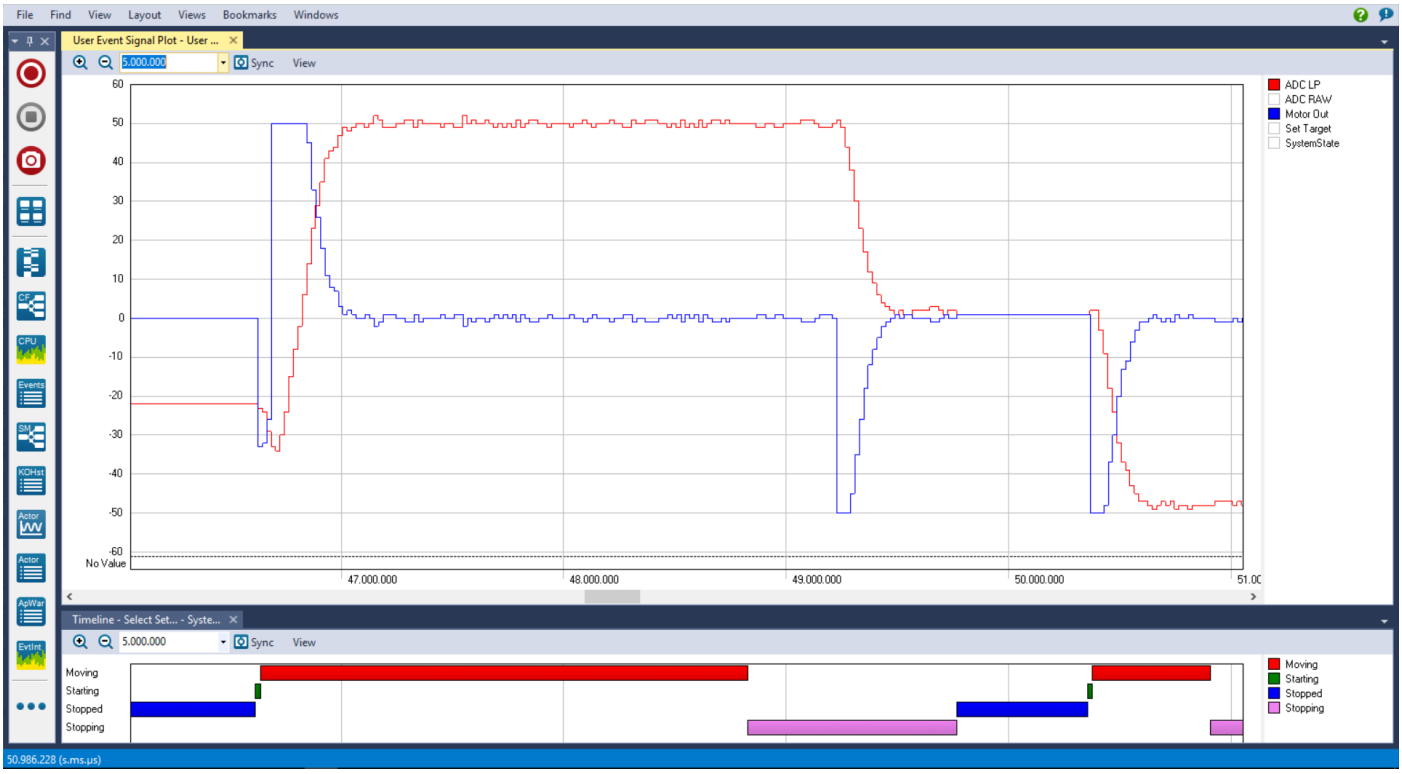
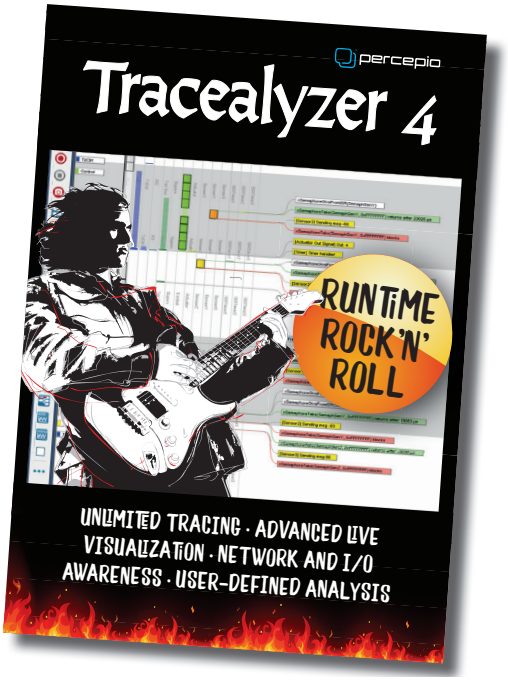


Tracealyzer Hands On

Getting the most out of Tracealyzer

percepio®



Download Tracealyzer at percepio.com/download
Free time-limited evaluation license is included

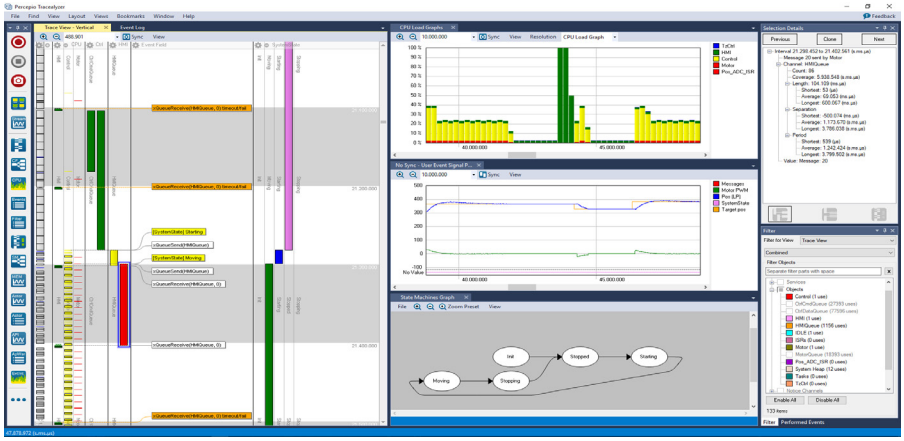
Tracealyzer Hands On is a series of blog posts with handy tips how you can get more out of Percepio Tracealyzer. This brochure contains a selection of our Hands On posts; the complete set is available at percepio.com/rtos-debug-portal/

Percepio AB, Västerås, Sweden
https://percepio.com

Tracealyzer – more than a debugging tool

When developers are stuck with a tough debugging problem, they'll often turn to Tracealyzer to help them gain insights into their system so that they can solve the problem. This has led to Tracealyzer gaining the reputation of being a great debugging tool; However, Tracealyzer is so much more than just a debugging tool.

In this blog series, we will examine several additional ways that developers should be using Tracealyzer besides debugging.



Start early and spot mistakes

First, developers should be using Tracealyzer the moment they set-up their software project and begin developing their application. The reason for using Tracealyzer so early is that it will help developers spot bugs and performance issues the moment that they occur! Most users wait until they have a problem to start tracing but if you trace your application periodically through-out development, you'll immediately spot strange behavior or mistakes during implementation. The result will be less time spent debugging which quickly correlates to less development time and lower costs.

Next, Tracealyzer can be used to trace embedded platform framework code or software stacks that behave as black boxes in application code. Many developers are starting to use embedded platforms or software stacks that are developed by 3rd parties and could span tens to hundreds of thousands of lines of code. There is no fast and efficient method available to understand how all that code executes and interacts without using a tool like Tracealyzer. Tracealyzer allows a developer to see what that black box code is doing and then properly account for it in their design and implementation.

A great example on how Tracealyzer can be used to understand black box software was recently published by Jacob Beningo in *A peek inside Amazon FreeRTOS* and *A peek inside Amazon FreeRTOS: Communication and memory* (both available online at beningo.com). In these articles, the author explores the Amazon FreeRTOS demonstration code, which contains little to no documentation, with Tracealyzer to examine and understand how the baseline code executes and functions.

Simple information such as how many tasks are in the application are discovered along with more difficult to find information such as malloc and free being called on average more than 350 times per second. Without Tracealyzer, you would either have to examine all the source, a time-consuming endeavor in an application that is ~418 kBytes, or cross your fingers and hope for the best.

Reverse engineer those stacks

Finally, one can use Tracealyzer to reverse engineer a software stack. There may be instances where a piece of software is open source, no longer supported or has major quality issues and the only way to really move forward is to start from scratch. If the software behavior is at least close, a developer could trace the stack and use that as a baseline to compare against the more robust software that is developed to take its place.

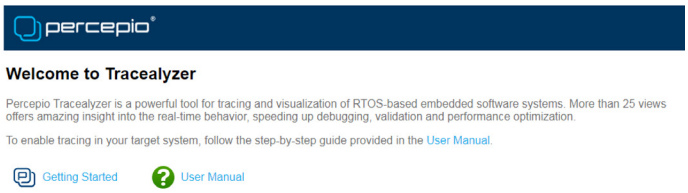
As we have started to see through-out this post, Tracealyzer is much more than just a debugging tool. It can be used through-out the entire development process to help developers monitor and understand their application. It can also be used to understand existing software in an efficient manner that doesn't require digging deep into source code. In the next several posts, we'll examine Tracealyzer in more detail and understand how to setup and use many of its functions. We'll also trace several software stacks to understand how commonly used open source software sizes up.

Analyzing Communication and Data Flow in an Unknown Software Stack

One of the biggest problems in embedded development today, besides spending too much time debugging, is understanding what a software stack or demo that you didn't write is doing. Embedded systems are becoming so complicated that the only way to build them in a cost-effective way and within a realistic time frame is to leverage existing components provided by 3rdparty stack providers and the microcontroller manufacturer. To be successful, we need to understand what this software is doing and how data flows around the application without spending weeks or months instrumenting or performing code reviews. In this post, we'll examine how we can do this quickly and easily using the Tracealyzer communication flow.

Set up the Tracealyzer recorder

Before using the communication flow, you'll have to set up the recorder library and trace the application code that you are interested in. If you have never acquired a trace before, you'll want to review the user manuals "Recording Traces" section for more details. This can be done in three simple steps:



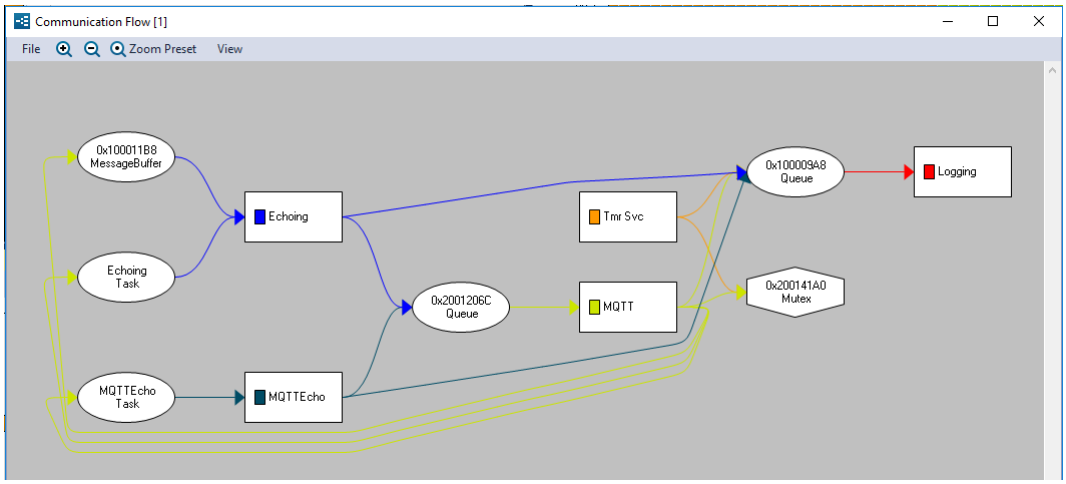
1. Open Tracealyzer.
2. Click the User Manual on the welcome screen next to the green question mark.
3. Click on the Recording Traces section and review the material.

Once a developer has acquired an application trace, they are then ready to start exploring the applications communication flow. For this post, we will be examining the application flow from the Amazon FreeRTOS demo application that connects embedded targets to Amazon Web Services (AWS). We acquired a trace using the base demo application on a STMicroelectronics IoT Discovery Node which supports Amazon FreeRTOS out of the box. Amazon FreeRTOS is an excellent example because at the time of this writing, there is little to no documentation that describes how the demonstration application is architected or behaves and it's easy to imagine an IoT developer wanting to leverage this example in their own code.

You can see all tasks

To get started with the communication and data flow analysis, after acquiring the trace and saving it, click on the Views dropdown from the menu and select "Communication Flow". The communication flow window will then present itself and for Amazon FreeRTOS will look something like what you see at the bottom of this page..

As you can see, there is a lot of useful information displayed. Let's look at how we can understand what exactly is happening with the application.



First, you'll notice that there are several different shapes in the view. The rectangles represent actors which in this example are the different tasks that are in the application. As you can see, there are five different tasks that are interacting:

- Tmr Svc
- MQTT
- MQTT Echo
- Echoing
- Logging

This doesn't mean that there are only five tasks in the application but five tasks that are communicating and passing data around the application. To get a full listing, we would want to examine the Trace View.

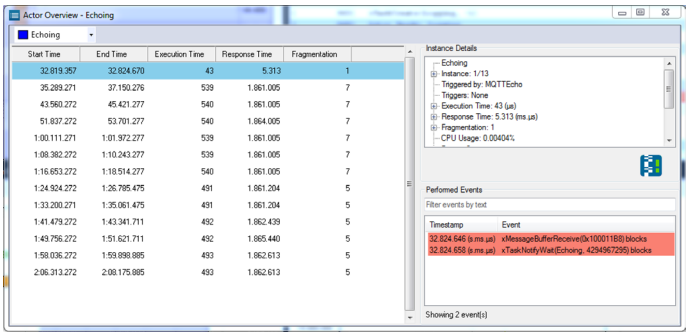
Next, you'll notice that there are ellipses and hexagons within the view. These shapes indicate the direction that data and communication is flowing. For example, the ellipses represent unidirectional communication and synchronization objects such as semaphores. The hexagon is used for bidirectional communication such as a mutex.

Click to highlight

You can easily filter and trace the view by clicking on an object or actor that you are interested in. For example, if you click on the mutex, the mutex, Tmr Svc and MQTT shapes are all highlighted along with the lines showing how they interact. Additional information is shown on the right-hand side of the view. From this information, we can see that the mutex is both sent and received by both tasks. This tells us that there is a shared resource that is being protected by this mutex.

If we were trying to understand how the Amazon FreeRTOS application works, we could review each actor and object and note how communication is flowing through the application. Note that there is a message buffer that is being populated and sent to the echo task. From looking at communication flow, one can deduce that the message buffer is data that will be sent to the AWS cloud. Echoing prepares the data and then puts the final message into a queue that will be sent to the MQTT task for transmission. At the same time, Echoing also posts a message in a queue that will record the action in Log task.

Whenever you want to learn more about an object or actor, simply double click on it to reveal an overview. The overview will filter all the events that involve that object and allow you to further investigate its behavior. For example, if you double click on the Echoing task actor, you'll see this:



There are several useful pieces of information that we can glean from this overview:

- Every execution including useful information such as start, end, execution, and response times.
- Individual instance details that include CPU utilization among other stats.
- Performed events which include when the actor was blocked.

We can use this information to not only learn about how an unknown code base is behaving but to also verify that the application is behaving the way that we expect it to. For instance, we might look through the Echoing actor overview and notice that the first time it executes its response time is milliseconds while subsequent executions it is nearly 2 seconds!

We might see this and determine that there is something not right about the way the application is behaving, and we can then dive deeper to understand why the code is behaving this way.

The communication view is a very powerful tool for developers to understand how their application and third-party code is behaving. It can be used to debug issues with the way the application is communicating and can be used to just understand what the application is doing.

The best way to fully understand how the communication view can benefit you is to acquire your own trace and experiment with the communication view.

Verify Task Timing and Scheduling

Let's face the dirty truth. There are quite a few embedded software developers creating real-time applications that don't know whether their applications meet their timing requirements. Early in the design phase, hopefully a rate monotonic analysis (RMA) is performed to estimate whether the application will be schedulable or not. But once the implementation phase is started, the real results are rarely fed back into the initial design assumptions through verification. In this post, we will explore how to use Tracealyzer to verify task timing and scheduling using the Amazon FreeRTOS trace that we acquired in the previous post.

The first tool one can leverage to verify task timing is the Actor Statistics Report. This report allows a developer to quickly gauge information about every task in the system such as:

- CPU Usage
- Execution times
- Response times
- Periodicity
- Separation
- Fragmentation

The execution and periodicity values can be extremely interesting to developers who are working to verify an RMA model or who just want to verify timing. The Actor Statistics report can be accessed by:

1. Clicking the Views menu
2. Clicking Actor Statistics Report
3. Selecting the desired tasks
4. Checking the desired data such as CPU Usage, Instance Periodicity, and so on
5. Pressing Show Report

The report data selection window presents you with all the options that can be seen in the top left image on next page.

Pick your statistics

The generated report may look a little bit different based on what data you are interested in. For example, generating a report for the Amazon FreeRTOS demo that includes CPU usage, execution and response times result in the report further down on the same page.

Notice how quickly we can get critical information from this trace data. We can immediately see that through-out the entire trace, the IDLE task is utilizing 45.521% of the CPU. This tells us that there should still be room for us to expand our application if it is done in the right way. What really stands out is that the MQTT task is using approximately 50% of the CPU! From the report, I can immediately ask whether this makes sense or whether there is something not right with the way MQTT is implemented.

Min and max execution times

We can also examine how long each task is executed from a minimum, average and maximum standpoint. I have always found it useful to review the spread between these and make sure that the minimum and maximum times make sense for the application.

For example, I would not expect much variation in the MQTTEcho task which we can see varies between 465 and 1735 microseconds. On the other hand, I may look at the MQTT task which varies from 66 microseconds to 26 seconds! Something about this task seems very fishy if it is executing for 26 seconds, which lets me know that I need to dive in and further investigate what is happening with this task.

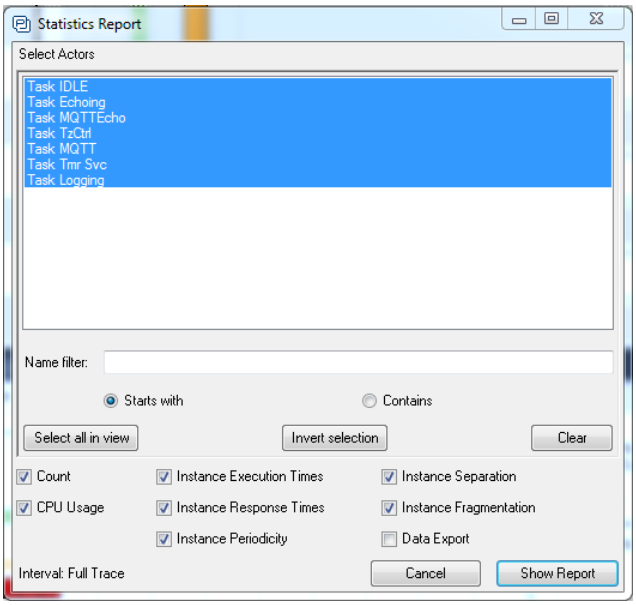
A second tool that you can use to help determine whether the timing on your tasks is enough to be scheduled successfully is the CPU Load Graphs. The CPU Load Graphs can tell a developer if they are getting close to maxing out the CPU at any point during the execution cycle.

The CPU Load Graphs can be accessed using the following steps:

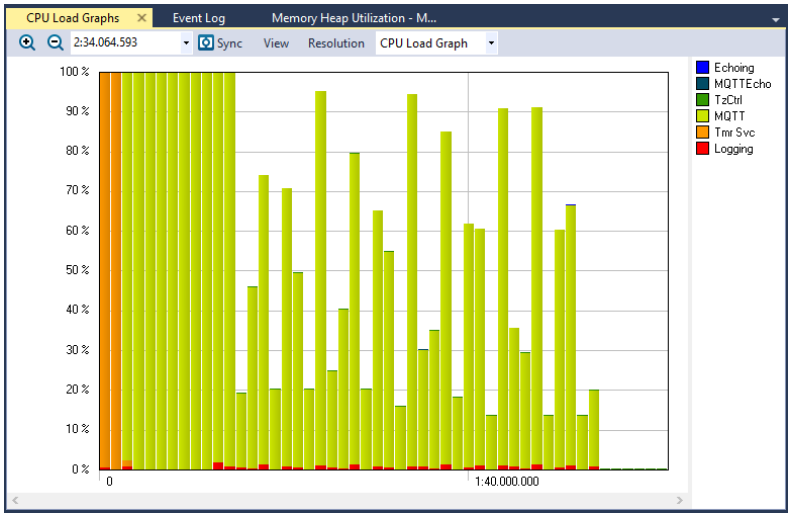
1. Click Views from the main menu
2. Click CPU Load Graphs
3. If you want the graph to be synchronized to the other views, click sync

For the Amazon FreeRTOS demonstration, the CPU Load Graphs appears as in the top right image.

Just by glimpsing at this graph we can see that there are periods during the



Select data you want to include in your report.



MQTT consumes a lot of CPU.

trace where the CPU was struggling to keep up. First, right at start-up, the CPU is at 100% utilization for approximately 37 seconds. Any attempt to add additional code during this period will result in schedules not being met (and they may not be met at the moment either).

data from Amazon Web Services (AWS), this most likely corresponds to those communication points. Again, providing us with some insights that if more code is going to be added to this application, we will need to carefully coordinate with the MQTT task to make sure that all deadlines are met.

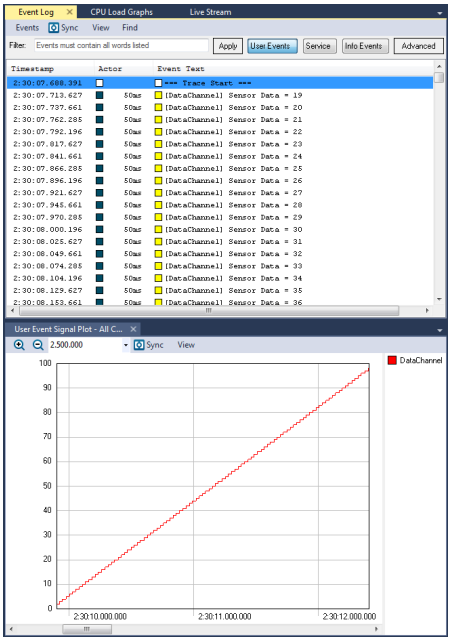
Actor	Count	CPU Usage	Execution Time			Response Time		
			Min	Avg	Max	Min	Avg	Max
IDLE	1	45.521	1:10.124.584	1:10.124.584	1:10.124.584	2:27.845.514	2:27.845.514	2:27.845.514
Echoing	13	0.004	43	479	540	5.313	1.719.220	1.865.440
MQTTEcho	13	0.005	465	568	1.735	409.187	5.181.579	30.919.700
TzCtrl	7088	0.142	31	31	31	31	10.905	26.568.101
MQTT	76	49.806	66	1.009.647	26.563.786	209.069	1.093.060	26.577.236
Tmr Svc	1	4.013	6.183.176	6.183.176	6.183.176	6.218.355	6.218.355	6.218.355
Logging	174	0.509	2.941	4.505	8.585	2.955	4.518	8.599

We can also notice that the main culprit appears to be the MQTT task. Again, multiple views seem to be suggesting that there is something going on with this task that requires further investigation.

Looking at the rest of the CPU Load Graphs shows that there are periods when the MQTT CPU utilization spikes. Since this application sends and receives

Armed with the data from the CPU Load Graphs and the Actor Statistics Report, we can use run-time data to determine whether our application code is indeed meeting the real-time deadlines and responses that we designed it to meet. These views can be critical in catching unexpected behavior and discovering potential issues within the code without having to wait for a bug to present itself.

Understanding Your Application with User Events

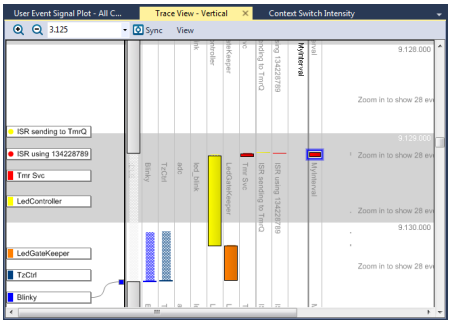


User events are a great way to visualize both application data and events.

Tracealyzer can automatically visualize how an RTOS-based application is behaving, which is a huge improvement over the "hope and pray" approach often used by developers. But what about events that don't automatically show up? What if you want to visualize some application data, measure the time between two events or monitor how a state machine in the application is behaving?

In this post, we will examine how you can set up such logging in FreeRTOS and view this information using Tracealyzer. Note that this post assumes you have already done the basic integration of the trace recorder library in FreeRTOS, as described in the Tracealyzer user manual.

The first step to visualizing custom information that is specific to your application is to create a user event channel. This is basically a string output channel that allows a developer to add their own custom events, called User events in Tracealyzer. For example, if I wanted to transmit sensor event data, I would first create the channel using the following code:



Intervals show up in the main timeline.

```
traceString MyChannel = xTraceRegisterString("DataChannel");
```

In case your compiler does not recognize this function, you need to #include "trcRecorder.h"

This function registers a user event channel named DataChannel in the trace. This makes Tracealyzer show a checkbox for this channel in the filter panel, so you can easily enable/disable the display of these events. Next, I am able to use either the vTracePrint() or vTracePrintF() functions to record my event data. I could transmit fixed string messages using vTracePrint as follows:

```
vTracePrint(MyChannel, "Button Pressed!");
```

Notice that we need to include the channel as the first parameter and then we issue our fixed string. If we wanted to record variable event data, such as changing sensor data, we could use the vTracePrintF() function as follows:

```
vTracePrintF(MyChannel, "Sensor Data = %d", SensorData);
```

While the format specifiers (%d etc.) are very similar to the classic printf function, vTracePrintF is separate implementation where most of the heavy lifting is done in the Tracealyzer application and it does not yet support all the numerous "printf" options. Specific documentation can be found in the Tracealyzer user manual and in trcRecorder.h.

Push the button

Once user event tracing is set up, we can record a trace containing both FreeRTOS events and the new user events from the application code. For example, if I recorded events from a push button (PB_Tx_1, PB_Tx_2) along with transmit and receive events (Tx, Rx) to understand my system timing, I can filter the event log for User Events to see only these events.

User events via vTracePrint or vTracePrintF are typically much faster than a classic printf because the real formatting is done in the host-side Tracealyzer tool, not in runtime. Further, vTracePrint is faster than vTracePrintF since the latter needs to scan the string to count the number of arguments. This requires a few more clock cycles but it's a great way to visualize data from the system.

For example, if I have an ADC that is sampling a sensor and I expect to see the data ramp up over a period of time, I can log that sensor data (as we have already seen) and then graph it using the User Event Signal Plot! All I need to

do is run my system and open the User Event Signal Plot and I would expect to see something similar to the diagrams in the beginning of the post.

Once we have a user event channel set up, we can start to consider the time between events – in Tracealyzer known as *intervals*. An interval represents the time between any two events in the trace, such as a button press and a button release. Intervals can be defined for any kind of event, kernel events or a user event like the ones we just defined.

Intervals are defined by you

Intervals can be extremely useful to understand important events in our system such as:

- How long it takes to get from point A to point B in a system (perhaps the time between when a USB device is plugged in and the USB stack is ready to use)
- The execution time of a function
- The time required to start-up the system

As you can image there are limitless possibilities that we may be interested in examining within the system. From the developer perspective, at a minimum, we can use vTracePrint() to create "Begin" and "End" events that we can then use to define our own custom intervals.

Intervals are defined in Tracealyzer using the "Intervals and State Machines" view. The steps are as follows:

1. Click Views -> Intervals and State Machines
2. Click Custom Intervals
3. Provide an Interval Name
4. Enter the text associated with the starting event, such as PB_Tx
5. Enter the text associated with the ending event, such as PB_Rx
6. Click Save

You'll notice that the new interval "MyInterval" gets added to the Trace View, highlighting the time between the PB_Tx and PB_Rx events, as shown below:

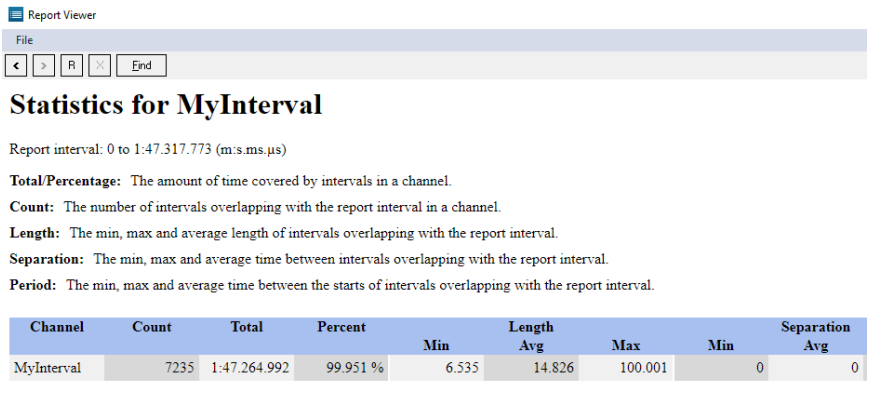
You can show any number of intervals in parallel to highlight important parts of your code, as long as there are corresponding events in the trace.

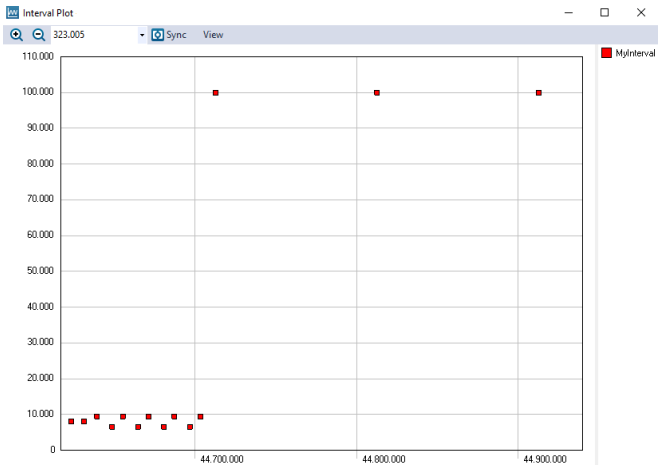
It is also possible to get statistical information on all occurrences of an interval such as min, max, average time, etc. If you right-click on the interval entry in "Intervals and State Machines" view, you will find further options like "Statistics" and "Show Plot".

Tables and diagrams

The "Statistics" option gives you a report with descriptive statistics of the interval durations, like the one shown below. Here you can see the longest and shortest durations of all such intervals in the trace, as well as other metrics like separation and periodicity. All min/max values are actually links, so when you click them Tracealyzer will show you the corresponding location in the trace view.

To see more detailed information about the intervals, select "Show Plot" instead. This shows a plot over time, where X is the timeline and the Y-axis shows the duration of each interval, like in the example below showing 10 short intervals (around 5-10 ms) followed by three 100 ms intervals.



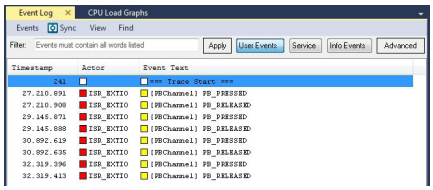


Interval statistics can be viewed as a table (above) or a plot (right).

Analyzing State Machines

In the previous Tracealyzer Hands On post, we discussed how a developer can create a user event channel to monitor events in their application. As you may recall, we also introduced the concept of intervals which is the time between any two events and can be added to the timeline. In this post, we will take the interval concept one step further and see how we can monitor state machines.

Let's start with a simple example. Suppose that a developer has created a user event channel for a push button, where user events are generated in the interrupt handler. The push button can have two possible states; PB_PRESSED and PB_RELEASED. If the developer runs the code and occasionally presses and releases the push button, they might capture a user event log that looks like the following.

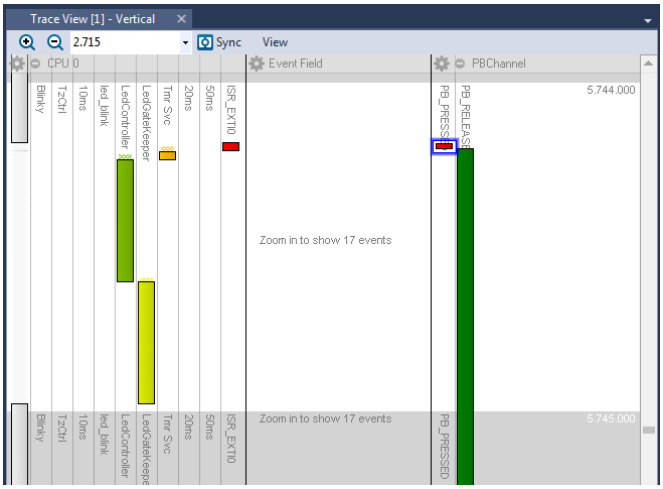


You can see that we have PB_PRESSED and PB_RELEASED events that are being generated by the ISR_EXTIO interrupt. These events can be viewed as a state machine for the push button. In Tracealyzer, state machines generate a special kind of interval, representing the time between state changes as well as the logged state. Let's create a state machine for our push button example using the following process.

1. Click Views -> Intervals and State Machines
2. Click "Add Custom State Machine"
3. The New State Machine window will appear with the options for Simple and Advanced. Click Simple.
4. From the dropdown, select the user channel name.
5. Click Create

At this point, you should see PBChannel added to the Intervals and State Machines list. Go ahead and close the State Machines list.

State machines can be visualized just like any other interval in Tracealyzer. The state visualization is very much like a logic analyzer, but for software rather than physical signals. You can see how your system behavior correlates with the system states, and you can show multiple states of relevance in parallel, to see how they overlap. The state machine that we just created will now show up in the trace view alongside our tasks.



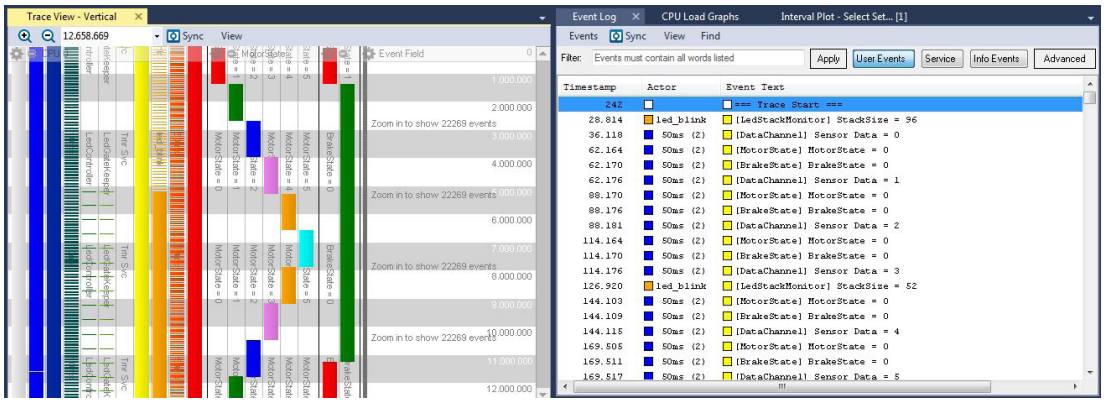
You can see that the PBChannel has been added in the far right of the trace view. We can now examine the trace and see how the push button state changes over time in addition to how the system task behavior may change when the button is pressed. This allows us to more easily identify any potential bugs or issues that may exist in our code by carefully examining states and tasks in parallel.

Note that you can flip the view to a horizontal orientation if you prefer that. Just select "View" -> "Horizontal View".

This approach for state machine visualization can be used to show any kind of state in the system, as long as the state can be logged on a user event channel. For example, developers can log application state changes, low-level driver states such as USB and TCP/IP states or even hardware states. This only requires the developer to take the time to instrument their state changes with the vTracePrint or vTracePrintF function calls.

Fields in the Trace View

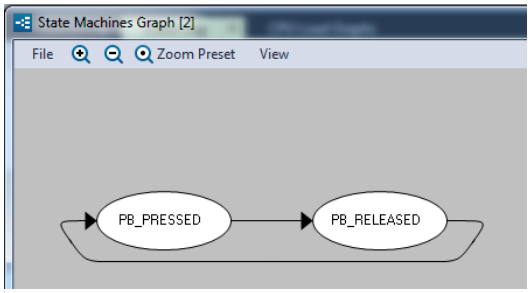
The trace view is composed of fields. In the above screenshot of the trace view, these are labeled CPU0, Event Field and PBChannel. The two first are shown by default, while PBChannel was added when we defined the state machine.



You can however create any number of fields using the View -> Add Field option, e.g. multiple scheduling fields that divides your tasks into logical groups, or multiple state machines. Fields can be reordered and individually configured. You can minimize them on the timeline or close them when no longer relevant. Note the settings gear next to the field name, which provides various options:

- Display size – To adjust the size of the field
- Collapse – to minimize the field
- Select Interval – to change the interval that is displayed in this field
- Close – closes the field in the timeline

One last trick to discuss today is that once the state machine has been defined, we can view all observed state changes as a graph. To do this, simply click the Views -> State Machines Graph. All the states and the transitions for those states will be graphed which then allows you to go through the graph and make sure that no illegal state transitions occur in the code. For our simple example, the state machine graph is equally simple.



Case study: motor control

Many embedded applications that involve motors make use of two different state machines; the motor state and a brake state. A motor may be in a state such as locked, stopped, low speed, medium speed and high speed. A brake state might be enabled and disabled. Obviously, if the motor is running, the brake should be disabled, otherwise we would undoubtedly start to see some smoke or at a minimum a feel a bad smell from the brake pads rubbing on the motor. Running the motor with the brakes engaged will force the motor to work harder, potentially resulting in a failure or damage to the motor. Let's examine a trace where a motor state machine ramps up then down and verify that the brake behaves the way it is supposed to.

First we add user event logging for the state changes, as described in the earlier post, *Understanding your Application with User Events*. This results in two user event channels, named MotorState and BrakeState (see above).

After acquiring the trace, we need to make Tracealyzer aware of this state information by defining state machines for these user event channels. As we discussed in the previous blogs, we can do this using the "Intervals and State Machines" menu option under Views. This view provides a list of all defined state machines and interval sets, initially empty, and provides three options for adding new data sets. These options include:

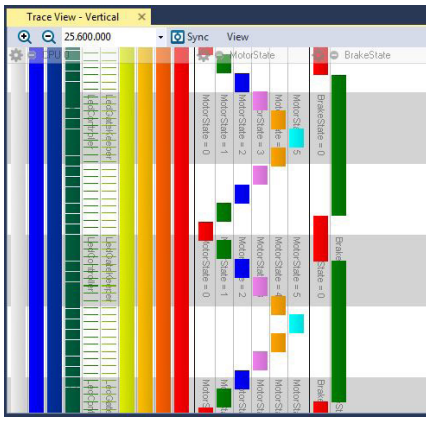
Add Predefined – This enables predefined intervals and state machines that Tracealyzer is aware of. For example, Tracealyzer automatically generates two interval sets for each message queue in your trace, "Message Processing" and "Queue Messages", and if using Keil RTX5, the states of TCP sockets can be included this way.

Add Custom State Machine – Here you can define a state machine using either the simple or the advanced method. The simple method assumes that there is a dedicated user event channel where only state names for the specific state

machine is logged, while the advanced option makes use of regular expressions which allows you to extract state information from any event in the trace.

Custom Intervals – This option allows a developer to specify how to match events to produce a custom interval set. You specify strings to match for the Start and End events of the interval, "Interval Start" and "Interval End", and intervals are then created for all matching event pairs.

For this blog, we are just going to use Custom State Machine (the simple option) to define state machines for the BrakeState and MotorState user event channels. The result can be seen here.



The brake state machine has two states, 0 = Engaged and 1 = Released. The motor states are in increasing order 0 = Locked; 1 = Stopped; 2 = Low; 3 = Medium, 4 = High, and 5 = Max speed.

Notice that on the right-hand side you can see the MotorState and BrakeState state machines. From a visual inspection, we can see that the brake is on at the start, is released when the motor is unlocked and then engages again when the motor has stopped.

Visual inspections are great but having a more in-depth reporting system that can automatically analyze the trace is preferred. From the "Intervals and States Machines" window, a developer can right click on their custom data set and then generate a plethora of useful views and reports to help them understand their application. These include:

Statistics – Generates a report for the data set, showing min, max, average lengths etc. This is useful for finding extreme cases, like the longest time between two events, and for making a more systematic analysis of the application by tracking important metrics that may change as the system evolves.

Show Timeline – Opens a separate horizontal trace interval, for easy correlation with other horizontal views.

Show Plot – Gives a scatter plot showing the durations of the intervals over time (much like the Actor Instance graphs for task execution times).

Compute Overlap – Creates a report on the intersection of two data sets (i.e. is there any time when motor and brakes are on at the same time).

Create Inverted – Generates a new "negated" data set, where you have intervals corresponding to the gaps in the original data set. This can be really useful when combined with "Compute Overlap".

State machine reporting can provide us with a wealth of information about how a state machine is behaving but also how it acts compared to other state machines. This is information that would be difficult to acquire and analyze if a developer was not using Tracealyzer. Of course, you could hook up a logic analyzer instead, but then you would need one output pin for each state, and you need to analyze the results manually. Tracealyzer makes this a lot easier.