Minimally intrusive data logging and tracing toolkit

Introduction: The firmware should have data logging capabilities to enable real-time synchronous data and exception logging, just as we do for hardware. Unattended systems also need a logged history in case problems are discovered in the field (e.g., IoT). Real-time systems cannot be stopped because either the conditions would change after a restart (producing different results), or the loss of control could cause damage. New data is often generated much faster than anyone can observe it. Fast variable sampling by debuggers or custom applications is limited to low kHz sampling rates. Data is sampled asynchronously, and the tester cannot know the exact timing of the data – for example, whether the values of variables are the result of the same control cycle.

The new toolkit is suitable for all types of embedded systems, from complex hard real-time systems to non-real-time, resource-constrained systems where the programmer wishes to avoid printf-style testing due to slow execution, memory constraints, and re-entrancy issues. The solution is the equivalent of a re-entrant timestamped *fprintf()* running on the host instead of the embedded system. The *RTEdbg* toolkit provides minimally intrusive firmware instrumentation for testing, debugging, system analysis, etc.

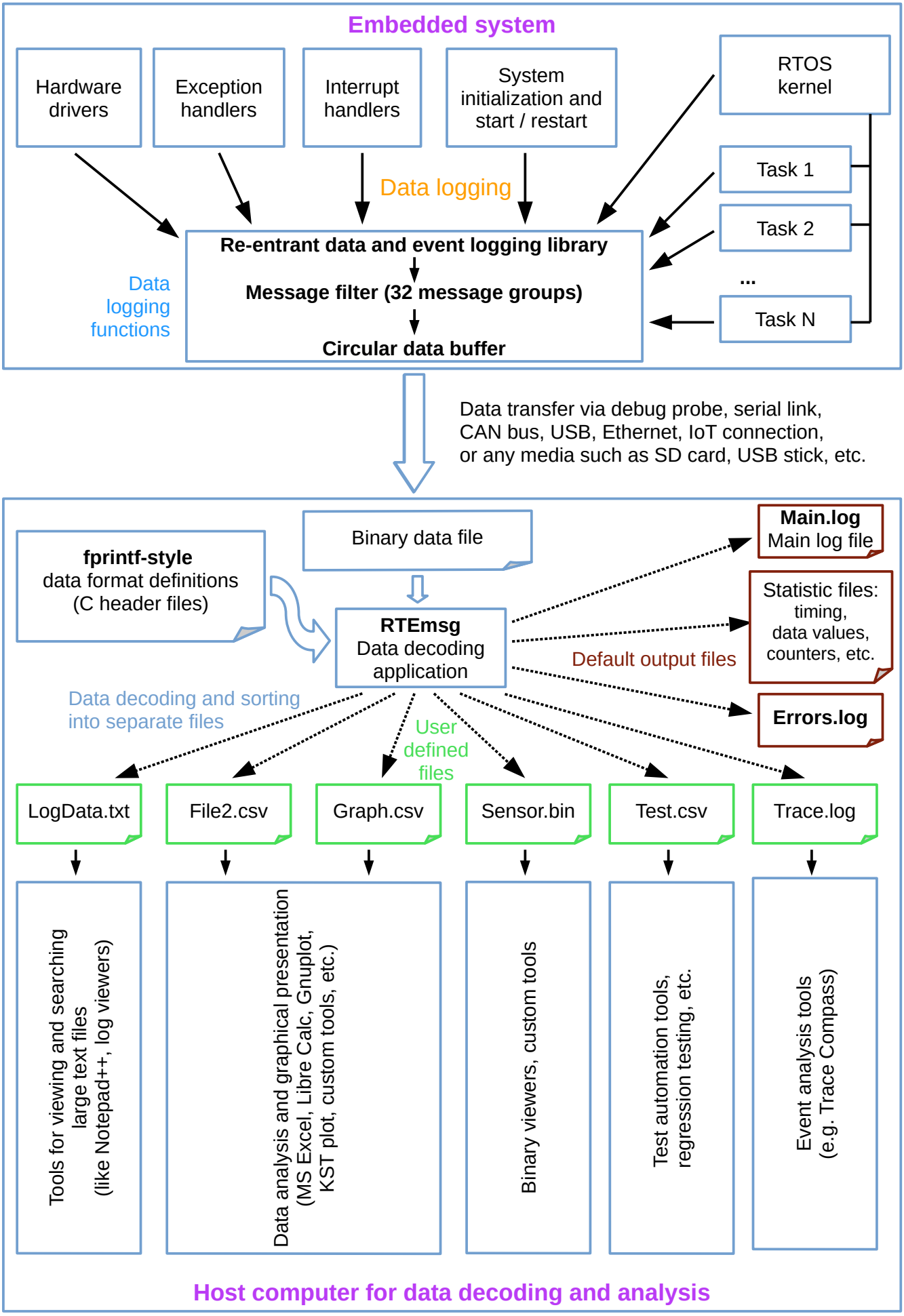
Note: Data captured by calling a function of the *RTEdbg* library is referred to as a **message** in this document.

Key features of the toolkit include:

- **Fast time-deterministic data logging** (*e.g. 27 CPU cycles to log an event on the Cortex M7*).
- **Small logging library footprint** (*typically from 0.3 kB to 1 kB – depending on compile-time options, e.g. code optimized for minimal size or stack usage, number of data logging functions used, etc.*).
- **Minimal stack usage** *(logging functions typically use 0 to 24 bytes on an ARM Cortex M CPU).*
- No data tagging or *printf* functions/strings are required in the firmware. Only an automatically assigned ID of an *fprintf-style* format definition (message ID) is logged along with the timestamp and data.
- A small number of functions and familiar printf string syntax make it **easy to learn and use**. Functions differ only in data size, not type.
- **Any type of data can be logged** – atomic or data structure, buffer, bit field, packed structure, etc.
- **Low circular buffer usage.** The minimum record size is 4 bytes – enough for a timestamp, message ID, and up to 8 bits of data. The maximum record size is set by the programmer.
- Powerful filtering lets you select the data you want to log on the fly (enable/disable 32 message groups).
- Single shot, snapshot, streaming, and post-mortem logging are possible.
- Custom triggers and message filter allow you to select data before or after a specific event. Logging can be limited to the most relevant data, making the most efficient use of the circular buffer.
- Designed to remain in production code (available if problems are discovered later).

How the tools work: The schematic on the following page shows how data logging can be integrated into embedded system firmware. Only a C source file with logging functions and some header files need to be added to the project. Any C or C++ code can be instrumented – including drivers, exception handlers, RTOS kernel. The logging functions store raw binary data in the circular buffer without time-consuming data tagging or encoding. Any interface or media can be used to transfer data to the host for decoding and analysis, since only the contents of a data structure need be transferred. The *RTEmsg* application performs the decoding of binary data. Logged messages are decoded (printed) to files using fprintf-style format definitions (defined in C header files). If errors are found in the binary file or in the format definition files, they are reported in the *Errors.log* file.

Real-time systems generate a flood of data, and it is important to sort the data before analyzing it to quickly identify potential problems (data extremes, errors, warnings). Programmers specify how the logged data is printed and into how many output files it is sorted (printed). Important messages or values can be written to more than one output file, each value in a different format. Existing data and event analysis tools can be used because flexible data formatting allows adaptation to the needs of those tools.



Schematic of the RTEdbg Toolkit

Examples of Data Collection Macros and Format Definitions

The following examples demonstrate some of the data logging and decoding (printing) capabilities of the new library. At the top of each table is the firmware as it would be implemented for an embedded system where the data would be printed to some sort of console such as serial port. The lower part of the table shows how this can be done in a minimally-intrusive way using the RTEmsg library macros. Data format definitions are shown in **green**. They only need to be defined for the host side and are located in the header file as comments starting with '//'. All data logged by the RTEdbg library macros includes timestamp information, whereas timestamps would have to be added manually to data printed with *printf()* or *fprintf()*. Only one initialization function needs to be called before logging begins.

Example 1: Print simple event information to the main log file

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

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

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

The **RTE_MSG0()** macro stores the MSG0_OVERTEMP message ID (format definition index) and timestamp in the circular buffer. Both values are encoded in a single 32-bit word and logged if the F_EVENTS message filter is enabled. The macro RTE_MSG0 packs both values into a single function parameter and calls a logging function with only one parameter. This further shortens the code because the logging functions have one less parameter. Both values (MSG0_OVERTEMP message ID and F_EVENTS filter) are automatically enumerated during the pre-build format definition syntax check with the *RTEmsg* application that is part of the toolkit. The enumeration is transparent to the programmer. The message ID is used on the host side to select the appropriate format definition with which to print (decode) the message data (print a text as shown in this simple example). By default text is printed to the *Main.log* file, along with a sequential message number and timestamp.

Example 2: Print the data logged by the firmware to the main log file

```
   struct batt_data {float voltage; float current;} battery;
   printf("Battery voltage %.2f, current: %.2f\n", battery.voltage, battery.current);
```

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

```
   RTE_MSG2(MSG2_BATT_DATA, F_BATTERY, battery.voltage, battery.current)
```

The **RTE_MSG2()** is another example of a data logging macro. It allows logging of two 32-bit values, which can be anything from a float, signed or unsigned integer, packed data or bit field. Other macros allow logging of one to four 32-bit values with a single call, logging of entire data structures, strings, memory dumps, etc. MSG2_BATT_DATA is the format definition ID and F_BATTERY is the filter number that can enable or disable logging of a group of messages that use the same filter number.

If programmers are logging only 32-bit values or data structures that contain only 32-bit values, then a simple way to log and decode data as shown above is sufficient. However, if they want to reduce circular buffer usage to store longer history, print data structures containing values of different sizes (or packed structures and bit fields), display timing information such as time difference between events, combine data from different messages into a CSV file, print indexed text messages instead of, for example, error codes, and much more, they need to become familiar with the extensions to the standard printf syntax. For example, it is possible to define which part of the message (how many bits of the captured data and starting from which data bit address) is used to print a value. In this way, it is possible to print all types of data, including bit fields and packed data structures. The length of a single printed value can range from 1 to 64 bits.

Integer or quasi-float values are often used in embedded systems. Values can be scaled before printing, as shown in the following example. This eliminates the need to scale the data into a user-friendly format in the embedded system prior to logging (saving program memory, time, and circular buffer space).

Example 3: Print the data to custom CSV file "battery data.csv" (including CSV header – first line)

```
   struct batt_data {uint16_t voltage; int16_t current;} batt;

   FILE * battery_data;
   battery_data = fopen("battery data.csv", "w");
   fprintf(battery_data, "Battery voltage [V];Current [A]\n");
   ...
   fprintf(battery_data, "%.2f;%.2f\n", batt.voltage * 0.01, batt.current * 0.01);
```

```
// OUT_FILE(BATT_DATA, "battery data.csv", "w", "Battery voltage [V];Current [A]\n")
// MSGN1_BATT_DATA >BATT_DATA "%[16u](*0.01).2f;%[16i](*0.01).2f\n"
```

```
   RTE_MSGN(MSGN1_BATT_DATA, F_BATTERY, &batt, sizeof(batt))
```

The data is printed to a custom CSV file with the header `"Battery voltage [V];Current [A]\n"` written after creation. The format definition extension example "%[16i](*0.01).2f" specifies that the 16-bit signed integer value is multiplied by 0.01 before being printed with the format string "%.2f". The RTE_MSGN() macro allows logging of whole structures or data fields, strings, buffers, memory dumps, etc. The MSGN1 naming convention in the message ID name **MSGN1**_BATT_DATA tells the decoding application that the RTE_**MSGN** macro is used for data logging and that the data size is **one** 32-bit data word.

Example 4: ARM Cortex M4/M7 exception handler

It is easy to add data logging to the firmware in the event of a firmware crash. Instead of just an error code like "system error #3" or a blinking LED, we can have the error information we need to find the cause. A full-featured and very lightweight ARM Cortex M4/M7 exception logging function is included in the toolkit demo code. The size of the function is only about 200 bytes and the size of the message in the circular buffer is 116 bytes. On a Cortex-M7 processor, preparing and logging raw exception data takes only about 360 CPU cycles. Logging the same information in printf mode would require several kilobytes of program memory and a lot of CPU time. This function prepares data in a structure and logs it with a call to the RTE_MSGN() macro (used to log complete data structures or buffers) - see below.

```
    RTE_MSGN(MSGN23_FATAL_EXCEPTION, F_SYSTEM, &g_exception, sizeof(g_exception))
```

Decoding this message using the format definition prints exception information in a user-friendly manner – see the example below. The "Status" register is first printed as hex, and then the "Flags" sub-value is printed as binary (demonstrating single bit value printing). The "CFSR" is also printed first as hex and then as a descriptive text message (demonstrating the use of the indexed text printing format extension). The same value can have multiple print outputs. The MSGN23_FATAL_EXCEPTION is a fairly complex format definition, but it only needs to be created and tested once and can be used in any number of projects. Programmers can easily add logging of other information, such as which RTOS task was running, important global variables, stack dump, CPU floating point registers, etc.

```
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
```

=> **N02804** – consecutive message number, **603.797** – time [ms], **MSGN_FATAL_EXCEPTION** – format ID

Conclusion

The library of data logging functions enables minimally intrusive instrumentation of embedded systems. It is optimized for 32-bit devices. Static RAM usage is 40 bytes for the data structure header plus space for the circular data buffer. Data logging can be paused by setting the message filter to zero without affecting code execution. Data can be logged, transferred to the host, decoded, and displayed/analyzed while the firmware is running normally. Each logged message is also an event with a timestamp for timing and performance analysis. The library is designed to be portable. It is ready for use on devices with an ARM Cortex-M core and other little-endian 32-bit devices. Two new generic CPU core drivers have been added to the library. To port *RTEdbg* to a new family of processors, use one of the two generic drivers and add a timestamp timer driver that matches your hardware.

The *RTEdbg* library can be used to capture various types of data, such as:

- **non-periodic data values and events**,
- **periodic data**, such as control loop data (data from each control loop can be written to a separate CSV file; data from multiple recorded messages can be combined into a single CSV file)
- **events**, such as RTOS tracing or application-specific events
- **exceptions –** any exception data can be printed (see the example above).

The universality of the solution eliminates the need to use two or three different solutions for logging different types of data.

There is no data processing or tagging in the embedded system during data logging, so there is no run-time overhead and no wasted program memory (and circular data buffer) for the data tagging or printf-like functions and strings. This results in fast code execution and low stack usage. Typically, 35 CPU cycles and 4 bytes of stack are required to log an event (Example 1) using the *RTEdbg* library function on a device with a Cortex-M4 core and only 20 bytes of stack when logging full data structures or buffers with macro RTE_MSGN. According to the Segger SystemView documentation, nearly 200 CPU cycles and a maximum of 150 to 510 bytes of stack (depending on the configuration) are required for event generation and encoding under the same conditions. Solutions such as SystemView or Tracealyzer are primarily designed for event analysis and are less suitable for logging application-specific data. Both tools are compatible with a wide range of processors, while the *RTEdbg* library has been optimized for 32-bit devices.

Code instrumentation requires additional effort from the programmer, and the learning curve should not be too steep. The number of functions and macros in the *RTEdbg* toolkit is small (see list on next page), and the syntax of the *printf* string is already familiar to programmers. The new toolkit complements existing debugging tools by overcoming several problems of existing code instrumentation solutions, such as: slow data logging, printf-style strings are part of the firmware and must be copied to the circular buffer, large memory footprint, high stack usage, missing or inflexible message filtering, inflexible customization to project requirements, flood of data in the log file, steep learning curve, etc.

The toolkit does not require any special hardware such as a dedicated debug probe. Decoding binary data with the *RTEmsg* application is very fast because the format definitions are precompiled and not interpreted. This tool not only decodes and sorts the recorded data, but also generates statistical reports of the decoded values and times, allowing testers to quickly identify extreme values. Format definitions can be prepared in any international character set, as UTF-8 encoding is transparent to the *fprintf()* function used to decode the data.

The toolkit is also useful for reverse engineering poorly documented code and analyzing complex, difficult-to-reproduce problems. Its low stack requirements virtually eliminate the possibility of stack overflows when instrumenting code (no extra stack space required for each RTOS task and instrumented driver or handler).

The open source toolkit is available on github.com/RTEdbg/RTEdbg. It includes a manual and demo projects and is licensed under a permissive MIT license.

Branko Premzel, PhD, 2024-11-25

Appendix – list of data logging functions and macros

The listed macros and functions can be used to log any type of data/event – see also note 1. This table is not intended to be an exact representation, but only to illustrate the simplicity of the set of functions and macros.

Data Logging Macros

void **RTE_MSG0**(fmt_id, filter_no) // Event logging (no data)
void **RTE_MSG1**(fmt_id, filter_no, data1) // 1 × 32 bit data logging
void **RTE_MSG2**(fmt_id, filter_no, data1, data2) // 2 × 32 bit data logging
void **RTE_MSG3**(fmt_id, filter_no, data1, data2, data3) // 3 × 32 bit data logging
void **RTE_MSG4**(fmt_id, filter_no, data1, data2, data3, data4) // 4 × 32 bit data logging
void **RTE_MSGN**(fmt_id, filter_no, data_address, data_length) // Log a complete structure or buffer
void **RTE_MSGX**(fmt_id, filter_no, data_address, data_length) // Special version of the RTE_MSGN
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 – automatically enumerated
uint32_t **filter_no** – Number of the filter bit that enables the message (0 … 31) – automatically enumerated
rte_any32_t **data1 … data4** – Any 32-bit data (from char to float, packed values, etc.)
void * **data_address** – Pointer to the data that is to be stored in the circular buffer
Complete data structures, strings, or buffers can be logged and decoded.
uint32_t **data_length** – length of data in bytes (size of the logged variable, data structure, string, or buffer)

Various Functions

void **rte_init**(initial_filter_value, init_mode);
Must be called before logging begins to initialize the data logging structure and timestamp timer.

void **rte_long_timestamp**(void);
Full time synchronization with host – write the message with a long timestamp value to the circular buffer. Each message contains a timestamp with a minimum length of 15 bits. This message records an additional 32 bits of time stamp (total of at least 47 bits).

void **rte_timestamp_frequency**(uint32_t frequency);
Should be called after changing the timestamp timer frequency during streaming data transfer to the host.

Message Filter Manipulation Functions (not necessary if the filter is manipulated over the debug probe)

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);*

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.

Notes:

1. The table contains the macros and functions that a programmer needs to know. Extended versions of the first five macros in the table above are not included in this list for simplicity. They allow additional short data to be packed into the same message size – for example, up to an additional 8 bits in a 32-bit long message. **Example:** For an event recorded with the RTEMSG0 macro, the size of the message in the circular buffer is 32 bits; with the RTE_EXT_MSG0(short_data) macro version, up to 8 bits of additional data can be recorded in the same 32-bit message.
2. The macro data parameters must be in the same order in the macro as they are in the printf-style string. The same is true for variables in a data structure if a complete data structure is logged.