Time in Embedded Analytics Systems

AN1001 v1.1

Tessent Embedded Analytics

23 August 2024

SIEMENS

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Contents

Revision history

Revision	Details	
v1.1	Initial release	

Related documents

Summary

Tessent Embedded Analytics modules can report the time a downstream message is created and the time this message arrives at the time-enabled Message Engine. Software can use these values to create a software timestamp for the message which can be used to reconstruct activity within the Embedded Analytics subsystem.

This application note is intended for Embedded Software Engineers who want to reconstruct time using the Embedded Analytic timestamps embedded in downstream messages.

Overview

Message Engine time messages

Message Engines have a free running 32-bit counter that is based on the Embedded Analytics clock whose frequency determines the available time resolution in the analytics subsystem. As each subsystem can have many Message Engines, the time counter should be enabled in one as a reference for the whole system, typically the one at the bottom of the hierarchy.

The time-enabled Message Engine reports its 32-bit counter value in downstream *time* messages, which can be triggered for one of three reasons (which is reported in the *time* message):

- *Periodic* – sent when the Interval Timer reaches the user-defined *timer_interval* cycles. Periodic time messages are only output if the Interval Timer is enabled in the time-enabled Message Engine.
- *Overflow* – the time-enabled Message Engine keeps track of downstream messages from its lower ports and if it hasn't seen a downstream message from a port for half an epoch (0x8000 clock cycles) when it receives a new downstream message with a time tag, it sends out an *overflow* time message immediately before dispatching the new message over the lower port to a communicator.
- *Reset* – sent when the time-enabled Message Engine time counter is reset (by software or a hardware generated real-time event). The time reported in the output *time* message is the value of the counter immediately before it is reset.

The value reported in the *time* message is the absolute time in EA clock cycles from the previous time reset event.

Regardless of what triggered it, the Message Engine always generates three *time* messages with the same payload but on different flows (0, 1, 2), to make sure that each communicator into which at least one flow is routed will always get the *time* message. If more than one flow is routed to a communicator, it will receive the time message for each flow.

Analytic module time tags

Analytic modules have their own 16-bit counters that are also clocked in the Embedded Analytics domain. These counter values (which are independent of the time-enabled Message Engine 32-bit counter and the counters in other modules) can be included as time tags when a downstream message is created. For example, to include time tags in ***te_inst*** downstream messages from the Enhanced Trace Encoder the **timestamp** field in the ***set_trace*** message needs to be set. The time period taken for the 16-bit counter to wrap is known as an *epoch*.

Triggering a reset event

All 32-bit Message Engine counters and 16-bit module counters can be reset at the same time by sending an Embedded Analytics real-time reset event (0x1).

The 16-bit time counter associated with each module can also get out of sync when system time is reset using real-time event 0x1 as event propagation can take time, particularly in large hierarchical systems. To minimize unnecessary complications, the reset real-time event 0x1 should always be triggered by the time-enabled Message Engine; triggering the event from another module may cause additional overhead that is more difficult to quantify.

All counters start at 0 at a hardware reset and there is no internal support for a real-time clock.

Creating software timestamps

Software can use a combination of the upper bits of a 32-bit time message and the module's 16-bit time tag to construct a 32-bit timestamp to indicate when a downstream message is output from the Message Engine lower port. Although there may be minor delays between when a downstream message is created and when the Message Engine receives it due to internal latencies, the concatenated timestamp can be used to provide an understanding of activity within a system.

Periodic and overflow time messages

In Figure 1 a periodic time message with value 0x1000_000 is output followed by a burst of downstream messages that have been dispatched immediately after each was created. The time tag in the first message is less than 0x8000 cycles (half an epoch) and each subsequent message follows less than half an epoch between each other.

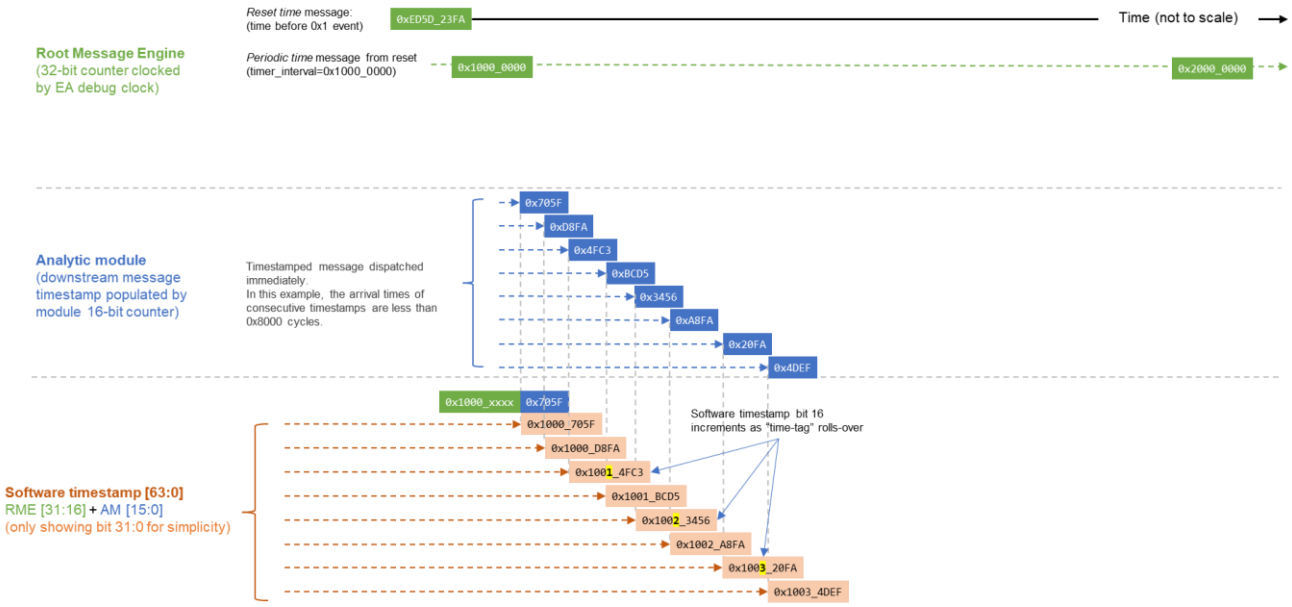

Figure 1: Software timestamps based on periodic messages

- As the first downstream message arrives less than 0x8000 cycles after the periodic time message, software can replace bits [15:0] with the 16-bit time tag from the downstream message (0x705F) to create the message timestamp 0x1000_705F.
- The second downstream message arrives less than 0x8000 cycles after the first, so the time is still in the original epoch. Software can create a timestamp from the original periodic time message and the downstream message time tag (0xDF8A) to create a timestamp 0x1000_DF8A.
- By the time the third message (0x4FC3) arrives, software detects that time tag has rolled over, increments the value of the upper bits of the periodic time message to 0x1001_xxxx and then replaces bits [15:0] with the time tag to create a timestamp 0x1001_4FC3.
- As additional messages in the burst arrive at the Message Engine, software can increment the value of the periodic time message bits [31:16] and calculate timestamps correctly providing the delta between each time tag is less than 0x8000.

Note that the burst of messages can come from one or more modules. The Message Engine is only interested in whether the message has a time tag and its destination.

In Figure 2 the initial burst of messages has finished and a new message with time tag 0x31FD arrives at the Message Engine.

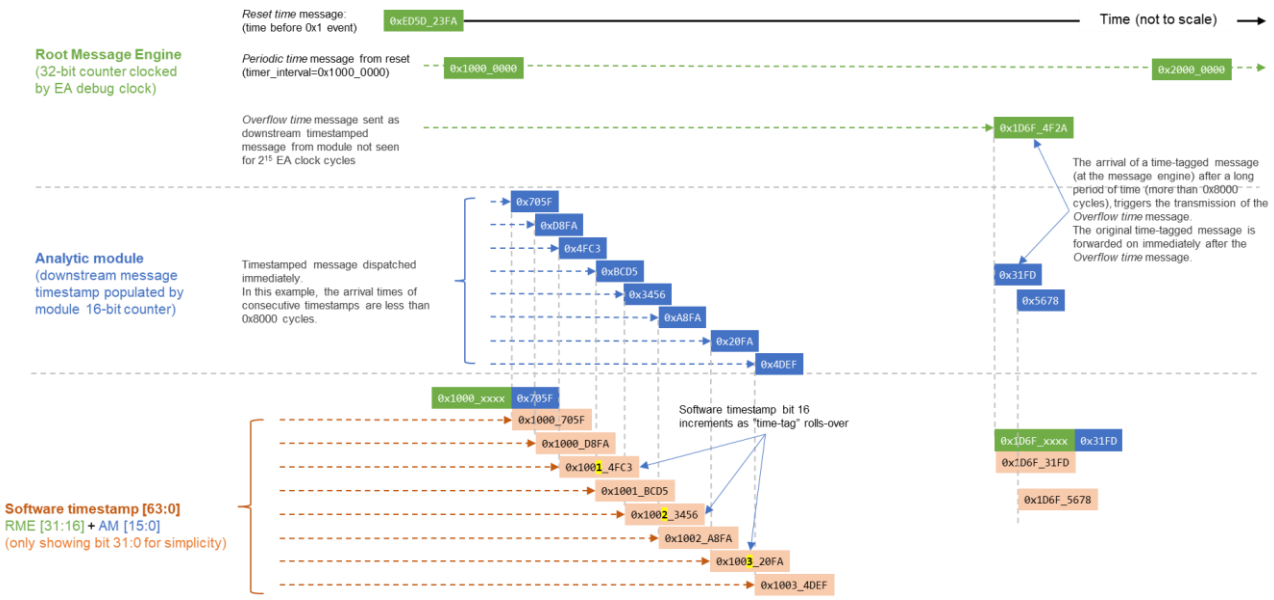

Figure 2: Software timestamps based on overflow messages

- As the Message Engine has not seen a downstream message with a time tag destined for the same lower port for more than half an epoch (0x8000 cycles),
    - it triggers an overflow time message that contains the number of cycles since the interval timer last rolled over (0x1D6F_4F2A);
    - forwards the downstream message immediately to the lower port after the overflow time message has been output;
    - software can then use the overflow time message and time tag to create the timestamp 0x1D6F_31FD.
- Software can create timestamps for further downstream messages using the periodic time value providing they appear no more than 0x8000 cycles after the 0x31FD message – see Figure 1.

Out of order timestamps

While messages with time tags often appear in bursts, they can be out of order. Different types of messages from the same analytic module (for example, match or counter messages from a Bus Monitor) are independently buffered and so have different latencies between creation and dispatch. Messages of the same type but from different modules may be out of order because of differences in buffer depth or fullness resulting in differing latencies between creation and dispatch.

The following pseudo-code shows how to reconstruct message time from a software-maintained 64-bit time and the message 16-bit time tag. The code assumes that the time that elapsed between creating the time tag and the arrival of the downstream message at the Message Engine is not long, and that time counters in all modules are in sync (there is no need for time counter difference compensation).

1. Startup actions for time application.
   - Define real time as a global 64-bit time `time64`.
   - Enable the Interval Timer and set `period=0x80000000`.
   - Capture real-time at the same time as the system is reset.
2. Receive the time tagged message.
   - Update `time64` with the 32-bit value from the Message Engine. If this is a reset event message the current time can be set to `0`; if it's a periodic or overflow message the 32-bit timer wrap must be accounted for:

```
update_time64(time32_new, reason):
if reason == time_reset:

    time64 = 0
else:
    if time32_new < time64[31:0]:
        time64[63:32] += 1
time64[31:0] = time32_new
```

3. Reconstruct the message time.

```
timestamp16_to_time64(timestamp16):
    result64 = time64
    # Determine distance from ``time64[15:0]`` to ``timestamp16``
    time16 = time64[15:0]
    delta16 = (timestamp16 >= time16 ? 0 : 0x10000) + timestamp16 - time16;
```

   The message timestamp may have been captured before or after time64 as maintained by the time application. If it was captured later, the overflow mechanism makes sure that `delta16` is smaller than `0x8000` (a *time* message would have been sent in the meantime) by treating it as being between -0x7FFF and 0x0000.

```
timestamp_earlier = delta16 > 0x8000
if timestamp_earlier:

    # If timestamp is earlier and timestamp is larger number
    # 16-bit wrap occurred: decrement result64[63:16]
    if timestamp16 > time16: result64[63:16] -= 1
else:
    # If timestamp is later and timestamp is smaller number
    # 16-bit wrap occurred: increment result64[63:16]
    if timestamp16 < time16: result64[63:16] += 1

# Replace the lowest 16 bits of the result with the received timestamp
result64[15:0] = timestamp16

# If the result indicated later time than current time64, update time64
if result64 > time64: time64 = result64
return result64
```

Table 1 and Figure 3 show a set of output values calculated using this time reconstruction algorithm:

	time64	timestamp16 (incoming)	time16 (calculated)	delta16 (calculated)	timestamp earlier?	wrap?	result64	update time64?	new time64
1	0x0002D000	0xF000	0xD000	0x2000	No	0	0x0002F000	yes	0x0002F000
2	0x0002F000	0x1000	0xF000	0x2000	No	1	0x00031000	yes	0x00031000
3	0x00031000	0xE000	0x1000	0xD000	Yes	-1	0x0002E000	no	0x00031000
4	0x00031000	0x0000	0x1000	0xF000	Yes	0	0x00030000	no	0x00031000
5	0x00031000	0xFF00	0x1000	0xEF00	Yes	-1	0x0002FF00	no	0x00031000
6	0x00031000	0x3000	0x1000	0x2000	No	0	0x00033000	yes	0x00033000
7	0x00033000	0x5000	0x3000	0x2000	No	0	0x00035000	yes	0x00035000
8	0x00035000	0x7000	0x5000	0x2000	No	0	0x00037000	yes	0x00037000
9	0x00037000	0x8000	0x7000	0x1000	No	0	0x00038000	yes	0x00038000
10	0x00038000	0x6500	0x8000	0xE500	Yes	0	0x00036500	no	0x00038000
11	0x00038000	0xFF00	0x8000	0x7F00	No	0	0x0003FF00	yes	0x0003FF00
12	0x0003FF00	0xFF00	0xFF00	0x0000	No	0	0x0003FF00	no	0x0003FF00

Table 1: Reconstructed 64-bit time from system time and message timestamps

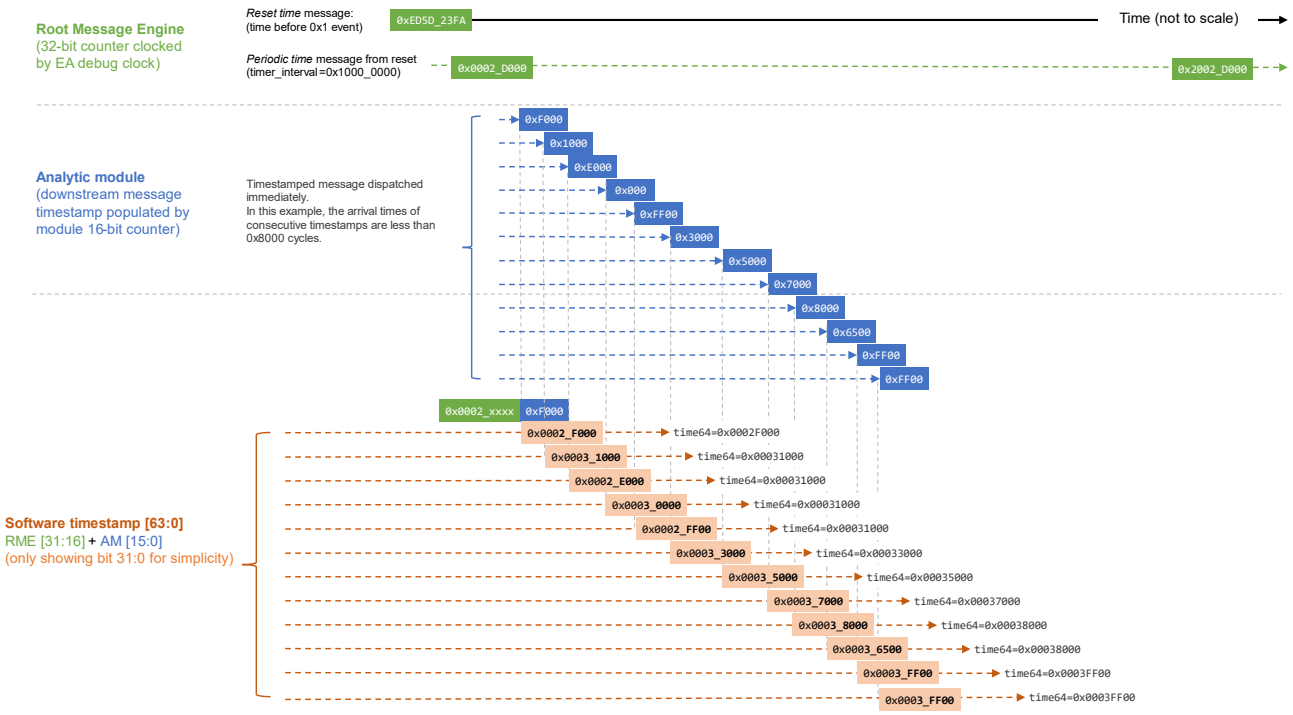

Figure 3: Reconstructed 64-bit time from out-of-order timestamps

Delayed messages

If there are more than 2^{15} cycles between when a message is created and when it arrives at the Message Engine, the message may appear from the wrong epoch. This uncommon behaviour can be found with messages that use features like the *trace to* mode with a filter - if the filter matching condition is very occasional, a message may not be dispatched for hours or days after the message has been created. It may also be evident if there is extreme backpressure in the subsystem limiting the rate at which messages can flow to the Message Engine. In this case the software lacks the information (number of times the overflow time messages have wrapped) necessary to identify when the message arrived at the lower port in order to create an accurate timestamp.

Contact Tessent Embedded Analytics for further information if this arises.