the code size limited versions of the Keil MDK and IAR EWARM toolchains. Modify the data transfer batch file in the TEST folder after changing the RTE_BUFFER_SIZE value (size of data transferred from the embedded system).

2.5 Simple Demo Project

The simple demo (see *Demo\Simple_STM32H743* folder) shows how to integrate data logging and decoding into a small project. It contains a typical configuration that you can use for projects with Cortex-M3, M4, M7 and similar processors that support mutex commands and have a CYCCNT timer (cycle counter). It is recommended to use a tool like WinMerge to compare the original source code generated with the STM32CubeMX code generator in the folder

`c:\RTEdbg\Demo\STM32H743.CubeMX` with the demo code in **`c:\RTEdbg\Demo\Simple_STM32H743`**

This allows programmers to see how the data logging functions have been integrated into the project. The demo project has also been prepared for and tested with the Keil MDK and IAR EWARM toolchains.

The simple demo project was tested on the NUCLEO-H743ZI board (ARM Cortex-M7 core) and should run on all boards with STM32H742/743/750/753 since no I/O is used. The CYCCNT (CPU cycle counter) is used as timestamp timer and the 64 MHz HSI clock generator is used as CPU clock. The RTE_TIMESTAMP_SHIFT is set to 1 (lowest value) to allow fine-grained timestamps. The timestamp timer counter value is divided by (2^RTE_TIMESTAMP_SHIFT = 2). This is possible if the messages are logged frequently enough so that the time between successive logged messages is less than a quarter of the timestamp period.

Note: The STM32CubeMX tools copy some source code and header files that are not required for typical projects. It is not possible to disable the generation of these files. They have been removed from this demo as well as from the *Demo/STM32H743.CubeMX* folder. In the file *RTEdbg/Inc/stm32h7xx_hal_conf.h* some of the default enabled (unnecessary) header includes have been disabled.

The following source code files have been added to the demo project (from the *Library* folder):

File name	Description
`rtedbg.c`	Data logging library source code.
`rtedbg.h`	Definitions of data structures, functions and macros. The file must be included in files where RTEdbg library functions are used.
`rtedbg_int.h`	Internal definitions for data logging functions.
`rtedbg_config.h`	RTEdbg configuration file – see RTEdbg Configuration File.
`rtedbg_cortex_m_mutex.h`	Buffer space reservation macro using exclusive Read-Modify-Write (mutex). This driver is suitable for devices with Cortex-M3, M4, M7, etc.
`rtedbg_timer_cyccnt.h`	Timestamp timer driver – ARM Cortex CYCCNT cycle counter is used for the timestamp. For other timestamp drivers see Timestamp Drivers. **Note:** When debugging, the debugger may use the DWT for its own purposes. This can interfere with the data logging code. When you step through with the debugger, it affects the behavior of CYCCNT. Consider running your code without single steps or breakpoints to get accurate timing results.
`rte_main_fmt.h`	Main format definition file - list of INCLUDE() directives for all definition files.
`rte_system_fmt.h`	Format definitions for the RTEdbg system messages.

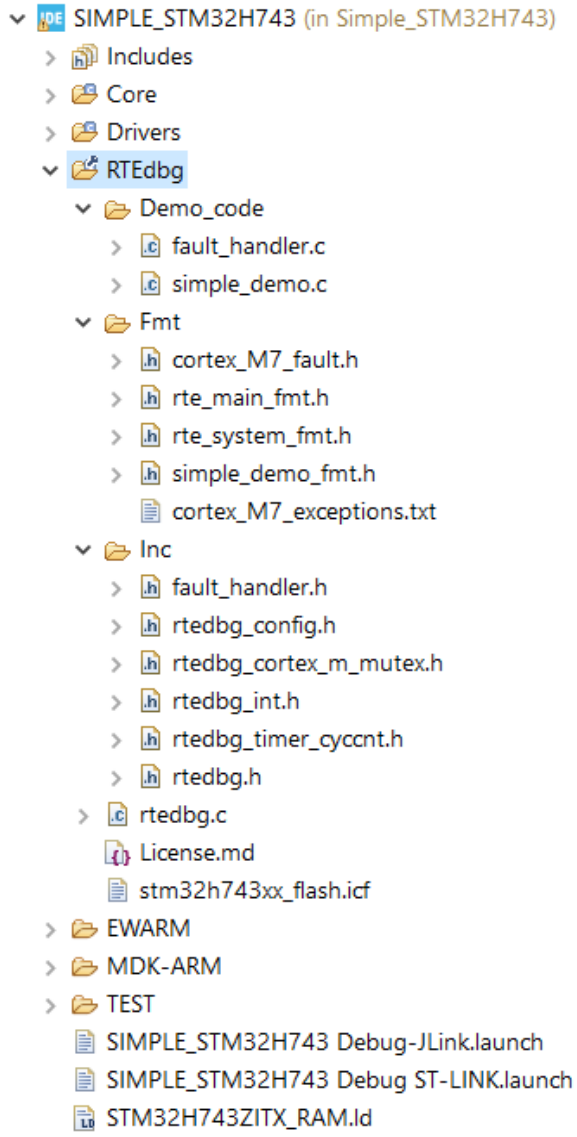
The **bold** marked files are RTEdbg library files that must be added to each project. The green marked files are the hardware driver files. The **brown** marked files are library files that have been modified according to the requirements of the demo project. For a particular user's project, take the versions from the c:*RTEdbg\Library* folder and modify them according to the project's requirements, or adapt the files from one of the demo projects.

Note: See also the descriptions in the comments of the driver files and in the Readme files.

The following files have been added to demonstrate some of the *RTEdbg* functionality (see the ***Fmt*** and ***Demo_code*** folders).

`simple_demo_fmt.h`	Format definitions for the demo code in the simple_demo.c.
`simple_demo_fmt.c`	Demonstration of the functionality of the RTEdbg library.

`fault_handler.c`
`fault_handler.h`
`cortex_M7_fault.h`
`cortex_M7_exceptions.txt`
	Cortex-M7/M4 Exception Handler and Exception Message Logging Demo and the header file
	Format definitions for the Cortex-M7 exception handler
	Exception vector names for the Cortex-M7 based devices



The image on the left shows the files that are part of the *Simple_STM32H743* project.

Exception handler example
Data logging demonstration

Format definition files

Header files (RTEdbg headers and configuration, drivers)

RTEdbg library source code

The startup code in the file *startup_stm32h743zitx.s* has been modified to start the exception handler instead of the default infinite loop. The exception handler demo is part of the demo projects. It shows how to log the exception data by calling the *RTEdbg* macros – see [ARM Cortex-M4 / M7 Exception Handler Example].

```
Default_Handler:
//*********************************************
//Infinite_Loop:
  b start_exception_handler
//  b  Infinite_Loop
//*********************************************
```

The above snippet is from the GCC version of the *startup_stm32h743zitx.s* file. The *Simple_STM32H743* folder also contains versions for the IAR EWARM and Keil MDK toolchains, including the linker setup.

The following lines have been added to the link definition file *STM32H743ZITX_RAM.ld* to set the circular data logging buffer to address 0x24000000 (start of AXI SRAM1). The following example is for the GNU linker.

```
/* RTEdbg data logging memory section */
. = ALIGN(4);
.RTE (NOLOAD):
{
  *(RTEDBG)
  *(RTEDBG*)
} >RAM_D1
```

Time Synchronization Between the Embedded System and the Host

All messages are logged with a timestamp. There are only a limited number of bits available for the timestamp in each message, and the available timestamp range is exceeded once in each timestamp period. The message decoding application *RTEmsg* detects the overflow and correctly updates the full (long) timestamp if the time between successive logged messages is less than a quarter of the timestamp period (see also Note 1). Thus, if messages are logged frequently enough in the embedded system, and only the sequence of events and the relative time between events are important for analyzing system performance, it is not necessary to log the long timestamp value with the rte_long_timestamp() function. This function enables absolute time synchronization (e.g., time from power-on or reset) between the embedded system and the host where message decoding is performed.

The long timestamp value is important:

- when you need information about how much time has passed, for example, system reset,
- if there may be longer pauses between logged messages (over a quarter of timestamp period),
- when there may be brief interruptions in the data transfer to the host – for example, because there is not enough bandwidth to transfer data to the host, or because the operating system on the host has not allocated enough processor time to the data transfer software.

Notes:

1. The timestamp can only be processed for messages that were actually written to the circular buffer (not disabled by the message filter) and could be transmitted to the host (not lost due to a buffer overflow). The length of the circular buffer in which messages are logged is limited. The programmer should ensure that at least one long timestamp is recorded in it. This is especially important when bursts of messages may occur.
2. Recording long timestamps too frequently can take up too much space in circular storage.

The following code has been added to the file *stm32h7xx_it.c* file in the demo project. This code logs the long timestamp periodically (every 8 ms in this example). This is just one of many possible implementations. Calling *rte_long_timestamp()* in an interrupt program like this guarantees that the long timestamp is always logged. This also ensures that the values of the upper 32 bits of the timestamp are updated correctly. The function *rte_init()* should be called before this interrupt is enabled.

The example below shows how long timestamps are logged periodically in the demo.

```
void SysTick_Handler(void)
{
  /* USER CODE BEGIN SysTick_IRQn 0 */

  /* USER CODE END SysTick_IRQn 0 */
  HAL_IncTick();
  /* USER CODE BEGIN SysTick_IRQn 1 */
  // Log the long timestamp every eight tick periods (8 ms) <= code added to the Systick handler
  if ((uwTick & 7u) == 0)
  {
      rte_long_timestamp();
  }
  /* USER CODE END SysTick_IRQn 1 */
}
```

The following folders have been added to the demo project include path list – see below.

Project properties ⇨ C/C++ Build ⇨ Settings ⇨ MCU GCC Compiler ⇨ Include paths:

```
../RTEdbg/Inc        <= Note: Add the CPU and timer drivers to this folder
../RTEdbg/Fmt        <= Note: Add the project specific format definition files to this folder
```

Note: Do the same for the MCU G++ compiler if you have a C++ or C/C++ project.

The following data logging modes can be evaluated with the Simple Demo project code:

- Data logging in 'post-mortem' mode (also allows snapshots and streaming data transfer).
- Data logging in single shot mode.
- Exception logging (see the *fault_handler.c* file).

See the description in the *simple_demo.c* file for how to select the 'post-mortem' or single shot logging mode.

Project Pre-build Settings

Each data logging macro requires format ID and filter number parameters. Their values are automatically assigned by the *RTEmsg* application before compilation is started. The application also checks the syntax of the format definitions. To perform this step, it is necessary to add the pre-build check to the IDE settings (or to the Makefile).

See the description below for how to do this for an Eclipse based IDE like the STM32CubeIDE:

Project Properties => C/C++ Build => Settings => Build Steps => Pre-build Steps => Command:

```
c:\\RTEdbg\\RTEmsg\\RTEmsg.exe . ..\\RTEdbg\\Fmt -N=10 -c
```

Note: In some IDEs, it is necessary to use one backslash in the file paths instead of two.

For information on how to set the pre-build step for Keil MDK or IAR EWARM IDEs, see the Verifying Format Definitions with RTEmsg. Demo projects already contain these settings.

The *RTEmsg* application reports error messages to the console and to the Errors.log file. Most error messages are self-explanatory. The RTEmsg Error Table contains a more detailed description for common errors that most programmers make and for errors where a short description may not be sufficient. The way the errors are reported is configurable to allow the IDEs to recognize the file names and line numbers correctly – see the description of the *RTEmsg* **-e** command line parameter. Some examples for popular IDEs are given in Verifying Format Definitions with RTEmsg.

The simple_demo.c Demo Code

The *simple_demo.c* file demonstrates how to use data logging macros and functions. The corresponding format definitions are in the header file *simple_demo_fmt.h*. See the comments for a more detailed description of the various data logging macros and how to throw an exception to test the exception handler. To keep the code simple and portable, the demo does not use hardware peripherals such as an ADC for data generation. For this reason, the data was processed using mathematical formulas (to get data that changes over time).

See also the description of the exception handler in the ARM Cortex-M4 / M7 Exception Handler Example, which is also part of this demo. The fault handler is fired when the code shown below is enabled.

```
#if defined INVALID_ADDRESS
        // The fatal error will be triggered after 600 ms
        if (uwTick > 600U)
        {
            (void)*INVALID_ADDRESS;    // Read from invalid address
        }
#endif
```

Note: Disable this part of code if you intend to test how to repeatedly get snapshots from a running system.

Compile and Test the Demo Code

Settings for GCC, Keil MDK, and IAR EWARM are already included in the project. Compile the project and download the code. Data is logged to the circular buffer that is part of the RTEdbg data structure (*g_rtedbg*). The complete data structure must be transferred to the host computer where decoding is performed. The structure header contains information such as timestamp frequency, data logging mode, number of bits used for format ID, etc. This information is necessary to properly decode the logged data.

The batch files for transferring data to the host are in the TEST folder. Examples are provided for ST-LINK and J-Link debug probes. See the *Readme.md* file in this folder for details. Run the *Snapshot_ST-LINK.bat* or *Snapshot_JLINK.bat* batch file to get a snapshot of the logged data, decode the binary file, and display the data. Notepad++ and a CSV viewer (e.g., Flow CSV) should be installed as described in Installing the Library and Tools to start the data display automatically. The execution of the code is paused during the data transfer (data snapshot). The message filter value is set to zero to temporarily disable message logging during the RTEdbg data structure transfer to the host. The batch file can be started any number of times without affecting the execution of the firmware. The same batch file can be used to transfer data from a running application or after firmware execution has been stopped by a breakpoint, system exception, or manually.

Set RTE_SINGLE_SHOT_ENABLED to 1 in the *rtedbg_config.h* configuration file to enable single shot data logging. This allows the firmware to select either single shot or 'post-mortem' logging mode during RTEdbg initialization. The second parameter of the *rte_init()* function defines the logging mode.

The following examples show how to do this using a debug probe. The *g_rtedbg* data structure is located at address 0x24000000 and the circular buffer size is 4096. Both examples provide the same functionality.

Example 1: Data transfer using the STMicroelectronics ST-LINK command line debug probe application

```
"c:\ST\STM32CubeProgrammer\bin\STM32_Programmer_CLI.exe" -c port=SWD mode=HOTPLUG shared
-q --read 0x24000004 4 "Filter.bin" -fillmemory 0x24000004 size=4 pattern=0 --read
0x24000000 0x4028 "data.bin" --write "Filter.bin" 0x24000004
```

Example 2: Data transfer using the Segger J-LINK command line debug probe application and a command file

```
"c:\Program Files\SEGGER\JLink\JLink.exe" -If SWD -ExitOnError -CommandFile
Snapshot_JLINK.cmd
```

File: Snapshot_JLINK.cmd

```
speed 12000                              <= Set the speed that is supported by your debug probe
device Cortex-M7
savebin Filter.bin 0x24000004 0x4        // Save the current message filter value
W4 0x24000004 0                          // Pause data logging (set filter to zero)
savebin data.bin 0x24000000 0x4028       // Save the g_rtedbg structure contents
LoadFile Filter.bin 0x24000004 noreset // Restore the message filter (resume data logging)
exit
```

The message filter value is located at the address 0x24000000 + offset 4. The batch file reads the filter value and writes it to the temporary file. The filter is set to zero to pause the data logging without stopping the firmware execution. Then, the complete contents of the *RTEdbg* data structure are transferred to the *data.bin* file. Finally, the filter is reset to its previous value and data logging resumes. If the ST-LINK debug probe is used for data transfer, the shared mode must be enabled in the debug settings. This allows simultaneous access for the debugger and the STM32 Programmer CLI application. The J-Link debug probe driver supports the shared mode of operation by default.

Binary Data Decoding

Raw binary data is logged during code execution using the *RTEdbg* library functions. The *RTEmsg* command line application decodes (prints) this information according to the programmer's format definitions into specified output files or the default *Main.log* file. For a complete description of data decoding, see RTEmsg MESSAGE DECODING APPLICATION.

All files are created in the specified output folder. See an example of a batch file that decodes the binary file *data.bin* and writes the files to the *output* folder.

```
C:\RTEDBG\RTEMSG\RTEMSG.EXE output "..\RTEdbg\Fmt\" -N=10 data.bin
```

The first two parameters are the name of the output folder and the name of the folder containing the format definition header files. The -N=x parameter specifies the number of bits for the format ID that identifies the format definition. It must have the same value as the compile time parameter RTE_FMT_ID_BITS in the *rtedbg_config.h* file.

Note: Output files cannot be created if they are locked, e.g., by text editor. Many programs do not lock files, and it is also possible to set up automatic content updates – see how to do this with Notepad++.

Default Log Files

Information generated during binary file decoding is written to the Main.log file, unless otherwise specified in the format definitions. Any errors detected during format definition processing and binary data decoding are written to the Errors.log file. See The Default Output Files section for a detailed description of all output files that are created either by default or when enabled with a command line argument.

The *Main.log* file contains a header with information from the RTEdbg data structure header: how logging is configured, timestamp frequency, etc. By default, data is printed to this file unless otherwise specified by its format definition. The message number, timestamp value, and format ID name are printed to *Main.log* for every message whether the programmer defines them or not – see below for a snippet of *Main.log*.

```
N00112, 1.950121, MSG0_CAN_TIMEOUT: CAN Bus timeout
```

The first value is a sequential message number, followed by the timestamp and format ID name. The colon is followed by the programmer-defined text (in this example, event description text).

The Stat_main.log file contains basic statistical information. It also shows the messages with the highest frequency of occurrence (number of repetitions) and the messages with the highest circular buffer usage (size in bytes). This information is useful to the programmer, for example, in determining which messages to disable if

the data transfer bandwidth to the host is too low and data loss is occurring. Other statistics files must be enabled with the *RTEmsg* -stat command line argument.

Additional Tips and Tricks

1. **Format definitions for writing to *Main.log* and other files should:**
   - Ensure good readability – note that the log files are typically large, and important data can easily be missed during analysis.
   - Use consistent formatting (this also improves readability).
   - Sort the data into files and prepare them for visualization (e.g., also write the data into a CSV file).
   - Use statistics to automatically find data and time extremes – see Statistical Features.
   - Prepare the data in a format that will be suitable for automatic processing by the software.
     See also: 7 Best Practices for Readable Log Files

2. **Errors reported in Errors.log and Main.log files**
   Many error detection and reporting mechanisms are built into the *RTEmsg* application for decoding logged data. This is to avoid false messages that could mislead the tester. When individual errors are reported in the log files, the tester must pay attention not only to the messages for which the errors were reported, but also to the message that follows. If the error occurs every time a certain type of message is decoded, then the error is usually either in the format definition or in the macro call for logging that message (i.e. incorrect data logged). See also RTEmsg Error Messages.

3. **Error found in a format definition after data decoding**
   If the programmer finds errors in a format definition, he can fix them and re-decode the binary file. There is no need to recompile the project, repeat testing, and wait for a hard-to-repeat problem to occur again. The advantage of the *RTEdbg* toolkit is that the complete format definitions reside on the host computer and can be modified if an error is found in any of the format definitions. The disadvantage is that for a given firmware version, it is necessary to use the appropriate version of format definitions for decoding. This is why we need to archive them along with the code of the project. See also How to Avoid Problems When Adding or Changing Format Definitions.

4. **Data logging in production code**
   It is recommended to leave the data logging in the production code (do not disable it by setting RTE_ENABLED to zero). Removing the code changes the timing of the firmware. It also reduces the ability to find the cause of problems that occur after the system is deployed in the field.

5. **Enabling/disabling message groups with message filter in the firmware**
   Since the amount of memory in the embedded system is limited, it is important that the firmware store data that is important according to the type of problem the programmer or tester is investigating. This can also be done by the firmware using the message filter. Three functions are provided: *rte_get_filter()*, *rte_set_filter()*, and *rte_restore_filter()*. The firmware can start or stop (pause) logging on a specific event (trigger). The firmware can also choose to either log until the circular buffer is full (single shot logging mode) or overwrite old data until stopped ('post-mortem' logging). Stopping data logging has no effect on the execution of the program code. The only difference is slightly faster execution of the data logging functions because they discard the data when the message group is not enabled.
   Examples:
   A) For an IoT system, a message filter value can be used to remotely select which message groups to log and forward the logged data to the host for analysis. A custom trigger can be used to start or stop logging to get the most relevant data.
   B) For an embedded control system (e.g., a mobile robot), the firmware can log data until a fatal event

is detected and then completely stop logging (by setting the message filter to zero), but not stop executing control and protection algorithms. The programmer or service technician can transfer data from the embedded system (after arriving on site), for example by connecting to it with a notebook computer, and analyze it or send it back to the developers.

6. **Add or change format definitions in new firmware releases**
   If the firmware of the embedded system is only instrumented for internal test purposes, it is not a problem if the format ID numbers change from one firmware revision to the next. If we want to add new format definitions or change existing ones for a specific project, and we want to use the same set of format definition files for both the existing and the future firmware versions, then we have to follow certain rules – see How to Avoid Problems When Adding or Changing Format Definitions. A single set of format definition files for decoding multiple firmware versions solves the problem of knowing exactly which version of the format definitions to use to decode a particular binary data file from the embedded system. This is useful, for example, for data transmitted from embedded IoT systems, especially when many IoT modules have not received updated firmware.

For Individuals Interested in Experimenting with Format Definitions

The *TEST* folder of the *Simple_STM32H743* project contains the *data.bin* file that was downloaded from the embedded system when an intentionally bad memory access triggered a bus error. This code was added to demonstrate the error handler (see ARM Cortex-M4 / M7 Exception Handler Example). The *TEST\output* folder contains the decoded contents of *data.bin*.

Two batch files have been prepared for data decoding with different locale settings:

- **`Decode.bat`** – The Windows computer locale setting ("Time & language" => "Language & region") forces the use of either a decimal point (.) or a decimal comma (,) as the separator for printed floating-point numbers.
- **`Decode-C_locale.bat`** – The RTEmsg *-locale=C* command line argument forces the decimal point (.) as the separator for printed floating point numbers.

The contents of the *TEST\output* folder were created using the second batch file. Run one of these two batch files after modifying the format definition file to re-decode the *data.bin* file.

Feel free to modify the format definition files in the *`Fmt`* projects folder. Use a text editor that supports UTF-8 character encoding. Most IDEs and text editors have UTF-8 support enabled by default. Use an UTF-8 compatible viewer or editor to view the files in the *output* folder.

The following format definition shows how to print messages using international fonts. You can combine any number of fonts in the same message since the UTF-8 encoding is transparent to the *fprintf()* function. The following format definition (defined in *`Fmt\simple_demo_fmt.h`*) is an example of such a message.

```
// MSG0_TRIGGER_SYS_ERROR
//  "\n  --> [Triggering a system error]-[触发系统错误]\n"
```

It is used in the following data logging macro (*`simple_demo.c`*).

```
    RTE_MSG0(MSG0_TRIGGER_SYS_ERROR, F_SYSTEM);
```

3 RTEdbg LIBRARY

This chapter describes data-logging features that can be used to instrument C and/or C++ program code. The logged information can be used to identify problems, check that the code is working as it should, measure the execution time of different parts of the code, test overall system performance, identify bottlenecks, and detect errors that may not be caught by traditional testing methods.

3.1 Data Logging Macros and Functions

All data-logging functions are reentrant. There are no critical sections within the data-logging library. Functions are non-locking and never return a busy state. No high-priority events are lost because a low-priority part of the firmware would be in a critical region while logging with this library. No recorder busy errors are possible during data logging. The only error that can occur is a buffer overflow during continuous message logging (streaming mode) if the data cannot be transferred to the host at least as fast as it is written to the circular buffer. Message filtering allows the selection of groups of messages to be logged during each test, if all data cannot be logged simultaneously due to data transfer rate limitations, or if a longer history is desired for some message groups. The rte_init() function must be called before data logging is started, e.g., after a system reset or reboot (best after the processor clock has already been initialized if this clock is the source to the timestamp timer).

The data logging macros and functions are robust. If programmers provide incorrect logging parameters, incorrect data would be logged, but this would not affect the execution of code instrumented with these logging functions. The only problem is the four macros that take the data address as a parameter. This address must be in a readable area of memory.

Format ID and Filter number: All data logging macros have a parameter called ***fmt_id***. It specifies which of the format definitions will be used to decode and sort the data on the host computer. The programmer does not have to worry about this because the RTEmsg MESSAGE DECODING APPLICATION automatically assigns numbers to the format IDs. The macro parameter ***filter_no*** is the filter number and defines which bit of the 32-bit filter variable is used to enable message logging. See the Format ID and Filter Naming Conventions section for a detailed description of this parameter and naming conventions.

Maximum message size: The maximum message size is defined by the parameter RTE_MAX_SUBPACKETS. The subpacket size is four 32-bit words or 16 bytes.

Maximum message size (bytes) = RTE_MAX_SUBPACKETS × 16

The RTE_MAX_SUBPACKETS is defined by the programmer in the *rtedbg_config.h* file.

The message size limit was added to indirectly define the maximum execution time of the data logging function and to prevent an erroneously long message from overwriting a large portion of the circular message buffer. The maximum message length should be defined as long as necessary and no longer. This avoids the problem of some part of the firmware repeatedly logging very long (or erroneously long) messages, thereby crowding out other important messages. Messages that are too long are either discarded or shortened to the maximum length depending on the value of the compile time parameter RTE_DISCARD_TOO_LONG_MESSAGES.

Programmers must ensure that the maximum possible message size logged with a call to e.g., *RTE_MSGN()* is greater than or equal to the maximum data structure intended to be logged in the buffer. For information on how to check whether the data structure fits into the maximum possible message, see Compile Time Verification if a Structure Fits Into a Message.

Note: In this manual the symbol **N** is used instead of the compile time parameter RTE_FMT_ID_BITS. It defines how many bits are used for the format IDs (message identifiers).

Timestamps: Each message logged to the circular buffer contains a timestamp. The size of the timestamp included with each message is 32 – 1 – **N** (bits). See Timestamps for a complete description. The *RTEmsg* application decoding the data on the host computer uses this data to determine the relative times between messages or events. With message calls for logging long timestamps (rte_long_timestamp()), it is also possible to log the absolute value of the time in the embedded system, e.g., the time from power-on. In such a case, an additional 32 bits are logged in the circular buffer for a long timestamp, for a total of 64 – 1 – **N** bits.

The following types of macros and functions are available in the *RTEdbg* library

- Macros for Logging Short Messages and Events: RTE_MSG0 … RTE_MSG4
- Macros for Logging Data Structures, Strings, and Buffers
- Message Filter Manipulation Functions
- rte_init - Initialize logging data structure
- Timestamp Support Functions

Macros for Logging Short Messages and Events: RTE_MSG0 … RTE_MSG4

Data-logging functions are called by macros that combine the two or three macro parameters into a single function parameter. The goal is to reduce the overall footprint of data-logging function calls and speed up execution.

```
void RTE_MSG0          (fmt_id, filter_no)                          // Event logging (no data)
void RTE_EXT_MSG0_y    (fmt_id, filter_no, short_data)              // Event logging + up to 8 bit data (y = 1 … 8)
void RTE_MSG1          (fmt_id, filter_no, data1)                   //  1 × 32 bit data logging
void RTE_EXT_MSG1_y    (fmt_id, filter_no, data1, short_data)       // 1 × 32 bit + up to 7 bits data
void RTE_MSG2          (fmt_id, filter_no, data1, data2)            // 2 × 32 bit data logging
void RTE_EXT_MSG2_y    (fmt_id, filter_no, data1, data2, short_data)// 2 × 32 bit + up to 6 bits data
void RTE_MSG3          (fmt_id, filter_no, data1, data2, data3)     // 3 × 32 bit data
void RTE_EXT_MSG3_y    (fmt_id, filter_no, data1, data2, data3, short_data) // 3 × 32 bit data + 5 bits
void RTE_MSG4          (fmt_id, filter_no, data1, data2, data3, data4)      // 4 × 32 bit data
void RTE_EXT_MSG4_y    (fmt_id, filter_no, data1, data2, data3, data4, short_data) // 4 × 32 bit + up to 4
void RTE_RESTART_TIMING()   // A special version of the rte_long_timestamp() function call
```

Note: The macros are specified here with void to show that the functions they call return no value, and the parameters are aligned to show that the first two are the same for all macros.

Description: The functions allow logging data with a size of zero to four 32-bit words (signed, unsigned, float, struct, etc). The macros *RTE_EXT_MSG0…4()* allow the logging of additional data bits – in addition to the 32-bit values defined with the data1 … data4. The value '**y**' can be from 1 to 8 and defines how many bits of additional information are stored in the message without increasing the number of words stored in the circular buffer – see Extended Messages RTE_EXT_MSG0_y … RTE_EXT_MSG4_y and Short Message Type Comparison.

Function parameters:

uint32_t **fmt_id** – Format ID identifies the format definition to be used for message decoding
uint32_t **filter_no** – Number of a filter that enables a message group (0 … 31)
rte_any32_t **data1 … data4** – Data to be stored in the circular buffer
uint32_t **short_data** (1 to 8 bits) – Additional short data (from 1 to 8 bit data)

The type ***rte_any32_t*** is defined in *RTEdbg.h*. It allows the use of values with a maximum size of 32 – from char to float. All pointers to const strings, arrays or structures used as parameters for the above macros must be cast to (void *) or (uint32_t).

Notes:

1. It is not possible to use 64-bit or larger values as parameters for the macros described above. They must be logged by calling the *RTE_MSGN()* macro with the address and size of either the variable or the complete structure.
2. Function parameters are of type ***rte_any32_t*** (almost any value with a maximum size of 32 bits is acceptable). If a different type (not defined with the rte_any32_t type definition) is used, the compiler will return the following error (example for the GNU C compiler):

   error: cast to union type from type not present in union

   Programmers must cast the variable type to one of the types defined in *rte_any32_t* or add the new type to the *rte_any32_t* union. Only variables with a size of 32 bits or less are allowed.
   If the parameter is a pointer, it should be cast to either (void *) or (uint32_t).
3. **Important:** Some compilers do not allow casting values to union type *rte_any32_t*. In this case, the RTE_USE_ANY_TYPE_UNION definition must be disabled – removed from the *rtedbg_config.h* file.

   This has the following effects:

   a) The compiler cannot check whether the value fits into 32 bits (it will truncate it if the programmer specifies a 64-bit value, for example).

   a) A 32-bit floating point value (float) cannot be used as parameter because the compiler converts the value to an unsigned integer if the value would be used as parameter for the data logging macros *RTE_MSG1()* … *RTE_MSG4()*. In this case the macro *RTE_MSGN()* must be used and a pointer to a float value or a structure containing float values should be used. See Workaround for Missing/ Disabled Compiler Support for the rte_any32_t Union for a description of solutions. Also follow the description if you have a C++ or mixed C/C++ project.

Extended Messages RTE_EXT_MSG0_y … RTE_EXT_MSG4_y

The macros *RTE_EXT_MSG0_y()* to *RTE_EXT_MSG3_y()* allow the logging of short values (up to eight bits long) together with the 32-bit values. The packing and decoding of this extended information is transparent to the programmer. The effect of this functionality is a reduction in circular memory consumption and faster data logging in cases where we need to log additional data up to 8 bits long.

During message logging, a single word (referred to here as FMT) containing a format ID and a timestamp is part of each message. In addition to the timestamp and format ID, it is possible to store up to 8 bits of additional information without increasing the consumption of the circular buffer in which the data is logged. Such a message is one 32-bit word shorter than it would be if the short data were logged using a simple macro version (non-extended), but this method of logging has its price if you have a lot of different messages. In the FMT word, N bits are reserved for the format ID. To record additional bits, the same bit space is used as for the format ID (low bits), which reduces the number of different messages that can be recorded. Therefore, it makes sense to use the *RTE_EXT_MSG()* macro type for messages that are recorded frequently or when it is not necessary to record a very large number of different messages for a certain project. The programmer must also decide how many bits of the FMT word to allocate for format IDs and additional data bits and how many for the timestamp. In general, a small number of timestamp bits reduces the resolution of the time information.

To record additional bits, it makes sense to use a macro type that allows only the necessary number of bits to be recorded. For example, if we want to store the information whether a state is true or false at the time of the event, we use the macro *RTE_EXT_MSG0_1()*. If we want to store information about the 6 states of the state machine, three bits are sufficient to encode 6 values. The Short Message Type Comparison table shows how

many format IDs are occupied by each message type. Multiple values can be packed into a 32-bit long word – see Pack Multiple Short Variables Into a 32-bit Word or Structure. The logging format is flexible, allowing programmers to make tradeoffs between how many different messages can be logged and how many bits are available for timestamp and additional data information.

Caution: There is no compilation or runtime check if the additional data for the *RTE_EXT_MSG()* type macros contains more data bits than can be stored in an extended message. The extra bits are discarded. The programmer must ensure that a macro is used that allows a sufficient number of bits to be recorded. The data logging functions are robust and will not crash if incorrect values are specified for the message filter, format ID, or additional data.

Table: Short Message Type Comparison

Macro name	Number 32-bit data words logged (without the FMT)	Additional data bits logged	Total 32-bit data words in the circular buffer (including the FMT)	Number of format IDs used (occupied)
RTE_MSG0	0	0	1	1
RTE_EXT_MSG0_**y**	0	**y** = 1 … 8	1	2 ^ **y**
RTE_MSG1	1	0	2	2
RTE_EXT_MSG1_**y**	1	**y** = 1 … 7	2	2 ^ (**y** + 1)
RTE_EXT_MSG2	2	0	3	4
RTE_EXT_MSG2_**y**	2	**y** = 1 … 6	3	2 ^ (**y** + 2)
RTE_MSG3	3	0	4	8
RTE_EXT_MSG3_**y**	3	**y** = 1 … 5	4	2 ^ (**y** + 3)
RTE_MSG4	4	0	5	16
RTE_EXT_MSG4_**y**	4	**y** = 1 … 4	5	2 ^ (**y** + 4)
RTE_MSGN(address, length) // length in bytes	(length + 3) / 4	0	(length + 3) / 4 + (length + 15) / 16 // minimally 1	16
RTE_MSGX(address, length) // length in bytes	len / 4 + 2	0	2 + (length / 4) + (length / 16)	16

Workaround for Missing/Disabled Compiler Support for the rte_any32_t Union

This description applies to compilers that do not support casting values to a union type, or to projects where strict C compatibility must be used. The RTE_USE_ANY_TYPE_UNION compile-time parameter must not be defined in *rtedbg_config.h*. The RTE_USE_ANY_TYPE_UNION parameter can be defined for mixed C/C++ projects since its use is disabled for C++ files in the *RTEdbg* header file. In this case, the description below also applies to the use of logging functions in C++ source files.

Two inline functions are included in the *rte_dbg.h* header file:

```
uint32_t float_par(float number);
uint32_t double_par(double number);
```

They do not convert the float to an unsigned integer. They just instruct the compiler to use the unmodified binary value so that the raw float value is logged. The functions allow you to use a floating point value as a parameter for macros from *RTE_MSG1()* to *RTE_MSG4()*. See the example below.

```
float   value1 = 3.14F;
double  value2 = 9e9;
int32_t value3 = -1000;
RTE_MSG3(MSG3_COMBINED, F_DEMO, float_par(value1), double_par(value2), value3)
```

Notes If the value1 were used directly, the compiler would convert the value from *float* to *uint32_t*, and the value would be truncated to 3. The *double_par()* function allows logging a shorter (32-bit) value instead of 64-bit, thus reducing the circular buffer usage. A 64-bit float value cannot be used directly as a parameter. If the programmer wants to log a 64-bit float (double), it must be logged with the *RTE_MSGN()* and its address and size must be provided as parameters. A 64-bit value can also be a part of a data structure – see below.

Macros for Logging Data Structures, Strings, and Buffers

void **RTE_MSGN**(fmt_id, filter_no, address, length)

Description: Log a data structure, buffer, string, etc. It is possible to log messages whose length is not known at compile time, but for such messages it is only possible to print them in string (text) or hexadecimal format (e.g., hex dump) or write them to a binary file (see the description of the %W type in Format Syntax and Type Field Extensions).

Parameters:

uint32_t **fmt_id**	– Format ID that identifies the format definition
uint32_t **filter_no**	– Number of a filter that enables a message group (0 … 31)
void * **address**	– Pointer to the data to be written into the circular buffer
uint32_t **length**	– Length of data in bytes (zero is a valid value)

This macro allows you to log complete or partial buffers or structures. The variables that are part of a structure can have different lengths from one bit up to 64 bits (e.g., 64-bit floating point).

Note: If the fastest possible code execution is required and the structure to be logged is short (contains four words or less), it is more optimal to log it using the macros *RTE_MSG0() ... RTE_MSG4()*.

Limitations of the current *RTE_MSGN()* implementation:

1. The function is optimized for maximum execution speed. The data address must be 32-bit aligned (address divisible by four) if the CPU does not allow unaligned access or if unaligned data access has been disabled by firmware although the CPU core supports it.
2. For more information on instrumenting embedded systems with MPU protection, see RTEdbg and Memory Protection Unit (MPU).

void **RTE_STRING**(fmt_id, filter_no, address)
void **RTE_STRINGN**(fmt_id, filter_no, address, max length)

Description: Log null-terminated string

Parameters:

uint32_t **fmt_id** – Format ID that identifies the format definition
uint32_t **filter_no** – Number of a filter that enables a message group (0 … 31)
char * **address** – Pointer to the data to be written into the circular buffer
The address must be 32-bit aligned if unaligned addressing is disabled for the core.
uint32_t **max_length** – Maximum string length (string length is also limited by maximum message size)

Note: If the length of a zero terminated string is known and the CPU allows access to unaligned data, the macro *RTE_MSGN(string, length)* can be used directly instead of *RTE_STRING()*. This reduces the execution time since the string length does not have to be determined.

Caveat: In general, compilers for 32-bit CPU cores align string variables to word addresses. This is true for string variables, but not necessarily for constant strings. If the string address may not be aligned (e.g., the programmer specifies non-aligned compilation/linking to use memory more efficiently), non-aligned addressing must be enabled for the CPU core. Be careful how you use this macro on processors that do not support access to 32-bit values at unaligned addresses.

void **RTE_MSGX**(fmt_id, filter_no, address, length)

Description: Log message of unknown size at compile time or non-zero terminated strings.

Parameters:

uint32_t **fmt_id** – Format ID that identifies the format definition
uint32_t **filter_no** – Number of a filter that enables a message group (0 … 31)
void * **address** – Pointer to the data to be written into the circular buffer
uint32_t **length** – Length of data in bytes (0 … 255)

This macro allows you to log data whose length is not divisible by 4 and whose length is unknown at compile time, as well as non-zero terminated strings (strings that may contain zeros). The length of such a message is limited to max. 255 bytes or (RTE_MAX_SUBPACKETS × 16 – 1) – whichever is less. The macro should also be used for logging data structures that are on unaligned addresses and where unaligned access is not supported by the CPU core.

Notes:

1. If the message is recorded using the *RTE_MSGX()* macro, then a name beginning with MSGX_ must be used for the message name. The MSGX_ prefix tells the *RTEmsg* application that the message is written in a different format than other messages. If the MSGX_ prefix is not used for the message name, the content of the message may not be decoded correctly.
2. If unaligned data access is possible and the data length is divisible by 4, it is recommended to use the *RTE_MSGN()* macro. The execution speed of logging is typically two to three times slower than logging a message of the same size with *RTE_MSGN()* – see Execution Times, Circular Buffer, and Stack Usage.
3. Messages logged with this macro are not as circular buffer efficient as messages logged with other macros. For example: To log 32-bit data, three 32-bit words are stored in the buffer. When logging with *RTE_MSG1()*, only two are stored. When logging up to 24 bits with *RTE_MSGX()*, two words are used in the buffer.

Message Filter Manipulation Functions

A 32-bit message filter variable can enable or disable up to 32 message groups to either select the relevant messages or reduce message overload. The message filter value can be manipulated via firmware or the debug probe. See Message Filtering for a complete description of message filtering.

void **rte_set_filter**(uint32_t filter_value);

Description: Set a new message filter value.

Parameter:

filter_value – New filter value (zero disables data logging altogether)

If the RTE_FILTER_OFF_ENABLED parameter was set to 1 in *rtedbg_config.h*, the new value will not be set if the current filter value is zero. This feature allows programmers, for example, to disable data logging after a fatal error has been logged and a system restart is required. The history of what happened before the fatal error is preserved long enough for someone to access the embedded system and connect to it via a communication interface, or for data to be transferred over an IoT connection or written to non-volatile memory. Meanwhile, the code continues to run normally.

Use the filter_value RTE_FORCE_ENABLE_ALL_FILTERS to enable all message filters even if filtering is completely disabled (current filter value is zero). If it is not desired that all message groups are enabled, then after calling the function *rte_set_filter(RTE_FORCE_ENABLE_ALL_FILTERS)* it is necessary to call it again with a different filter value. Once the filter value is non-zero, it can be changed again.

Note: Filter number 0 (F_SYSTEM) can only be disabled if all messages are disabled (filter_value = 0). This filter allows important messages such as long timestamps, timestamp timer frequency changes, etc. to be logged. When setting the message filter value via the debug probe, be careful not to set this bit to 0 unless you are setting the entire filter word to zero.

Examples:

```
// Enable a group of messages (set a bit of the filter variable)
rte_set_filter(rte_get_filter() | (1U << (31U - F_FILTER_NAME))
// Disable a group of messages (set a bit of the filter variable):
rte_set_filter(rte_get_filter() & (~(1U << (31U - F_FILTER_NAME)))
```

uint32_t **rte_get_filter**(void);

Description: Returns the current message filter value.

The firmware can read the current filter value and set it to the current value later if the message filter is changed or cleared.

void **rte_restore_filter**(void);

Description: Restore the filter variable to the last non-zero value used before the filter variable was set to zero. Can be used to restore the previous value if message logging has been temporarily disabled with a call to *rte_set_filter(0)*. Each non-zero filter value is also written to a copy of the filter by the rte_set_filter() function.

rte_init - Initialize logging data structure

void **rte_init**(initial_filter_value, init_mode);

Description: Initializes the *g_rtedbg* data logging structure and the timestamp timer.

This function must be called before any data logging is performed to initialize the data logging structures (e.g., after a reset or system reboot). It is preferable to call this function after the CPU clock has been initialized if the CPU clock is used as an input to the timestamp timer. If the function is called from a multitasking system, see the description below.

Note: The *rte_init()* function is not needed (does not have to be called) if the *g_rtedbg* data structure is initialized by the debug probe at the beginning of the code test (e.g. after the breakpoint at the *main()* function is hit). This reduces memory usage for resource-constrained systems. See also the ***RTEgdbData*** utility *Readme* file.

If the data logging structure has already been initialized, it will not be initialized again unless requested by the *init_mode* parameter (RTE_RESTART_LOGGING or RTE_SINGLE_SHOT_AND_ERASE_BUFFER). If a reset or system reboot occurs during data logging and the data logging structure header is intact, logging will continue. Otherwise, the entire data structure is reinitialized (erased) as it was after power-up.

Parameters:

uint32_t **initial_filter_value** – specifies which filter numbers are enabled.

If the *initial_filter_value* is RTE_FORCE_ENABLE_ALL_FILTERS, all message groups will be enabled, even if filtering has been completely disabled by setting the filter to zero.

Use a zero value if data logging is to be enabled later (e.g., using a software trigger).

uint32_t **init_mode** – Specifies whether to continue or restart data logging, and whether to enable the 'post-mortem' or single shot data logging mode (see below).

init_mode values:

a) **RTE_CONTINUE_LOGGING**

The data logging structure is not completely cleared if its header has already been initialized (data logging continues afterwards). Use this value if you want message logging to continue after a watchdog or software reset, for example. The data in the circular buffer is not cleared, which makes execution very fast. Only the config ID and buffer size are set in the *g_rtedbg* data structure, and the timestamp timer is initialized.

b) **RTE_RESTART_LOGGING**

The data logging structure header and circular buffer are completely cleared each time the *rte_init()* function is called. In general, this value should be passed to the function when the firmware detects a power-on reset.

c) **RTE_SINGLE_SHOT_LOGGING**

Enables Single Shot Data Logging. The circular buffer is not cleared, only the buffer index is set to zero, allowing fast code execution.

Note: There is a (not very likely) situation where some data would not be written completely and old data in that part of the buffer could produce a false "history". Such data would typically be at the end. See the description in Interrupt During Message Logging Followed by Fatal Exception. The *RTEmsg* data decoding application tries to detect such suspicious messages, but this is not always possible. If the execution of your code is interrupted, e.g., due to a fatal system error, pay attention to the messages at the very end of the history (at the end of the log file), especially if the *RTEmsg* application reports an error.

d) **RTE_SINGLE_SHOT_AND_ERASE_BUFFER**
Enables single shot logging and completely clears the circular data buffer. Partially written (interrupted or preempted and then never completed) messages can be reliably detected.

Notes:

1. The data logging buffer is fully initialized when RTE_SINGLE_SHOT_AND_ERASE_BUFFER or RTE_RESTART_LOGGING is used. The same is true when the data logging mode is changed from single shot to 'post-mortem' or vice versa. This can take a long time if the circular buffer size is large.
2. The following description refers to the example timestamp drivers as included in the *RTEdbg* function library and in the demo examples. The complete timestamp counter is reset during the execution of *rte_init*(). This counter should normally be set to zero to indicate a system restart. Adjust the time-stamp driver if you need a different implementation. See Time Measurement from Power-On.

Why is circular buffer initialization required?
The logging library is optimized for maximum execution speed and does not allow the logging function to pause (wait within the logging function until the buffer has been transferred to the host and there is enough space for a new message). Therefore, there is only one index that points to the location where the next message will be inserted into the buffer. In streaming mode, the application on the host must maintain an index of the data that has already been transferred. In order to detect how much data is in the buffer in a case where there is little data in the buffer (less than its size), the entire circular buffer is overwritten with the value 0xFFFFFFFFFF, a type of data that cannot occur during proper data logging. This allows the host software to determine how many words have already been written to the circular buffer, or if any of the messages have not been completed – see a possible cause Interrupt During Message Logging Followed by Fatal Exception.

Calling *rte_init()* while the multitasking firmware is running
The *rte_init()* function should generally be called during system initialization before logging begins. The message filter value is set to zero to disable logging during initialization. If the function must be called during normal firmware operation, note the following

a) In a multitasking system, any task or interrupt handler could set the message filter value and enable data logging. If the RTE_FORCE_ENABLE_ALL_FILTERS filter parameter is used, the message filter cannot be completely disabled. If logging is triggered by a custom trigger, care must be taken in the implementation to ensure that logging does not start before initialization is complete.

b) If the initialization is started by an RTOS task, make sure that all lower or equal priority tasks can finish logging before the initialization is started. A lower priority task might be preempted during data logging (writing to the buffer), and some of the data might be written to the reserved buffer space after the circular buffer initialization is completed in a higher priority task. See the description of possible solutions in RTOS Task Starvation. See also Data logging in RTOS-based applications.

Initialization execution time
The *rte_init()* function is executed quickly when data logging continues and the buffer does not need to be cleared (*init_mode* has a value of RTE_CONTINUE_LOGGING or RTE_SINGLE_SHOT_LOGGING). The execution time for a complete buffer flush depends on the size of the circular buffer defined by the parameter RTE_BUFFER_SIZE, the CPU clock, memory latency, etc.

3.2 Timestamps

Logged messages contain timestamps. The RTEdbg library functions are reentrant and non-blocking, so they must log absolute timestamp values, not relative (time differences between logged messages) as in many other data logging solutions. In this document, the name timestamp is used for the short version of the timestamp that is part of each message. The length of the timestamp in bits depends on the number of bits used for the format ID (**N** in this document). The short timestamp length is 32 – 1 – **N** (minimum 15 bits). A long timestamp is a value recorded with the rte_long_timestamp() message. Its total length is 64 – 1 – **N** bits (minimum 47 bits). The programmer defines which timer is used for the time measurement and the counter divisor value. If long timestamp information is not logged in the circular buffer, only the relative times between events can be calculated on the host computer, not the absolute time (e.g., time from power-on). If messages are logged at a high enough frequency (see below), the relative time between any pair of messages can be reconstructed without using the *rte_long_timestamp()* function. See also Time Measurement from Power-On.

The *RTEmsg* data decoding application can determine the time between logged messages when:

- The time between consecutive logged messages is less than a quarter of the timestamp period. This applies to messages that are actually logged and sent to the host (not blocked by a message filter or lost due to a buffer overflow). For real-time systems, where events and other data are typically logged at least once per ms, one message per quarter of the timestamp period is not a problem. For systems where this is difficult to achieve, the time stamp period can be extended by reducing timestamp resolution (using parameter RTE_TIMESTAMP_SHIFT or timestamp counter prescaler).
- The firmware (or debugger) does not pause data logging for long periods by setting the message filter to zero or not call the logging functions at all.

At least one long timestamp message must be in the circular buffer if the *RTEmsg* is also to determine absolute time (e.g., time since reset). The programmer should estimate how much data will be logged in a given time period and call *rte_long_timestamp()* often enough.

The **timestamp period** is defined by the following equation:
(2 ^ (32 – 1 – **N**)) * (2 ^ RTE_TIMESTAMP_SHIFT) / timestamp_frequency
It specifies the time in which the timestamp contained in each message is wrapped around (overflow occurs).

Example: **N** = 12, RTE_TIMESTAMP_SHIFT = 1 (divide by 2), and timestamp_frequency = 2 MHz
timestamp_period = 2^19 / (2e6 / 2) = 524.3 ms (1 µs timestamp resolution)

The long timestamp has an additional 32 bits (51 total for this example) and would not overflow for 71 years at 1 µs resolution. See the timestamp period calculator in the spreadsheet file ***"RTEdbg calculator.ods"*** (in the ***RTEdbg/DOC*** folder). It allows you to estimate recommended values for **N** and RTE_TIMESTAMP_SHIFT.

Timestamp Support Functions

void **rte_timestamp_frequency**(uint32_t frequency);

This function logs the timestamp timer frequency. It only needs to be called if the firmware changes the timestamp timer frequency on the fly – e.g., the processor frequency is adapted to the needs of the embedded system and the same clock is used as the input for the timestamp timer.

How to avoid timestamp timer frequency changes: Use a peripheral timer for timestamping, and change the timer's prescaler immediately after the clock frequency is changed to keep the timer counting independent of the frequency. The timestamp timer frequency is also written to the header of the *g_rtedbg* data structure. It is

sent to the host with all other logged data and is used by the *RTEmsg* application for message decoding.

Note: If the microcontroller has a clock security system that automatically switches to a backup clock source when the current clock source is not running its interrupt handler should log the new clock frequency with this function when the CPU core clock is also timestamp timer clock.

RTE_RESTART_TIMING() – a special version of *rte_long_timestamp()*

This macro logs a message that is a request to restart time decoding in the *RTEmsg* application. This macro logs a long timestamp value of 0xFFFFFFFF – a value that does not appear during normal message logging. This information tells the *RTEmsg* data decoding application that the following timestamps are not continuations of previous ones – the *RTEmsg* application restart statistical timing data processing. Use the macro after a system restart, after waking from sleep, after data logging has been disabled for a long time, etc.

Note: If a relative timestamp value is used as an input to the statistics, the value calculated as the difference between the current message timestamp and the timestamp of a message logged before the pause is likely to be incorrect. The first time difference (for every message type) processed after this message is discarded from the statistics.

The *RTEmsg* application processes messages in the order they appear in the circular buffer. Each message is assigned a message sequence number. This number is written to the Main.log file along with the message timestamp. Occasionally, the timestamps in the log file are not in sequential order. A message may have a slightly smaller timestamp than the previous one. This is because interrupts are not disabled while messages are being logged to the circular buffer. It is possible for a low priority function (or task) to be interrupted by a high priority interrupt (or task) after space in the circular buffer has already been reserved and the timestamp counter value has not yet been read from the timestamp timer counter.

Timestamp Drivers

This section describes how to implement a time stamp driver. The following two functions must be implemented in all timer driver files (they are called by the logging functions). The ***rte_long_timestamp()* is optional** – implement it if the absolute time information is needed (not just relative time between events).

void rte_init_timestamp_counter()

Initializes the timer used for the timestamp. This function is called by *rte_init()* to initialize the timer. The timestamp counter and the long timestamp value are set to zero in the sample drivers included in the *RTEdbg* distribution. Change this behavior if you do not want the timestamp to start from zero after calling *rte_init()*.

uint32_t rte_get_timestamp()

Returns the current value of the timestamp counter. A hardware timer used for the timestamps need not be a 32-bit unit. The minimum recommended timer counter resolution is 16 bits. The timer driver file must contain the compile time parameter RTE_TIMESTAMP_COUNTER_BITS which defines the number of timer counter bits. The timestamp counter value must be incrementing. If it is decrementing (such as the ARM Cortex SYSTICK), this function must modify the value read from the timer counter so that the function returns an incrementing value. See the *rtedbg_timer_systick.h* driver for how this is implemented.

void **rte_long_timestamp**(void);

The function logs the long timestamp value to the circular buffer and enables absolute time synchronization between the embedded system and the host (absolute time from, for example, power-on and not just relative time between logged messages). This function is part of the time stamp driver. It is important to call it periodically for systems where data loss may occur if there is insufficient data bandwidth to transfer data to the host. The *rte_init()* function should be called before long timestamp logging is enabled. Call the function periodi-

cally if you plan to stream data to the host and there would be data loss either because of insufficient bandwidth or because the program on the host would not get enough processing time.

The upper part of the timestamp (upper 32 of the 64 – 1 – **N** bits total) is updated when the firmware detects that the lower part of the timestamp (hardware counter) has overflowed. The upper part of the long timestamp variable is updated regardless of whether logging is enabled or disabled with a message filter (F_SYSTEM). After data logging is re-enabled with the message filter, the correct long timestamp value is logged.

Custom versions: A developer may replace this function with a custom version, if there is a better way to generate at least a 48-bit timestamp – e.g., with two linked hardware timers (e.g., 32-bit + 16-bit), an interrupt routine triggered by a timer counter overflow, etc.

Check RTEdbg and Memory Protection Unit (MPU) if you'll use a MPU in your project or protected RTOS.

Note: The *rte_long_timestamp()* function, as implemented for the demo timestamp drivers, is not reentrant. An erroneous increment of the upper part of the timestamp could only occur if an interrupt (or task switch) occurred during long timestamp logging and the interrupt program or another task called this function again. The likelihood of this happening in practice is very small, but it is better to avoid the problem anyway. Therefore, it is wise to include long timestamp logging in only one part of the code – for example, in an interrupt program that runs periodically.

The timer's clock frequency may be divided by the prescaler to get the desired timestamp frequency value. If the timer does not have a prescaler, such as the ARM Cortex CYCCNT, the value must be divided after reading the timestamp from the counter. The data logging functions are designed for the fastest possible execution, so only a division by a power of 2 is implemented. Therefore, shifting is used instead of division. The parameter **RTE_TIMESTAMP_SHIFT** defines the division factor. The minimum value of this parameter is 1, so the value read from the timer counter is always divided by at least 2. The reason for such a minimum value was a more optimal (faster) code. If the peripheral timer with prescaler is used for time counting, its prescaler must be smaller by a factor of 2, since the value is divided by (at least) 2 during message logging.

The value of RTE_TIMESTAMP_SHIFT must be less than (32 – 1 – **N**) if a 32-bit timer is used. This ensures that the high bit of the timestamp written in the message also changes. The *RTEmsg* data decoding application uses this bit of the timestamp to reconstruct the upper 32 bits of the timestamp, even if the long timestamp value is not logged by the *rte_long_timestamp()* function. See sample drivers for timers that do not have a 32-bit counter - how the value of RTE_TIMESTAMP_SHIFT is checked.

The timestamp frequency value is required to properly decode the time data on the host computer. The macro

```
#define RTE_GET_TSTAMP_FREQUENCY()
```

must be defined in the *rtedbg_config.h* configuration file. The value it returns can be a constant value, the name of a variable that contains the timestamp frequency value, or the name of a function that returns the current frequency value.

The RTE_TIMER_DRIVER must be defined in the *rtedbg_config.h* configuration file (see example below)

```
#define RTE_TIMER_DRIVER "rtedbg_timer_cyccnt.h"
```

Time Measurement from Power-On

The sample timestamp drivers included in the distribution measure time from the time the *rte_init()* function is called, since it resets the long timestamp counter and the working variables. The *rte_init()* function should be called after every system restart, whether due to a software restart or reset, watchdog reset, resume from sleep, or anything else, to reinitialize the *g_rtedbg* data logging structure. It could be corrupted by erroneous firmware execution leading to a restart.

If we want to measure the absolute time from power up, we need to modify the driver according to the following instructions. The example below is for a timer driver based on the CYCCNT timer in the DWT unit (file *rtedbg_timer_cyccnt.h*).

```
__STATIC_FORCEINLINE void rte_init_timestamp_counter(void) {
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk; // Enable the DWT unit
    DWT->CYCCNT = 0;                                // Reset the cycle counter
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;            // and enable it
#if RTE_USE_LONG_TIMESTAMP != 0
    t_stamp.l = t_stamp.h = 0;                       // Reset the long timestamp
#endif
}
```

The code in **bold** must be moved to a separate function that is called only on power-on reset to reset the long timestamp value and the timer counter.

Also consider the following:

1. The *rte_init()* function should be called with the *init_mode* parameter set to
   RTE_RESTART_LOGGING – **after power-on reset** (forces full structure initialization)
   RTE_CONTINUE_LOGGING – **after restart** (to continue data logging)
2. If for some reason no message has been logged for a longer period of time (e.g., after a sleep or a system hang and a watchdog triggers a restart), the macro RTE_RESTART_TIMING() should be called to log the information that data logging has been paused. This information instructs the *RTEmsg* application to restart statistical data processing and time difference measurement because the values may be incorrect (time between a message logged before and after the pause).
3. It is also useful to log application-specific information about the cause of the restart (was it a software restart, a watchdog restart, a wake from sleep, etc.) and any other information that might help determine the cause of the restart. Most 32-bit microcontrollers provide it in peripheral register(s).

Timestamp Drivers Included in the RTEdbg Project

The ARM Cortex SYSTICK timer driver *rtedbg_timer_systick.h* and the ARM Cortex DWT cycle counter *rtedbg_timer_cyccnt.h* driver files are ready to use and don't need to be modified. Other driver files should be adapted to project specifics – e.g., timer clock speed. See also *Readme.md* files in the *c:\B\SWdbg\RTEdbg\Library\Portable* folder or the *github.com/RTEdbg/RTElib/tree/master/Portable*. Additional drivers will be published there.

rtedbg_timer_cyccnt.h It contains the initialization of the ARM Cortex CPU cycle counter CYCCNT. This file can be used with all ARM CPUs that have it (available on Cortex-M3/M4/M7/M33/M55/M85). The RTE_TIMESTAMP_SHIFT value must be adjusted according to the CPU clock and the desired timestamp resolution. The timer value is divided by (2 ^ RTE_TIMESTAMP_SHIFT). **Note:** The CYCCNT counter number is only correct if the program is executed continuously with the debugger without stopping. It is incremented by more cycles than it took to execute the instructions when the code is executed step-by-step with the debugger. The value of the CYCCNT counter is not incremented while the ARM Cortex-M core is in sleep mode. The cycle counter is only incremented when the core is active and executing instructions.
rtedbg_timer_systick.h The ARM Cortex-M SYSTICK timer is a 24-bit down counter. The *rte_get_timestamp()* function modifies the counter value so that the function returns an incrementing value. If the microcontroller does not have a 32-bit timer, use this timer for the timestamp and a 16-bit timer for the system tick.

rtedbg_timer_stm32h7_tim2.h
This example assumes a 64 MHz timer clock and shows how to use the 32-bit timer TIM2 as a timestamp counter. If the CPU frequency is changed during system operation and the timestamp prescaler is adjusted accordingly to the new frequency, then rte_timestamp_frequency() function does not need to be called after the CPU frequency change.

rtedbg_timer_stm32L4_tim2.h
Same as above, except the driver is for the STM32L4 family.

rtedbg_timer_stm32L0_tim2.h – shows how to implement a timestamp driver with a 16-bit timer.

The STM32L0 family only has 16-bit timers. The only exception is the 24-bit SYSTICK timer, which is usually not available for the timestamp timer because it is typically used for RTOS or system ticks. There are some limitations when using a 16-bit timer as a timestamp counter. The format ID size (defined with RTE_FMT_ID_BITS) has a fixed value of 15 to simplify the driver code. To avoid a reduction of the timestamp precision due to a right shift with RTE_TIMESTAMP_SHIFT = 1 (minimal value), the counter value is multiplied by 2 by a left shift. The example assumes a 16 MHz core clock for the STM32L053.

rtedbg_timer_test.h
This is a special version of the timer that does not actually count time, but only the number of times the data has been logged. It is intended only for testing the data logging functions after they have been ported to the new CPU or compiler. Since the message number is logged instead of the time, the logged timestamp does not depend on CPU timing and timer counter peripheral specifics. The output files generated by *RTEmsg* after decoding can be easily compared to verified file versions (e.g., version for the ARM Cortex-M).

Compile Time Parameter RTE_DELAYED_TSTAMP_READ

The compile time parameter RTE_DELAYED_TSTAMP_READ affects whether the message timestamp is read from the timer counter before space is reserved in the circular buffer or after it has already been reserved. The effect of this setting on timing is mostly negligible, since the difference between the time read before and after the memory reservation is typically only 10 to 30 CPU cycles. Since we are measuring time relatively between events, the difference is even smaller since the it is similar for all data logging functions. The impact on compiled code is shown below.

RTE_DELAYED_TSTAMP_READ = 1

The timestamp value is read from the timestamp counter late in the execution of the data logging function – just before the value is needed for data logging. This setup is recommended for firmware running on simpler CPU cores (i.e. Cortex-M0 … M4) since the code is typically faster, and the stack usage is slightly lower.

RTE_DELAYED_TSTAMP_READ = 0
The timestamp value is read from the timestamp counter at the beginning of data logging functions:

- The execution speed increases for complex CPU cores, especially if timestamp timer is connected to the CPU core via a slow peripheral bus.
- Stack usage may be slightly higher.

This setup is recommended for complex devices with multiple data buses, such as microcontrollers based on the Cortex-M7 core. The difference is not large when using a fast timer such as CYCCNT or SYSTICK that is directly coupled to the CPU core. The difference can be more than 10 cycles if a peripheral timer is connected to the CPU via a peripheral bus and the bus is running at a fraction of the CPU core speed. The smaller number of cycles is due to the fact that the value of the timer counter can be loaded in parallel with other instructions that prepare data for logging.

3.3 Resources Needed for Message Logging

The data shown below is an example of the code size for a typical project with an ARM Cortex-M7 core processor. The code size is almost the same for processors with Cortex-M4 core. The code sizes shown are for the code with the compiler optimization parameters set as for the example described in the Demo Project for the NUCLEO-H743ZI Development Board. The project was compiled using the GNU toolchain for ARM 11.3.Rel1. The same is true for the data shown in Execution Times, Circular Buffer, and Stack Usage.

The following resources are typically required for data logging (see Note 2 below):

1. Program memory requirements for the data logging functions of the *RTEdbg* library
   a) Compile time parameters and compiler settings as in the STM32H743 demo – functions optimized for size (-Os) except *RTE_MSG0() … RTE_MSG4()* which are optimized for speed (-O2), long timestamps enabled, RTE_MINIMIZED_CODE_SIZE = 0, RTE_FIRMWARE_MAY_SET_FILTER = 0
      516 bytes – for the selected macros and functions (see the Note 1 below)
      748 bytes – as above plus *RTE_MSG3()*, *RTE_MSG4(),* and *rte_long_timestamp()*
      396 bytes – as the top value without the *rte_init()* - *the g_rtedbg* structure initialized by a debug probe, and long timestamps disabled
   b) Compile time parameters as above plus RTE_MINIMIZED_CODE_SIZE = 1, all functions optimized for size, long timestamps disabled
      356 bytes – for the selected macros and functions (see Note 1 below)
      404 bytes – same as above plus *RTE_MSG3()* and *RTE_MSG4()*
      292 bytes – as the top value without the *rte_init()* - data structure initialized by a debug probe
2. **Static RAM usage** – example for RTE_BUFFER_SIZE = 1024 (logging of up to 1024 events)
   - 1024 × 4 = **4096** bytes for the circular data logging buffer, plus
   - **40** bytes for the data structure header (24) and trailer (16)
3. **Stack** usage is low, especially when the RTE_MINIMIZED_CODE_SIZE compile time parameter is set to 0. See the table on the following page for typical stack usage requirements.

Notes:

1. The above examples assume the use of the following functions and macros for a typical project: *rte_init()*, *RTE_MSG0()*, *RTE_MSG1()*, *RTE_MSG2()* and *RTE_MSGN()*, as typically not all functions are needed for simple projects. The code size also depends on the implementation of the timestamp timer driver. For this example, the timer driver *rtedbg_timer_cyccnt.h* (*ARM Cortex* CYCCNT) is used.
2. There are no format strings and no data tagging in the firmware, only format ID and pure binary data. However, the program memory is also occupied by calling logging functions and preparing the data. On a device with an ARM Cortex-M core, a function call is 4 bytes long and the size of the instruction to load data into register R0 is typically 4 bytes (this example is for the event logging with the macro *RTE_MSG0)*. Therefore, only 8 bytes of code is typically required to call a function that logs an event.
3. The code size is larger when single shot recording is enabled and when the circular buffer size is not a power of two. It is smaller when message filtering is disabled. Actual program memory requirements depend on the number of *RTEdbg* library functions used, CPU core, *rtedbg_config.h* settings, compiler options, etc. The RTE_DELAYED_TSTAMP_READ parameter has been set to a value of 1.
4. Extended message logging macros of type RTE_EXT_MSG call the same functions as macros of type RTE_MSG, therefore the memory consumption of the library functions is not higher when using a mix of these macros.

Execution Times, Circular Buffer, and Stack Usage

This table is provided for reference only, to give a quick insight into the execution speed and stack usage. The values shown are for the demo code in the STM32H743 demo project running on the NUCLEO-H743ZI board (Cortex-M7 core). The code optimizations for the whole project were set as follows:

- **size:** RTE_MINIMIZED_CODE_SIZE = 1 and -Os compile option for all functions listed below
- **speed:** RTE_MINIMIZED_CODE_SIZE = 0 and -Os for all functions with the exception of RTE_MSG0/RTE_EXT_MSG0_y … RTE_MSG4/RTE_EXT_MSG4_y (optimization -O2)

Data logging MACRO or function	Circular buffer space occupied (bytes)	Stack usage in bytes - code optimized for **speed** / **size**	CPU clock cycles		
			Code optimized for **speed**	Code optimized for **size**	Code optimized for **speed** Message filter = 0
RTE_MSG0 / RTE_EXT_MSG0_y	4	4 / 24	27 (35)	51	16 (14)
RTE_MSG1 / RTE_EXT_MSG1_y	8	8 / 40	33 (43)	63	18 (21)
RTE_MSG2 / RTE_EXT_MSG2_y	12	8 / 40	35 (48)	68	19 (22)
RTE_MSG3 / RTE_EXT_MSG3_y	16	12 / 48	41 (54)	73	23 (25)
RTE_MSG4 / RTE_EXT_MSG4_y	20	20 / 52	47 (63)	73	26 (31)
RTE_MSGN (size = 16B)	20	20 / 24	60 (89)	63	23 (28)
RTE_MSGN (size = 64B)	80		122 (206)	130	
RTE_MSGX (size = 15B)	20	24 / 24	130 (173)	130	23 (31)
RTE_MSGX (size = 63B)	80		369 (536)	369	
rte_long_timestamp	8	8 / 40	41 (57)	72	26 (35)

Additional information:

1. The timing values represent what can be achieved with zero memory latency. Actual execution speed depends on the CPU core, compiler type and version, optimization options, memory latency, data and instruction caches, program memory latency, bus width, cycles required to read the timer counter, etc.
2. Actual logging time also depends on the time required to prepare the data for logging. The times shown in the table include the function call and return and the time required to load the function parameters, which are constant values for timing measurement purposes. If the data for logging is read from memory or from a peripheral connected to the processor core with a slow bus, or if the data is additionally processed, the time for logging the data will be correspondingly longer. The code was in ITCM (Cortex-M7 Core Coupled RAM) for the measurement, message filtering was enabled, the buffer size was divisible by a power of two, and the CYCCNT cycle counter was used for the timestamp.
3. The values for CPUs with a Cortex-M4 core are shown in **green**. The compilation options are set as in the Demo Project for the NUCLEO-L433 Development Board. The code runs slower on devices with a Cortex-M0 or M0+ core. When the code is size-optimized for Cortex-M0+, it takes 49 cycles to log data/event for *RTE_MSG0()*, 80 cycles for *RTE_MSG2()*, and 91 cycles for *RTE_MSG4()* macros.
4. Stack usage depends on CPU core, compiler type and version, and optimization settings. Stack usage is the same for the Cortex-M4 and M7 cores. Data is shown for the next two versions and includes stack space for parameters where needed:
   - code optimized for speed – compiled with `#define RTE_MINIMIZED_CODE_SIZE 0`
   - code optimized for size – compiled with `#define RTE_MINIMIZED_CODE_SIZE 1`

See also section Optimize Data Logging Code Size and Speed for tips on how to get even faster.

3.4 Message Filtering

The message filter allows programmers to choose which message groups are recorded and which are not. Up to 32 message groups can be defined, and each group can be individually enabled or disabled. All messages logged with the same filter parameter belong to the same message group. The filter variable can be changed during code execution by the firmware or by the debug probe. The firmware can set the message filter during initialization with the *rte_init()* function parameter or at any time during code execution with the *rte_set_filter()* function. The firmware (or the programmer using a debug probe) can selectively disable message groups if the amount of logged messages is greater than the bandwidth of the transmission path to the host computer. The message filter variable bits are like the buttons on a digital oscilloscope that enable individual oscilloscope channels. The data logging buffer is shared by all enabled message groups (channels), just like memory on a digital oscilloscope. Separate message groups should be reserved for messages that generate a lot of data, such as large messages logged in interrupt routines.

Filter numbers are assigned automatically during the format definition file processing. Only filter number 0 *(F_SYSTEM)* is mandatory. It can also be used for other important messages such as system errors or fatal exceptions. *The rte_set_filter()* function can only set this bit to zero if logging is completely disabled by setting the filter value to 0. Such an operation is only enabled if RTE_FILTER_OFF_ENABLED is 1.

Note: The *F_SYSTEM* filter (bit 31 of the filter variable) should normally remain enabled unless the filter variable is set to zero to disable all message groups. Disabling *F_SYSTEM* would disable all system messages, including the long timestamp used for full timestamp synchronization. Keep this in mind when using a debugger to manipulate the filter value.

How filtering is implemented

Each bit of the *g_rtedbg.filter* variable enables a group of messages. Bit 31 of this variable is filter number zero, and bit 0 corresponds to filter number 31. The *rte_set_filter()* function also writes a copy of the filter variable to *g_rtedbg.filter_copy*. If logging is temporarily stopped by setting the *g_rtedbg.filter* variable to zero, the value of *g_rtedbg.filter_copy* can be copied back to resume logging as it was before logging was stopped.

The following compile time options in the *rtedbg_config.h* configuration file define the filtering behavior.

Compile time parameter	Description
RTE_MSG_FILTERING_ENABLED	0 – No filtering possible. 1 – Message filtering enabled. **Note:** The filter number parameter must be specified each time the logging function is called, even if filtering is disabled. In this case, the filter number F_SYSTEM or zero should be used as the filter parameter.
RTE_FIRMWARE_MAY_SET_FILTER	0 – The filter can only be changed on the fly by the debug probe – the initial value is set with *rte_init(initial_filter_value, init_mode)*. 1 – The firmware can change the message filter value. **Note:** The *rte_set_filter()* function is not available if this parameter is set to zero.
RTE_FILTER_OFF_ENABLED	0 – The firmware can completely disable message logging by calling *rte_set_filter(0)*. 1 – Message logging cannot be completely disabled. See the description Locking Data Logging After a Fatal Error.

Filtering can be completely disabled at compile time. Disabling message filters reduces code size and eliminates the possibility of inadvertently disabling data logging at runtime. On the other hand, message groups cannot be selectively disabled if there are too many messages in a given time period. It is also not possible to take snapshots of a running system.

How to Temporarily Pause and Resume Message Logging

The *RTEdbg* library allows data logging to be temporarily paused. Either the programmer using a debugger, the host data transfer application, or the embedded system firmware can set the filter value to zero to:

- temporarily stop logging before the buffer data (e.g., snapshot) is transferred to the host,
- stop single-shot logging, or
- stop logging after a fatal error is detected.

If the firmware needs to temporarily pause logging (e.g., to send the snapshot to the host), the current filter value can be retrieved with *rte_get_filter()* and set to the old value with *rte_set_filter()*. The *rte_restore_filter()* function can be used instead to restore the filter value to the previous non-zero value.

Locking Data Logging After a Fatal Error

Message filtering can be disabled by setting the filter to zero after a fatal or other critical error. Many real-time embedded systems must automatically restart after a fatal error. If data logging were to continue, the fatal error data in the circular buffer (and the history prior to the error) would be overwritten.

If RTE_FILTER_OFF_ENABLED is set to 1 at compile time, the message filter is not simply enabled by calling *rte_set_filter()* unless

a) *rte_set_filter()* is called with the parameter new_value = RTE_FORCE_ENABLE_ALL_FILTERS.

b) *rte_init()* is called with the parameter *init_mode* = RTE_RESTART_LOGGING (this also clears the logging buffer) or with the parameter *initial_filter_value* = RTE_FORCE_ENABLE_ALL_FILTERS.

Manipulating a Filter Variable During a Debugging Session

The filter value can be manipulated during logging to select which message groups to log. If some of the frequent messages are disabled, then the same buffer size will allow longer logging of all other messages. Modern debuggers allow on-the-fly modification of global variables while the firmware is running. Global variable values such as *g_rtedbg.filter* can be manipulated in the *Live Expressions* window. See the description of Stat_main.log – it includes a list of the top 10 message types that consume the most circular buffer space.

It is not necessary to stop the firmware while saving data. Data logging can be temporarily paused by setting the *g_rtedbg.filter* variable to zero. Message logging can then be re-enabled by setting the message filter back to its previous (or other) value.

3.5 Single Shot Data Logging

In 'post-mortem' logging mode the logging buffer is used as a circular buffer. In single shot mode, the same buffer is used in a linear fashion. It is filled from the beginning to the end and then the filter value is set to zero to prevent buffer overflow. Even if the firmware tries to re-enable data logging, it is immediately disabled again. The *rte_init()* function must be called to re-enable either single shot or 'post-mortem' mode.

If you need to use single shot logging in a particular project:

- Single shot mode must be enabled (RTE_SINGLE_SHOT_ENABLED set to 1).
- Message filtering must be enabled (RTE_MSG_FILTERING_ENABLED set to 1).
- Single shot mode must be enabled during *RTEdbg* initialization. See the *rte_init()* function description for more details on single shot logging. The data logging buffer can be cleared or simply overwritten - the mode is selected with the *rte_init()* function parameter.

Data logging is started when the filter variable is set to a non-zero value. This can be done with a debug probe or with firmware or immediately with the *rte_init()* function filter parameter.

See the example below:

```
if (some_variable > trigger_parameter)
{
   rte_set_filter(filter_parameter);
}
```

Messages in single shot mode are recorded in the circular buffer until it is full. Messages that do not fit into the remaining space are discarded, and messages that do fit are stored. If the programmer wants logging to stop at the first message that is too large, he must add this to the *rtedbg_config.h* file

```
#define RTE_STOP_SINGLE_SHOT_AT_FIRST_TOO_LARGE_MSG
```

In general, it is better to allow smaller messages to be logged after the larger one(s) could not be saved, because we get additional history. However, when analyzing log files, we must be aware that at the end of such a history in the log files, some data may be missing because one or more messages could not be saved due to their excessive size.

Notes:

1. The single shot logging method results in larger code and a slightly longer logging execution time (for the 'post-mortem' logging also). It should not be enabled for projects where it is not needed.
2. It is not possible to change the logging mode while data is being logged. *The rte_init()* function must be called to set or change the logging mode – see the rte_init() description. The entire logging data structure, including the circular buffer, is cleared when the logging mode is changed from single shot to 'post-mortem' or vice versa.

3.6 RTEdbg Library Structure

The distribution package (ZIP file) contains the ***Library*** folder with the following files and subfolders. The files marked with a gray background are mandatory and must be copied to the project folder.

Name	Description
rtedbg.c	Data logging library source code.
Inc\rtedbg.h	Definitions of data structures, functions, and macros for data collection. This file must be included in all files from which the data logging functions are called.
Inc\rtedbg_int.h	Internal *RTEdbg* library definitions.
Inc\rtedbg_config.h	Configuration file that defines the data logging functionality, including the size of the data logging buffer, etc. This file must be renamed to *rtedbg_config.h and* modified according to the embedded system CPU and application specific capabilities and requirements. The *rtedbg_config.h* in the demo projects is ready for use with GNU C, ARM compilers 5 and 6 (included in the Keil MDK IDE), and the IAR ARM compiler. Programmers should review it and remove any redundant settings that are not appropriate, for example, the toolchain used in their project.
Inc\rtedbg_inline.h	Inline logging function versions – see Optimize Data Logging Code Size and Speed.
Fmt\rte_main_fmt.h	Main formatting file containing format ID and filter name definitions.
Fmt\rte_system_fmt.h	Mandatory format definition file containing format definitions for the RTEdbg system messages.
Folder **Portable/Timer**	This folder contains files with timestamp timer initialization code and timer counter reading code. It is up to the programmer to decide which timer to use for the timestamp. Only one of the files from this folder should be copied/integrated into the project and modified (if necessary) according to the project and hardware requirements. See the more detailed description of the timestamp timers in the Timestamp Drivers. See also the Readme file.
Folder **Portable/CPU**	This folder contains files for buffer space reservation using either mutex instructions or other methods to ensure that the logging functions are reentrant. Only one of the files in this folder should be copied into the project. See additional descriptions in the Readme.md file (same folder as the driver files) and in the comments of the driver files.
	rtedbg_generic_atomic.h (uses the atomic operations library) Generic space reservation using mutex instructions. Use the *rtedbg_cortex_m_mutex.h* for ARM Cortex-M3/M4/M7… as it is more optimized than the generic version. **rtedbg_generic_atomic_smp.h** (uses the atomic operations library) Generic space reservation using mutex instructions – a version for multi-core devices.
	rtedbg_generic_irq_disable.h Buffer space reservation using interrupt disable/enable. Interrupts are typically disabled for a few CPU cycles. The driver can be used for all devices as it is the most universal solution. While it provides a robust and broadly compatible solution, it is the only possible driver for devices with cores that don't have hardware mutex (atomic) support.
	rtedbg_cortex_m_mutex.h Buffer space reservation using mutex instructions – for all cores that support it, such as ARM Cortex-M3/M4/M7/M33/M85. This driver version is smaller and faster as the

	generic one in the *rtedbg_generic_atomic.h*.
	rtedbg_generic_non_reentrant.h Buffer space reservation without re-entry protection – see Optimize Data Logging Code Size and Speed.

3.7 Linking Requirements

The data logging structure *g_rtedbg* must be located in a part of the RAM to which the entire instrumented firmware must have access. This is especially important when using an RTOS with memory protection support. If data logging is to continue after a system restart (e.g., watchdog reset), the *g_rtedbg* structure should be located in a non-initialized part of memory. See the online article: How to implement and use `.noinit` RAM. It is up to the programmer where to place the *g_rtedbg* structure. The address is generally only important if the data transfer to the host is done with a debug probe and the logging data structure needs to be at a fixed address to simplify the data transfer to the host. If the firmware sends the data to the host or writes it to non-volatile memory, the address is not important – e.g. if the logged data is transferred to the host via a serial channel using RTEcomLib functions.

GCC Toolchain

Below is an example of what could be added to the GCC linker file (example from a demo project).

The following definition is in the *rtedbg_config.h* file*:*

```
#define RTE_DBG_RAM __attribute__ ((section ("RTEDBG"))) __USED
```

The following linker specification has been added to the linker definition file:

```
/* RTEdbg data logging memory section */
. = ALIGN(4);
.RTE (NOLOAD) :
{
  *(RTEDBG)
  *(RTEDBG*)
} >RAM_D1
```

Note: This definition is placed at the beginning of a RAM section in the demo code so that the data logging structure is always at the same address and thus easily accessible to the host data transfer program.

The definitions should be modified according to the capabilities of your toolchain. If the __attribute__ keyword does not provide similar functionality, then either a compiler-specific definition can be used (e.g., the @ SECTION for the IAR EWARM) or a _Pragma keyword (available with C99-compatible and other compilers). See the link gcc.gnu.org/onlinedocs/cpp/Pragmas.html for a description of _Pragma.

The RTE_DBG_RAM definition can be empty. In this case, either the address of the *g_rtedbg* structure must be defined in the link specification or the address must be searched in the map file generated by the linker if it is to be used by the data transfer tool.

See the STM32CubeIDE User Guide section ***Place variables at specific addresses*** for a detailed description of how to place variables at a specific address. The description is valid for other GNU C based toolchains as well.

IAR EWARM Toolchain

The EWARM subfolder in the STM32 demo projects contains setup and other files necessary for compilation and testing with the IAR EWARM toolchain.

If you want the data logging structure to be at a fixed address so that it is easily available for data transfer using a debug probe, you must modify the link file as in the demo. See the description below for instructions on how to link variables to a specific address:

Placing a group of functions or variables in a specific section

Below is the linker definition file of the STM32H743 demo project. The changed and added lines are highlighted in **green**.

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__     = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__       = 0x081FFFFF;
define symbol __ICFEDIT_region_RAM_start__     = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__       = 0x2001FFFF;
define symbol __ICFEDIT_region_ITCMRAM_start__ = 0x00000000;
define symbol __ICFEDIT_region_ITCMRAM_end__   = 0x0000FFFF;
define symbol __ICFEDIT_region_RAM2_start__    = 0x24000000;
define symbol __ICFEDIT_region_RAM2_end__      = 0x2407FFFF;

/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x400;
define symbol __ICFEDIT_size_heap__   = 0x200;
/**** End of ICF editor section. ###ICF###*/

define memory mem with size = 4G;
define region ROM_region = mem:[from __ICFEDIT_region_ROM_start__ to __ICFEDIT_region_ROM_end__];
define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];
define region RAM2_region = mem:[from __ICFEDIT_region_RAM2_start__ to __ICFEDIT_region_RAM2_end__];
define     region     ITCMRAM_region     =     mem:[from     __ICFEDIT_region_ITCMRAM_start__     to
__ICFEDIT_region_ITCMRAM_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit };
do not initialize  { section RTEDBG };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region  { readonly };
place in RAM_region  { readwrite, block CSTACK, block HEAP };
place in RAM2_region { readwrite section RTEDBG };
```

The *rtedbg_config.h* configuration file contains an example of how to set the parameters for optimizing the functions that are in the *rtedbg.c* file for the IAR C compiler. To enable static parameter checking of format ID and filter number, compilation must be done with the c11 or gnu11 option enabled.

Keil MDK Toolchain

MDK-ARM subfolders in the STM32 demo projects contain setup and other files for compilation and testing with the Keil MDK toolchain. Compilation with ARM Compiler 6 is enabled by default.

When the project is compiled with the ARM Compiler 5 and the default values of the parameters in the *rtedbg_config.h* configuration file, the compiler returns the following error:

error: #119: cast to type "rte_any32_t" is not allowed

Casting to unions is not allowed on the ARM Compiler 5. For a more detailed description of this issue, see Workaround for Missing/Disabled Compiler Support for the rte_any32_t Union.
Also see the description if you have a C++ or mixed C/C++ project.

The following definition must be disabled (line deleted) in the *rtedbg_config.h* file:

```
#define RTE_USE_ANY_TYPE_UNION    /* Use casting to rte_any32_t for RTE_MSG macros */
```

This allows the project to be compiled, but indirectly disables the ability to log floating point values with the macros *RTE_MSG1()* to *EXT_RTE_MSG4()*. If a floating point value is used as parameter for the listed macros, the value is converted to an integer when casting from *float* to *uint32_t*. Thus, an integer value is logged in the circular buffer and could only be printed as an integer on the host side. The floating point values can still be logged with the *RTE_MSGN()*, because it uses a pointer to the floating point value or structure. It is also no problem if floating point values are part of a structure that is logged with *RTE_MSGN()*.

Linker definition file *Keil_STM32L433.sct*

```
; *************************************************************
; ***  Scatter-Loading Description File for STM32L433 DEMO  ***
; *************************************************************
LR_IROM1 0x08000000 0x00040000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00040000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }
  RW_IRAM2 0x10000000 0x00004000  {  ; RW data       ⇦ Added definition
   RTEDBG*(+RW)
  }
  RW_IRAM1 0x20000000 0x0000C000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

See Keil MDK – definition of a section and the following link Methods of placing functions and data at specific addresses for linking hints.

The *rtedbg_config.h* file in the demo code has been prepared to be used for compiling the project with either ARM compiler 5 or 6. If the ARM Compiler 5 is enabled, the linker may not place variables as *g_rtedbg* structure at the same addresses as for the Compiler 6. Check this and modify the linker definitions or adjust the address of the *g_rtedbg* data structure in the batch files located in the TEST folder.

The *rtedbg_config.h* configuration file contains an example of how to set the parameters for optimizing the functions in the *rtedbg.c* file for the v6 and v5 ARM compilers.

To enable static parameter checking of format ID and filter number, the compiler must be run with the c11 or gnu11 option enabled. This is only available with the ARM Compiler 6.

4 FORMAT DEFINITIONS

The format definition files define how the data from the logged binary messages data should be processed and printed to the Main.log file and custom output files. The data can be anything from event information to real-time control loop data to an exception dump. Format definitions specify how binary data is decoded (printed) in output files.

The following descriptions refer to the format definitions contained in the *rte_main_fmt.h* file (main format definition file) and the files included in that file. These files define format IDs, message types, message filters, output and input file names, format strings, etc. The format definition files are parsed by the *RTEmsg* application. It checks the syntax and prepares the header files that are included in the project source files instrumented with calls to data logging functions. The format ID and filter numbers are automatically assigned when the format definition file is processed.

The *rte_system_fmt.h* file contains format definitions for the system messages and must be included as the first include file in the *rte_main_fmt.h* file. See several examples of data logging and format definition files in this manual and in the Demo Projects.

The *RTEmsg* application parses the format definition header files and adds the *#define's* with filter numbers and format IDs. It must be run before the start of the compilation process (pre-build phase) to prepare the values for the C or C++ compiler. See Verifying Format Definitions with RTEmsg. The syntax of all format definitions is checked. This eliminates many errors that programmers can make. The format definition file is updated only when its contents are changed. This prevents recompilation if no changes have been made to the format definition files that affect format IDs and filter numbers.

Most editors and IDE's automatically reload the modified header files without notification. If the notification is annoying to the programmer, it can usually be disabled in the IDE or editor settings. See the Keil uVision example below:

> *To allow uVision to always reload externally modified files, check the Automatic reload of externally modified files checkbox, in Edit – Configuration, Editor tab, File & Project Handling.*

4.1 Format Definition File Syntax

The format definition files should only contain:

1. `/* One-line C-style comments */` <= Comment must end on same line
2. Blank lines.
3. `// Format directives and definitions – as C++ style comments`
   See the sections Directives in Format Definition Files and Format ID Names and Format Strings.
   A space or tab after the '//' is not required.
   The comment in the line containing a format definition does not have to start in column one. The format definition for a single message type can span multiple lines.
4. The directives inserted by the *RTEmsg*.
   It processes the format definitions and adds the `#define` directives with filter and format ID names and numbers. It also adds the preprocessor directives that prevent multiple inclusions.

   ```
   #ifndef __SOME_UNIQUE_PHRASE
   #define __SOME_UNIQUE_PHRASE
   #endif
   ```

4.2 Directives in Format Definition Files

INCLUDE() – Include a format definition file

// INCLUDE("file name")

For large projects, it is recommended to split the definitions into multiple files, e.g., one format definition file for each part of the firmware functionality. This eliminates the need to recompile the entire project if only a few format definitions are added or changed. See also the FMT_ALIGN() and FMT_START() keywords.

The *RTEmsg* application processes the INCLUDE() format definition files according to their extension:

Extension	How the include file is processed
.fmt	The format definition file is processed and written to the output file with a '.h' extension. For example, if a file *filename.fmt* is processed, the output file name is *format.fmt.h*. The *RTEmsg* application adds #define directives for the filter numbers and message ID definitions. The input file remains unchanged. The output file is overwritten only if the new content is different from the previous one. It is deleted if errors are detected during compilation. The output file is created in the same folder as the original.
Extension other than **.fmt** (e.g., **.h**)	The *RTEmsg* application adds #define directives to the format definition file for the filter numbers and format IDs. The file is modified each time the syntax check (compilation) is started and the new content is different from the old. It is not changed if errors are detected during the syntax check/compilation. The names and numbers of the filter and format IDs are in the same file as the format definitions. This makes it easier to navigate in the code – for example, it is possible to click on the format ID name and jump directly to the format definition in the header file. **Note:** Use the *RTEmsg* -back option if you want the program to automatically create a backup file (.bak) before changing the contents of the format definition header file or adding or changing #define with format ID numbers and filter numbers.

The paths to the include files must be specified either absolutely or relative to the folder containing the *rte_main_fmt.h* file. The `INCLUDE("rte_system_fmt.h")` must be the first keyword or format definition in *rte_main_fmt.h*.

Examples:

```
// INCLUDE("user_format.h")
// INCLUDE("..\rtos\format.fmt")
```

When processing a format definition file, the *RTEmsg* application first creates a temporary file with a '*.work'* extension. If the new version is the same as the current one, it deletes it. If it is modified, it replaces the current one with the '*.h'* extension if the format definition file has the '*.fmt'* extension, or the original format definition file if it has a different extension.

The header file where the #defines for the filter numbers and format IDs are written is only modified by the *RTEmsg* application after compilation if one of the filter numbers or format IDs is changed or added. This results in the compiler not having to compile the program module that contains the particular header file.

OUT_FILE() – Specify the output file into which the data will be printed (sorted)

// OUT_FILE(NAME, "file_name", "mode")
// OUT_FILE(NAME, "file_name", "mode", "Initial content – optional")

Programmers can use the output file definition to print (sort) the logged data to any number of output files. By default, the file is created in the output_folder defined with the *RTEmsg* command line argument.

Legend:

NAME – Specifies the file to which the message value is printed when used in the format definition
`Example: >NAME "Format string"`
file name – Filename as in the file system (absolute or relative path can be specified)
mode – Type of access to enable (as for the fopen function) – see the link and examples below:
"w" – Opens an empty file for writing. If the specified file exists, its contents will be destroyed.
"ab" – Opens a binary file for writing to the end of the file (appending to the existing contents).
The following modes are supported: w, a, b, +, t and x.
Initial contents – String containing the initial content to be written to the file when it is created.
The ***initial content*** can be used, for example, to define field names in a CSV file.

Writing to binary files – mode = "wb"
See the description of the "%W" type in the Format Syntax and Type Field Extensions section.

Appending to existing files – mode = "wa"
The format type extension "%D" writes the creation date of the binary logged data file to the output file (this is the time when the data was transferred from an embedded system). This option allows you to append data including the date/time to the output file.

Note: The output file is not created when the syntax check is started with the '**-c**' command line argument.

Examples:
```
// OUT_FILE(ERRORS, "errors.txt", "w")
// OUT_FILE(BATT_DATA, "battery.csv", "w", "Time[s];Voltage[V];Current[A];Temp.[°C];\n")
// OUT_FILE(BIN_DATA, "sensor-data.bin", "wb")
```

FILTER() – Define a filter name

// FILTER(F_FILTER_NAME)
// FILTER(F_FILTER_NAME, "Optional filter description")

Legend:

F_FILTER_NAME – Filter name as used in the C code.
Filter description – Filter description (optional parameter)

Filter names must begin with the prefix **F_**, followed by uppercase letters, numbers, and underscores.

Filter numbers and descriptions are written to the Filter_names.txt file in the output folder (to be used by the data transfer tools). If no filter description is given for a particular filter, the filter name is used instead.

Examples:
```
// FILTER(F_SYSTEM, "System and other important messages")
// FILTER(F_TASK1)
```

The F_SYSTEM filter is defined in the *rte_system_fmt.h* header file and is required for logging system messages such as long timestamps. It can also be used, for example, for important application-specific messages that should only be disabled when message logging is completely disabled.

IN_FILE() – Specify the input file for the indexed text (%Y format type)

// IN_FILE(NAME, "file_name")

This definition can only be used with the "%Y" format type. See Format Syntax and Type Field Extensions.

Legend:

- **NAME** – File name as used in the format definitions
- **file_name** – The input file must be located in the *Fmt* folder along with the format definition files. An absolute path must be specified if the file is not there.

The input file contains text messages to be printed instead of numbers. Can be used, for example, to report error messages instead of error numbers. The first text line has an index of zero. The file must contain at least two lines of text. If the index is greater than the number of text lines, the content of the last line is used. Some of the possible uses are: error and warning messages, exception vector names, sequencer states, etc.

Notes:

1. The length of text messages (line length) in the file must be between one and 255 bytes. Note that the number of bytes can be larger than the number of characters if a variable-width character encoding such as UTF-8 is used. For example, Latin characters are encoded with one or two bytes, while Asian characters use up to three bytes.
2. The input file is accessed when the binary file is decoded. It does not need to even exist when the definition syntax check is started with the '**-c**' command line argument.
3. Escape sequences are not available for indexed text.

Example:

```
// IN_FILE(ERR_NAMES, "Error_messages.txt")
```

MEMO() – Define a memory variable

// MEMO(M_NAME)	Define a memory named M_NAME
// MEMO(M_NAME, optional initial value)	Define memory and set initial value

Define the name of the memory used to temporarily store a numeric value.

A typical use of the MEMO command is to store a value from a message and later print that value, along with values from other messages, to a CSV file for analysis and graphing. In this way, it is possible to group data from multiple messages into a single CSV file for common display and analysis. An initial value for individual MEMOs allows us to place the value in the expected range for the variable and prevent the value from being 0 on the first recall. Some data graphing programs allow automatic scaling, and a single zero in the data could cause the program to take too large a range to display.

Example:

```
// MEMO(M_STATE_OF_CHARGE)          ← Define the name of the memory (default initial value is 0)
// MEMO(M_MEMO1, 100.2)             ← Define the memory name and set its initial value to 100.2
```

Notes:

1. The memory name must begin with the prefix **M_**. See the Store the Value in Memory <M_NAME> section for a description and additional examples.
2. A decimal point must be used for the initial value. The locale parameter, which specifies the use of decimal points or commas, is active only for printing values.

FMT_ALIGN() / FMT_START() – Reserve space in the format ID area

The *RTEmsg* application automatically assigns numbers to the format IDs. The ID value is incremented according to the message type. See the Short Message Type Comparison for how many consecutive ID values are assigned to each message type. When one or more format ID definitions are inserted or modified, all subsequent format IDs may be renumbered, and the result is a recompilation of all source files containing the modified format ID headers.

The following two directives allow you to reserve numeric space for format IDs:

FMT_ALIGN(param)

param – Align the format ID value to the specified value.
The parameter must be a power of 2 and its minimum value is 16.

Example: If *param* is 256, round the ID format to the first greater or equal value divisible by 256.

FMT_START(start_code)

start_code – Set the format ID for the following definitions to the specified value.
It is not allowed to set the format ID value to a value before the current format ID position to a value that has already been assigned or skipped.

The FMT_ALIGN directive can be used to skip some format ID values, so that the new header file always starts at the same value, unless many format IDs are inserted at once. The FMT_START directive specifies the numeric value at which subsequent format IDs should start.

Format ID Names and Format Strings

Each message type must have a format ID name that identifies the format definition.

The format definition has the following fields:

```
// FORMAT_ID_NAME
// >OUTPUT_FILE <INPUT_FILE "printf style format string"
```

FORMAT_ID_NAME defines of the format ID name. See the Format ID and Filter Naming Conventions section for a description of the naming conventions. The name must not be on a separate line, as shown above. The entire format definition can be on a single line.

The ***>OUTPUT_FILE*** and ***<INPUT_FILE*** fields are optional. They select the output file to which the message is printed or the input file from which the text messages are read.

>OUTPUT_FILE – The name of the file where the output should be written.
<INPUT_FILE – The input file name is required only if the format type "%Y" is specified in the format definition for this value.

The **"printf style format string"** specifies how the message will be printed (decoded). Format styles and extensions are defined in the Format Syntax and Type Field Extensions section. This string can be on the same line as the format ID name, or in multiple lines following the first line after the format ID name. Blank lines or comments are allowed between lines of format strings that belong to the same message.

Using output files:

1. If the ***>OUTPUT_FILE*** field is omitted, output is printed only to the Main.log file.
2. If ***>OUTPUT_FILE*** is used, output is printed only to the specified file.
3. If the field is specified as ***>>OUTPUT_FILE***, text is printed to the specified file and also to *Main.log*.

Notes:

1. Multiple lines of format strings can be added after the format ID name definition. This allows you to either use long and complex format definitions or to output value(s) from the same message to more than one file (e.g., to a CSV file in addition to the *Main.log* file).
   Limitation: A format definition for a single printed value must be on one line. This applies only to the part of the format definition that begins with the '%' character and ends with the format type character – as 'f' in the example: `"%[8u](-100*0.5).1f"`.
2. If a format definition is split into multiple lines and the output is to be redirected to a custom file, then **the redirection with >FILE_NAME or >>FILE_NAME must be before each string (in each line)**.
3. The format ID name, message number, and timestamp value are automatically written to the *Main.log* file and do not need to be specified in the format string for that file. If any of these are defined in the format definition, which specifies that the copy of the printed data should also be written to Main.log then the '%t', '%N', and '%M' type definitions are ignored when printing to the *Main.log* file. Also, the '\n' at the end of the string does not need to be added to format strings that go to this file, as it is automatically added.
4. Support for all common ESC sequences is implemented with the exception of Unicode ("\uhhhh" and "\Uhhhhhhh"). Hex values "\xhh" are always treated as bytes. If a hex number that is too large is used, its value is truncated to 8 bits.

IMPORTANT: The *RTEmsg* application writes errors only to Errors.log and Main.log, not to the custom (user-defined) files. If the specified value does not exist in the message logged by the embedded system, it will print zero if it prints a number, or empty text if it prints a string. This can happen, for example, if the embedded system logged less data than specified in the print format definition, or if there is an error in the definition. If the data in a custom file is suspicious, you should take a look at *Errors.log* and *Main.log*.

Compile Time Verification of Format ID Values

The data logging functions are robust and will not throw an error or exception if incorrect values are specified for the listed parameters. However, incorrect values will be logged (potentially misleading the testers) if an incorrect format ID is used or not logged at all if an incorrect filter number is used as a parameter. Since the functions of the *RTEdbg* library are added for testing and debugging, it is important to ensure that the correct parameters are used. It would be embarrassing to discover, while analyzing the hard-to-replicate problem, that the logged data does not contain the information we intended to record.

Format IDs and filter numbers are automatically assigned by the *RTEmsg* application prior to code compilation. It is recommended to allow the compiler to check the parameter values of the data logging macros during compilation because programmers can make mistakes and, for example, use the wrong format ID or filter number for a particular message. If incorrect values are used, some data may not be logged or decoded properly.

Format ID and filter parameters are checked at compile time when the

RTE_COMPILE_TIME_PARAMETER_CHECK

is defined in the *rtedbg_config.h*. This macro triggers a check with a static assert. Compilers conforming to the C11 specification have support for this directive. If for some reason a C99 compliant compilation is required, the parameters cannot be checked at compile time. The *RTEmsg* application also checks many things while parsing the format definition files, but it cannot catch all programming errors.

Format ID and Filter Naming Conventions

All format ID names must have a prefix as defined in the table below, followed by a number that defines the length of the message (in 32-bit words). Names can only contain the characters 'a-z', 'A-Z', '_', and '0-9'. The naming convention allows the message decoding application to verify that the message being decoded has the correct length. If the length is not known at compile time, the prefix MSGN_ or MSGX_ must be used instead (depending on the data logging macro used – prefix MSGN_ if logging with the RTE_MSGN macro).

Prefix	Description
MSG**nn**_	**nn** – message length in 32-bit words (**nn** = 0 … 4) The prefix 'MSG0_' must be used for messages logged with *RTE_MSG0()*, 'MSG1_' for *RTE_MSG1()* and so on.
EXT_MSG**n_y**_	**n** – message length in 32-bit words (**n** = 0 … 4, **y** = 1 … 8) Same as above, except that it indicates to the decoding application that additional data has been added to the message by using *RTE_EXT_MSG_()* macro type for data logging. Example: Use the EXT_MSG0_5_ prefix for a message logged with RTE_EXT_MSG0_5().
MSGN_	Must be used for messages logged with *RTE_MSGN()* in cases where the length of the message is unknown at compile time, such as null-terminated strings logged with the *RTE_STRING()* or *RTE_STRINGN()* macros.
MSGN**nn**_	**nn** – message length in 32-bit words Should be used for messages whose length is known at compile time. The length specifies the raw message data size in words (not including the timestamp/format ID word). If the structure size is not divisible by 4, the result must be rounded up to the nearest integer. Example: If the `sizeof(struct)` logged with the call to the *RTE_MSGN()* macro is 20 bytes (5 words), the prefix MSGN5 should be used. **Note:** When logging structures, defining the length of the message is also important for size control if the programmer changes the structure and forgets to change the format definition.
MSGX_	Must be used for all messages logged with the *RTE_MSGX()* macro. This macro is intended for logging messages whose length is unknown at compile time.

It is important to specify the message size if it is known at compile time. The *RTEmsg* message decoding application uses the size to properly assemble the messages on the host and to verify that the defined size matches the actual size found in the binary file. The size is very important in cases where a message may not be logged properly – e.g., messages may be overwritten if the data transfer bandwidth to the host system is too low.

Examples of data logging macros and format ID names (prefixes shown in bold):

```
RTE_MSG0(MSG0_SYSTEM_STARTED, F_SYSTEM)                                  // Data size  = 0 bits
RTE_EXT_MSG0_4(EXT_MSG0_4_SYSTEM_SHUTDOWN, F_SYSTEM, status)             // Data size  = 4 bits
RTE_EXT_MSG1_2(EXT_MSG1_2_DATA, F_SYSTEM, data1_32b, two_bit_data)       // Size  = 1 word + 2 bits
RTE_MSG2(MSG2_BATT_DATA, F_SYSTEM, battery.voltage, battery.current)     // Size  = 2 words
RTE_MSGN(MSGN_BUFFER_DUMP, F_FILTERX, &buffer, data_size)                // Size not specified
RTE_MSGN(MSGN3_BATTERY_DATA, F_SYSTEM, &battery, sizeof(battery)         // Size  = 3 words
RTE_MSGX(MSGX_SENSOR_DATA, F_SYSTEM, &sensor, sizeof(sensor)             // Size not specified
RTE_STRING(MSGN_STRING_DATA, F_FILTER_NO, p_string)    // Message length unknown at compile time
```

The format ID name should begin with the same combination of letters as the macro name ends – see the examples above (**in bold**). This helps programmers to determine if the format ID name matches the data logging macro name. The ***MSGx_*** prefix must be used for all messages logged with the ***RTE_MSGX()*** macro. Otherwise, the message content may not be decoded correctly.

4.3 Syntax of the Printf Style Format Definition String

The *RTEmsg* application uses the standard library *fprintf()* function for message printing (decoding). It is possible to print all types values using the standard printf string syntax. RTEmsg specific printf syntax extensions allow printing of multiple shorter data packed into 32-bit values, packed structures, bitfields, and others.

See the following link for a detailed explanation of the standard C format syntax:
docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions

The additional definition fields must be placed between the '%' character and before the formatting characters (**fwpt** – see the legend below) that define the printing behavior. All of these fields are optional. The order is not important, only the **[value]** definition must be listed first (if defined).

If the value size is not specified, a 32-bit value is used and it is interpreted as specified with the given format type – float32_t, uint32_t or int32_t (i.e. float if "%g" or another float type is used).

Additional fields in the format definition string provide the following functionality:

"**text** %[**value**](**scaling**){msg1|msg2|…}<**memo**>|**statistics**|**fwpt text**"

Note: No spaces are allowed between the **'%'** character and the **fwpt** field.
The only exception is the definition of indexed text using {text1|text2|…|textN}.

Legend:

- **text** – Plain text part of the message that is written to the output file without any additional processing (can be any text, either plain ASCII or UTF-8).
- [**value**] – Specifies either which part of the logged message to use when printing the value, or what other information (e.g., timestamp, message number) to use instead – see [Value…].
- (**scaling**) – Defines value scaling (offset and/or multiplier) – see (±Offset*Multiplier).
- **<memo>** – Name of the memory where the value is stored for later use – see <M_NAME>.
- |**statistics**| – Enables statistics for the current value and defines the value name – see |Value Name|.
- {**msg1**|**msg2**|**…**} – Defines text for the %Y format type – see {text1|text2|…|textN}.
- **fwpt** – Fields that define **f**lag characters, **w**idth and **p**recision specifications, and **t**ype conversion specifiers (+.1f in the example below). See Format Syntax and Type Field Extensions for special types.

International character set support: It is recommended to use the UTF-8 variable-width character encoding for the format definition files. This allows virtually any script to be used, as the UTF-8 encoding is transparent to the *fprintf()* function.

Example:

```
// MSG1_CPU_TEMP "CPU temperature = %[8u](-100*0.5)<M_CPU_TEMP>+.1f ºC\n"
```

`[8u](-100*0.5)<M_CPU_TEMP>` – format definition extension
`+.1f` – **f**lag character (+), **p**recision (.1) and **f**ormat type (f – float)

The sample definition allows the temperature in the range -50.0 to 77.5ºC to be printed from a logged 8-bit unsigned value (uint8_t). The value is stored in the 'M_CPU_TEMP' memory for later use.

Note: The text between the '%' and before the **fwpt** field ('`+.1f`' in the example above) is used to prepare the value to be printed. Only the text "`CPU temperature = %+.1f ºC\n`" is used to print the value with *fprintf()*.

Limitation: The standard library printf() function accepts an asterisk (*) for width and/or precision (example: "%*.*f"). This is not possible for the RTEdbg message printing definitions.

Specify the Value to Print [Value…]

This field specifies which part of the logged message to use when printing the value, or what other information (timestamp, message number, stored value, etc.) to use instead. If this field is omitted, 32-bit values are used in the same order as they were logged with a macro or in the same order as in the data structure. The data address is the address of the first bit of the decoded message from which the value will be used in the next output. The data address is incremented by the length of the data taken from the message data content buffer.

The data logged in a message can be output to more than one file. The OUT_FILE() section describes how to redirect the text to a file other than the default Main.log. The list of format strings for a single message can be arbitrarily long, and multiple consecutive format strings can be redirected to the same file. Each time the redirection goes to a different file than the one defined for the previous format string, the start bit address is set to 0.

[**nn**:**mmF**] or [**mmF**]	**Specify the start bit address, number of bits, and data type of the printed value.** This field defines which and how many bits of the logged message are used for the printed value. **If the size of the value (mm in bits) is omitted, then** • A 32-bit data length is assumed from the current bit address, and the current bit address must be divisible by 32. • The type of the value (integer, unsigned integer, float) is determined by the type field defined in the printf string (e.g., %d, %u, %f, %g). • The entire message content is used when one of the following string printf type specifiers is used %s, %W, %1H, %2H, %4H. **Legend:** • **mm** – Specify the number of bits used for the printed value • **nn** – define an absolute address of the first data bit (if omitted, it continues from the current bit address) **+nn** – Skip **nn** bits forward (e.g., skip data not to print) **–nn** – Skip **nn** bits backward (e.g., jump to the previously printed value) **Examples:** [32:16u] – Absolute addressing. It selects a 16-bit unsigned value starting from the 32nd bit for output. [+8:8u] – Relative addressing. Skips 8 bits and uses the next 8 bits as an unsigned value. [-32:32f] – Relative addressing – i.e. jump back 32 bits if the same value was first printed as hex and should also be printed as float. • **F** – data type: **u** – unsigned integer (from 1 to 64 bits) – **default if omitted** **i** – signed integer (from 2 to 64 bits) **f** – floating point – 64-bit (double), 32-bit (float), and 16-bit (half-float) values are supported **s** – null-terminated string or string of known maximum length (the length must be divisible by 8 and byte aligned) Example: [32s] = string of up to four characters The substring can be only a part of the complete message, which contains other information. A null character is appended to the end of the substring.

	Notes:
	1. The message length used for printing of the extended messages logged with *RTE_EXT_MSG0...4* is one word (32 bit) longer – e.g., a message logged with the *RTE_EXT_MSG2()* type macro is three 32 bit words long. The additional (extended) value is the last 32 bit word.
	2. The maximum length of a single value is 64 bits.
	3. The code has been tested for CPU cores that support little-endian byte order in memory. Most current microcontroller families use little-endian architecture by default. Some of them offer the option to configure endianness.
	4. Support for reversing bytes or bits of the printed value (e.g., converting values from big-endian to little-endian) will be added in a later release of the *RTEmsg*.
	Examples:
	`//"Error number: %[16u]u\n"`
	`//"Packed structure: %[8]03u, %[8]u, %[16i]d\n"`
	`//"A 32-bit float value: %g and a 32-bit unsigned value: %u\n"`

[N]	Use the current message serial number of the current message as the printed value.
[t]	Use the current message timestamp as the printed value (time in seconds).
[T]	Calculate the message time period and use it as the printed value. This is the difference in seconds between the current message timestamp and the previous timestamp for the same message. The value is 0 for the first occurrence of the message during binary file decoding and for the first occurrence after the message logged with the *RTE_RESTART_TIMING()* macro. **Note:** Accessing the timestamp value with [t] or [T] allows the value to be printed with a user-specified format string instead of the format defined for printing timestamps to the Main.log file with %t or %T.
[t-MSG_Name]	**[t-MSG_Name]** – Uses the value of the time difference between the message timestamp of the current message and the timestamp of the message whose name is given (in seconds). If the message with the MSG_Name has not yet been decoded (found in the binary file), the time difference is not known and a value of zero is used. There must be at least one occurrence of MSG_Name after the start of decoding or after the message logged with the *RTE_RESTART_TIMING()*. **Example:** `"%[t-MSG0_START_EXEC](*1000.).3f"` – prints the time difference in milliseconds between the timestamp of the current message and the timestamp of the last message with the format ID name MSG0_START_EXEC.
[M_name]	Use the value from the memory named ***M_name***. All values are stored as 64-bit double values. The retrieved value is prepared as floating point (double), unsigned (uint64_t), and signed integer (int64_t) to allow the use of format type conversion specifiers for floating point and fixed point numbers (not just floating point values). If a 64-bit integer value is stored in the memo with a value greater than the number of mantissa bits of a 64-bit float value, the recall memo will return a value with the least significant bits set to zero. There are no such problems with 32-bit values.

Value Scaling Definition (±Offset*Multiplier)

Any logged value can be scaled to a human-friendly form. Fixed-point notation is often used in embedded systems. Either a fixed number of bits is used for the fractional part, or the number is multiplied by a scaling factor – for example, voltage is scaled by 100 to get an integer variable with 10 mV resolution. Any value from the logged message (including timestamps and time differences) can be scaled with offset and multiplier. The offset value is added first, and then the result is multiplied according to the scaling definition.

Note: Fixed point notation can be used to pack data during message recording, e.g., when floats are used for internal firmware representation, to reduce the number of 32-bit words used in a circular buffer for a particular message type.

(±offset*multiplier)
or
(±offset)
or
(*multiplier)
	Scaling:
	value_used_for_printing = (value + offset) * multiplier;
	The value taken from message, memory, timing, etc. can be scaled to convert a value into a human-friendly representation. The result is returned as a floating point value and also as an unsigned and signed integer (if the programmer uses the '%d' or '%u' format type). Values are rounded when converted from float to integer.
	Scaling allows to print quasi floating-point values (e.g., an integer value is scaled by 0.001 → the integer value 1025 is printed as 1.025). Scaling and offsetting can be done during decoding on the host computer instead of within the embedded system. This speeds up logging and reduces circular buffer usage.
	Notes:
	1. A period '.' character must be used for floating point scaling values. The locale parameter, which defines the use of decimal points or decimal commas, is only active for printing values. See the -locale=xxx command line argument.
	2. If the offset value is omitted, a value of 0 is used. If the multiplier is omitted, a value of 1.0 is used instead.
	3. It is possible to use scaling for any value specified for printing between the square brackets [...]. This includes definitions such as [t] and [MEMO].
	4. **The value definition field […] is mandatory if scaling is defined for the printed value.** A definition like the first one is not allowed:
	`"Value = %(*0.1)f"` – wrong (missing size)
	`"Value = %[32f](*0.1)f"` – correct
	`"Value = %[8](-50*0.5)f"` – correct
	Format string examples:
	`// "Voltage: %[16u](*0.01).2f V, current%[16i](*.001).3f A\n"`
	`// >TEMPERATURES "%N %t: T1 = %[8](-50)d, T2 = %[8](-50)d\n"`

Store the Value in Memory <M_NAME>

This function allows you to store variable values for later use (recall). All values (integers or floats) are always stored as 64-bit floating point values. Memo names must be defined with the MEMO() directive. The name must be defined before use. The individual value is stored using the <M_MEMO_NAME>. See the example below.

<M_NAME>	Store the current decoded value in memory named M_NAME. This allows you to store data that can later be printed along with some other values (e.g., print data to a CSV file for graphing as shown below). The memo name must be specified before using the MEMO(M_NAME) directive in the format definition file before use. Any value processed during printing can be stored – either value or timestamp from the logged message or calculated values as a time difference [t-MSG_NAME]). **The value is saved exactly as it is prepared for printing** – for example, if scaling is used, the scaled value is saved.

Example: Combine fast- and slow-changing data in the same CSV file

Often, some system variables or peripheral inputs change infrequently. If all variables that should be written to a CSV file (e.g., for graphing) were to be logged each time other (more rapidly changing) variables were logged, the circular buffer would take up a lot of space, and the data transfer to the host would require a lot of bandwidth. The MEMO function can be used to store a slow value and retrieve it each time a message with fast changing values is decoded. The value is retrieved using the [M_MEMO_NAME].

See below for a practical example. The temperature is measured and logged once per second with the macro *RTE_MSG1()*. The voltage is logged with, for example, a period of 100 ms. Both values can be written together to a CSV file - to be imported into a spreadsheet for analysis.

```
/* Define memory named M_TEMPERATURE (initial value is set to zero by default). */
/* The memo name definition must precede the first use in a format definition. */
// MEMO(M_TEMPERATURE)
```

```
/* Format definition for temperature message (logged with 1 second period). */
/* The message is printed to Main.log and stored in the M_TEMPERATURE memory. */
// MSG1_TEMPERATURE
// "Temperature: %<M_TEMPERATURE>.1f"
```

```
/* Format definition for voltage message (logged with a period of 100 ms). */
/* The message is first printed to Main.log, then temperature and voltage with timestamp. */
/* to a CSV file. The OUT_FILE directive writes the first line of a CSV file. This is the header and contains */
/* the field names. */
// OUT_FILE(LOG, "log.csv", "w", "Time [s];Voltage [V];Temperature [°C]\n")
// MSG1_VOLTAGE
// "Voltage: %.2f"                          ← Voltage printed to the Main.log file
// >LOG "%t;%.2f;%[M_TEMPERATURE].1f\n"     ← Print time, voltage and temperature to Log.csv
```

Define Variable Name for Statistics |Value Name|

The RTEmsg application automatically collects the maximum and minimum values for the selected values. This can be enabled for any value that is printed. See the Statistical Features section for a detailed description of the statistical functions. Statistics must be enabled using the *RTEmsg* -stat command line argument.

|Value name|	**Enable statistics for a value** from the specified message.
	"Value_name" is the name of the current value written to the *Statistics.csv* file. The value can be any numeric value from the decoded message – including the [t], [T], and [t-MSG_Name] value definition for timestamps. If scaling is defined, the scaled value is used. The 'Value name' does not have to be unique. It is not a problem if the same name is used in format definitions for different messages because the name is always used together with the 'Format ID name'.
	Several minimum and maximum values are logged and an average value is calculated for this data, and all of them are written to the statistics file when the data decoding is finished. Each value is logged together with the number of the logged message. This allows us to easily find out in which message the logged value was in and what happened before and after that message, since all messages are written to the Main.log file.
	Example of **time value** statistics:
	`// MSG1_TASK1_CYCLE_TIME`
	`//  "Task1 cycle time: %[T]|Task1 period|(*1000)%g ms"`
	The value appears as "**Task1 period**" in the *Statistics.csv* file.
	The value used for statistics is multiplied by 1000, just like the printed value. The order of the definitions is not important. The (offset*multiplier) can be before or after the statistics definition.
	Example of logged **data value** statistics:
	`// MSG1_TEMP1`
	`// "Motor temperature %[16u](*0.01)|Motor temp. #1|.2f"`
	The text |**Motor temp. #1**| defines the name under which the value will appear in the statistics file.

Indexed Text Definition {text1|text2|…|textN}

The format type extension "%Y" allows to print selected text according to the decoded value used as index. Text descriptions can be printed instead of simple numbers, making log files easier to read. Since the same message value can be printed more than once, it can be printed both as a number and as text (see the first example below).

This definition allows the text table to be defined inline. Several separate text messages can be specified. The first text is used if the index is 0, the second one if the index is 1, and so on. If the index is greater than the number of texts, the last text is used. The '|' character (logical OR in C code) is used as a text delimiter.

Any ASCII or UTF-8 character can be used within the inline definition, with only two exceptions:

'}' – end of indexed text definition

'|' – text delimiter

Examples:

`"Operating mode: %[2]u:%[-2:2u]{Init|Run|Stop|Idle}Y"` – Prints text "Stop" if value is 2

`"Return value: %[16u]{False|True}Y"` – Prints "True" if the value is not zero and "False" if is is zero

`"Eco mode: %[24:1]{Off|On}Y"` – The text "Off" is printed if the value is zero and "On" if it is not zero. The variable does not have to be a single bit one, as shown in the example above.

For a complex example, see Format Definition for Printing Exception Data for a Cortex-M7 CPU. It also shows how to print the indexed text from the input file.

Notes:

1. At least two text fields must be defined, and the minimum length of each field is one character (can be a space).
2. The length of each text field (characters separated by '|') can be from one to 255 bytes. The length of international UTF-8 characters varies from one to four bytes. The actual number of characters is shorter in this case.
3. Escape sequences are not allowed when defining indexed text.
4. Longer definitions should preferably be in a separate file – see the description of the IN_FILE() – Specify the input file for the indexed text (%Y format type).

4.4 Format Syntax and Type Field Extensions

The following table shows the format string syntax and extension. The data decoding/formatting application was developed and compiled using the Microsoft Visual Studio C Compiler. See the following document for a list of all type, field size, precision, and other specifiers:

docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions

The *RTEmsg* application supports the following type conversion specifiers (subset of the Microsoft printf):

`%d, %i`	– Signed integers up to 64 bits in length
`%c, %o, %u, %x, %X`	– Unsigned integers up to 64 bits in length
`%e, %E, %f, %F, %g, %G, %a, %A`	– floating point values
`%s`	– strings

and the following characters that define [flags][width][.precision][size]:

`'-', '+', '0', '#', ' '`	- flag characters
`'0' .. '9'`, and `'.'`	- width and precision specification
`'l' and 'h'`	- argument size specification (`l` – 32-bit, `ll` – 64-bit value, etc.)

Microsoft printf format syntax extensions such as 'I32', 'I64', 'z,' 'w' etc. are not supported.

Notes:

1. The exact syntax of the printf style format definition is not checked (e.g., if someone were to define %20.2.1f). It is up to the programmer to ensure that the syntax is correct according to the printf syntax. If errors are detected while decoding binary file, the erroneous format definitions can be changed and the decoding restarted without having to recompile the firmware and restart the test.
2. All integer values from the logged message are internally treated as 64-bit signed or unsigned integers during printing. Their size in the logged message can range from 1 bit for unsigned, 2 for signed, and up to 64 bits.
3. When the value is scaled (see Value Scaling Definition (±Offset*Multiplier)), the result is prepared as a 64-bit signed integer, a 64-bit unsigned integer, and a 64-bit float (double). Thus, the value can be printed as an integer or as a float. If the result of the scaling is a negative value, the unsigned integer value is zero.

Additional Type Field Characters

The following format extensions are valid for the *RTEmsg* application. They allow values to be printed in ways not supported by the standard C printf syntax. The *[flags][width][.precision][size]* specifiers are not allowed for the additional type field characters defined below.

%N	**Print current message serial number.** The number is printed as 'Nxxxxx', where xxxxx is the serial number of the decoded message. The serial number of the message is printed in the same format as in *Main.log*. It is recommended to include message numbers or timestamp values when decoding data into custom files. This allows programmers to find the data found in one file (containing a selection of data) in other files to check what happened before (or after) an event. The character 'N' is inserted at the beginning of the message number to distinguish it from other numbers in the same output file. The format is the same as printing the message serial number in *Main.log*. A hash '#' symbol is printed before the message number in the *Main.log* file if the timestamp value for that message is suspicious and not when printing to a custom file. See Message Numbers.
%t	**Print the current message timestamp** data using the same number format as the Main.log file. "%t" is a simpler but less flexible version of "%[t](xxx)F", where xxx is the timestamp scaling and F is the number formatting specification (e.g., "%[t](1000.)8.3f").
	Note: Message number, timestamp, and format ID name are automatically written to *Main.log* for each message. If the '%t' is in a format definition that should go to a defined file and *Main.log*, then the '%t' is ignored when writing to *Main.log*. The same is true for '%N' and '%M'.
%T	**Print the message repetition period.** This is the difference between the current message timestamp data and the timestamp of the previous occurrence of the same message. A zero is printed if there is no previous occurrence of the same message type after starting decoding or after decoding a message logged with the RTE_RESTART_TIMING() macro.
%W	**Write binary data from current message to the output file** This function can be used to collect binary data from the embedded system. The complete message content is written if no data size is specified with the size definition – see the description of [nn:mmF] / [mmF] in section Specify the Value to Print [Value…]. If the value size is defined, it must be less than or equal to 64 bits and divisible by 8. The value address must be divisible by 8 also. **Note:** If the length of the logged data is not known at compile time, the MSGN or MSGX prefix must be used for the format ID name. In this case, the complete message content is written to the output file. The *RTE_MSGN()* enables logging of data with variable length (length in bytes divisible by 4) and the *RTE_MSGX()* enables logging of variable-length messages (length in bytes not divisible by 4).
%B	**Print an integer as a binary number.** The value size must be defined (from 1 to 64). It defines how many digits to print. Leading zeros will not be skipped. The size is 32 if no size is specified. The number of digits printed is equal to the number of bits defined for the value (or 32 by default). **Example:** "%[16u]B" - prints a 16-bit binary value, "%B" - prints a 32-bit value
%D	**Insert date and time** as: YYYY-MM-DD HH-MM-SS Example: `2023-06-23 14:30:55` This is the date the binary file was created after data was transferred from the embedded system.
%M	**Print the Format ID name of the current message.** The Format ID name is automatically printed in Main.log for each message, along with the message number and timestamp. It can be printed to other output files with the %M type.
%Y	**Print an indexed text from the specified input file or from an inline list of strings.**

	This option can be used, for example, to print error messages instead of error numbers to make the log files more readable. A list of text messages must be specified with an inline definition (see Indexed Text Definition {text1|text2|…|textN} or input file (see IN_FILE() – Specify the input file for the indexed text (%Y format type)).
	The value prepared for printing (from message, memory, etc.) is used as the index of the text list. If the index is 0, the first member of the list is printed. If the index is 1, the second text is used, and so on. If the index is greater than the number of text items or lines in the file, the last text is used.
	Format string example for inline text definition:
	`MSG1_SAFETY_MONITOR "Safety monitor: %[24:1u]{Disabled|Enabled}Y"`
	A single bit (bit 24) is printed as *Disabled* if it's value is 0 and *Enabled* if it's value is 1.
	Example of text strings read from a file:
	`IN_FILE(ERROR_MESSAGES, "Errors.txt")`
	`MSG1_ERROR_MSG <ERROR_MESSAGES "Error #%[16:8u]u: %[16:8u]Y"`
	In this example, an 8-bit information (error number) from a logged message is printed first as an unsigned number and then as a text message. If such an error message file has 20 lines with 20 error messages, then the message is printed from line 1 if the value is 0 and from line 20 if the value (error code) is 19. If the value is 20 or more, the message from the last line (20th) is printed. For example, if the error message text is to be printed for the first 19 error messages and the last line (20th) contains the message "internal error code", this text will be printed for error numbers 19 and above.

%nH	**Print the message data as hex numbers.** **n** = 1 – message printed as bytes, **n** = 2 – as 16-bit words and **n** = 4 – as 32-bit words This format definition is intended for printing message contents when the message length is unknown at compile time (formatting cannot be defined for individual values), or when the length is known and the entire message is to be printed in hex only. **Notes:** 1. The value size (number of bits/bytes to be printed) cannot be specified in the [Value...] field. The entire contents of the decoded message will be printed, or the remainder of the message data if some of the values have already been printed (from the current address on). This makes it possible to print the entire contents of a message in hexadecimal format, or just the rest of the message if only the first few words of the message are known - for example, for a stack dump. 2. If there is no data in the message or the rest of the message that it should print, it prints nothing. 3. If the data does not fit into one line (is longer than 16 bytes), it is split into several lines and an index is added at the beginning of each line. Examples: `// MSGN_DUMP_BUFFER1 "%1H"` // Print the entire message content as hex bytes `// MSGN_DUMP2 "%d %4H"` // Print the first value as a 32-bit integer and the remainder as hex words

5 RTEdbg LIBRARY – PROGRAMMER TIPS

5.1 General Tips

This section outlines key considerations for programmers to avoid potential issues when instrumenting code and analyzing results.

Message Numbers

The message number is the sequential number of the message. All messages are counted during message decoding, not just the valid ones that were decoded correctly. Invalid data can be the result of problems such as buffer overrun. By default, message numbers are printed starting with the letter 'N' and five digits. The *RTEmsg* application command line parameter '**-nr**' defines the format of the message number printing.

A number sign '#' before the letter N, representing the message number (for example, #N01234), indicates that the timestamp for that message is suspicious. The number sign is not printed to a custom file if the '%N' format extension is used. The time difference between successive messages is usually relatively small. A timestamp is suspicious if the time difference between successive messages is greater than a quarter of the timestamp period. A large difference can have several causes, such as insufficient bandwidth for data transfer from the embedded system to the host, message logging paused for some time, watchdog reset, resume from sleep, etc.

Open Additional Data Channel

Some debugging solutions use channels (e.g., Segger RTT) to send information to the host. A new channel can be created here by simply defining an additional format ID and a corresponding format string that redirects print output to an additional file. All data received with this format ID can be redirected to the custom file – e.g., a complete communication between CPU and GSM modem can be written to a separate file.

Example: The format definition is shown in green, followed by the call to string data logging.

```
// OUT_FILE(CHANNEL1, "channel1.txt", "w")
// >CHANNEL1 MSGN_CHANNEL1 "%s"
```

```
RTE_STRING(MSGN_CHANNEL1, F_FILTER_CH1, String_buffer)
```

Sorting Data

Real-time and non-real-time systems typically produce a lot of data. It is recommended that the data be printed (decoded) so that all events go into the Main.log file, and special events and real-time control data are also printed (sorted) into separate files. This allows testers to quickly review small amounts of sorted data (i.e. special events, error messages, out-of-bounds data, etc.). If the data in the custom files has either timestamps or message numbers of messages containing special events, testers can search for data in other files and in *Main.log*. This makes it possible to quickly find more detailed data and check what happened before (or after) a special event. Data formatting is left up to the programmers (except for the *Main.log* file), so they'll have to make sure that the message number and/or its timestamp are also written to the separate files.

Special events with data or timing extremes can also be detected using the statistics functionality of the *RTEmsg* application. It collects the maximum and minimum times between the selected events and the maximum and minimum values of the logged data – see the statistics definition |Value Name|.

Data in separate files should be tagged with either a timestamp (printed with "%t") or a message number ("%N") or both. This allows the tester to locate information for other logged data and events (before or after the partic-

ular event or data) in other files from the same test session. The anomaly usually has a cause that may be several ms or even seconds back. It is also interesting to check what happened after the event.

Compile Time Verification if a Structure Fits Into a Message

The entire data structure can be logged with a single call. The size of the largest structure must be less than or equal to the maximum message size. The compile time parameter RTE_MAX_SUBPACKETS defines how many of the 16 byte blocks can be logged with one function call.
The maximum message size in bytes is RTE_MAX_MSG_SIZE = RTE_MAX_SUBPACKETS × 16. This is the maximum structure size that can be logged with the *RTE_MSGN()* macro. Messages that are too long are either discarded or shortened to the maximum length depending on the value of the compile time parameter RTE_DISCARD_TOO_LONG_MESSAGES.

The following example shows how to check if the message size does not exceed what can be logged with a call to *RTE_MSGN()*. This is a compile time check that will return a compile error if the size of the structure or data field is too large to be logged with a single function call.

```
static_assert(sizeof(data_struct) <= (RTE_MAX_MSG_SIZE), "Structure size too large");
    // Example how to do compile time check if the structure is not larger
    // as it could be sent with one call to message logging function
static_assert(sizeof(data_struct) <= (RTE_MAX_MSG_SIZE));
    // Shorter version of assert introduced with C17 can be used (message is omitted)
RTE_MSGN(MSG2_DATA, F_FILTER_NO, &data_struct, sizeof(data_struct))
    // Save the contents of 'data_struct' to the circular message buffer
```

Minimum Circular Buffer Size

During streaming (continuous) data logging, the software running on a host computer takes some time to transfer the logged data. Data transfer tools run on the Windows operating system, which is not a real-time operating system. It may deny CPU time to the process transferring data for a few tens of milliseconds or more. If the circular buffer size is too small, some data may be lost, especially with fast data bursts. The recommended size (RTE_BUFFER_SIZE) is at least a few kB, but for slower processes, small, infrequent messages, and microcontrollers with small RAM, 1 kB may be sufficient to store a useful amount of history.

The use of circular buffer space can be reduced if

- bitfields are used (see Bitfield Logging) or packed structures,
- smaller values are packed together – see Pack Multiple Short Variables Into a 32-bit Word or Structure,
- multiple values are logged with one call – e.g., using one *RTE_MSG4()* instead of two *RTE_MSG2()*,
- Short data are logged with the Extended Messages RTE_EXT_MSG0_y … RTE_EXT_MSG4_y.

Data packing reduces the amount of RAM used for logged data, but may increase the amount of program memory used and/or slow down data logging also.

RTEdbg and Memory Protection Unit (MPU)

All instrumented code (privileged and non-privileged) must have write access to the portion of RAM where the *g_rtedbg* structure resides. If an *MPU-protected* version of the RTOS is used, no task (even a privileged one) has access to system peripherals such as the ARM Cortex-M CPU cycle counter (CYCCNT) or the system clock timer (SYSTICK), even if the memory area containing the cycle counter is enabled for that task. In this case a different timer peripheral must be used and access to the peripheral timer memory area must be enabled by the MPU for all tasks in which the data is logged using the functions and macros of the *RTEdbg* library. Check the RTOS documentation for the availability of functions that provide information about the current cycle counter and CPU operating frequency – such as *sys_clock_hw_cycles_per_sec()* and *k_cycle_get_32()* for the Zephyr RTOS. Use them in your timer driver instead of the direct access to the timer counter and frequency value – macro *RTE_GET_TSTAMP_FREQUENCY()* in the *rtedbg_config.h* file. In general, a task with normal privileges cannot disable interrupts. Some other mechanism must be used to enter/exit a critical section if necessary.

Interrupt During Message Logging Followed by Fatal Exception

This is just one example of what can happen because the logging functions are reentrant and interrupts are not disabled during logging. If the space in the buffer has already been reserved, but the message has not been completely written, the following may happen The space for the interrupted message may just be reserved, but not written, or only partially written. If the fatal exception is thrown within the interrupt function that interrupted message logging, then the previous message will not be completely written. The possibility of such a scenario is not very high because the logging functions are fast, but it cannot be neglected. Therefore, if the last message is a message from an exception handler that stops the execution of the application, then the message(s) just before this logged fatal error may not be correct. See the description of exception data logging in the ARM Cortex-M4 / M7 Exception Handler Example.

RTOS Task Starvation

In embedded systems that use multitasking, low-priority tasks may not get enough CPU time for extended periods of time. This is called task starvation. Let's look at an example where a low-priority RTOS task has called a data logging function and is in the process of writing data to the circular buffer after the buffer space has already been reserved. In such a case, the following could happen. If this task does not get CPU time for a long time because higher priority task(s) do not release the CPU, this task will not finish writing to the data logging buffer until it gets CPU again. This can cause problems when using streaming data transfer. Data from the circular buffer may be read (by a debug probe, for example) before the lower priority task has finished writing data to the buffer. Data logging in higher-priority code continues, and the buffer index may have already come around, and the area of the data logging block reserved by that task may have already been overwritten with new data. When the interrupted task finally gets CPU time, it will continue writing to the previously reserved portion of the buffer. This would overwrite newer data and produce unexpected logged data.

If such a scenario is likely, workarounds such as the two examples below can be used.

Solution 1: A larger block of memory is reserved for logging if enough RAM is available. This minimizes the possibility of such a scenario.

Solution 2: Disable task switching before calling the data logging function within a low priority task that can be put aside for a longer period of time. See below for an example for FreeRTOS.

```
taskENTER_CRITICAL();
RTE_MSGN(MSGN_STRUCT, F_FILTER_NO, &Structure, sizeof(Structure))
taskEXIT_CRITICAL();
```

Use this workaround for low-priority tasks.

See FreeRTOS taskEXIT_CRITICAL and Spinlock API for the Zephyr RTOS.

Using the RTEdbg Library on Multi-Core Processors

Read the description below if you are using data logging with the RTEdbg library on a multi-core processor and all cores write to a common circular data buffer located in shared RAM.

When a program runs on a single-CPU machine, the hardware does the necessary accounting to ensure that the program runs as if all memory operations were performed in the order specified by the programmer (program order), so memory barriers are not necessary. However, when memory is shared with multiple devices, such as other CPUs in a multiprocessor system, additional considerations are required to ensure data coherency. This does not apply to cases where, for example, the DMA unit is involved in the process of sending logged data to the host or writing it to nonvolatile memory.

Use the *rtedbg_generic_atomic_smp.h* driver to reserve space in the data logging buffer as it is the most universal. Adapt it to the memory sharing features of the microcontroller family and compiler used, if necessary.

See also the document Application Note 321 ARM Cortex-M Programming Guide to Memory Barrier Instructions if your are using a device with an ARM Cortex-M core and wish to optimize the CPU driver. **Summary:** The DMB instruction ensures that any explicit data memory transfers prior to the DMB are completed before any subsequent explicit data memory transactions after the DMB starts. This ensures the correct order between two memory accesses. The use of DMB is rarely necessary in Cortex-M processors because they do not reorder memory transactions. However, it is required if the software is to be reused on other ARM processors, especially in multi-master systems. Refer to the appropriate programming and hardware manuals if a different core type is used in your project.

Note: There is no simple solution for multi-core systems with completely different core types, such as the combination of ARM Cortex-M0 and M4. Use a separate data logging structure for each of them. There may be a problem even if both CPU cores support atomic (mutex) instructions and data memory barriers, e.g. a dual-core device with Cortex-M4 and M7. Check the documentation for the microcontroller family you are using.

Accessing the RTEdbg data structure with a debug probe: The debug probe should perform cache maintenance when accessing the device memory. If you suspect that a problem accessing data in circular memory is caused by the debug probe, implement the memory barrier macro or use the MPU as described below.

Using the Memory Protection Unit (MPU): The other solution to the data cache coherency problem is to use the MPU. Configure the cache policy of the memory region where the *g_rtedbg* structure resides from write-back to write-through, or alternatively to shareable (non-cacheable). The address of the *g_rtedbg* structure should preferably be aligned to the cache line size (32 byte alignment in the case of a Cortex-M7 core).

How to Avoid Problems When Adding or Changing Format Definitions

After a firmware release, the format definition header files must also be archived. This allows data collected by the production firmware version to be decoded if problems are discovered. The formatting files are required to properly decode raw data from the data logging structure. Therefore, the files must be archived with the project code (object files generated during project linking). If an error is found in the format definition, it can be corrected later, but the enumeration of format IDs must not be changed.

When data logging is used for a product that evolves over time, and firmware updates don't reach all systems at the same time, it is inconvenient to have to use a different set of format definition files to decode data collected from systems using different firmware versions. It is possible to make changes to the format definitions without changing the format IDs for existing message types and use the same set of format definition files for all firmware versions.

Follow the rules presented below to avoid the potential problems:

- Keep all project-specific format definitions in one file, and add each new format definition at the end of the file. Add instructions for adding and modifying format definitions at the beginning of this file.
- If an existing format definition needs to be updated, modify it only
  - if the change does not affect the number of format ID numbers required for that ID, as this could change all subsequent format ID numbers (see the Short Message Type Comparison), and
  - if the encoding of the logged data has not changed (the same binary value has the same meaning and scaling).
- In all other cases, do not change the current format definition. Instead, add a new format definition with new name to the end of the format definition file (keep the old format definition for data collected from systems with older firmware versions).
- Make sure there is enough format ID space (number of possible format IDs) reserved. Check the percentage of used format IDs in the Stat_main.log file before the first official firmware release and

increase the number of bits used for format ID representation (RTE_FMT_ID_BITS) if necessary to have enough free format IDs for future firmware releases.

Pack Multiple Short Variables Into a 32-bit Word or Structure

When logging, it is important how much space the message occupies in the circular buffer, as this affects the length of the history of events before, for example, a fatal error that caused the operation to stop. The following two examples show how this space can be used effectively.

The *RTEdbg* library macros support logging of 32-bit values only. Shorter variables (char, uint8_t, int16_t, etc.) can also be used as parameters for the data logging functions. However, this would unnecessarily waste space in the circular buffer since a 32-bit value would be logged for a short variable. Values can be easily packed into 32-bit words using shift and logical OR operations.

The first example below shows how to pack three variables into one word and decode (print) them on the host.

```
// Log three variables as one 32-bit data word
   int16_t var1;
   uint8_t var2;
   uint8_t var3;

   RTE_MSG1(MSG1_PACKED_VARIABLES, F_FILTER_NO, var1 | (var2 << 16u) | (var3 << 24u))
```

Format definition for the example above:

```
// MSG1_PACKED_VARIABLES "var1 = %[16i]d, var2 = %[8]u, var3 = %[8]u\n"
```

The programmer must make sure that the values fit into the number of bits in which they are packed. Otherwise, some of the upper bits or even the sign may be lost. The problem of cutting off upper bits when packing into a smaller format can be easily solved by the programmer by limiting the values before packing to the maximum size possible in the packed format.

The second example shows how to pack complete battery information (seven values) into a small structure and log it into a circular buffer with just one function call. The total length of the logged structure is 12 bytes.

```
   struct _battery_data
   {
       uint16_t voltage_x100;      // Battery voltage [V x 100]
       int16_t  current_x100;      // Battery current [A x 100]
       uint16_t charged_ah_x10;    // Charged capacity [Ah x 10]
       uint16_t discharged_ah_x10; // Discharged capacity [Ah x 10]
       uint8_t  temperature1_x2;   // Sensor #1 [ºC x 2], offset 100 (0 = -50ºC)
       uint8_t  temperature2_x2;   // Sensor #2 [ºC x 2], offset 100 (0 = -50ºC)
       uint16_t soc_x10;           // SOC [% x 10]
   } battery;


// Data formatting specification example
// MSGN3_BATT_DEMO "%4H"
// "\n  Battery voltage: %[16u](*.01).2f V, Current: %[16i](*.01).2f A, "
// "Charged: %[16](*.1).1f [Ah], Discharged: %[16](*0.1).1f [Ah], "
// "Temperatures: %[8u](-100*.5).1f [°C], %[8u](-100*.5).1f [°C], SOC: %[16](*.1).1f %%"
// "\n  Time used for calculation: %[t-MSG0_START_SINCOS](*1e6).3f us"
// >BATTERY_DATA "%[16u](*.01).2f;%[16i](*.01).2f;%[16](*.1).1f;%[16](*0.1).1f;"
// >BATTERY_DATA "%[8u](-100*.5).1f;%[8u](-100*.5).1f;%[16](*.1).1f\n"


// Data logging with a macro RTE_MSGN()
RTE_MSGN(MSGN3_BATT_DEMO, F_BATTERY_DATA, &battery, sizeof(battery))
```

The format ID *MSGN3_BATT_DATA* above starts with *MSGN3_* for this example. It indicates that the message size is three 32-bit words and that the message will be logged using the *RTE_MSGN()*. The *RTEmsg* application needs this information to check the syntax during the pre-build phase.

Bitfield Logging

Message logging is based on 32-bit words because this is a minimal amount of data in the circular buffer. When logging many values of small size measured in bits, logging is faster and uses less buffer space if the values are packed into bitfields. See the bitfield example below.

```
typedef union
{
    struct
    {
        unsigned int data3:3;
        unsigned int data5:5;
        int          data24:24;
    };
    uint32_t u32;
} demo_struct1_t;

   demo_struct1_t test_struct;
   test_struct.data3 = some_data1;
   test_struct.data5 = some_data2;
   test_struct.data24 = some_data3;
   RTE_MSG1(MSG1_STRUCT1, F_DEMO_MSG1, test_struct.u32)

/* Format definition for this example – in the format header file */
// MSG1_STRUCT1  "Bitfield: %[3u]d, %[5u]d, %[24i]%d\n"
```

Programmers must understand how the compiler packs the data (bitfields) into memory. The packing algorithm can be influenced by compile-time parameters, options, pragmas, etc. See the example of pragma options for the IAR C compiler.

```
#pragma bitfields={disjoint_types|joined_types|reversed_disjoint_types|reversed|default}
```

Bitfields may not be portable to another compiler or CPU core type (especially if signed values are used). However, for most compilers, a bitfield allocation strategy can be selected that overcomes most problems of this type. Consult your compiler documentation. Below are links to popular ARM Cortex-M compiler descriptions related to bitfields.

- gcc.gnu.org/onlinedocs/gccint/Storage-Layout.html
- developer.arm.com/.../C-and-C---Implementation-Details/Structures--unions--enumerations--and-bitfields
- www.keil.com/support/man/docs/armcc/armcc_chr1360775115791.htm

Note: Any bitfield data formatting should be tested during debugging before actual use to verify that the compiler is packing the data as the programmer intended (or assumed), and that your format definition matches the way the data is being packed.

How to Completely Disable Data Logging Functionality at Compile Time

Data logging can be disabled by setting the RTE_ENABLED parameter in the *rtedbg_config.h* configuration file to 0. This replaces all data logging functions with empty macros, completely removing data logging calls.

It is recommended to keep the message logging in the production code for several reasons:

1. Logged data is available for testing and diagnostic purposes if problems are discovered later.
2. The benefits of real-time data are great compared to the small cost of additional flash and RAM. Even a small data logging buffer is better than no buffer at all. Message filters can be used to select what data to log during a particular test or troubleshooting.
3. Data logging was an integrated and tested part of the system, and removing it could change the behavior of the system (at least the timing).

5.2 Data logging in RTOS-based applications

The *RTEdbg* library functions are reentrant if the circular memory allocation is done correctly. For details on the drivers that are part of the library, see the *Readme* file in the Library\Portable folder and the documentation in the driver comments in the Library\Portable\CPU folder and subfolders. Reentrancy is easily achieved if the processor core supports mutex instructions. For example, this is true for the ARM Cortex-M3/M4/M7/M33/M85, Renesas RXv2, RISC-V (extension A), etc. processors. In general, if one of the following three CPU drivers can be used, no further action is required:

- rtedbg_generic_atomic.h
- rtedbg_generic_atomic_smp.h
- rtedbg_cortex_m_mutex.h

All other processors must temporarily disable interrupts while reserving circular memory. The CPU driver *rtedbg_generic_irq_disable.h* must be used. Two macros must be defined in the *rtedbg_config.h* configuration file. They are not portable as they depend on the CPU core, compiler and RTOS type. See additional descriptions in the *Readme.md* file (same folder as the driver files) and in the comments of the driver files.

The problem is shown for CPU cores that support unprivileged code execution, such as Cortex M0+ and M23. The privileged access level allows access to all resources in the processor, while the unprivileged access level means that some memory areas may be inaccessible and some operations may not be available (such as interrupt disable/enable). The unprivileged access level is not available in the Cortex-M0 processor and is optional (device-specific) in the Cortex-M0+ processors. Since the CPU cannot temporarily disable/enable interrupts, it must call the kernel service to disable/enable them. See FreeRTOS Kernel > API Reference > RTOS Kernel Control > taskENTER_CRITICAL(). The *taskENTER_CRITICAL*() and *taskEXIT_CRITICAL*() functions cannot be called from an interrupt service routine. They also cannot be called before the kernel is started.

The examples below are for devices with ARM Cortex-M0, M0+ or M23 core running FreeRTOS. More examples can be found in the *Readme* file (see Portable folder).

```
#define RTE_ENTER_CRITICAL()            \
    uint32_t irq_tmp = __get_PRIMASK(); \
    __disable_irq();

#define RTE_EXIT_CRITICAL()             \
    if (irq_tmp == 0U)                  \
    {                                   \
        __enable_irq();                 \
    }
```

Such an implementation is correct for projects where the code either does not run under the RTOS or runs under an RTOS that does not use unprivileged mode. Thus, the above macros are also applicable to FreeRTOS on the above processor cores if the Memory Protection Unit (MPU) support is not used - see below.

Information about the FreeRTOS implementation *(see FreeRTOS Kernel Forum):*

1. *If you use a FreeRTOS port without Memory Protection Unit (MPU) support, all the tasks execute as privileged.*
2. *If you use a FreeRTOS port with Memory Protection Unit (MPU) support, the application tasks execute as unprivileged.*
3. *In both the cases (MPU and non-MPU), the kernel code always executes as privileged.*

Use the above simple version of the macros in conjunction with FreeRTOS for ARM Cortex-M0/M0+/M23 only if MPU support is not used. When using this version, the data acquisition is the fastest and the code is shorter.

The following macros are for devices with ARM Cortex M0+/M23 cores running FreeRTOS with MPU support.

```
#define RTE_ENTER_CRITICAL()                                                \
    uint32_t privileged_mode = 0U;                                          \
    uint32_t irq_disabled = 0;                                              \
    /* Running an exception handler or in privileged mode? */               \
    if ((__get_IPSR() > 0U) || ((__get_CONTROL() & 1U) == 0))               \
    {                                                                       \
        privileged_mode = 1U;                                               \
        irq_disabled = __get_PRIMASK();                                     \
        __disable_irq();                                                    \
    }                                                                       \
    else                                                                    \
    {                                                                       \
        taskENTER_CRITICAL();   /* RTOS kernel disables interrupts */       \
    }

#define RTE_EXIT_CRITICAL()                                                 \
    if (privileged_mode)                                                    \
    {                                                                       \
          if (irq_disabled == 0U)                                           \
          {                                                                 \
              __enable_irq();                                               \
          }                                                                 \
    }                                                                       \
    else                                                                    \
    {                                                                       \
        taskEXIT_CRITICAL();    /* RTOS kernel enables interrupts */        \
    }
```

Disabling and enabling interrupts is done by the FreeRTOS kernel only when logging is called from a non-privileged task.

Additional problems arise when the program code is executed on a multi-core device. In such cases, memory reservation must be done using atomic/mutex instructions, data memory barriers, hardware semaphores, etc. Using shared memory on multi-core devices is not trivial.

On processors that have, for example, one core that supports mutex instructions (e.g., Cortex M4) and another that does not (e.g., Cortex M0+), data cannot be logged in a common structure in shared RAM. Use a separate *g_rtedbg* data structure for each core in such cases.

Running the code under the operating system may cause additional data logging problems – see the description in RTOS Task Starvation.

5.3 Optimize Data Logging Code Size and Speed

The *RTEdbg* library contains low complexity code – see the *rtedbg.c* file. Compilers produce short and fast code with low stack usage when higher levels of optimization are used. For most projects, the settings of one of the demo projects can be used and the details described in this section are not relevant. Fine-tuning is important when the microcontroller has small memory or when ultimate code execution speed is required.

For ultimate logging speed, you can use the following measures in addition to those in the table below:

- **Inline data logging functions**
  Include "rtedbg_inline.h" instead of "rtedbg.h" in source files where the most time-critical part of the code is located. In these files, the logging code is inlined directly into the functions from which logging is called. Inline function versions are provided for all macros that log up to four data words.
  If the source file contains interrupt programs that have such a high priority that no other interrupt program can interrupt them, a faster non-reentrant version of circular buffer reservation can be used for logging, allowing even faster (less intrusive) logging.
  Re-entry protection can be disabled by adding the macro
  ```
  #define RTE_USE_LOCAL_CPU_DRIVER "rtedbg_generic_non_reentrant.h"
  ```
  before `#include "rtedbg_inline.h"` in the C source file where logging is added.
- **Sacrifice re-entry protection**
  If the programmer can ensure that the logging functions are only called from parts of the program that can never be executed simultaneously, then re-entrancy protection can be disabled. Use the following CPU driver in the *rtedbg_config.h* file (see above for how to do this locally).
  ```
  #define RTE_CPU_DRIVER "rtedbg_generic_non_reentrant.h"
  ```
  Note: A function that logs data with re-entry protection disabled can be interrupted by functions that do not log data.
- **Sacrifice timestamping**
  If only the logged values and message sequence are important, use the following macros in the *rtedbg_config.h* file.
  ```
  #define RTE_TIMER_DRIVER  "rtedbg_zero_timer.h"
  #define RTE_TIMESTAMP_SHIFT 1U
  #define RTE_GET_TSTAMP_FREQUENCY() 1U
  ```

These measures also reduce program memory and stack usage. Each of these measures has certain consequences that the programmer must be aware of, from increased memory consumption in the case of code inlining to reduced functionality (missing timestamps). However, the execution speed of short logging functions can be much higher if all measures are taken – e.g. for RTE_MSG0() on Cortex-M7 the number of cycles is about half of the minimal value given in the chapter Execution Times, Circular Buffer, and Stack Usage.

The table shows the recommended values of the compile-time parameters to satisfy each criterion.

Compile time parameter	Min. code size	Min. stack usage	Max. execution speed
RTE_MINIMIZED_CODE_SIZE	1	0	0
RTE_SINGLE_SHOT_ENABLED	0	-	0
RTE_DELAYED_TSTAMP_READ	1	1	*
RTE_USE_LONG_TIMESTAMP	0	-	-
RTE_MSG_FILTERING_ENABLED	0	-	0
RTE_BUFFER_SIZE is a power of 2	yes	-	yes
Compiler optimization option	size or balanced	speed	speed or balanced

Legend:

'**1**' – should be set to 1, '**0**' – should be set to 0, '**-**' – no influence,

'*****' – set to 1 for simple cores like Cortex-M0+ … M4 and to 0 for complex cores like Cortex-M7 or if the timestamp timer is connected to the CPU core via a low-speed peripheral bus

If only a small subset of functions is used, RTE_MINIMIZED_CODE = 0 may result in the smallest footprint. Optimization settings do not always produce the expected result for all functions. Code optimized for speed may be slower (and larger) than code optimized for size for a particular function. Play with the compiler optimization settings and examine and test the code to get the best results.

Code optimization parameters can be set for the whole project, for *rtedbg.c* file separately or with macros in *rtedbg_config.h* file. See the description of the RTE_OPTIM... macros and sample optimization settings for the GCC, ARM and IAR compilers in *rtedbg_config.h* in the demo projects.

Further reduction of the impact of instrumentation on code execution

The description in this paragraph applies only to very time-critical, e.g., hard real-time, systems, and only to the most time-critical parts of their code. For the vast majority of real-time systems, it makes no sense to worry about the things described here.

Code instrumentation for logging and performance analysis involves adding additional logging code that can subtly affect program execution. This is also true for the otherwise very efficient instrumentation using the functions of the *RTEdbg* toolkit library. In addition to the execution time of the logging functions, instrumenting code can affect how compilers generate assembly code, in particular how registers are allocated for working variables. The stack usage for logging with the *RTEdbg* toolkit is generally very small. However, by including data logging, for example, in the middle of a complex function, it can be larger than without instrumentation in the function that calls the logging functions if the compiler needs to store some of the locally used values on the stack. With logging function inlining, stack usage is generally lower than without inlining.
To minimize performance overhead, developers should follow some basic principles when implementing instrumentation techniques. The goal is to gain insight into code behavior while maintaining the efficiency of the original code and minimizing performance impact.

Optimal placement of data logging function calls: Programmers should consider where to insert the data logging function macros. If the function has no input parameters and no output value (such as an interrupt handler), the preferred location is at the entry or exit of the function. If the function has input parameters, the preferred location is near the end of the functions within which the values are to be logged. Near the end of the function, the values of local variables are already discarded by the compiler, and calling the logging function does not cause the need to store local variables on the stack or in scratch registers. All this does not apply to inline functions or static functions, which can be included directly (not called as functions) at higher levels of compiler optimization. Inline versions of smaller logging functions (functions that log a small number of values) generally have less impact on the generated assembly code (including stack usage).

5.4 Typical Problems Faced by Programmers

If you are new to the RTEdbg toolkit, you should first check your format definitions in case you have problems decoding logged data, for example, *RTEmsg* application writes a lot of error messages in the *Errors.log* file. If you are not sure if the format definition is correct, or if there is an error in your program that is logging the data, the RTEdbg library, or the *RTEmsg* application, then test the matter on a small example. Test the complex format definitions with known data values before using them in a real application. See also Errors Reported During Pre-Build and Compilation.

Here is a list of common mistakes that programmers make when defining format definitions or decoding data:

1. The position of the first bit of the value to be printed is not properly defined. For example, if the logged value is printed as an 8-bit value, the first byte used as the value to be printed is the least significant byte of a 32-bit value.
2. The bit address of the next printed data value is reset to zero when the value is printed to a different output file as defined by the format definition above the current one. This happens after each change of output file (output redirection to a different file).
3. The format definition is not updated after a data structure change (e.g., an additional element is added).
4. The wrong format ID is being used to log the a variable or structure.
5. Many errors can also occur if one of the format definition files has been modified in such a way that the numbering of the format IDs changes after the program code has been compiled and transferred to the embedded system. It is not a problem if the format strings are changed. The problem occurs when format definitions are added or deleted, or the message type is changed from, for example, MSG2_xxx to MSG3_xxx. Also check if the *g_rtedbg* data structure is at the address it should be. If not, check that the link setup is correct. See also the Linking Requirements section.
6. Many errors can also be caused by using the wrong version of format definition files. Always archive the format definition files with the source code of the project so that the logged data can be decoded correctly later. New format definitions can also be added without changing the format IDs already used to log existing messages – see How to Avoid Problems When Adding or Changing Format Definitions.

How to Check for Potential Data Logging Problems

When analyzing the data, we need to focus on messages with unusual timing (time jumps, data gaps, undefined format IDs, incomplete messages, etc.). This can be caused by data bursts that cause buffer overflows (more data being logged than can be sent to the host in real time).

The *-timestamps* command line argument of the *RTEmsg* application instructs the software to write information about relative timestamp values (the difference between the timestamps of successively logged messages) to the *Timestamps.csv* file. If large relative timestamp values are seen for an application that periodically logs various messages, this indicates potential problems, such as:

- Too much data in a given period of time when streaming to the host computer.
- Because Windows is not a real-time operating system, data transfer software may occasionally not get CPU time for up to a few hundred milliseconds, even when the computer is lightly loaded.
- Data logging of a low-priority task or part of the firmware was interrupted for a long time (longer than the time required to transfer the circular buffer contents to the host) – see RTOS Task Starvation.

Large Number of Errors Detected During Message Decoding

A large number of errors detected during data decoding usually have the following main causes:

- Corrupt binary file or binary file header:
  - data may be transferred from the wrong part of memory (not from the part of RAM that contains the g_rtedbg structure with logged data).,
  - the size of the structure passed to the host is different from the actual size of the g_rtedbg structure,
  - the circular buffer may be corrupted due to errors in the firmware, DMA settings, etc.
- The wrong set of format definitions is used to decode the binary file.
  The format definitions must match the firmware that logged the data. The format definitions must be archived with the firmware for later use after the embedded system is deployed in the field.
- The format IDs have changed after the firmware was compiled (e.g., format ID names have been added or deleted from the format definition files).
  The format IDs do not change when the format definition text is changed. However, they may change if the format ID names are changed (e.g., from MSG1_xxx to MSG2_xxx), if we move the format definition in the format definition file or move it to another file, if definitions are added or deleted, etc.
- Data corruption during transfer from embedded system to host.

Suspicious Data in the Output File

If some data of a particular message in the output file is suspicious, first check the format definition for that message type and also the call to the message logging function. Note that the position of the data in the decoded message is reset each time we start writing the data to a new file.

The probability of incorrect data in the circular data logging buffer is very small. See the descriptions in sections RTOS Task Starvation and Interrupt During Message Logging Followed by Fatal Exception for two of the possibilities. Since the data logging buffer must be accessible to all parts of the firmware (parts that call data logging functions), it is possible that, for example, a faulty RTOS task could overwrite some data in the buffer.

Suspicious data may be the result of a buffer overflow due to insufficient bandwidth to the host. Check the relative timestamps file – see the description of the Timestamps.csv file.

Other possible causes include:

- Incorrect data: Testers interpreting the logged information should review the data logging call (right data recorded?) if suspicious data is repeatedly found in the same logged message type.
- A low-priority RTOS task was interrupted while the data was being written to the circular buffer. And when the data was transferred to the host, the interrupted task was still not ready to run again (to complete the write). For single-shot logging, the host application can usually fix this because the data can be transferred to the host after a pause. However, this is not always possible for 'post-mortem' logging.
- Due to an error in the application, data in the circular buffer could be overwritten. Since all parts of the code have access to the circular buffer, they can also overwrite data and cause memory corruption. The most likely cause is programming errors (such as non-initialized or corrupt pointers). Other types of errors, such as the DMA unit writing to the wrong part of memory, or errors due to electromagnetic interference, or soft errors, are usually less likely.
- When transferring the snapshot to the host for data logged in 'post-mortem' mode, logging is temporarily disabled by setting the filter to 0. This ensures that the firmware cannot modify the data in the circular buffer during transfer to the host. If the firmware periodically sets the filter value during logging, it may re-enable the filter and write to the circular buffer in the middle of the transfer to the host. If this happens, strange data will appear in *Main.log* or other log files, or the *RTEmsg* application will report one or more errors while decoding the data. Set RTE_FILTER_OFF_ENABLED in *rtedbg_config.h* to a

value 1. This prevents the firmware from simply changing the filter value after it has been set to zero (unless the RTE_FORCE_ENABLE_ALL_FILTERS filter parameter is used).

- The *RTEdbg* library contains memory reservation drivers. Especially if you have created or configured them yourself (e.g. for a new CPU family), check if the solution is really reentrant. Also follow the instructions in the readme files in the subfolders of the *c:\RTEdbg\Library\PortableCPU* folder.
- Also note that the NMI (Non Maskable Interrupt) interrupt cannot be temporarily disabled. For example, if the interrupt occurs during circular buffer space reservation, two different messages would be written to the same part of the circular buffer, giving an unpredictable result. This description applies only to processors that do not support mutex instructions (where the *rtedbg_generic_irq_disable.h* CPU driver is used).

Timing Information not as Expected

If the timing data doesn't match the actual application timing, check the following:

1. Are the timestamp driver, timestamp prescaler, timer counter shift value (RTE_TIMESTAMP_SHIFT) correct? Also check if the macro *RTE_GET_TSTAMP_FREQUENCY()* returns the correct value.
2. If the application has changed the CPU timer frequency during execution, the programmer must change the timestamp timer prescaler after the frequency change or log the frequency change with the *rte_timestamp_frequency()* function. The message with the new frequency information may be lost if the data transfer bandwidth to the host is insufficient (information is preserved in the *g_rtedbg* structure).

6 TESTING AND DEBUGGING WITH THE RTEdbg TOOLKIT

6.1 Testing Instrumented Code in Low-Power Modes

Power saving modes range from reducing the clock speed, turning off the clock(s) partially or completely, to cutting power to parts of the chip (e.g., some memories). Different power-down measures have different names with different chip vendors and chip families (sleep, stop, standby). Programmers must ensure that the debug probe has access to the memory with the data logging structure. For a Cortex-M core, the CPU clock must remain running when the core enters sleep mode, and the RAM memory used for data logging should not be turned off. Follow the recommendations of the debugging chip vendor described in the sections labeled "Debugging in Low Power Modes.

Transfer data from the embedded system only when the probe access conditions are met, when the CPU core clock is running, or at least the core clock is not turned off. Set the Debug MCU Configuration Register during code debugging. It is possible to prevent the CPU from completely switching the core clock. This allows access to the CPU core and memory. Be careful not to completely disable the memory containing the *g_rtedbg* data logging structure during sleep, as the memory is not always accessible. See the "Debug MCU Configuration Register" description in the ARM Cortex CPU Family Reference Manual.

The programmer must decide how to update and report the elapsed time during standby. If the *rte_long_timestamp()* function is not used in the project firmware, at least the fact that the code execution was interrupted for a while should be logged in the circular buffer. The macro RTE_RESTART_TIMING() is provided for this purpose. It restarts the timestamp recovery algorithm in the *RTEmsg* application and also restarts the timing statistics. When using the timestamp statistics, the time difference between an event before and after the sleep would be incorrect. Therefore, the timing statistics algorithm is reset each time the message logged with the *RTE_RESTART_TIMING()* macro is detected in the circular buffer data.

The firmware should update the entire timestamp counter (including the upper 32 bits) if the correct time is to be logged after the system wakes from sleep. If an RTC is available for the particular embedded system, then the RTC time (and date) can be logged after the system wakes up. If the CPU operates at a different frequency after sleep / wake up and this clock influences the timestamp timer counter, then the frequency change must be logged with the function *rte_timestamp_frequency()*.

6.2 Test and Debug Functional Safety Applications

The *RTEdbg* library is designed to meet functional safety requirements: statically allocated data logging structure, fail-safe circular buffer indexing, interrupts are never disabled, non-blocking (execution does not stop when the logging buffer is full), etc. No system failure is possible if the data logging variables are corrupted. The *RTEdbg* library is designed to conform to the BARR Group Embedded C Coding Standard and MISRA C:2012 guidelines, with deviations listed in a separate document. Compliance is checked using PC-lint and complexity is checked using GNU complexity.

Great care has been taken during development to keep code complexity low and robustness high. This applies not only to the library of data logging functions, but also to tools that run on the host computer and support automatic format ID assignment and decoding of logged data. Even if the *RTEmsg* application were to generate an incorrect number of format IDs or message filters, this would not affect the execution of the instrumented program code. The only consequence could be the recording of incorrect data.

Data and event logging is not directly related to safety and security. It is only an add-on for testing and optimizing the firmware and the overall system. It is not guaranteed that the logged data will be correct under all circumstances. It is only important that no abnormal situations occur during the logging process. The only problem that can occur with logging is if the wrong address is passed to the logging function, which would cause an error such as a bus error or memory protection error when reading the data. There are no data checksums in the circular logging buffer. They are intentionally omitted to speed up the logging process and have minimal impact on the timing of the embedded system.

The data logging macros *RTE_MSG0()* … *RTE_MSG4()* run fast and the execution times are known. For the *RTE_MSGN()* and *RTE_MSGX()* macros, the programmer defines the maximum size of the data structure or field in the configuration file and thus limits the maximum execution time of these two data logging functions.

There are a few hardware-related things to consider when choosing which part of RAM to place the data logging structure in, such as

1. **RAM parity:** After the data logging function is called, only two values are read from the data logging structure (message filter and buffer index). A parity or ECC error can occur during these two accesses. The data logging functions are robust (always checking and limiting the buffer index value) and will not crash on an incorrect value. Therefore, it is not necessary to place the buffer in a parity or ECC protected part of the memory. Data logging is not directly part of the functional safety firmware. It is there to verify that the firmware is working as it should.
2. **RTEdbg data structure in ECC or parity protected RAM.** If the RAM parity error triggers an exception handler, and the recovery firmware detects that the access was in the address range where the *g_rtedbg* structure resides, then only the following two variables within the structure need to be reinitialized to continue application execution:
   g_rtedbg.buf_index – should be set to zero
   g_rtedbg.filter – should be set to the desired value (only if the message filtering is enabled).
   Since data logging is designed to be immune to random errors in RAM, it is easier to use non-ECC or parity-protected RAM for this purpose.
3. **RAM block enable:** Some microcontrollers have the ability to enable/disable individual RAM blocks to optimize power consumption. If the logging buffer is in such memory, the application firmware should periodically check to see if the RAM block containing the logging data structure and buffer is still enabled, and enable it if it is not.

Data logging should be left in production code because the compiler may make different decisions during general code optimization and object code generation if it is disabled. This can result in object code with different timing or even code that behaves differently.

6.3 Data Logging for Unattended, Remote, and IoT Systems

Many embedded systems do not have a permanent connection to the host computer, or in the case of Internet of Things (IoT) systems, the connection is intermittent (data is sent occasionally) or has low bandwidth. As a result, continuous data logging is not possible. *RTEdbg* functions log data in raw binary format, which reduces the amount of the information that needs to be sent to the host (reducing the cost of data transfer).

The RTEdbg library provides two data logging modes suitable for such applications:

1. **'Post-mortem' data logging:** Data is continuously logged to the circular buffer and then stopped when the firmware detects an event (i.e. fatal error, application error, etc.). The history in the buffer can be helpful in determining the cause of a particular error. This is the default logging mode.
   A programmable trigger can be set in the firmware to stop logging when a specific condition is detected.

The firmware must set the filter to zero to disable data logging until the data is transferred either to the host computer or to non-volatile memory for long-term storage.

2. **Single shot data logging:** When this mode is enabled, logging stops when the buffer is full. This logging mode can be used, for example, to log the system response to a special event. The firmware can start single shot logging after a special condition is detected by setting the message filter to the desired value. When logging stops, the data can be sent over an IoT connection and analyzed or stored in non-volatile memory. See Single Shot Data Logging.

Trigger parameter(s): A trigger can stop data logging in the case of 'post-mortem' logging, or start it in the case of single shot logging. The trigger can also be a special condition, detected application error, system error, etc. The integration of custom trigger variables allows flexible start and/or stop of data logging activities. These variables can be part of the embedded system parameters – either stored in the non-volatile memory of the embedded system or set over a remote connection during a remote test session. One of the variables should be the message filter value – see below. The trigger functionality is application specific and is not included in the *RTEdbg* library.

Message filtering: Since data logging memory is limited, it is important to implement the ability to set the message filter parameter. This can be either an embedded system parameter or a command received from the embedded system via a connection (serial, USB, TCP/IP, etc.). Messages can be divided into 32 groups, each of which can be enabled or disabled separately. Reducing the number of enabled message filter bits increases the logging time for the same logging buffer. It is not necessary to recompile and reload the firmware to change the data logging if the filtering is designed wisely. Either longer logging time (fewer message groups) or shorter logging time and more detailed information (more message groups) can be selected with the message filter value. See also Programmers should plan for the use of filters.

Filter names: Message groups have names that the application on the host can display on the screen. This allows the user to enable filters using names rather than numbers. The *RTEmsg* application writes the filter names to Filter_names.txt file. Each line contains the name of one message group. The first line corresponds to filter number 0 (bit 31 of the filter variable) and the last line corresponds to filter 31 (bit 0). The data transfer program can use the names to make it easier to select groups of messages to log.

Data Transfer to a Host Computer for Analysis

Data transfer to the host can be done through all types of interfaces since only the contents of the *g_rtedbg* structure must be transferred to the host – see the example in *rtedbg_demo.c* file. If logging in a low-priority task is interrupted by a high-priority task (or interrupt handler), the function that sends the logged data to the host or writes it to nonvolatile memory must wait a bit before sending or writing the data – in case the task executing the data transfer to the host has not the lowest priority. See the description of RTOS Task Starvation.

Checksums: The firmware should add a checksum to the data packets sent to the host when using a potentially unreliable communication channel or media. There is no checksum in the *g_rtedbg* data logging structure. The firmware must ensure that the data that arrives in the binary file on the host is intact (and resend it if necessary).

Log buffer size: It is not possible to change the size of the data logging buffer on the fly. It must be defined at compile time because all data structures of the *RTEdbg* library are static.

Non-volatile memory: It is also possible to store the data in non-volatile memory and access it on demand. Multiple events can be logged in, for example, flash memory. Programmers can store the logged information permanently (until cleared by the operator) or allow the firmware to overwrite the oldest information. Inexpensive flash memory allows large amounts of data to be stored (e.g., many 'post-mortem' logs). If possible, they should also be time-stamped, e.g., with real-time clock time and date.

7 RTEmsg MESSAGE DECODING APPLICATION

RTEmsg is a Windows-based application that decodes embedded system log data. Binary data is decoded according to the format definition files. The application is also a pre-build tool for syntax checking the format definition header files and enumeration of format definitions and filter numbers (inserts #define directives with filter and format ID numbers) in the C/C++ header files.

7.1 RTEmsg Command Line Arguments

Syntax: `RTEmsg output_folder fmt_folder {options} {bin_data_file}`
or `RTEmsg @file`

output_folder – Folder where all output files are created – system files like Main.log and user files specified with the OUT_FILE() directive.

fmt_folder – Folder containing format definition files (at least the *rte_main_fmt.h* file must be in it). The files included with *IN_FILE()* and *INCLUDE()* directives must also be in this folder if relative or absolute path names are not defined.

options – Optional arguments (see the list of command line arguments in the table below) "-N=x" is the only argument required.

bin_data_file – Binary input file from the embedded system (not necessary if the '-c' parameter is used)

@file – file with parameters – the first two lines contain mandatory parameters:
line 1: output_folder
line 2: fmt_folder
The file encoding must be UTF-8 if non-ASCII text is used for the parameters.

Exit value: 0 – no error, > 0 – errors found (see the Errors.log file for more information)

Example 1: Process the binary file and create all statistics files:

```
RTEmsg ..\output ..\RTEdbg\Fmt -stat=all -N=10 data.bin
```

Example 2: Parse the format definition files and output format definition headers:

```
RTEmsg . ..\RTEdbg\Fmt -c -N=10
```

Notes:

1. Use either absolute path names or relative to the *RTEmsg* application start folder. If the application is started from an IDE, this is usually the folder where the output files (obj, hex, elf, …) are created.
2. Put quotation marks around the folder or file name if it contains spaces. Remove the backslash '\' character at the end of the folder name if the name is between quotes. The Windows command line interpreter interprets (\") as an escape character.
3. Almost all of the text strings used by the *RTEmsg* application are located in the *Messages.txt* file, which is located in the same folder as the *RTEmsg.exe* file. The default English version currently needs to be manually replaced with a translated one. See the translation description in the `"c:\RTEdbg\RTEmsg\Translation"` folder. Support for automatically using the correct language version will be added later.

List of RTEmsg Command Line Arguments

Argument	Description
-N=**xx**	**xx** – Number of bits used for the message ID (from a minimum of 9 to a maximum of 16 bits) The value should be large enough to accommodate all message types in the current project. The value of the parameter RTE_FMT_ID_BITS in the file *rtedbg_config.h* must be equal to the number of format ID bits defined here. Number of possible format IDs: 2^xx. **Example:** -N=10 → Use 10 bits for the format ID. 2^ 10 = 1024 → up to 1024 different message types – see Short Message Type Comparison.
-c	Perform the pre-build syntax check of the format definition files and update the filter and format ID definition header files with #defines for the filter numbers and format ID numbers. The output files other than Filter_names.txt and Errors.log are not created. Input files defined with *IN_FILE()* are not accessed. Format definition parsing errors are also written to the console (the toolchain error parser can catch and process them). Use the -e command line argument to configure error reporting if necessary. If the format definition file has a '.fmt' extension, the compilation output is written to the header file with the '.fmt.h' extension. If the format definition file has a different extension, the #defines for filter numbers and format IDs are written to the processed file. The output file is not updated if the new version is the same as the current one. This reduces the overall compilation time of the project, since the C source code does not need to be recompiled if the format definition files have not changed.
-back	Create a backup file for each modified format definition header file (.bak suffix is added). By default, no backup files are created after the #defines with format IDs and filter numbers are inserted into the header file.
-stat=all	Run full statistics and create all statistics files – see Statistical Features.
-stat=value	Process the statistics data and create the *Statistics.csv* file.
-stat=msg	Create two files with information about the logged messages: • *Stat_msgs_found.txt* - list of all detected messages with counters (how many times each one was found) • *Stat_msgs_missing.txt* - list of all defined message ID names that were not found in the binary file during message decoding.
-nr=**xxx**	**xxx** – Custom message number printing definition. The leading '%' must not be part of this argument to avoid potential problems with the Windows command line parser. **Example:** `-nr=N04u` – print the message number as: `printf("%N04u", message_no);` By default, the number is printed as "%N05u".
-timestamps	Create the Timestamps.csv file. It contains absolute and relative timestamp values – the difference between the timestamps of successive logged messages.
-time=*unit*	This option specifies the unit of time for timestamps printed during message decoding and all other time-related values: -time=s - values printed in seconds (default) -time=m or -time=ms - values printed in milliseconds -time=u or -time=us - values printed in microseconds

-T=**xxx**	**xxx** – Custom timestamp print definition The leading '%' must not be part of this argument to avoid potential problems with the Windows command line parser. The *RTEmsg* application does not check the syntax of this argument. **Example:** `-T=.3f` – print the timestamp with three decimal places **Note:** The timestamp is printed with the "%1.6f" printf type string when timestamps are printed in seconds by default, "%1.3f" for milliseconds and "%1.2f" for microseconds.
-newline	Add an extra line break after each complete message printed to the Main.log file. This is the default when the -debug command-line argument is used.
-e="**xxx**"	Definition of *RTEmsg* error reporting – see Verifying Format Definitions with RTEmsg. "**xxx**" is a string that defines how the error message is reported to the console and to the *Errors.log* file. It is convenient if the IDE can automatically detect the file name and line number where the error was detected. Different IDEs have their own settings for detecting and parsing error messages, and this is usually not configurable. The following types are available: %L – Line number where the error was detected %E – Error number %F – File name (the file where the error was found) %P – Full pathname (the file where the error was found) %D – Error description %A – Additional information (shows which part of the format definition has a problem) The default error string: `-e="%F:%L: error: ERR_%E %D => \"%A\"\n"` Add the RTEmsg '-utf8' command line argument if you have problems with console output.
-locale=**xxx**	The locale defines the decimal separator for printing values in output files. By default, the locale is set to the ANSI code page obtained from the Windows operating system. This argument allows you to use a custom runtime locale that is different from the system default. **Examples:** -locale=C – default setting for the C code (decimal point used as a decimal separator) -locale=en – English, de – German, fr – French, zh – Chinese, etc. See the ISO Language Code Table for a complete list. **Note:** The decimal separator setting applies only when printing numeric values to output files. The dot must be used for numeric values in the format definition files.
-utf8	Use the UTF-8 code page when printing error messages to the Windows console. Add this option if you have problems displaying non-ASCII messages in the console window.
-debug	This option provides additional diagnostic information. Data is also printed in hex for each message. This option is intended for programmers planning to extend the functionality of the RTEdbg library or *RTEmsg* application, porting code to a new CPU core family, or looking for potential bugs in their format definitions or data logging library or application.
-purge	This option causes all directives inserted by the *RTEmsg* to be removed from the format definition files being processed. It can be used, for example, for format definition files that are included in software libraries. If someone were to add the format definition file from a library and forget to include it with the *INCLUDE()* directive to the *rte_main_fmt.h*, the remaining #define's could cause a mess when logging binary data since the correct format ID values would not be assigned by the *RTEmsg* application. This option only works with the -c option.

-ts=**a**;**b**	Set the sensitivity of checking the order of timestamped messages (for advanced users). The value 'a' defines the maximum expected delay in completing logging of a message where space has already been reserved in the circular buffer for a message but the timestamp value has not been retrieved from the timestamp counter when logging was interrupted by a higher priority task. The value 'b' defines the maximum expected difference in timestamps between successively logged messages. Both values are in ms. The default value for 'a' is -0.10 and for 'b' is 0.33 of the timestamp period. Too large values reduce the probability of detecting message loss, e.g. due to too slow data transfer to the host. The *RTEmsg* application marks suspicious (out-of-sequence) messages with a hash sign '#' before the message number in the Main.log file – see Message Numbers. Each logged message contains the lower part of the long timestamp, which has a length of 32 - 1 - N bits, thus limiting the maximum timestamp value contained in each message. The long timestamp has an additional 32 bits for a total of 64 – 1 – N. The (short) timestamp logged in each message overflows once per timestamp period. The *RTEmsg* application detects this and increments the upper 32 bits of the long timestamp even without the long timestamp. The data logging functions are reentrant and can be interrupted. In the vast majority of cases each successive message has a slightly larger timestamp than the previous one, and the *RTEmsg* application can detect when an overflow has occurred when the upper bit flips. Any *RTEdbg* logging function can be interrupted during data logging after it has already reserved space in the circular buffer but has not yet acquired a timestamp from the timestamp timer. Because of this, the timestamps of consecutive messages may not always be in ascending order. A large difference can occur if the logging of a low-priority RTOS task was interrupted by a higher-priority task and the latter did not release a CPU core for a long time. Both small and large negative differences between timestamps must be considered by the *RTEmsg* application and it must attempt to correctly recover the higher part of the timestamp. The logging functions are executed quickly, and the probability of such an event is very low in practice. However, there is a much higher probability that the circular buffer will overflow due to insufficient bandwidth for data transfer to the host in streaming data transfer mode. In such a case, the missing messages with their timestamps do not hinder the long timestamp recovery if the missing gap is less than a quarter of the timestamp period. Such overflow recovery is more reliable if long timestamps are also recorded. The procedure for recovering the long timestamp and detecting possible circular buffer overflows can be influenced by setting this command line argument. The long timestamp value should be logged with the *rte_long_timestamp()* function at least once per timestamp period to synchronize timestamp decoding for embedded systems where the circular buffer may overflow (e.g., because the data transfer rate to the host is slower than the data logging rate in the embedded system). **Example:** The argument values -ts=-10;25 define that the *RTEmsg* application should assume that the maximum timestamp difference between any two consecutive messages should be less than 25 ms and that no message is logged with a delay greater than 10 m. If the time difference of the decoded message is greater, the *RTEmsg* application assumes that the message was logged one timestamp period later (messages lost due to data overflow) and increments the upper part of the long timestamp value (upper 32 bits).

7.2 The Default Output Files

The *RTEmsg* application generates several files by default – see the table below. Programmers can define additional output files and print (sort) data to them.

File name	Description
Main.log	Main log file with information about all messages
Errors.log	Errors messages
Filter_names.txt	Message filter names
Stat_main.log	General statistics containing information about the decoded messages

Output files are created in the ***output_folder*** specified as a command line argument. This applies to all custom output files unless otherwise specified. See below for a detailed description of all standard output files. Files with the ***Stat_*** prefix contain the statistical information – see Statistical Features.

Note: Data decoding will stop if one of the output files cannot be created or opened for writing – for example, if the text editor or log viewer locks the open file.

Main.log

When decoding binary messages, all information is printed to this file by default, unless otherwise specified. If the data is sorted into multiple output files, the *Main.log* file can be used to find a sequence of events – for example, what happened before or after a particular event.

The *Main.log* file contains the following information (in the header and trailer):

- Version and revision of the *RTEmsg* application that created the file.
- Date/time when the binary file was decoded.
- Name and date/time when the binary file was created (when data was transferred to the host).
- List of *RTEmsg* command line arguments used to decode the data.
- Binary file header information – information about how logging is configured: buffer size, timestamp, single shot or 'post-mortem' data recording, etc. The binary file size is reported in 32-bit words and includes the size of the circular data buffer and trailer (four additional words). For data loaded directly from the embedded system (*g_rtedbg* logging structure), the length is RTE_BUFFER_SIZE + 4. The last_index is the position in the circular buffer where the next information would be written.
- Message filter information (which message groups were enabled)
- Warnings and notes at the end of the file (if any)
- The "`Execution time`" values show the time spent processing the format definitions (first value) and the time spent decoding the binary file from the embedded system.

For each decoded message, the following information is written to the file:

- Message number, timestamp, format ID name, and the values as specified in the format definition.
- Error message if the message could not be decoded, a buffer overflow was detected, etc.

The following command line arguments affect the generation of the Main.log file.

-T=xxx	Custom timestamp print definition.
-time=unit	Specify the time unit for timestamps printed during message decoding.
-newline	Add an extra line break after each complete message printed in the *Main.log* file.
-nr	Define a custom message number format.
-locale=xxx	Allow the use of a custom runtime locale that is different from the user default.

-debug	Additional diagnostic information (for testing the *RTEdbg* library and *RTEmsg* application).

The message serial number, timestamp, and name are automatically printed for each message. For all other specified output files, the message serial number and timestamp must be added using printf definition extensions such as "%t" and "%N" - see below for an example of a message with a long timestamp.

```
N02786 0.601793 MSG1_LONG_TIMESTAMP: 0x1239
```

Legend:

N02786 – Consecutive message number (increases with each decoded message)
A hash before N (for example, #N01234) indicates that the timestamp is suspicious.
See the description of Message Numbers.

0.601793 – Timestamp (seconds, unless otherwise specified with the command line argument)

MSG1_LONG_TIMESTAMP – The name of the format ID of the message.

The data following the first three values is as defined by the programmer in the format definition.

Each message in the *Main.log* file is tagged with the message serial number and timestamp. Either the timestamp or the message serial number can be used to locate the message in the *Main.log* file and check what happened before or after the erroneous message was detected.

How are errors reported when decoding binary files?

1. If a data error (i.e. incomplete message, corrupted message data, unknown format ID) is found, only the content (hex data) is displayed together with the message number. The timestamp is not displayed because its value cannot be reliably determined for such messages.
2. Error information is also printed to the Errors.log file.
3. An error may be reported for the first message of a snapshot (the oldest message), as it may not always be fully preserved – see the datails in The first message of a snapshot may be partially overwritten.

The value of the filter variable at the time of the data transfer to the host computer is also displayed – see below:

```
Message filter: 0x00000000 (filter copy: 0xFFFFF800)
  0 = 0(1) "System and other important messages"
  1 = 0(1) "Filter test 1"
  2 = 0(1) "Filter test 2"
  ...
```

The first column shows the filter number, followed by the enable bit of the filter variable and its copy for the displayed filter number/name. A zero value for "Message filter at end of logging" can be the result of:

- Call to the *rte_set_filter(0);* – the firmware has disabled the message logging
- Data logging stopped with a debug probe to get a snapshot of the logged data – see the snapshot batch file example: Transferring Data to the Host Using the ST-LINK Debug Probe
- User intervention (e.g., setting the filter variable using a debugger)

The following notes or warnings are written to the *Main.log* file when appropriate

Long timestamps were defined for the firmware, but no such message was found in the binary file.

In the firmware, the functionality to record long timestamps was enabled, allowing full time synchronization between the embedded system and the host computer where the data is decoded. Despite the functionality being enabled, no long timestamp was found in the recorded data.

Messages with a '#' in front of the message number have suspicious timestamps (the data may also be suspicious) - a total of %u suspicious messages.

When decoding binary data, the *RTEmsg* application expects the data to have been recorded continuously without large time differences between the timestamps of successive recorded messages. If it detects a large difference, it will report this error. It doesn't have to be a real error. Data logging may have been disabled by the firmware for a period of time, or it may not have been logged for some other reason (e.g., going to sleep). The *RTEmsg* **-ts=a;b** command line parameter allows the programmer to set the sensitivity of this check. The '#' in front of the message may not indicate a real error. Data logging may have been disabled by the firmware for a period of time, or it may not have been logged for some other reason (e.g., going to sleep).

The first message of a snapshot may be partially overwritten when the last message is written, and this may be the cause of the error shown above.

This message may appear after the first message decoded in a snapshot. In the circular memory where the *RTEdbg* library functions record data, the newest message always partially or completely overwrites the oldest message or several smaller older messages at the same time. The oldest message in the buffer is therefore not necessarily preserved in its entirety. This can lead to an error message when decoding the first (oldest) message.

Note: If the size for the partially overwritten message is a message whose size is defined, *RTEmsg* discards the "truncated" message, but if it is not defined (e.g., format definitions whose name starts with the prefix MSGN_ or MSGX_), then such a message is decoded and an error may be reported during decoding.
If the length is known, use MSGNxx_ (xx is the number of words logged) instead of the MSGN_ prefix.

Unfinished words with content = 0xFFFFFFFF
Data without FMT word
Valid subpacket(s) with format ID

This text marks messages that RTEmsg could not correctly assemble and recognize. Such data is output in hexadecimal format only. It is up to the programmer to extract useful information from this data if possible. Such data in circular memory occurs, for example, due to interruption of data logging in the middle of writing a message (all logging functions are reentrant), memory corruption, transfer of data from the wrong part of memory or wrong size of memory block transferred to the host computer, etc.

Errors found while decoding the message. The value(s) displayed with the number and printf string may be incorrect.

This warning tells the programmer that either the message is of a different size than the format definitions specify (for example, it contains too little data), there is an error in the format definition, or something else. If the single value to be printed cannot be properly prepared according to the format definition, the value 0 is printed (or an empty string if the printed value it is a string).

Example:

```
N00490 65,324 MSGN_STRING: String dump: F0 82 FE A7
  Full string: "x_(J", characters 3-7: ""
N00490 => Errors found while decoding the message. The value(s) displayed with the number
and printf string may be incorrect.
-->#3 - "", characters 3-7: "%s"
 ERR_127: Value end address (64) is above last bit address (32) of this message.
```

Format definition string for the above example

```
// MSGN_STRING "String dump: %1H\n  Full string: \"%s\", characters 3-7: \"%[16:48]s\""
```

Note: The data '-->#3' shows which sequence '%x' from the format definition was used for output to the log file (x - data type - '%s' in the above example). In the first two output methods '%1H' and '%s' the size was not defined and the message was output first in hexadecimal format and then as a string. In the third output method, '%[16:48]s', the data was not output because it requested the output of characters 3 to 7 and there were only four characters (1 … 4) in the message.

Errors.log

This file contains the complete list of errors, including format definition parsing errors. This file is usually empty if the format definitions are correct and the messages have been properly logged / transmitted to the host.

Consider the following:

1. Some errors detected during data decoding are reported only to the *Errors.log* file and not to the *Main.log* file as well. These are errors that are detected before decoding the data from a binary file:
   - Errors found in format definition files.
   - Output file management problems (creating files, writing to files, etc.).
   - Problems with input files (e.g., file does not exist, cannot be accessed).
2. Errors detected while parsing format definitions are reported according to the -e="..." command-line argument (or using the default definition).
3. Messages that do not have the correct length according to the format specification are not decoded and are not written to files other than *Main.log*. This prevents possible data transfer errors, write errors, etc. from resulting in erroneous data in custom output files (e.g., CSV) and strange data plots that would confuse the tester.

See the RTEmsg Error Messages section for additional explanations of various error messages and the Typical Problems Faced by Programmers section for descriptions of possible causes.

Filter_names.txt

The *Filter_names.txt* file is created in the output folder during the syntax check. It contains filter descriptions for 32 message filters. Each line represents the description of a filter that enables a group of messages. The first line corresponds to filter number 0 (bit 31 of the filter variable) and the last line corresponds to filter number 31 (bit 0 of the filter variable). The first line contains the name of the filter with the number 0.

This file is intended for use with software tools that allow, for example, continuous logging of messages (streaming mode) and enabling and disabling of individual message groups. It is created in the specified output folder when the '-c' command line argument is used.

Timestamps.csv

The file contains absolute and relative timestamp values. Relative values are differences between the time-stamps of successive logged messages. Data is prepared as a CSV file to allow graphical display of values, making it easier to identify suspicious values and extremes. The file can be used to check for things like

- whether data overruns have occurred (a large time difference between consecutive messages indicates that some messages may have been lost),
- whether the data logging function in low-priority firmware parts has finished writing data.

The '***-timestamps***' command line argument must be used to enable this file.

Below is an excerpt from the *Timestamps.csv* file:

```
N03996;465.4140;0.0011
N03997;466.3980;0.9836
```

The first value is the message number, the second is the timestamp, and the third is the difference between consecutive message timestamps. The time difference is given in seconds by default (or ms, or µs, as defined by the command line argument). Timestamps for messages decoded with error(s) are not written to this file.

7.3 Statistical Features

The *RTEmsg* application collects some statistical data during message decoding by default. The Stat_main.log file is created each time the *RTEmsg* performs message decoding. All other files listed in the table below must be enabled using the RTEmsg Command Line Arguments.

The statistics function allows the programmer to

1. Detect minimum and maximum values of decoded data value.
2. Detection of minimum and maximum values of time between specified messages (timing analysis).
3. Count the number of occurrences for each message (e.g., number of specific events). The number of occurrences for all messages found in the binary data file is printed to Stat_msgs_found.txt.
4. Write a list of messages for which no occurrences were found in the logged data (missing messages) to Stat_msgs_missing.txt. This information may also indicate that some parts of the code were not executed (not tested).

Data logging in embedded systems generates a lot of values. It is time consuming to manually review so much data. Statistical functions in the *RTEmsg* application allow quick selection of extreme data and timing values. Timing and numeric data values are collected along with the message numbers where the values were found. This makes it possible to search for other data before or after the message where the extreme value was found.

The statistics functions allow you to easily define statistics for logged data values and time parameters (time between two messages – value defined with [%T] or [%t-MSG_NAME]). For a description of how to enable these functions in format definitions refer to the Define Variable Name for Statistics |Value Name|.

Note: It is important to note that for messages whose length is known in advance, the length is specified by the format ID name – see Format ID and Filter Naming Conventions. Then the *RTEmsg* application can check if the decoded message contains the correct amount of data before decoding it. The values from incorrect messages will not appear in the statistics.

All statistics files are listed below. By default, only the Stat_main.log file is created. The information is written to all other statistics files only if enabled with the '**-stat=...**' command line argument. All statistics files are created in the specified output folder.

Stat_main.log	General statistics – general information about the decoded messages.
Statistics.csv	Statistics file with extremes for the specified message data and time values.
Stat_msgs_found.txt	File with counters of format IDs processed during message decoding.
Stat_msgs_missing.txt	File with names of format IDs for which no message was logged.

Stat_main.log

This file contains general logging statistics:

1. The total number of messages processed (number of messages found in the data file).
2. The number and percentage of format IDs used.
3. The number of error messages logged. The possible error types are counted separately. The list is empty if everything is OK. See Interrupt During Message Logging Followed by Fatal Exception and RTOS Task Starvation for an explanation of two possible scenarios. Data in the circular buffer could be corrupted due to incorrect firmware behavior, during transfer to the host, etc.
4. A list of the top 10 message types, including their message sizes.
5. A top 10 list of message types that consume the most circular buffer space helps identify which messages use the most buffer or bandwidth for host data transfer.

Statistics.csv

The *RTEmsg* application allows easy collection of minimum and maximum values. These are either values from logged messages or time values (for example, the difference between two logged message timestamps). See section Define Variable Name for Statistics |Value Name| for description on how to enable value statistics for each numeric value. Several extreme minimum and maximum values are displayed in the *Statistics.csv* file, along with the message numbers where the particular value was detected. If only a small number of values were found in the input data, all values are displayed. It is possible to measure the time between any two types of messages and include the difference in the statistics – use the '%T' and '%t-MSG:NAME' format extensions.

Example of a format definition that includes a statistics field.

```
// MSG3_SINCOS_DEMO "%u, %g, %|Calculate sin/cos [cycles]|u"
```

Example of one field in the file *Statistics.csv*.

Calculate sin/cos [cycles] Value name	Maximum	862	860	860	860	857	857
MSG3_SINCOS_DEMO Format ID name for the message with this value	for message no	N00710	N00049	N00217	N01535	N00382	N01049
		Message number in which the value shown above appeared					
	Minimum	782	786	787	787	787	787
	for message no	N01835	N02523	N01672	N02493	N02499	N02502
Average / # of samples		820,67 Average	1246 # Samples				

The meanings of the fields in the table are shown in green in the example above. The leftmost column of numeric values shows the value of the maximum and minimum value found. Below each maximum/minimum value is the serial number of the message in which the value was found. This makes it possible to search for other data before or after the message in which the extreme value was found. The average of all samples is displayed at the bottom along with the number of samples processed.

Note: The RTE_RESTART_TIMING() macro should be called after the system has been restarted (e.g., after a watchdog reset), after the system has woken up from a sleep, or after the data logging has been paused by the firmware. This will log a special message informing *RTEmsg* application to prevent the statistics from taking into account the time difference between the last messages logged before the sleep and the same messages logged after the sleep.

Stat_msgs_found.txt

The file *Stat_msgs_found.txt* contains a list of all message types for which at least one message was found in the binary input file. The first column contains the number of message occurrences, and the second column contains the format ID names.

Stat_msgs_missing.txt

The file *Stat_msgs_missing.txt* contains a list of all message types for which no messages were found in the binary input file. This information can be used, for example, to check that all parts of the code have been executed (tested) and that there are no obsolete format IDs.

Static code analysis tools are one way to detect and then remove the obsolete format ID definitions. Some toolchains also provide this functionality. For macros defined in source files, the -*Wunused*-macros option is available in gcc and clang based toolchains. There is also a -*Wunused*-local-typedefs option in GCC.
See also the Cscout application https://github.com/dspinellis/cscout – a source code analysis and refactoring tool for collections of C programs that helps find unused extern's and #define's (not just unused format definitions).

7.4 Verifying Format Definitions with RTEmsg

The *RTEmsg* application does not only decode binary files. It's purpose is also:

- pre-build verification of format definitions, and
- enumeration of format ID definitions and filter numbers.

The table below shows how to add the *RTEmsg* pre-build check to some popular IDE's or Makefile projects. The examples below assume that the *RTEmsg.exe* application is located in the suggested installation folder:

```
c:\RTEdbg\RTEmsg\
```

Most IDE's set the start folder for external applications like the *RTEmsg* to the top of the folder where the compiler and linker write the output files. This is usually the Release or Debug folder in an Eclipse IDE project. The project folders are therefore typically one level above the *RTEmsg* start folder. The output folder for the *RTEmsg* is defined as '.' in the examples below. Therefore the output files will be created in the folder where the IDE puts the output files during compilation. The Errors.log file is created in this folder. Errors reported during the pre-build format definition check are written to it.

Note: The -N=xx *RTEmsg* parameter must have the same value as the RTE_FMT_ID_BITS parameter in the *rtedbg_config.h* file.

Eclipse based IDE's

Set: Project Properties => C/C++ Build => Settings => Build Steps => Pre-build Steps => Command:

```
c:\\RTEdbg\\RTEmsg\\RTEmsg.exe . ..\\RTEdbg\\Fmt -N=10 -c
```

Keil MDK

Set: Project Options => User => Before Build/Rebuild => Run #1 (or #2):

```
c:\RTEdbg\RTEmsg\RTEmsg.exe . ..\RTEdbg\Fmt -N=10 -c
-e="%%P(%%L): error:  ##%%E: %%D => \"%%A\"\n"
```

Place a checkmark in front of the command and select the *Stop on exit: '>= 1'*.

This definition should be used within Keil uVision. A double '##' must be used because the uVision IDE uses '#' for macros. Add the '%%' because of the Windows command line parser.

The double '%%' and '##' should be replaced with single '%' and '#' when compiling from a Makefile.

IAR EWARM

Set: Project Options => Build Actions Configuration => Pre-build command line:

```
c:\RTEdbg\RTEmsg\RTEmsg.exe . ..\RTEdbg\Fmt -N=10 -c
-e="Error %E: %D => \"%A\"\nFile: \"%P\", Line: %L\n"
```

This is not the optimal configuration because the EWARM IDE does not recognize the file and line number information. The author did not find a description in the "IAR C/C++ Development Guide" or "IDE Project Management and Building Guide" on how to format the error information of the pre-build command. It was also not possible to get adequate information from IAR support. As a result, the programmer cannot click Error in the Build Message window info to jump to the line with the error.

7.5 RTEmsg Error Messages

The errors messages are written to the *Errors.log* file and, for format definition errors, also to the console. This allows the IDE to catch the errors and display them in its window. The format of the error output when parsing format definition files is configurable – see the RTEmsg Command Line Arguments section for a description of the '-e' parameter. See also RTEmsg Command Line Return Codes and Large Number of Errors Detected During Message Decoding if a large number of errors were found in the *Errors.log* file.

RTEmsg Error Table

Most of the error messages are self-explanatory. The *RTEmsg* application tries to display the part of the message that is most likely the cause of the error. The table below provides additional descriptions only for error numbers where a short one-line text cannot adequately describe the error or problem found.

Error	Additional explanation
ERR_037	*The binary file 'file_name' does not contain enough data (size = xxx)* In the header of the *g_rtedbg* data structure passed to the binary file, there is a size (buffer_size) that is greater than the number of 32-bit words in the file.
ERR_039 ERR_040	*Wrong binary file header – buffer size* *Wrong binary file header – reserved bit values not equal to zero* The *g_rtedbg* data structure contains the *rte_cfg* configuration word and part of this word is the header size. The same configuration word contains information about the number of format ID bits, whether single shot recording is enabled in the firmware, etc. It contains some reserved bits that must be zero.
ERR_063	*Missing name of the format definition folder or syntax error* The name of the folder containing the format definition files is a required parameter. This error may also occur if there is a `\' character at the end of the folder name specified between quotation marks.
ERR_100	*Block with %u unfinished words (0xFFFFFFFF) found. The preceding and/or following message may also be affected.* The value 0xFFFFFFFF cannot appear in normal logged data. This value overwrites the entire circular buffer during initialization. If this value appears in the data, it means that the recording of the individual message was interrupted and was not completed until the data was transferred to the host computer. **Example:** `N00148 ERR_100: Block with 4 unfinished words (0xFFFFFFFF) found. The preceding and/or following message may also be affected.`
ERR_104	*Incomplete message or too many consecutive DATA words found (x words skipped)...* Each data packet contains one to four 32-bit DATA words and one 32-bit FMT (format ID and timestamp) word, and the logged message consists of one or more such packets. Each of them must contain one FMT word. This message indicates that either a message has not been completely written or the circular buffer has been corrupted. Check the next and previous message to see if the data displayed is plausible. **Example:** `N00367 ERR_104: Incomplete message or too many consecutive DATA words found (3 words skipped). The following message(s) may also be affected.` `>>> N00367 index: 805  Data without FMT word: 0xF8627868 0x9B202006 0x20203BB4` Data: 32-bit data values as found in the binary file (values from the circular buffer) index: Index of the first data value belonging to the decoded message **Note:** The program assumes that the last part of an excessive number of consecutive DATA is a correct message, so it does not skip all consecutive DATA words, but only those it finds that are unlikely to belong to the next message or the following format ID. However, this is not necessarily

	always the case, so it also gives a warning that the next message could also be incorrect. The programmer must check whether the data in the following message makes sense.
ERR_106	*Not enough data in the binary file (less than specified in the structure header…* The typical cause is either data being transferred from the wrong memory address, or the size of the *g_rtedbg* data structure does not match the size of the data block being transferred to the host.
ERR_110	*Binary file contains more data (xx) than indicated in the header. The entire file is decoded, but data may be corrupted.* In the embedded system, data is logged into a data structure that contains a circular buffer. The header of this structure contains information about the length of the circular buffer. If the contents of the entire structure have been transferred to the binary file, then the header of the structure or the header of the binary file will contain a length that matches the length of the binary file. Such an error is reported by the *RTEmsg* application if, for example, more data has been transferred than the size of the data structure. The value shown as **xx** above is the buffer length from the *g_rtedbg* data structure (binary file header) and this length does not match the length of the binary file.
ERR_112	*No format definition for the ID number:* The cause of such an error may be, for example, the use of files with format definitions other than those used to compile the firmware, corruption of data in the data structure in which the data is recorded, incomplete data logging (e.g., there is an interruption during data logging and during the interruption a fatal error is triggered, causing the logging to stop), etc. **Example:** `N00382 ERR_112: No format definition for the ID number: 688` `>>> Format ID: 688, data: 0x55AF562D` If the message contains data in addition to the format ID, it is written to the log file in hexadecimal format as shown above. For messages with an unknown format ID, the timestamp is not processed or printed.
ERR_114	*The circular buffer index is out of range. Its value was reset to zero. Data may not be decoded in the correct order.* Messages are decoded as they are found in the file, not from the last index in the circular buffer because it was out of range (decoded data may not be in the correct order).
ERR_116	*The message size is xx bytes. According to the format definition, it should be yy bytes.* The message size found in the binary file during decoding does not match the defined message size. **Example:** `N00426 ERR_116: 'MSG3_COMBINED', The message size is 8 bytes. According to the format definition, it should contain 12 bytes.` In this case, the decoded message contained two 32-bit data words, but according to the format definition, there should have been three.
ERR_117 ERR_118 ERR_141 ERR_142	*Suspicious MSGX message …* The macro *RTE_MSGX()* allows to record data messages whose length is not divisible by four and is unknown during firmware development. The highest byte of the last DATA word with the *RTE_MSGX()* macro of the recorded message gives the length – the number of recorded bytes. These errors are reported by the *RTEmsg* application if it detects too few or too many 32-bit words in the message based on the number of bytes. Error 142 is reported if the last word of the message contains non-zero byte(s) in the unused part of the word that should be zero.
ERR_131	*Suspicious message: non-zero extended data found in a message (MSGN/MSGX) that should not contain it (extended value = xx).* The FMT word logged with messages of type *RTE_MSGN()* or *RTE_MSGX()* contain 1 to 4 bits of unused bits depending on the message type. They must be zero. This message informs the programmer that there is probably something wrong with the data (or format definition).
ERR_209	*File name too long* The full path name is limited to 260 characters. This is a Microsoft Windows limitation that can be

	circumvented, but the workaround has not yet been implemented in the current version of *RTEmsg*.
ERR_216	*Too many OUT_FILE, IN_FILE, MEMO, and {...} definitions* The number of definitions is limited to 2000 in the current implementation.
ERR_282	*Bit address of the value must be divisible by 32. Check the previous value or set address with the [...] definition* A simplified format definition (no value size and or address defined for the printed value) can be used to output data to a log file where it is not necessary to specify the bit address, length, and data type. For example, instead of "%[32:32f].2f", only "%.2f" can be used. The *RTEmsg* determines the data type based on the type of the output data ('f' = float in this case). If the format definition previously contained definitions for data types shorter than 32 bits or the data was not aligned to 32 bits, then the bit address of this data is not aligned to 32 bits and the *RTEmsg* does not know which bits of the message to decode and write to the file. In such a case, the bit address and the data size and type must be specified in the format definition.
ERR_283	*Value of bit address + size exceeds the message size* The format definition provides for the output of data with an address and size that exceeds the defined message size. This error can also occur if the programmer adds or removes variables from the structure without changing the format definition. **Example** for a message of type MSG3_…: Such a message contains 3 × 32 = 96 bits. The format definition "%[88:8u]u" is correct because the number of the last bit of the data 88 + 8 = 96 is still in the range 1 … 96 for a message of this type.
ERR_294	*The value definition field '[...]' is mandatory before the scaling field '(...)'* The *RTEmsg* must know the type of value to properly convert (prepare) the binary value. All integer values (signed and unsigned) must be converted to double before scaling can take place.

Notes:

1. Error descriptions may change (improve) in future releases, but error numbers should remain the same. Therefore, always include the error number when referencing.
2. If the error description warns that the previous and/or next message may also be incorrect, the programmer should verify that the contents of those messages are logical.
3. If a large number of errors or suspicious data are reported in the decoded messages, or a large number of error messages are reported, check the Typical Problems Faced by Programmers.
4. If the error message text contains UTF-8 characters, the console window may not display them correctly. If this happens, check the error messages in the *Errors.log* file. The -utf8 command line argument can be used to control how errors are printed to the console.
5. Every piece of information is important to a programmer looking for the cause of a malfunction. Especially if the error is difficult to reproduce. Therefore *RTEmsg* tries to print all data (as hex) even if it cannot decode it correctly (as the programmer intended).

Errors Reported During Pre-Build and Compilation

It is important that the parameters of the data logging functions are correct to ensure that the correct data is logged and can be decoded by the *RTEmsg* application running on the host. This is especially important when looking for hard-to-replicate problems. Programmers must ensure that the correct data is being logged. The *RTEmsg* application checks that the format definitions are syntactically correct and match the message logging macro types. The compile-time parameter check with static_assert() during the pre-build phase can automatically catch some additional problems. For example, the compiler checks that the filter number is within the allowed range (between 0 and 31) and that the format ID is within the range and correct. See the *rtedbg.h* header file for implementation details.

The error `"Invalid format ID value - lowest bit(s) must be 0."` is reported if the format ID does not match the message type – e.g., for the macro RTE_MSG2() the lowest two bits of the format ID must be 0. This error is usually caused by a wrong format ID definition or by using the definition in the wrong data logging function. For example: The definition MSG1_... which should be used as parameter for the macro RTE_MSG1() is used as parameter for the macro RTE_EXT_MSG1_y(). This error typically occurs when the programmer uses the wrong message name (format ID) - e.g., MSG0_... in macro RTE_EXT_MSG0_.

RTEmsg Command Line Return Codes

The return value of the *RTEmsg* application can be useful if you are compiling from a script, powershell, cmd or batch file. The software reports errors to the *Errors.log* file and to the console. Errors 3 and above are used to report fatal problems that cannot be reported to *Errors.log*. These errors should not occur in normal operation. See the Errorlevel and Exit codes for various ways to check an *errorlevel* (batch file execution).

Exit value	Error description
0	No errors were detected.
1	An error was detected during format definition processing. Could not decode binary file.
2	The decoding of the binary data file was not completed because a fatal error was detected while decoding the data. Check the contents of the *Errors.log* file.
3	Errors detected while decoding a binary file. Decoding of the binary file was completed because there were no fatal errors in the binary file. Check the contents of the *Errors.log* file.
4	Stack space exhausted.
5	Fatal exception when processing a format definition file (internal error).
6	Fatal exception when processing the binary file (internal error).
7 … 9	Reserved for future use.
10	Unable to retrieve current working directory information.
11	Cannot retrieve information about the *RTEmsg* application start folder.
12	Cannot change the working directory to the *RTEmsg* start folder.
13	Cannot change the directory back to the working folder (the folder where *RTEmsg* was started).
14	Cannot change the directory to the specified output folder.
15	The *Errors.log* file could not be created in the output folder – for example, the output folder does not exist, the file exists but is locked, and so on.
16	Bad command line argument (or argument in the parameter file @file)
17	Problem with *Messages.txt* file. The number of lines of text messages does not match the number of message indexes in the *RTEmsg* application.
18	The *Messages.txt* file cannot be opened or located. This file contains all the text messages used for reporting errors, writing reports, statistics, etc. This file should be located in the same folder as the *RTEmsg.exe* application.

8 Data Visualization and Analysis Tools

Tools for viewing log files or graphical data visualization are not included in the *RTEdbg* toolkit. Many existing tools can be used to view the data. The *RTEmsg* can generate custom output files with user and/or machine-friendly data. If long periods of time are logged, the output files can be very large. This list contains software tools that have been identified as useful for data analysis. Most of the tools listed here are also free for commercial use. Please let us know about tools that you find very useful and should be added to this list. These can be standalone tools or plugins for popular IDE's like Eclipse or VS Code.

Plain Text File Display and Analysis Tools

Notepad++	Very fast text editor with fast multi-file search, auto reload, etc.
fLogViewer	Log file visualization (live update, color coding, filtering and searching)
Monitor Text File Changes in Real Time	List of other log file viewers that allow you to view log information in real time.

CSV viewing and Analysis Tools

LibreOffice Calc	LibreOffice spreadsheet software
Microsoft Office Excel	Microsoft spreadsheet software
Flow CSV Viewer	A lightweight tool for viewing and editing time series data files in CSV format. It opens a CSV file instantly without any dialogs or complicated commands.
CSViewer	Fast viewer for large CSV files
KST Plot	Data visualization, large file and live streaming support, data plotting and manipulation
Gnuplot	Command-line driven graphing application
LabPlot	Data visualization and analysis
Matlab / Simulink	Commercial numerical computing environment, data analysis, data plotting, etc.
SciDAVis	Free scientific data analysis and visualization application
List of Free and Open Source Plotting Tools	A list of tools for mathematical and statistical analysis of data

Eclipse Based Tools

Eclipse LogViewer	Real-time viewing of log files within the Eclipse IDE
Eclipse Trace Compass	A powerful event viewing and analysis tool used in many large projects. Supports many trace formats. Of the supported data formats, Best Trace Format is probably the easiest to implement with the *RTEmsg* application. It is a simple ASCII trace format and can be easily defined using the format definitions.

Other Tools

Babeltrace	Trace manipulation toolkit

8.1 Tips for Use of Selected Tools

This section provides some tips on how to use selected tools more effectively. It is generally recommended to use the UTF-8 variable-width character encoding to ensure that international fonts are handled correctly.

Notepad++

Notepad++ is a fast and powerful text editor. It is suitable for viewing large files created by the *RTEmsg* application. It has several features that are very useful for viewing logged data. There are also several plugins available. It is easy to browse and quickly search large files.

Automatically reopen files when the software is started Settings => Backup Select the [*Remember the current session for next launch*] check box if you want all open files to be reopened the next time you start the program.
Auto reload open files (reload after changes are detected without prompting) Settings => Preferences => MISC Click on the [*Update silently*] check box – see under *File Status Auto-Detection*. If the [*Scroll to last line after update*] option is enabled, Notepad++ will automatically display the last lines after updating the file(s).
Run a batch file to transfer and decode the data **Set the shortcut:** Press <F5> or click on [Run=>Run], type the full path (or select the batch file from the file system), press the [Save], select the hotkey, and give the new shortcut a name. With a shortcut it is possible, for example, to load new snapshot data from a running system, decode it and display all log files again in Notepad++ with a single keystroke. For an example batch file, see Transferring Data to the Host Using the ST-LINK Debug Probe. Using a batch file like this, it is possible to temporarily pause data logging (set the filter to zero) without interrupting the firmware execution, transfer the data to the binary file, decode the file, and start (or restart) displaying the collected information. **Note:** If the batch file is run from within Notepad++, the working folder will be set to the folder where the *Notepad++.exe* file is located, not the folder where the batch file is located. The programmer needs to add a CD (change directory) command to the batch file – jump to the folder with the batch file (or a folder where the output files will be created) – see the batch files in the Demo projects (*TEST* folder).
Find specific message information across all open files The message number is intentionally written as Nxxxxx (xxxxx – message number) and not as a simple number. Such naming allows reliable searching for a specific number in large log files. Searching for a simple number would return too many false positives. **Notepad++ Find in files feature** Press <Ctrl+F> to open the search tool. Enter the search string and click the [Find All in All Opened Documents] button.
NppLogGazer plug-in NppLogGazer is a Notepad++ plugin for fast and convenient searching especially for log analysis.
Turn Notepad++ into a log file analyzer This online article shows how to set up Notepad++ to support a custom language.

View and Analyze CSV Files

Many programmers prefer tools like LibreOffice Calc or MS Office Excel for detailed analysis of CSV files. However, these tools are not suitable for quickly displaying the contents of the CSV file in a graphical form. They are also not optimal for large CSV files. Below are two examples of tools that can display CSV data.

KST Plot

The Kst software is a tool for viewing and plotting large data sets in real time and has built-in data analysis capabilities. Data presentation can be automated using a powerful command-line interface and can also be scripted using Python. It has a large set of powerful functions for manipulating plots with keyboard and mouse. Export to bitmap or vector formats allows results to be included in test reports.

Flow CSV Viewer

The Flow CSV Viewer displays the information immediately upon opening the CSV file. It automatically takes care of many things that would otherwise have to be set up by the programmer. The tool is very convenient when a CSV file does not have too many columns or a huge amount of data.

Check and set the preferences – see: *File => Preferences* (i.e. CSV Import, Open files from last session on startup etc.).

The tool is quite simple to use, but some useful features are not visible at first glance. So check the help: *Help => Keyboard Shortcuts*, *Mouse Operations* (i.e. *Create new group*, *Reorder groups*, etc.) and others. See below for an example screenshot (data from a demo project).

Notes:

1. The Flow CSV is distributed through the Microsoft Store. There is no easy way to start the viewer from the command line (from a batch file). It is recommended that you set the Flow CSV Viewer as your default CSV viewer if you want the files generated by the *RTEmsg* application to be automatically opened with the batch file after the data has been transferred to the PC and decoded.
2. If you start downloading and decoding new data too soon after the previous download and decoding, it may happen that the Flow Viewer has not finished processing the data in one of the CSV files and does not close the file. In such a case, the *RTEmsg* application will not be able to write new data to the file.

LibreOffice Calc

The following command can be used to automatically open a CSV file in read-only mode from a batch file:

```
"%ProgramFiles%\LibreOffice\program\scalc.exe" -view "file.csv"
```

If read-only mode is not enabled, the *RTEmsg* application cannot rewrite the file while it is open in Calc. The reload function must be started manually after the file is modified in the Calc software, it will not automatically reload the file or display a modification warning.

9 Transferring Collected Data to a Host

This section shows some of the methods that can be used to download binary logged data from the embedded system to the host computer, start data decoding, and automatically open the log files. It also shows how to temporarily stop logging without stopping firmware execution by setting the message filter to zero. The snapshot can be easily transferred to the host using a debug probe, any communication channel, or media.

9.1 Create a Batch File or Firmware for Data Transfer to a Host

First, look at the *g_rtedbg* data logging structure (see the *rtdbg.h* file).

Offset	Variable name	Variable description
0	**buf_index**	Index to the buffer location where the next message will be written.
4	**filter**	Each bit of the 32-bit filter variable enables a message group.
8	rte_cfg	RTEdbg configuration word with information on enabled features.
12	timestamp_frequency	Timestamp timer frequency [Hz].
16	**filter_copy**	A copy of the filter variable contains the last non-zero *rte_set_filter()* parameter value. Host software or firmware can restore the value after logging is paused (copy this value to the *filter* variable).
20	buffer_size	Data logging buffer size = RTE_BUFFER_SIZE + 4 (32-bit words)
24	buffer[buffer_size]	Logged data

The programmer (or tester) must know the address and size of the *g_rtedbg* data structure if its contents are to be transferred to the host using a debug probe.
The size is: 24 + 4 × (RTE_BUFFER_SIZE + 4) // For the header, circular buffer and the buffer trailer

Only the following three *g_rtedbg* structure variables are relevant when testing with the IDE's built-in debugger and transferring data with the debug probe.

filter	Set the *filter* variable to zero before sending data to the host to temporarily disable (pause) data logging. Code execution continues normally. See the batch file examples for how to do this with debug probes. If this variable is affected by the debugger, it should save the value of the variable before setting it to zero. When the transfer is complete, the old filter value is written back and logging resumes. Use the contents of the *filter_copy* variable to re-enable data logging if it has been stopped by the firmware. If the transfer to the host or media is done with firmware, use the *rte_set_filter(0)* function to disable data logging before the transfer begins. Use *rte_restore_filter()* to restore the previous filter value from the *filter_copy* or *rte_set_filter()* to set a different value.
buf_index	The buffer index must be reset to zero before restarting data logging in single 'shot mode' only. It must be reset before the filter variable is set to a non-zero value. In 'post-mortem' recording mode, it is not necessary to reset this variable.
filter_copy	When the firmware manipulates the value of the filter variable with the *rte_set_filter()* function, any non-zero filter value is also written to the *filter_copy* variable. The *rte_restore_filter()* function restores *filter* variable from *filter_copy* so that data logging can continue.

Notes:

1. The complete contents of the *g_rtedbg* structure must be transferred to the host and written to a file. The file must be written in binary mode to ensure that the byte order of the file is preserved exactly.
2. When manipulating the *filter* variable with the debugger, be careful not to set bit 31 to zero unless you need to set the entire variable to zero. This bit disables system messages such as long timestamps. This is prevented by the *rte_set_filter()* function when setting the filter with the firmware.
3. If an RTOS task writing log data could be interrupted while writing data by another task of higher or equal priority, add some delay before transferring data to the host or writing it to media. If the CPU is not fully occupied, a lower priority process will also get CPU time and complete any unfinished data writing. See also RTOS Task Starvation and Data logging in RTOS-based applications.

9.2 RTEdbg Toolkit Data Transfer Tools

RTEgdbData utility

The RTEgdbData command line utility allows data logged with the RTEdbg library functions to be transferred from the embedded system to a binary file on the host via a GDB protocol. GDB servers are available for most debug probes, allowing testing to be performed in a consistent manner. Because of its ubiquity, the first tool for transferring logged data from an embedded system uses the GDB server protocol to access the memory of the embedded system. This makes it possible to transfer data to the host even for debug probes that do not have a command-line utility for transferring data.

The GDB server software is either part of the Debug Probe software or a separate package, as in the case of OpenOCD. The IDE automatically starts the GDB server when we start debugging the code. The programmer can start the GDB server with custom settings and set the debugger in the IDE to use the already started GDB server. When testing without a debugger, the programmer has to start the GDB server himself.

This utility stops data logging by setting the message filter to zero (if not already stopped by the firmware), transfers the data to the host (writes it to a binary file), and then restarts data logging by restoring the message filter value. The firmware runs normally during the data transfer, but the messages are not logged during this time. They are discarded instead of being written to the circular buffer.

The RTEgdbData utility automatically detects the current data logging mode (post-mortem or single shot) and restarts logging when the data transfer is complete. It can be used to:

1. **Get a snapshot of a running system:** Logging is only paused during data transfer while the code is running normally. When the data transfer to the host is complete, logging is enabled again.
2. **Get single-shot data:** In single-shot mode, the firmware stops logging when the circular buffer is full. Single-shot mode logging is automatically restarted when the data transfer to the host is complete. It can also be disabled by setting the filter to zero with the command line argument "-filter=0". In this case, the firmware can enable data logging by setting the message filter to a non-zero value, e.g. using a software trigger.
3. **Get post-mortem data:** Transfer data after logging has been stopped by the firmware, e.g. due to a fatal system error or a fatal application error.

A persistent mode is also enabled, allowing multiple data transfers, automatic decoding start and display after decoding, etc. See the RTEgdbData Readme for complete documentation. Support for streaming (continuous) data transfer mode and other interfaces such as COM or TCP/IP port is not yet implemented.

9.3 Using a Debug Probe to Transfer Data to a Host

If a debug probe can be connected to the embedded system, the data can be transferred using the tools provided by the debug probe vendor.

First, check the address and size of the ***g_rtedbg*** data structure (e.g., in the map file or using a Build Analyzer). Usually this structure should be at a fixed address so that it is not necessary to search for it in the address map file – see the Linking Requirements. The size of the data structure in bytes = 40 + 4 × RTE_BUFFER_SIZE.

A command line application is available for most debug probes. By creating a batch file with startup parameters for the programmer's command-line application, the contents of the ***g_rtedbg*** structure can be transferred to a binary file. Data decoding with the *RTEmsg* application can be started with the same batch file.

The following examples show how to transfer the contents of RAM to the host computer using common debug probes. For these examples it is assumed that the

- *g_rtedbg* structure is located at RAM address 0x24000000,
- the circular buffer size is 2048 32-bit words and
- the complete data logging structure size is 8232 bytes = 2048 × 4 + 40 = 0x2028.

The message filter variable is located at offset 4 (address 0x24000004). Writing a zero to the 32-bit filter variable temporarily disables data logging.

Notes:

1. The following sections show some examples of transferring circular buffer contents using two common debug probes. See your debug probe documentation for software command line options. Sample batch files for data transfer and decoding are included in the demo projects – see Demo Projects. Each demo project has a TEST folder that contains batch files for automating the data transfer. This folder also contains the ***Readme.md*** file with additional description. See also the comments in the batch files and the Segger J-LINK cmd files.
2. It is recommended that you either install the software tools in user-defined paths (default paths may change with software version) or add their paths to the system PATH after installation – see: learn.microsoft.com/en-us/windows-server/administration/windows-commands/path
3. When attempting to retrieve data from an embedded system after a fatal error, such as the bus fault, and the watchdog is enabled, an error may be reported by the application used to transfer the data from the embedded system. The watchdog may interrupt the memory read operation. Not all applications provided with the debug probes are resistant to this problem. Repeat the operation if this problem occurs.

Transferring Data to the Host Using the ST-LINK Debug Probe

The sample batch file below pauses data logging by setting the filter to zero, transfers the data to the host, resumes data decoding with the previous filter value, and displays the contents. The same batch file can be used to transfer data from an embedded system whose code is running normally, a system where the firmware has stopped data logging by setting the message filter to zero, or a system where code execution has stopped (e.g., a breakpoint or exception has been thrown).

The file contains the following sequence of commands (example shown for the Simple_STM32H743 demo):

1. Delete the old binary data file.
2. Transfer the data from the embedded system using the ST-LINK debug probe
   - Read the 32-bit filter value from address 0x24000004 into the *Filter.bin* temporary file.
   - Write a zero value to the message filter variable to pause (temporarily stop) data logging.
   - Read some data into the *Temp.bin* temporary file to get a short (optional) delay for messages to be written to the buffer when multiple of RTOS tasks with different priorities are running.
   - Write back the 32-bit filter value (resume data logging).
3. Decode the binary data using the *RTEmsg* application. The data is written to the *output* folder.
4. Open the output files with Notepad++ and CSV Viewer.
5. Remove the temporary files.

```
@echo off
REM Take a snapshot of a running embedded system and decode the data with RTEmsg
REM and launch Notepad++ and CSV Viewer to view the decoded messages.

REM The command CD (change working directory) is only necessary if the batch file
REM  is started from another program - e.g., Notepad++
cd "c:\RTEdbg\Demo\Simple_STM32H743\TEST"
if exist data.bin del data.bin

"c:\ST\STM32CubeProgrammer\bin\STM32_Programmer_CLI.exe" -c port=SWD mode=HOTPLUG shared -q --read
0x24000004 4 "Filter.bin" -fillmemory 0x24000004 size=4 pattern=0 --read 0x24000000 0x800
"Temp.bin" --read 0x24000000 0x2028 "data.bin" --write "Filter.bin" 0x24000004
IF %ERRORLEVEL% NEQ 0 goto Error

REM Decode the binary data file. The files will be created in the output folder.
c:\RTEdbg\RTEmsg\RTEmsg.exe output ..\RTEdbg\Fmt -N=10 data.bin -stat=all -time=m -timestamps
REM Show data only if no error (0) or no fatal errors found (3)
IF %ERRORLEVEL% EQU 0 goto ShowData
IF %ERRORLEVEL% EQU 3 goto ShowData

:Error
REM Pause to display error message in console window
pause
goto end

REM Display the decoded information
:ShowData
start "" output\sin_cos.csv
start "" output\Battery_data.csv
start "" "%ProgramFiles%\Notepad++\notepad++.exe" output\*.log output\*.txt
goto end

:end
REM Remove the temporary files
del Filter.bin
del Temp.bin
```

The `cd "folder name"` line is only necessary if the batch file is started from the Notepad++, since the working folder from which the batch file is started is the folder where the Notepad++ software is installed – see the Notepad++ section.

Notes:

1. Program execution of running firmware is not stopped or interrupted at all during data transfer. If the filter variable is set to zero, message logging is temporarily paused. Messages are skipped instead of being logged.
2. The batch file should be customized to meet the needs of your application.
3. The `start "" (or start "title")` is used to open a separate Command prompt window. Otherwise, the batch file execution would stop until the started command is finished.
4. **Shared mode** must be enabled if the command line application is to be used in parallel with the debugger – e.g., GDB integrated in the STM32CubeIDE. The same has to be done in the debugger settings within the debug configurations – enable the [Shared ST-LINK].

Transferring Data to a Host Using the Segger J-Link Debug Probe

J-Link Commander is used to transfer data from the embedded system to the file in the example shown below. See the J-LINK Commander documentation for a detailed description. See also the description in the timestamp timer driver *rtedbg_timer_cyccnt.h* to avoid potential problems when using this timestamp timer.

```
@echo off
REM Take a snapshot of a running embedded system, run the RTEmsg application to decode the data.
REM and launch Notepad++ and CSV Viewer to view the decoded messages.

REM The command CD (change working directory) is only necessary if the batch file
REM  is started from another program - e.g., Notepad++
if exist data.bin del data.bin

REM Data transfer using the J-LINK debug probe
REM The address and size of the RTEdbg data structure are in the Snapshot_JLINK.cmd
"c:\Program Files\SEGGER\JLink\JLink.exe" -If SWD -ExitOnError -CommandFile Snapshot_JLINK.cmd
IF %ERRORLEVEL% NEQ 0 goto Error

REM Decode the binary data file. The files will be created in the output folder.
c:\RTEdbg\RTEmsg\RTEmsg.exe output ..\RTEdbg\Fmt -N=10 data.bin -stat=all -time=m -timestamps
REM Show data only if no error (0) or no fatal errors found (3)
IF %ERRORLEVEL% EQU 0 goto ShowData
IF %ERRORLEVEL% EQU 3 goto ShowData

:Error
REM Pause to display error message in console window
pause
goto end

REM Display the decoded information
:ShowData
start "" output\sin_cos.csv
start "" output\Battery_data.csv
start "" "%ProgramFiles%\Notepad++\notepad++.exe" output\*.log output\*.txt
goto end

:end
REM Remove the temporary files
del Filter.bin
del Temp.bin
```

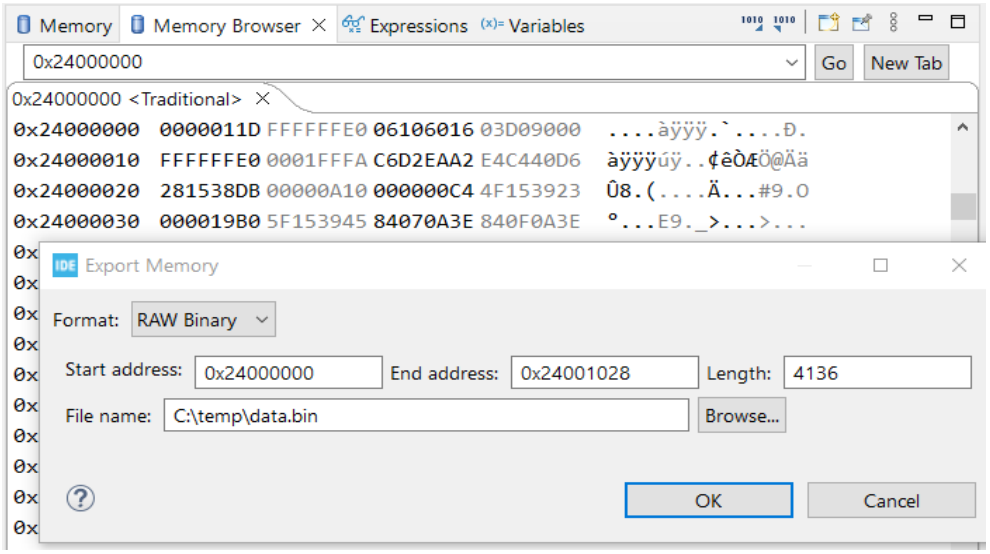
Some of the functionality is defined in the J-LINK command file – see the example below.

```
speed 12000
device Cortex-M7
savebin Filter.bin 0x24000004 0x4       // Save the current filter value
W4 0x24000004 0                         // Pause data logging (set filter to zero)
savebin Temp.bin 0x24000000 0x400       // Load data - delay to enable the remaining data
                                        // to be written
savebin data.bin 0x24000000 0x2028      // Save the g_rtedbg structure contents
LoadFile Filter.bin 0x24000004 noreset  // Restore the message filter (resume data logging)
exit
```

9.4 Save Embedded System Memory to a File Using an Eclipse IDE

If a command-line tool is not available for your debug probe, the data can be transferred to the host using the Eclipse-based toolchain functionality. Data logging must be stopped manually by stopping it through the debugger (pause code execution, breakpoint reached, etc.) or by setting the message filter to zero.

In Eclipse based toolchains it is possible to dump the contents of the memory window to a file. Click on the "▸1010" (Export) icon, then enter the start and end address of the *g_rtedbg* data structure.



Running a batch file from an Eclipse-based IDE

While testing the instrumented code simultaneously with the IDE's built-in debugger, the data snapshot and display in one of the windows or on a separate monitor can be launched directly from the IDE (without having to switch between applications).

A batch file, as described in Transferring Collected Data to a Host, can also be started from the IDE. A batch file can retrieve snapshot data from a running or stopped embedded system and decode the binary data. When the log files are opened within the IDE, the file contents can be automatically refreshed.

The following example shows how to do this for the STM32CubeIDE. The programmer has to adapt it to his preferences or project requirements.

1. Select: Run => External Tools => External Tools Configurations => [Main]
   Set the Name and select the batch file using [Browse Workspace] or [Browse File System].
   Set the [Working Directory] to the directory where the batch file is located – or any other directory that is appropriate for the particular batch file.
2. Then select the [Refresh] tab and check [Refresh resources upon completion] as well as [Entire workspace], and [Recursively include sub-folders] if necessary.
3. In the [Build] tab the [Build before lunch] should be unchecked.
4. Then select the [Common] tab and set Encoding to UTF-8 (or Default if it is UTF-8), [Allocate console] and [Launch in background].
   Set the [Display in favorites menu] to External tools.

The external tool (external batch file) can be started with Run => External tools => select the tool from the list. It is possible to bind the last used external command to a hotkey. See below for a description.

Select: Windows => Preferences => General => Keys, select the command "Runs the last launched external Tool" from the list and bind it to a hotkey. The external command must be run from the menu first.

9.5 Transfer recorded data to the host via a serial channel

The code in the RTEcomLib repository demonstrates how to transfer the data logged with the *RTEdbg* library to the host via a serial channel. The code shows how to transfer data between the embedded system and the host over a two-wire and a single-wire (half-duplex) connection. Single-wire communication is especially useful for microcontrollers with a very limited number of pins or where almost all pins are in use.

Serial data transfer can also be used if transfers with the *RTEgdbData* utility in parallel with the IDE's built-in debugger are not possible, or if it would limit the functionality of debugging (e.g. if it would be necessary to disable the *Live View* functionality). It is also useful in cases when, due to the limitations of the GDB server when transferring data with the *RTEgdbData* utility, the execution of the code in the embedded system stops when *RTEgdbData* connects to the GDB server.

The RTEcomLib_NUCLEO_C071RB_Demo repository contains demo code for the RTEcomLib library. The code runs on the NUCLEO-C071RB demo board (STM32C071RB – ARM Cortex M0+).

The data transfer utility is in the RTEcomData repository. This utility allows data to be transferred from the embedded system to the host via an asynchronous serial channel.

The address of the *g_rtedbg* structure used to log data is not important since the data transfer functionality is part of the firmware and it has the address. Therefore, there are no special requirements for project linking.

This utility stops data logging by setting the message filter to zero (if not already stopped by the firmware), transfers the data to the host (writes it to a binary file), and then restarts data logging by restoring the message filter value. The firmware runs normally during the data transfer, but the messages are not logged during this time. They are discarded instead of being written to the circular buffer. The utility automatically detects the current data logging mode (post-mortem or single shot) and restarts logging when the data transfer is complete. It can be used to get a snapshot of a running system, get single-shot data and post-mortem data. A persistent mode is also enabled, allowing multiple data transfers, automatic decoding start and display after decoding, etc.

Additional instructions can be found in the Readme files in the repositories mentioned above.

10 APPENDIX

10.1 Demo Projects

The RTEdbg distribution ZIP file contains the following projects in the *Demo* folder.

Sub-folder name	IDE's used	Chip	Core type	Clock freq.	Board used for test
Simple_STM32H743	STM32CubeIDE, IAR EWARM, Keil MDK - ARM Compiler 5 & 6	STM32H743	Cortex-M7	64 MHz	NUCLEO-H743ZI
STM32L433		STM32L433	Cortex-M4	16 MHz	NUCLEO-L433RC-P
STM32L053		STM32L053	Cortex-M0+	16 MHz	NUCLEO-L053R8
STM32H743		STM32H743	Cortex-M7	64 MHz	NUCLEO-H743ZI
lpcxpresso54628_ hello_world	MCUXpresso	LPC54628	Cortex-M4	220 MHz	LPCXpresso54628

The table shows the clock frequency of the CPU core as used in the particular demonstration project.

STM32 Projects: No peripheral pins are used. All projects use internal clock oscillators. Therefore, the demo code can be evaluated on any chip of the same CPU family and on almost any board with such a device, since all I/O pins are used as analog inputs.

NXP MCUXpresso project: Existing demo code (hello_world) for the LPCXpresso54628 development board was used. The demo was extended with the *RTEdbg* library and demo source files. Compilation and linking options have been modified where necessary.

Notes:

1. The ***STM32H743.CubeMX*** folder contains the original project created with the STM32CubeMX code generator. If you compare the contents of this folder with the *STM32H743* or *Simple_STM32H743* demo folders, you can see what has been added and changed to the project for data logging with the *RTEdbg* function library. The unnecessary files have been deleted from this and other demo projects for clarity. The STM32CubeMX copies too many files – from the HAL driver source and header files to the header files in the *Drivers\CMSIS\Include* folder.
2. Each demo folder for the STM32 CPU family contains the complete setup for the STM32CubeIDE (main folder), IAR EWARM (EWARM subfolder) and for the Keil MDK (MDK-ARM subfolder) toolchains. Not all compiler/linker settings are the same for different IDEs.

Demo Project for the NUCLEO-H743ZI Development Board

The demo project shows how to integrate the data logging functions into a project. The ***Demo\STM32H743*** folder contains code designed and tested for the STMicroelectronics NUCLEO-H743ZI/ZI2 development board. No I/O has been used (all pins except debug are initialized as analog I/O), so the code will run on any board with a processor from the same family. Therefore, it will at least run on boards with STM32H742/743/753/750 CPUs. The internal oscillator is used to allow operation on boards without crystal or external oscillator.

The folder ***Demo\STM32H743.CubeMX*** contains a project created with the STM32CubeMX. This project has been stripped of all unnecessary HAL driver and header files for clarity. The ***Demo\STM32H743*** folder contains the same project with the changes and additions listed in the table below. Both project folders have been included instead of a detailed recipe describing what to do in the new project to include data logging. A software tool such as WinMerge can show all the differences between the two projects. With this information and other details shown here, the demo can be easily ported to boards with other ARM Cortex-M4 or M7 CPUs. The

project was developed with the STM32CubeIDE and tested with the Keil MDK and IAR EWARM to verify the portability of the code, format definitions and the library code optimization settings.

The table shows the changes made to the original code and project settings.

The *RTEdbg* folder is added to the main project folder. It contains the *RTEdbg* library files. The format definition files are in the *Fmt* folder.
The *Demo_code* folder contains the following files: • *Inc/fault_handler.h* and *fault_handler.c* – Example of a common exception handler with exception handler data logging → See ARM Cortex-M4 / M7 Exception Handler Example. • *simple_demo.c* – Demo code for the RTEdbg library showing the basic functionality. • *rtedbg_demo.c* – Demo Code for the *RTEdbg* Library showing how to use the data logging library. This is a demonstration of various data logging functions and library test code. At the beginning of the *rtedbg_demo()* function is an example of logging the cause of a reset and frequency change. This code is specific to the STM32H7xx family and must be disabled if the demo is to be tested with other hardware. • *rtedbg_test.c* – Code to test the data-logging functions after porting to new compilers or CPUs.
The following changes have been made to the *main.c* file: 1. A call to *rtedbg_test(), rtedbg_demo()* and *simple_demo()* has been added. Activate/deactivate the functions with `#if 1 / #if 0` to test them individually. 2. The code for this demo is linked into RAM. Therefore, the vector table offset register must be set at the beginning of the *main()* function. 3. In the function *Error_Handler()* is a demonstration of how to implement logging of calls to error handlers. See Example: Logging Calls to the STM HAL Error_Handler() for a detailed description.
In the file *Core/Inc/stm32h7xx_hal_conf.h* some of the default enabled headers have been disabled for this demo to speed up compilation. These files are not required for this demo project. **Note:** The STM32CubeMX includes many files that are not required for code generation – from drivers (*Drivers\STM32H7xx_HAL_Driver* folder) to include files to files for other CPU cores and other compilers (*Drivers/CMSIS/Include* folder). Most of the unnecessary files have been removed to simplify the project, speed up compilation and reduce the size of the ZIP file containing the RTEdbg distribution.
In the file *startup_stm32h743zitx.s* `"b Infinite_Loop"` has been replaced by `"b start_exception_handler"`. This will start the shared exception handler each time an exception or interrupt is thrown for which the programmer has not provided an exception/interrupt handler. The exception handler with data logging is located in *RTEdbg\Demo_code\fault_handler.c*. The example above is for the GCC ARM toolkit. Also included in the demo are startup file modification examples for IAR EAWARM and Keil MDK. Linking and include settings have also been done.
The following folders have been added to the include path list – see below. Project properties ⇨ C/C++ Build ⇨ Settings ⇨ MCU GCC Compiler ⇨ Include paths: ../RTEdbg/Inc ../RTEdbg/Fmt
Linking: Compare the original linker configuration file *STM32H743ZITX_RAM.ld* with the linker file in the STM32H743 folder. The file has been modified to use the fast internal RAM (ITCM and DTCM) to test how fast the data logging functions can execute under optimal conditions. Only a section was added to specify in

which part of the RAM the data logging structure should be placed:

```
. = ALIGN(4);
.RTE (NOLOAD) :       /* RTEdbg data logging memory section */
{
  *(RTEDBG)
  *(RTEDBG*)
} >RAM_D1
```

This is an example of how we can place the *g_rtedbg* data structure at a specific address (to the RAM_D1).

GCC compiler settings for the demo:

1. The default optimization level for the project is -O0. For the demo, the level is set to -Og.
   Project properties => C/C++ Build => Settings => Tool settings => MCU GCC Compiler => Optimization => Optimization level = Optimize for Debug (-Og)
2. Linker script is changed to *${workspace_loc:/${ProjName}/STM32H743ZITX_RAM.ld}*
   Project properties => C/C++ Build => Settings => Tool settings => MCU GCC Linker => General
3. Convert to binary option for build output was disabled.
   Project properties => C/C++ Build => Settings => MCP Post build outputs = Convert to binary = Disabled

The pre-build settings:

The command

```
c:\\RTEdbg\\RTEmsg\\RTEmsg.exe . ..\\RTEdbg\\Fmt -N=10 -c
```

has been added to the

Project properties => C/C++ Build => Settings => Build Steps => Pre-build command:

to enable the automatic generation of #define's for the filter numbers and format IDs.

See the Verifying Format Definitions with RTEmsg section for a description of how to integrate pre-build validation and format ID generation into other toolchains..

How to Run and Test the Demo Firmware

This demo project can be opened with three different IDEs: STM32CubeIDE, Keil MDK and IAR EWARM. The code must be compiled, downloaded and executed. Each demo folder contains a subfolder called TEST. It contains demo batch files for transferring data to the host and decoding the binary data file. The batch file does not stop firmware execution when the data transfer batch files are run. See the *Readme.md* for details.

Example batch files for transferring data to the host computer using a debug probe are included in the demo projects (see the *TEST* folder). All file paths in the batch files assume that the project is located in the *c:\RTEdbg\Demo* folder.

Demo Project for the NUCLEO-L433 Development Board

This demo is located in the *Demo\STM32L433* folder. The CPU core in this demo is the Cortex-M4. It is basically the same as the demo for the NUCLEO-H743ZI.

The *g_rtedbg* data logging structure is located at address 0x10000000. For timestamp functionality, either the CYCCNT counter or the TIM2 timer can be used without code modification (enable one or the other in the *rtdbg_config.h*). An example for the SYSTICK timer is also included. However, to use it as a timestamp timer, the project code would have to be modified, since the SYSTICK is used for HAL tick functionality.

Demo Project for the NUCLEO-L053 Development Board

This demo is located in the folder *Demo\STM32L053*. The CPU core type is Cortex-M0+. The STM32L05x family does not have 32-bit timers. Therefore the SYSTICK timer was used for the timestamp counter. This is a 24-bit counter and full timestamp resolution is possible and there are no limitations regarding the value of RTE_FMT_ID_BITS. In such case, one of the 16-bit timers (e.g., timer TIM2) must be used for the HAL tick functionality instead of the default SYSTICK. The code is linked for execution in flash memory. The *g_rtedbg* data logging structure is located at address 0x20000000.

The NUCLEO-L053 demo also includes an example of how to implement a timestamp using a 16-bit timer counter – see the file *rtedbg_timer_stm32L0_tim2.h*. This driver has two limitations:

- The timestamp resolution is only 16 bits.
- The format ID size defined by RTE_FMT_ID_BITS must be 15.

Demo Project for LPCXpresso54628 Board with MCUXpresso

The demo for the LPCXpresso54628 board can be found in the folder *Demo\lpcxpresso54628_hello_world*. The basis for this demo project was chosen from the NXP example code because it is one of the simplest demo code projects available for the LPCXpresso54628 and thus suitable for demonstrating the integration of data logging.

The calls to the demo code necessary for testing the data logging and some additional functions have been added to the *hello_world.c* file. The *RTEdbg* subfolder containing the RTEdbg library and the demo files has been added to the project *source* folder.

Additional *g_rtedbg* structure definitions have been added to the link definition file

lpcxpresso54628_hello_world_Debug.ld

and the managed linker script was disabled. The automatically generated linker script files have been moved out of the debug folder because they might be deleted by the project clean command. They have been moved up one level for the demo project and the script path has been changed accordingly.

The *g_rtedbg* structure with the circular buffer has been placed into the 32 kB SRAMX memory with the following definition added to the linker definition file "*lpcxpresso54628_hello_world_Debug.ld*".

MCUXpresso IDE User Guide section ***Placing code and data into different Memory Regions***.

```
/* RTEdbg data logging memory section */
. = ALIGN(4);
.RTE (NOLOAD):
{
    *(RTEDBG)
    *(RTEDBG*)
} >SRAMX
```

This places the *g_rtedbg* data logging structure at address 0x04000000.

The following pre-build command

```
c:\\RTEdbg\\RTEmsg\\RTEmsg.exe . ..\\RTEdbg\\Fmt -N=10 -c
```

has been added to the

Project properties => C/C++ Build => Settings => Build Steps => Pre-build command

Caution: The main build will be executed regardless of the success/failure of the pre-build step – see link.

An additional file *main.h* has been added to the *source/RTEdbg/Inc* folder. It contains definitions of things which are not available in a project generated with the MCUXpresso toolchain or which are different from the code generated with STM32CubeIDE to maintain code compatibility between demo projects.

The debug configuration in the demo is for a Segger J-Link LITE running on the Link2 debug adapter integrated on the board. It was programmed with LPCScrypt.

The following C compiler settings have been changed:

1. 'Tool settings => MCU C Compiler => Dialect' has been changed to ISO C11. If the settings are for a C version older than ISO C C11 (or GNU C 11), the parameters of the RTEdbg data logging macro cannot be checked at compile time. In this case, the RTE_CHECK_PARAMETERS definition in the configuration file must be disabled (value set to 0).
2. Two additional folders have been added to the 'Tool settings => MCU C Compiler => Includes':
   ${workspace_loc:/${ProjName}/source/RTEdbg/Inc}
   ${workspace_loc:/${ProjName}/source/RTEdbg/Fmt}

Important: The SDK for the *lpcxpresso54628* must be installed before testing this demo project. The toolchain will not detect and report the problem if the SDK is missing. Various errors will be reported when trying to set the debug configuration or start a debug session instead of the real cause.

10.2 Example: Logging Calls to the STM HAL Error_Handler()

Programmers use, for example, numeric values or __FILE__ and __LINE__ macros or predefined identifiers (e.g., C99 __func__) to identify the location of the block in the source file that may be causing a problem (during error reporting). None of this can be used together with the code automatically generated by the STM32CubeMX tools. An example from the STM32H743 demo project (file *main.c*) is:

```
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
    Error_Handler();
}
```

The error handler is called with no parameters to indicate what has happened or what the cause might be.

The link register (LR) contains information about the program location from which the handler was called (program counter value). If this value is known, programmers can identify the source code part from the list file. Logging the LR value is easier than logging the __FILE__/ __LINE__ and uses only 8 bytes of the circular buffer. See below for the modified *Error_Handler()* function from the demo STM32H743 (file *main.c*).

```
/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */

    // Log the location from which this function has been called
    uint32_t link_reg = (__get_LR() & ~1) - 4;
        // Get the address from which the Error_handler() was called (see the Caveat below)
    RTE_MSG1 (MSG1_ERROR_HANDLER, F_SYSTEM, link_reg)
    shutdown_the_system();
  /* USER CODE END Error_Handler_Debug */
}
```

Caution: The link register (LR) information logged by the *RTE_MSG1()* macro is intended for experienced programmers familiar with ARM Cortex assembly language. The address in the LR register is not necessarily the address from which the *Error_Handler()* function was called. It may be the address of the function that called the error reporting function. In fact, the linker can replace the function call with the Branch with Link

command with the Branch command. Also, in cases where there are multiple calls to the *Error_Handler()* function inside a function, the compiler can only insert one call to this function. The LR (link register) value must be logged at the beginning of an error/exception handler – as shown above. If another function is called before *__get_LR()*, the LR value will be overwritten. Such a solution is only possible on CPUs based on a ARM Cortex-M core or other cores that store the return address in a CPU register such as LR.

10.3 ARM Cortex-M4 / M7 Exception Handler Example

If a fatal error occurs while the debug probe is not connected, it is important to keep the most important information such as the CPU core dump and important global variables. The core dump can be logged using the *RTEdbg* library functions. The logged data allows you to analyze not only the error itself, but also the embedded system data and events that preceded the error, which may provide information about the cause of the error.

In the demo project folder is a common fault handler demo – see the file *fault_handler.c*. It logs the fault information to the circular buffer. The fault handler replaces the default infinite loop in the startup file *startup_stm32____.s*. This fault handler can be used for all STM32 devices with either Cortex-M4 or Cortex-M7 core. Before using it for other Cortex-M7 and Cortex-M4 based processor families, you should review the fault handler and customize it if necessary. See the following page for an example of format definitions used to decode exception data and an example of data that is logged after the *Hard fault* exception is triggered. The programmer can add logging of other information such as stack contents, important global variables and/or important RTOS variables, etc.

The handler logs the values of general and system CPU registers using the data logging functions of the RTEdbg library. Data logging is stopped by setting the message filter to zero. An automatic restart with the watchdog does not overwrite the data in the circular buffer unless the programmer requests it with the *rte_init()* or *rte_set_filter()* parameters (force buffer clear or force new filter value) – see Locking Data Logging After a Fatal Error. This data can be analyzed later when an automatic restart has already been performed and, for example, the host computer is connected to the embedded module. The hardware drivers and communication functions typically require a restart to reconnect to the host after a fatal error. The complete logged data (contents of the data logging structure) can also be written to internal or external flash memory.

Even a moderately sized data-logging buffer can log information about fatal errors and what the code was doing before the error occurred. This allows for efficient 'post-mortem' analysis. If a larger circular buffer is available, some additional information can be logged – from RTOS internals to task or system stack and important system variables. A complete data structure or buffer can be logged with a single call to *RTE_MSGN()*. Note that the maximum size of a single message is limited. The compile time parameter RTE_MAX_SUBPACKETS defines how much information can be logged with a single call.

For the IAR EWARM and Keil MDK, the exception handler entry code is implemented in the startup file (in assembly language). It calls *main_exception_handler()*.

The bus fault is triggered by the following line in *rtedbg_demo.c* and *simple_demo.c* to demonstrate the handler.

```
(void)*INVALID_ADDRESS;  // Read from invalid address
```

See the exception handler format definition below – data logged with the *RTE_MSGN()* macro.

Format Definition for Printing Exception Data for a Cortex-M7 CPU

```
// MSGN_FATAL_EXCEPTION
// "Register dump"
// "\n  R00:0x%08X, R01:0x%08X, R02:0x%08X, R03:0x%08X"
// "\n  R04:0x%08X, R05:0x%08X, R06:0x%08X, R07:0x%08X"
// "\n  R08:0x%08X, R09:0x%08X, R10:0x%08X, R11:0x%08X"
// "\n  R12:0x%08X,  SP:0x%08X,  LR:0x%08X,  PC:0x%08X"
// "\n  Status(xPSR = 0x%08X): ISR_No=%[-32:9]d, Flags: Q=%[+18:1]u, V=%[1]u, C=%[1]u, Z=%[1]u, N=
%[1]u"
// "\n  EXC_RETURN:0x%[8u]05X, BASEPRI:0x%[8u]05X, CONTROL:0x%[8u]05X"
// "\n  CFSR:0x%[+8:32u]08X"
/*-------------------------------------------------------------------------------*/
/* Decoding of CFSR register (Configurable fault status register) */
/* Individual bits represent information about usage, memory management and bus faults. */
/* The RTEmsg processed values starting from bit 0. */
/* Therefore we start with bit 0 of the CFSR register. */
// ":%[-32:8]{ |\n     Memory management fault}Y"
// "%[-8:1]{ |\n        The processor attempted an instruction fetch from a location that does not
permit execution}Y"
// "%[1]{ |\n        The processor attempted a load or store at a location that does not permit the
operation}Y"
// "%[+1:1]{ |\n        Unstack for an exception return has caused one or more access violations}Y"
// "%[1]{ |\n        Stacking for an exception entry has caused one or more access violations}Y"
// "%[1]{ |\n        A MemManage fault occurred during floating-point lazy state preservation}Y"
// "%[+1:1]{ |\n        Valid fault address.}Y"
// "%[8u]{ |\n     Bus fault}Y"
// "%[-8:1]{ |\n        The processor detects the instruction bus error on prefetching an
instruction, but it sets the IBUSERR flag to 1 only if it attempts to issue the faulting
instruction.}Y"
// "%[1]{ |\n        A data bus error has occurred, and the PC value stacked for the exception
return points to the instruction that caused the fault.}Y"
// "%[1]{ |\n        A data bus error has occurred, but the return address in the stack frame is not
related to the instruction that caused the error.}Y"
// "%[1]{ |\n        Unstack for an exception return has caused one or more BusFaults.}Y"
// "%[1]{ |\n        Stacking for an exception entry has caused one or more BusFaults}Y"
// "%[1]{ |\n        A bus fault occurred during floating-point lazy state preservation.}Y"
// "%[+1:1]{\n        Non valid fault address|\n        Valid fault address}Y"
// "%[16]{ |\n     Usage fault}Y"
// "%[-16:1]{ |\n        The processor has attempted to execute an undefined instruction.}Y"
// "%[1]{ |\n        The processor has attempted to execute an instruction that makes illegal use of
the EPSR}Y"
// "%[1]{ |\n        The processor has attempted an illegal load of EXC_RETURN to the PC, as a
result of an invalid context, or an invalid EXC_RETURN value.}Y"
// "%[1]{ |\n        No coprocessor}Y"
// "%[+4:1]{ |\n        The processor has made an unaligned memory access.}Y"
// "%[1]{ |\n        Divide by zero}Y"
/*-------------------------------------------------------------------------------*/
// "\n  HFSR:0x%[+6:32]08X, ABFSR:0x%08X, Offending address:0x%08X\n"
// <EXC_NAMES "  ICSR:0x%08X, VECTACTIVE: %[-32:9]u-%[-9:9]Y, VECTPENDING: %[+3:9]u-%[-9:9]Y"
```

Example of error data printed to the Main.log file after decoding the bus fault exception info (Cortex-M7)

```
N02804 603.797 MSGN_FATAL_EXCEPTION:   Register dump
  R00:0xFFFFFFF0, R01:0x00000012, R02:0x07000000, R03:0x0000012D
  R04:0x20000030, R05:0xFFFFFFE0, R06:0x00000064, R07:0x007594F5
  R08:0x00001FE9, R09:0xFA481A48, R10:0x0000000A, R11:0x20000000
  R12:0x00001335,  SP:0x2001FF28,  LR:0x00000675,  PC:0x00000684
  Status(xPSR = 0x21000000): ISR_No=0, Flags: Q=0, V=0, C=1, Z=0, N=0
  EXC_RETURN:0x000E9, BASEPRI:0x00000, CONTROL:0x00000
  CFSR:0x00008200:
     Bus fault
        A data bus error has occurred, and the PC value stacked for the exception return
           points to the instruction that caused the fault.
        Valid fault address
  HFSR:0x40000000, ABFSR:0x00000000, Offending address:0x07000000
  ICSR:0x00000803, VECTACTIVE: 3-HardFault, VECTPENDING: 0-Thread mode
```

Legend: **N02804** – message number, **603.797** – time [ms], **MSGN_FATAL_EXCEPTION** – format ID name

10.4 Overview of Existing Data Logging / Testing Solutions

The table below lists some of the data logging and analysis tools (mostly open source).

Segger SystemView	(commercial tool, tied to J-Link debug probes) Data logging via J-Link and SEGGER RTT. Optimized for event logging. Event viewer software included.
Keil MDK Event Recorder	(commercial tool) Optimized for event logging and less useful for general embedded system data logging. Includes support for MDK middleware, Keil RTX5.
Percepcio Tracealyzer	(commercial tool) Powerful data analysis tools. Optimized for event logging and less suited for real-time control system data logging and analysis.
National Instruments - Real-Time Trace Viewer	(commercial tool) The Real-Time Trace Viewer displays the time and event data, or trace session, on the host computer.
VxWorks System Viewer	(commercial tool) Tool used to collect instances of pre-configured times tamped events and display them in graphical or tabular format.
Linux Trace Toolkit (LTT, LTTng)	A set of tools designed to log program execution details from a patched Linux kernel and then perform various analyses on them.
Spdlog	Host side fast C++ logging library
RTEMS Trace	Designed for RTOS-based systems.
Microsoft TraceX	Windows-based system analysis tool for Microsoft Azure RTOS
QP/Spy Software Tracing	Software trace and test system designed specifically for embedded systems.
TRICE	Code instrumentation and data logging. Decoding based on the Go language.
Embedded Logger	Logging solution that moves fixed strings outside the final binary
µP7 library	Lightweight C library for sending logs to the host
NanoLog / **article**	Fast logging system for C++ with a simple printf-like API
MCUViewer	Open-source GUI debug tool for microcontrollers (Variable and Trace Viewer)
ESP IDF	ESP Application Level Tracing Library
Dictionary-based Logging	Zephyr Dictionary-based Logging
elog	Embedded Logger
COVESA DLT	Log and trace based on the protocol from the AUTOSAR Classic Platform
Traces	API tracing framework for Linux C/C++ applications

Most of these tools were designed for event logging and are less suitable for general embedded system data logging and are only partially configurable. Many toolkits allow printf-like data decoding to be performed on the host rather than in the embedded system. In most cases, however, the printf-like strings must be transferred to the host, which increases the use of the data-logging buffer and slows the data-logging process. Many of the logging functions require a lot of stack space. Some of the solutions use rather inefficient data encoding and therefore require larger data logging buffers. Most solutions only support basic print-style formatting.

11 Revision History

The following table shows the revision history for the documentation in this manual. See the *Revision_history.md* files in the source code folders for the *RTEdbg* data logging library and the *RTEmsg* message decoding application. Minor corrections are not part of this list.

Date	Description
2024-05-11	Initial version of the document
2024-10-02	- Corrected stack usage for RTE_MSG4 in Execution Times, Circular Buffer, and Stack Usage. - Updated tables and data in Resources Needed for Message Logging. - Corrected OUT_FILE() examples. - Added information about the RTEgdbData utility. - Added information about the generic CPU Drivers – see RTEdbg Library Structure. - Updated sections Using the RTEdbg Library on Multi-Core Processors, Suspicious Data in the Output File. - Added section Data logging in RTOS-based applications. - the CPU driver *rtedbg_cortex_m.h* driver replaced by *rtedbg_generic_irq_disable.h*
2024-11-15	- Added section Transfer recorded data to the host via a serial channel - Updated section Library Source Code and Hardware-Dependent Drivers - Updated section Using the RTEdbg Library on Multi-Core Processors
2024-12-16	- Updated section Overview of Existing Data Logging / Testing Solutions - Updated section RTEdbg Library Structure - Updated section Data logging in RTOS-based applications - Updated section Optimize Data Logging Code Size and Speed - Updated section List of RTEmsg Command Line Arguments (description of -ts=a;b argument) - Modified `%nH` definition in Additional Type Field Characters - Added section: Further reduction of the impact of instrumentation on code execution - Updated section: Stat_msgs_missing.txt