Speed debugging, gain system behaviour insights with trace and visualizations

Dr Johan Kraft, CEO, Percepio AB - August 27, 2015

Anyone involved with software development will have most likely heard (and perhaps even said) the phrase ‘it’s not a bug, it’s a feature’ at some point, and while its origins remain a mystery, its sentiment is clear; it’s a bug that we haven’t seen before.

Intermittent ‘features’ in an embedded system can originate in either the software or hardware domain, often only evident when certain conditions collide in both. In the hardware domains, the timings involved may be parts of a nanosecond and where the logic is accessible, such as an address line or data bus — there exist instruments that can operate at high sample rates, allowing engineers to visualise and verify such ‘glitches’. In the software domain finding glitches becomes much more challenging.

Sequential Processing

While parallel processing is being rapidly adopted across all applications, single-processor systems remain common in embedded systems, thanks partly to the continued increases in the performance of microcontroller cores. Embedded MCUs are now capable of executing a range of increasingly sophisticated Real-Time Operating Systems, often including the ability to run various communication protocols for both wired and wireless interfaces.

Whether in a single- or multi-processing system, combining these tasks with the embedded system’s main application, written by the engineering team, can make embedded software builds large, complex and difficult to fault-find, particularly when visibility into the code’s execution is limited. It can also lead to the dreaded intermittent fault which, if part of the system’s operation is ‘hidden’, can make solving them even more challenging.

A typical example may be an unexplained delay in a scheduled task. Of course, an RTOS is intended to guarantee specific tasks happen at specific times but this can be dependent on the task’s priority and what else may be happening at any time. In one real-world example, where a sensor needed to be sampled every 5ms, it was found that occasionally the delay between samples reached 6.5ms, with no simple explanation as to the cause. In another example, a customer reported that their system exhibited random resets. The suspected cause was that the watchdog was expiring before it was serviced, but how could this be checked? In yet another example, a system running a TCP/IP stack showed slower response times to network requests after minor changes in the code, for no obvious reason.

These are typical examples of how embedded systems running complex software can behave in unforeseen ways, leaving engineering teams speculating on the causes and attempting to solve the problems with only empirical results from which to assess their efforts. In the case of intermittent

faults or system performance fluctuations, this is clearly an inefficient and unreliable development method.

Trace Tools

The use of logging software embedded in a build in order to record certain actions isn't new, of course, but it can offer a significantly improved level of visibility into a system. This is especially true if the logging software understands the general meaning of kernel calls, for instance locking a Mutex or writing to a message queue. However, while the data generated by such trace software is undoubtedly valuable, exploiting that value isn't always simple. Analysing trace data and visually rendering it in various ways, ranging from an event list to high-level dependency graphs and advanced statistics, is the key.

One of the main ways to view trace data is a timeline visualizing the execution of tasks/threads and interrupts along with other logged events, such as system calls. In the example from Percepio's Tracealyzer tool shown in Figure 1, trace data are displayed as annotations in a vertical timeline, using horizontal colour-coded text labels. Other timeline views are also possible, such as a horizontal orientation with multiple views combined on a common timeline.

While much important trace data is created by the operating system's kernel tracking system events, it's also helpful if developers can extend the tracing with user-defined events. This provides the ability to have any event or data in a user's application be logged along with system events such as function calls, interrupts, and the like. Creating user-defined events is similar to calling the classic 'printf' C library function but much faster. The actual formatting is handled in the host-side application and the tracing can therefore even be used in time-critical code such as interrupt handlers. And, of course, user-defined events can also be correlated with other kernel-based events in a timeline.

Example Insights

Some examples of different visualization views shows how different views can speed debugging. The vertical timeline view, for instance, can help identify the cause of problems in task execution timing. In the situation explored in Figure 1(a), a SamplerTask scheduled for 5ms intervals was intermittently being inexplicably delayed. Tracealyzer was used to graphically show the task in question, time-correlated with other tasks. By invoking an exploded view (Figure 1(a)) on the task of interest when a delay occurred, the developer found that a second, lower priority ControlTask was incorrectly blocking the primary, scheduled task from executing.

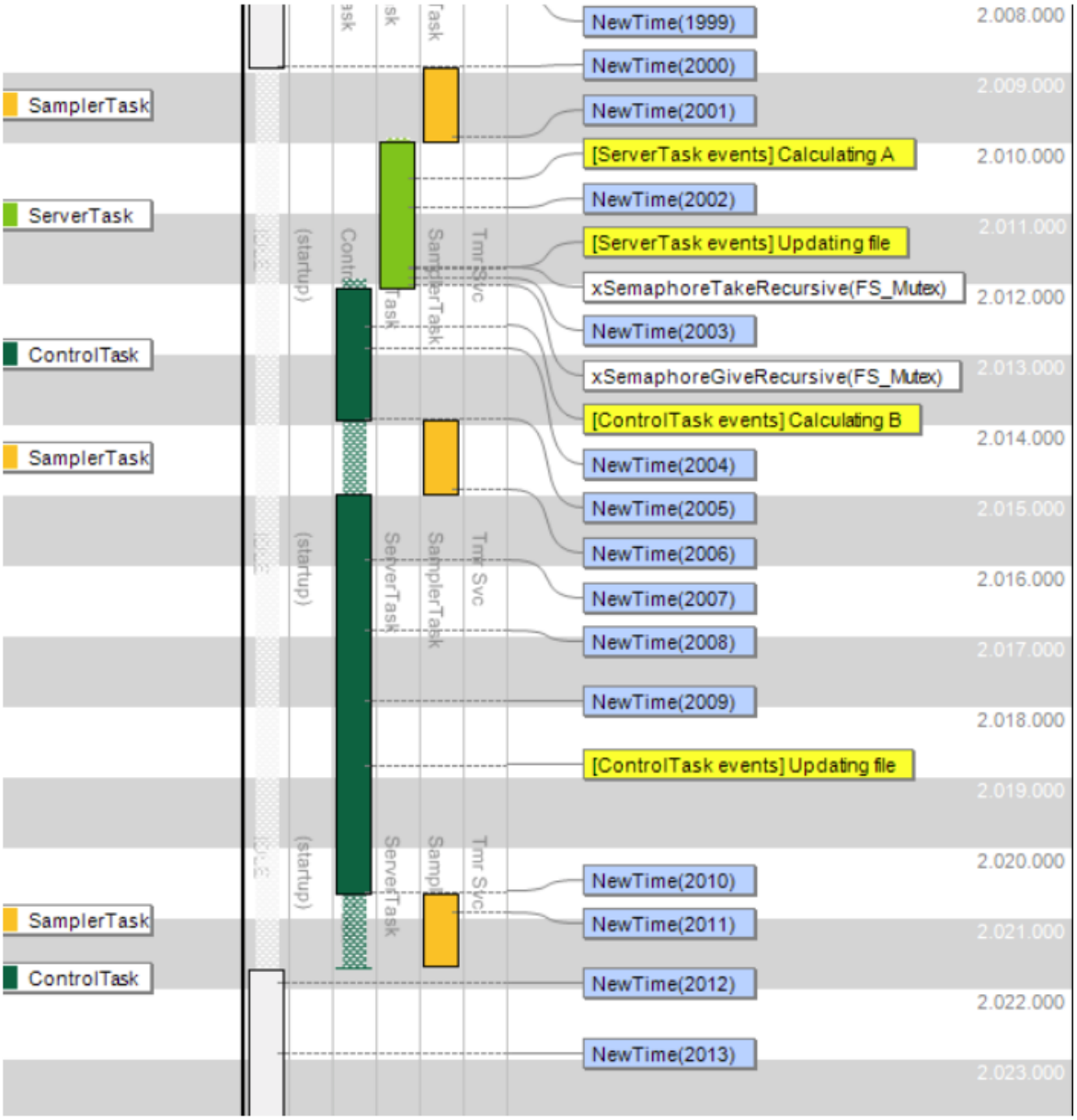

Figure 1(a): It appears that the ControlTask may be disabling interrupts.

Examination of the code revealed that the second task (Calculating B) was disabling interrupts to protect a critical section (Updating file) unrelated to the primary, scheduled SamplerTask. Disabling interrupts blocked the operating system scheduling, causing the delay. After the developer changed the second task to use a Mutex to protect the file update instead disabling interrupts, the primary task was able to meet its timing requirements. Figure 1(b) confirms that SamplerTask is now occurring every 5ms as intended.

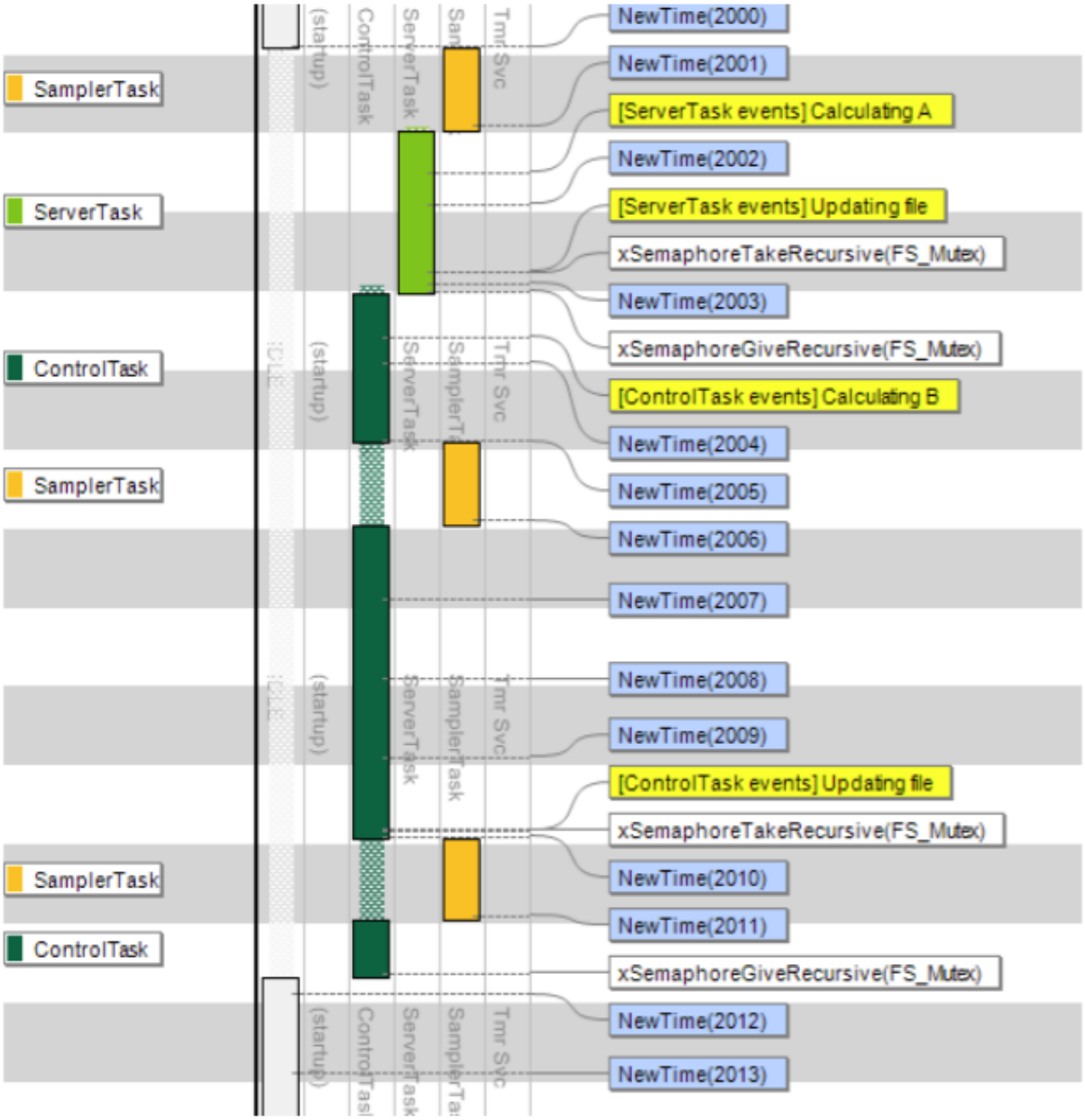

Figure 1(b): Changing the way ControlTask protects a critical section lets SamplerTask run as intended

In the second example, a user-defined event along with multiple views of the trace data helped identify the cause of an expiring watchdog timer. The developer created events to not only record when the Watchdog was reset or when it expired, but also to log the remaining Watchdog timer value. This event thus traced the time left in the Watchdog timer when it got reset, and could be used to identify instances when the watchdog timer came perilously close to expiring (too little time remaining), so they could be examined more closely.

By inspecting the logged system calls the developer found that the watchdog resetting task not only reset the Watchdog timer; it also posted a message to another task using a (fixed-size) message queue. Timeline analysis revealed that the Watchdog expiration seemed to occur while the message posting blocked the Watchdog task.

Once the situation was identified, the question then became ‘why is the message posting taking so long?' By visually exploring the operations on this message queue using a kernel object history it

became clear to the developer that the message queue sometimes became full.

To learn why the queue became full, the developer correlated a view of the CPU load against how the Watchdog timer margin varied over time, as shown in Figure 2. This correlation revealed that Fixed Priority Scheduling was allowing a medium-priority task (ServerTask) to use so much CPU time that the message queue wasn't always being read. Instead, it became full, leading to the Watchdog expiring. The solution was in this case to modify the task priorities.

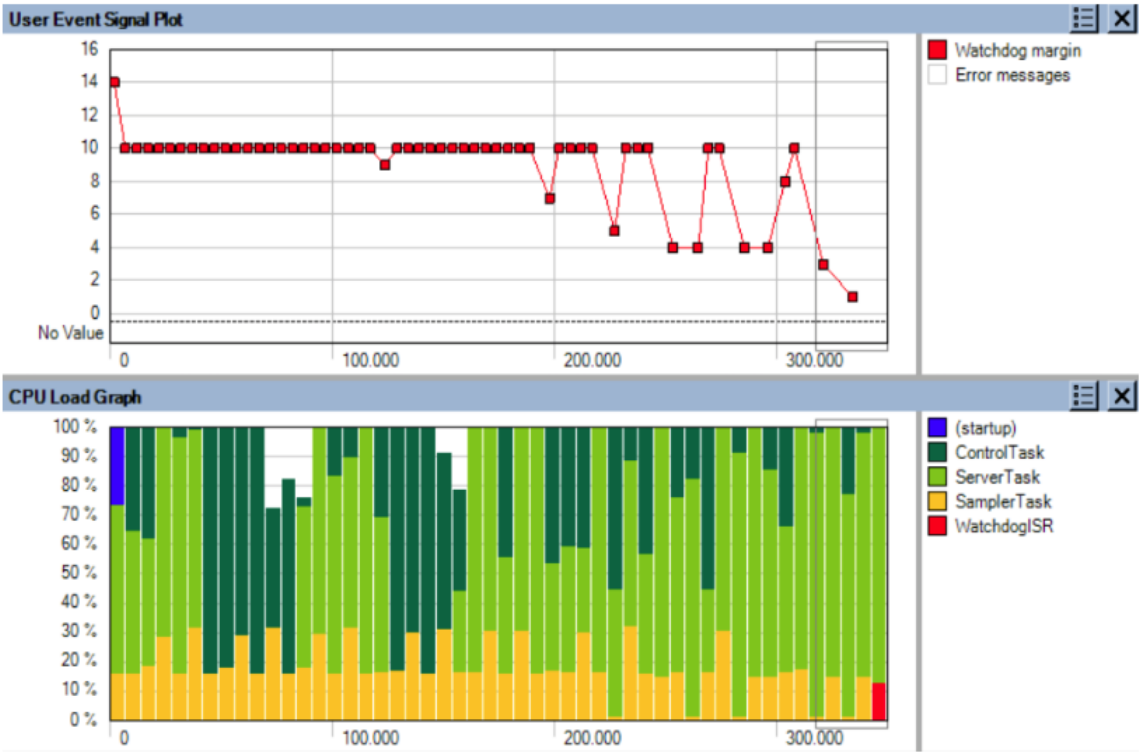

Figure 2: The CPU Load graph, correlated to the Watchdog Timer User Event, gives valuable insights.

In the last example, a software modification caused increased response time to network requests. By looking at a communications flow view (Figure 3) the developer found that one particular task — Logger — was receiving frequent but single messages from a variety of sensors containing diagnostics data to be written to a device file system. Writing the data upon receipt of a message caused a context switch. By modifying the task priorities, the developer enabled buffering of the messages until the network request had finished so they could thereafter be handled in a batch. This way, the number of context-switches during the handling of network requests was drastically reduced, thereby improving overall system responsiveness.

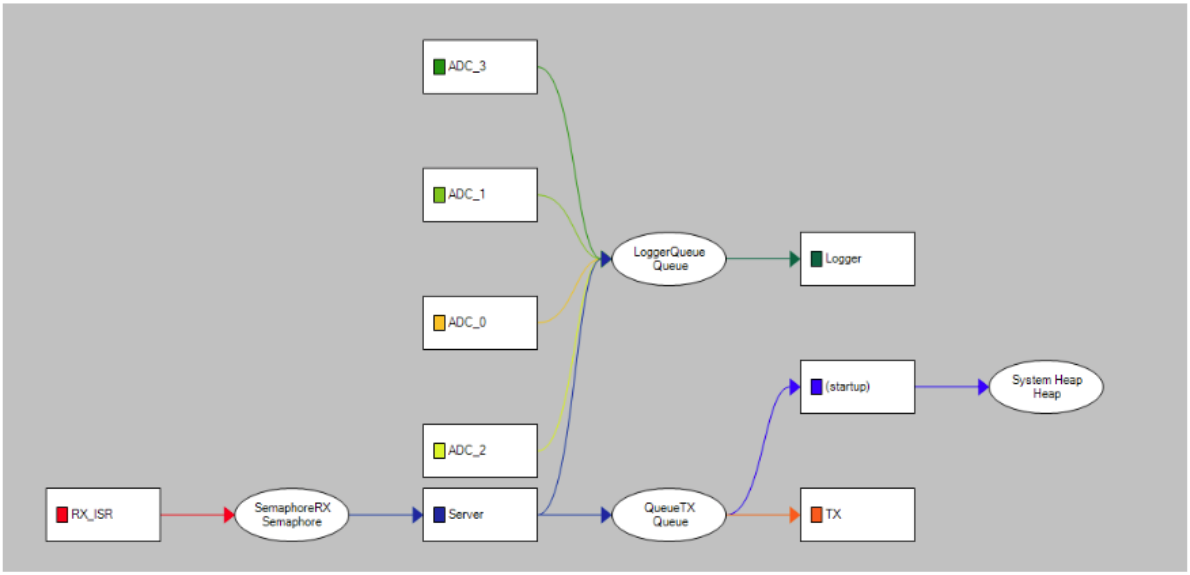

Figure 3: The Communication Flow reveals five tasks sending messages to Logger.

Conclusion

The complexity of embedded software is increasing rapidly, creating demand for improved development tools. While runtime data can be recorded in various ways, understanding its meaning isn't a simple process. Through the use of innovative data visualisation, from tools such as Tracealyzer, understanding becomes much easier. The logging and visualization can not only speed debugging during development, it can help identify field problems. By embedding trace tools in production code, companies can gather invaluable data about real systems running in the field. Embedded systems thus need no longer be a 'black box', leaving engineers to suppose what may be happening. Powerful visualisation tools now available turn that black box into an open box.