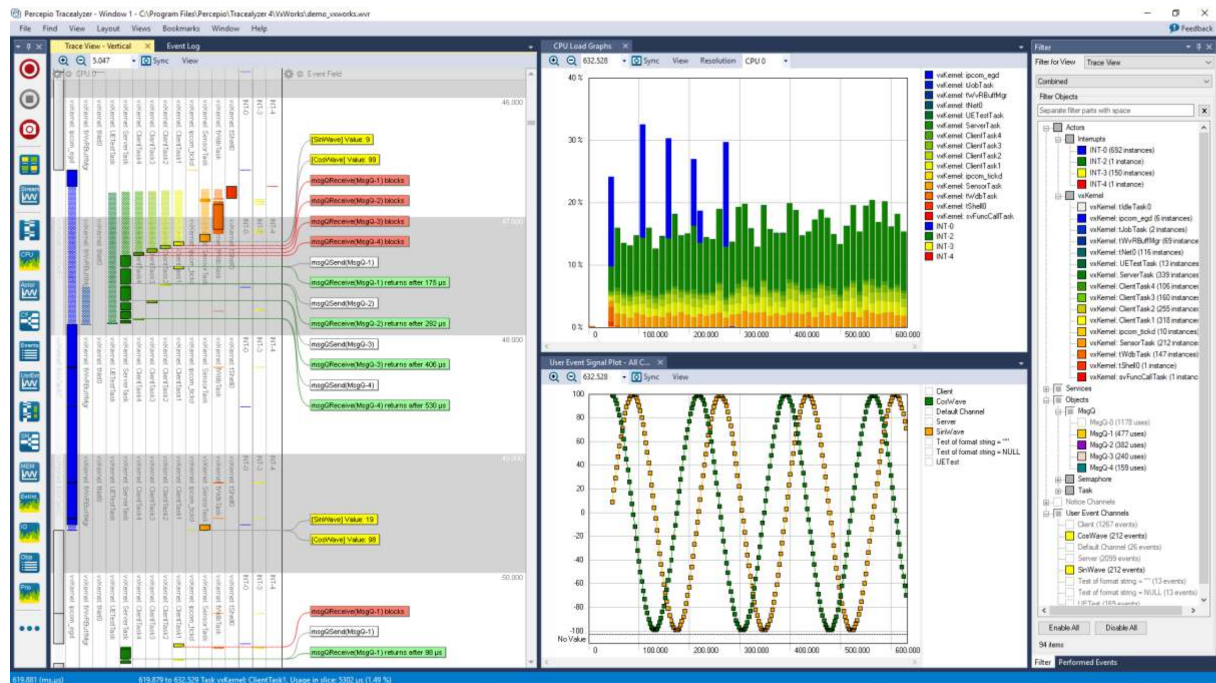
Tracealyzer for VxWorks – Overview and Getting Started Guide

Percepio Application Note PA-031



Tracealyzer is a trace visualization and analysis tool for embedded software developers. The recently released Tracealyzer version 4 for VxWorks gives the developer an unprecedented insight into the runtime behavior, leading to reduced troubleshooting time and improved software quality, performance and reliability.

Tracealyzer does not depend on special trace hardware which means that it can be used even outside the development lab, e.g. in field testing or as a flight recorder in deployed systems. The latter mode is particularly useful when developers are trying to capture a rare error that is hard to reproduce.

Tracealyzer does more than simply display the recorded data. It actually understands the meaning of VxWorks kernel events and leverages this to provide more advanced visualization of standard VxWorks traces. Apart from the innovative trace view, Tracealyzer also provides several other views like the Communication Flow, which is a dependency graph showing how tasks and other VxWorks objects are interacting in runtime. Another example is the Object Utilization view, showing the allocation of kernel objects over time (e.g. the size of message queues). All these views are interconnected in intuitive ways, just double-click on a data point to open a related view focused on the same data point, showing a different perspective and abstraction level. For instance, double-clicking on a “msgQReceive” event in the trace view opens the Object History view, showing all operations on that particular message queue. With multiple connected views at different abstraction levels, you can spot anomalies using the high-level overviews, like the CPU Load Graph and Statistics Report, and easily drill down to inspect the detailed events causing the anomaly.

The sophisticated visualization offered by Tracealyzer makes the trace data more accessible and usable, allowing developers to speed up debugging and verification.

Tracealyzer can be used side-by-side with traditional debugging tools such as those available in Wind River Workbench and complements a source-code debugger with several additional views on system level, ideal for understanding real-time issues and rare anomalies where a classic source-code debugger is not sufficient.

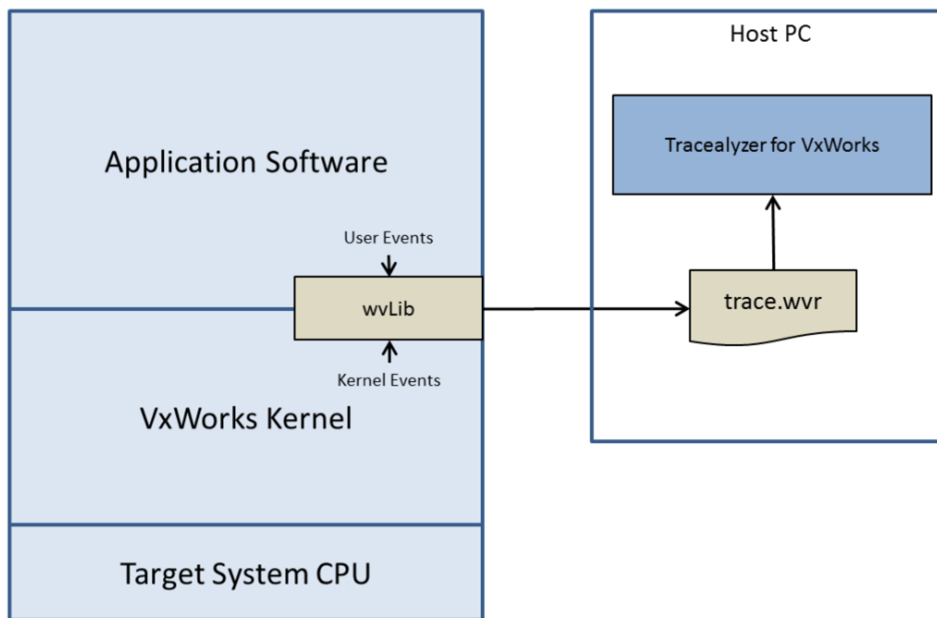
Installing Tracealyzer for VxWorks

Download the Tracealyzer package by registering at percepio.com/download and make sure to specify VxWorks as target platform. You may also request a free, time-limited evaluation license together with the download. Full licenses are available through Percepio's distributor network and from its web store. University students and academic researchers may request a free academic license. See percepio.com/licensing for more information.

Tracealyzer is a .NET application that runs natively on Windows and also on Linux assuming Mono, an open-source .NET framework endorsed by Microsoft, is installed. Mono is included in most Linux distributions, and can otherwise be downloaded from http://www.mono-project.com. We recommend using the latest version of Mono, at the time of writing v5.14. For further instructions, see this FAQ entry with distribution-specific instructions.

Using Tracealyzer with VxWorks

Tracealyzer for VxWorks works with trace files from the VxWorks tracing library, wvLib. Those traces, stored in .wvr format, contain kernel events like context switches and kernel API calls, and may also include "User Events" logged from the application code using the wvEvent function in wvLib.



The VxWorks tracing library buffers the trace data in an internal RAM buffer, which can be saved to a target-side file (continuously or when the tracing is stopped), or transferred continuously via a TCP socket.

To trace your VxWorks system, you first need to make sure that your VxWorks Image project has been configured to include wvLib (for Wind River System Viewer) and that it uses a high-resolution timestamp driver.

Tracealyzer includes a small library for VxWorks (see tracealyzer.c/.h in the VxWorks folder) that allows for configuring, starting and stopping the recording, and also for storing formatted user events. This is intended as a simplified interface to the integrated VxWorks recorder. Include tracealyzer.c and .h in your application and use the functions listed below to configure and control tracing.

The easiest way to record traces using the Tracealyzer library is with the default settings. In that case, you only need to call tzStart() to begin the recording and tzStop() when you have finished your test

case. In the default setup, the trace is kept in a RAM buffer during the recording and stored to a file ("tz.wvr") on the target device when tzStop is called. Thus, the trace length is limited to the size of the trace buffer (by default 256 KB). If the buffer becomes full before tzStop is called, older events will be overwritten.

`STATUS tzStart(int traceLevel)`
Starts tracing. The parameter "traceLevel" decides the level of detail, valid options are:

- `TZ_TRACE_LEVEL_CONTEXT_SWITCH`: Only context-switches (tasks and ISRs).
- `TZ_TRACE_LEVEL_TASK_STATE`: Adds task state changes (ready, etc.).
- `TZ_TRACE_LEVEL_OBJECT_AND_SYSTEM`: All details (recommended).

`STATUS tzStop(void)`
Stops tracing and may also store the trace, depending on the Storage configuration (see tzConfigStorage).

`STATUS tzEvent(char* channelName, char* formatStr, ...)`
Stores a formatted User Event (see section "Tracing Application Code with User Events").

All functions return 0 if successful.

Custom Trace Configuration

The functions `tzConfigStorage()` and `tzConfigBuffer()` may optionally be called prior to `tzStart`, in order to apply a custom tracing configuration.

```
STATUS tzConfigStorage(int storageMode, int storageMethod, char* arg1, int
arg2, int storeOnStop)
```
Configures how traces are stored.

You may omit the configuration call; default settings are then used, corresponding to the following setup:

```
tzConfigStorage(TZ_MODE_DEFERRED, TZ_METHOD_FILE, "tz.wvr", O_CREAT |
O_TRUNC, 1)
```

Valid options for parameter storageMode are:

- `TZ_MODE_CONTINUOUS` (= 0): Trace is continuously written to a file or socket.
- `TZ_MODE_DEFERRED` (= 1): Trace is stored on command (tzStop).

Valid options for storageMethod are:

- `TZ_METHOD_FILE` (= 0): Trace is stored on the target file system.
- `TZ_METHOD_SOCKET` (= 1): Trace is uploaded to host via a TCP socket. Tracealyzer can receive traces via TCP sockets, but this mode is not yet fully supported for VxWorks as of v4.2.12. We aim to support this option soon.

Parameters arg1 and arg2 depends on the setting for storageMethod.

If using `TZ_METHOD_FILE`:

- arg1 should be the path of the target-side trace file
- arg2 is the file open attributes/flags – should be "`O_CREAT | O_TRUNC`"

If using storageMethod `TZ_METHOD_SOCKET`:

- arg1 should be the address of the host computer that is to receive the trace.
- arg 2 should be the TCP port of the receiving computer,

The parameter `storeOnStop` is only relevant if storage mode is `TZ_MODE_DEFERRED`. If `storeOnStop` is set to 1, calling `tzStop` will also store the trace to the specified file.

Some example setups

Continuous tracing to file. Allows for long traces. Continues until `tzStop` is called.

```
tzConfigStorage(TZ_MODE_CONTINUOUS, TZ_METHOD_FILE, "tz.wvr", O_WRONLY, 0)
```

Deferred tracing to file. The latest events are kept in a RAM ring-buffer and saved to file when `tzStop` is called.

```
tzConfigStorage(TZ_MODE_DEFERRED, TZ_METHOD_FILE, "tz.wvr",
O_CREAT|O_TRUNC, 1)
```

Tracealyzer can receive traces via TCP sockets, but this mode is not yet fully supported for VxWorks as of v4.2.12. We aim to support this option soon.

```
STATUS tzConfigureBuffer(int bufferCount, int bufferSize)
```

Configures the size of the trace buffer (rBuff), expressed in the number of sub-buffers and the size of each sub-buffer. If this call is omitted, a default configuration is used (4 x 64 KB). When using deferred tracing to file, this decides the length of the trace. When using continuous tracing, this buffer size may impact the maximum throughput of the tracing.

To view the resulting .wvr trace file, select File -> Open in Tracealyzer, or simply drag the file to the Tracealyzer main window.

For more details about the Tracealyzer Library for VxWorks, including other tracing configurations, please refer to the code documentation in tracealyzer.h. For now, we recommend using `TZ_METHOD_FILE` with deferred or continuous storage, as socket mode is not fully supported in Tracealyzer 4.2. Note that you may also record traces using Wind River Workbench and view the resulting .wvr files in Tracealyzer.

Tracing Application Code with User Events

The VxWorks trace recorder (wvLib) allows you to store custom events from your application using the function `wvEvent()`. This allows for writing binary data to the trace buffer.

```
STATUS wvEvent(event_t usrEventId, char* buffer, size_t bufSize)
```

Tracealyzer can display these events, but makes no interpretation by default and displays them as raw data. You may however use the User Event Interpretations (see the View menu) to set up formatting rules for displaying VxWorks events in Tracealyzer. This allows Tracealyzer to extract strings and values that can for instance be plotted in a User Event Signal Plot.

There is another, easier way to add formatted log messages to your trace – use the `tzEvent`() function found in the Tracealyzer Library for VxWorks. When using `tzEvent`, it is not necessary to use the User Event Interpretations feature needed for native wvEvents.

```
STATUS tzEvent(char* channelName, char* formatStr, ...)
```

The `tzEvent` function provides an interface similar to a classic printf, but the actual formatting is done off-line in the Tracealyzer visualization which makes `tzEvent` much faster than a printf. Moreover, unlike a printf you get the information visualized in the Tracealyzer views, which allows you to correlate the event with other recorded events, such as scheduling and system calls.

The parameters of `tzEvent`() are:

- `channelName`: A textual name that specifies the User Event Channel of the event, that allows the user events to be filtered easily. This name is displayed in the Filter view, under the User Event category.
- `fmtString`: A format string allowing for text and format specifiers like a classic printf, that allows you to embed data arguments into the string. This does not use the stdio formatting, but most common format specifiers from printf are supported. See tracealyzer.h for details.
- `data arguments`: Optionally, you can include a variable number of data arguments, as specified by the format string.

Example usage

```
tzEvent("Debug", "Entering function foo()");

tzEvent("Voltage U1", "U1: %lf v", voltage_u1);

tzEvent("MyState", "%s", fsm1StateNames[newStateID]);
```

By default, formatted user events are limited to 256 bytes, which shall contain channel name and format string (including null-terminated), any data arguments and four initial bytes that identifies the event as a Tracealyzer formatted user event. This limit can be increased by adjusting `MAX_ARG_SIZE` in tracealyzer.h but beware that your application will then require more stack space.

The VxWorks wvEvent function uses a numerical identifier for all events. Tracealyzer does not use this identifier formatted user events, so tzEvent uses a constant for this purpose (`WVEVENT_ID`) which by default is set to 0. `WVEVENT_ID` is defined in tracealyzer.h in case you wish to change this value, e.g. to avoid collision with existing custom events.

Why Choose Tracealyzer?

Tracealyzer for VxWorks uses the same data source as System Viewer but provides a more powerful visualization featuring over 30 views, interconnected in clever ways.

Smart trace visualization: The trace view (figure 1) gives a visual timeline of all recorded events, with many options for filtering and visualization. By default, the timeline has a vertical orientation, which makes it natural to show events (e.g. VxWorks API calls) in clear text, as *event labels*. The event labels can also be filtered in several different ways to focus the view on the most relevant events.

The timeline can be rendered vertically as in this example, or horizontally; in both cases it works exactly the same.

The trace view is composed of *view fields* of several types for showing different kinds of information (task scheduling, event labels, intervals, state machines etc.) Fields can be added, collapsed and expanded, rearranged and closed individually, to enable the user to always focus on the most relevant information at any time. You can even have multiple fields of the same type (e.g. scheduling) with different settings and filters.

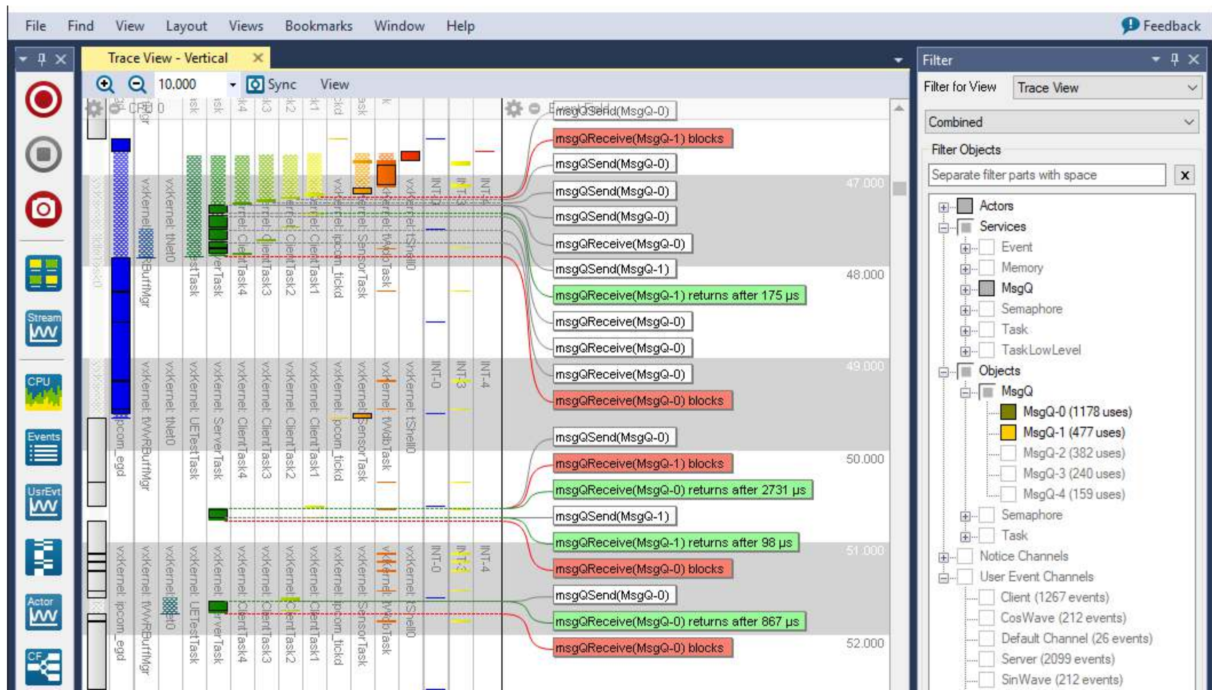


Figure 1: Event labels in main trace view

Understand Runtime Dependencies: Dependencies between system calls, tasks and other kernel objects are understood and highlighted by Tracealyzer. For instance, when selecting a "return from blocking" event (green label) the corresponding "blocking" event (red label) can be found using the "Go to Entry event" option, as shown below.

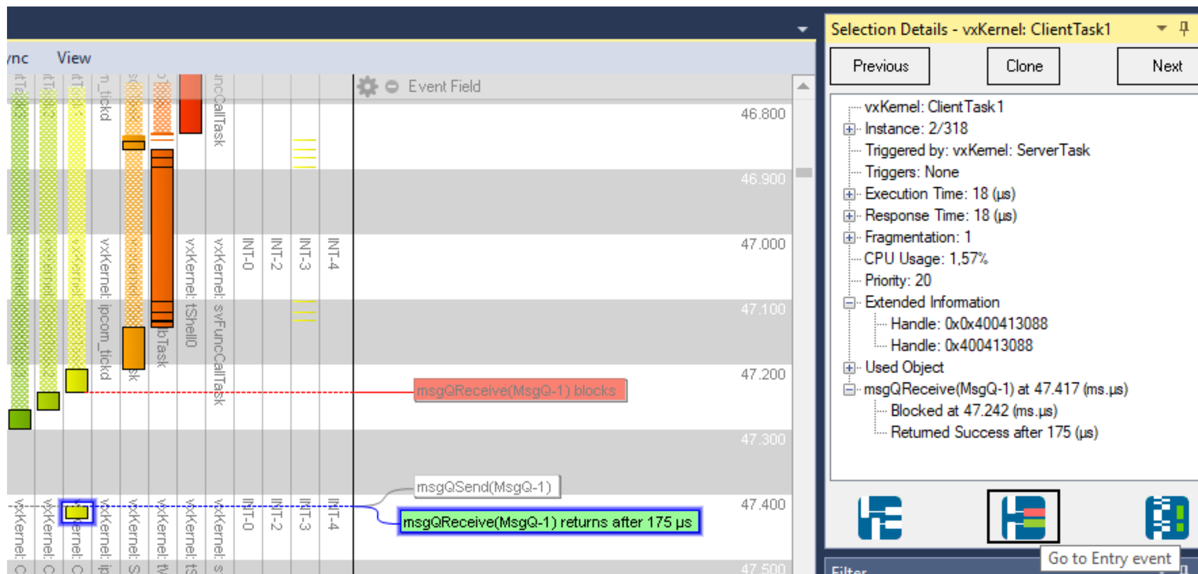


Figure 2: Find related events

Moreover, the Communication Flow graph shown in Figure 3 provides a dependency graph of the task interactions via VxWorks services. This is generated from the trace data and can be regarded as a summary of the recorded VxWorks API calls. This is a very useful high-level view of your system's runtime architecture. Double-clicking an object in the graph shows a list of the corresponding events in the Object History view.

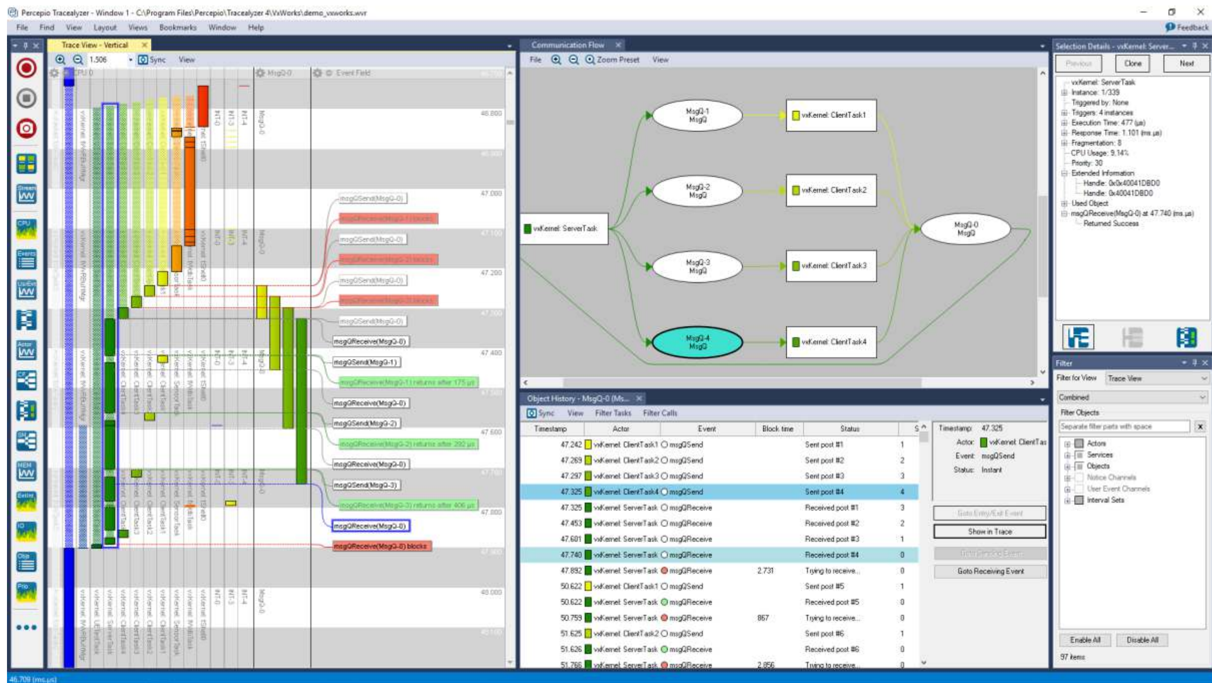


Figure 3: Trace, Communication Flow and Object History views

Integrated data plotting: Tracealyzer allows you to log any application data or event as User Events, shown in the trace view as yellow labels. Any data arguments can be plotted in the User Event Signal Plot view, as shown in figure 4, and clicking on a data point will cause Tracealyzer to highlight the corresponding event in the trace view. That way one can correlate the data with task scheduling, interrupts, system calls, and other events.

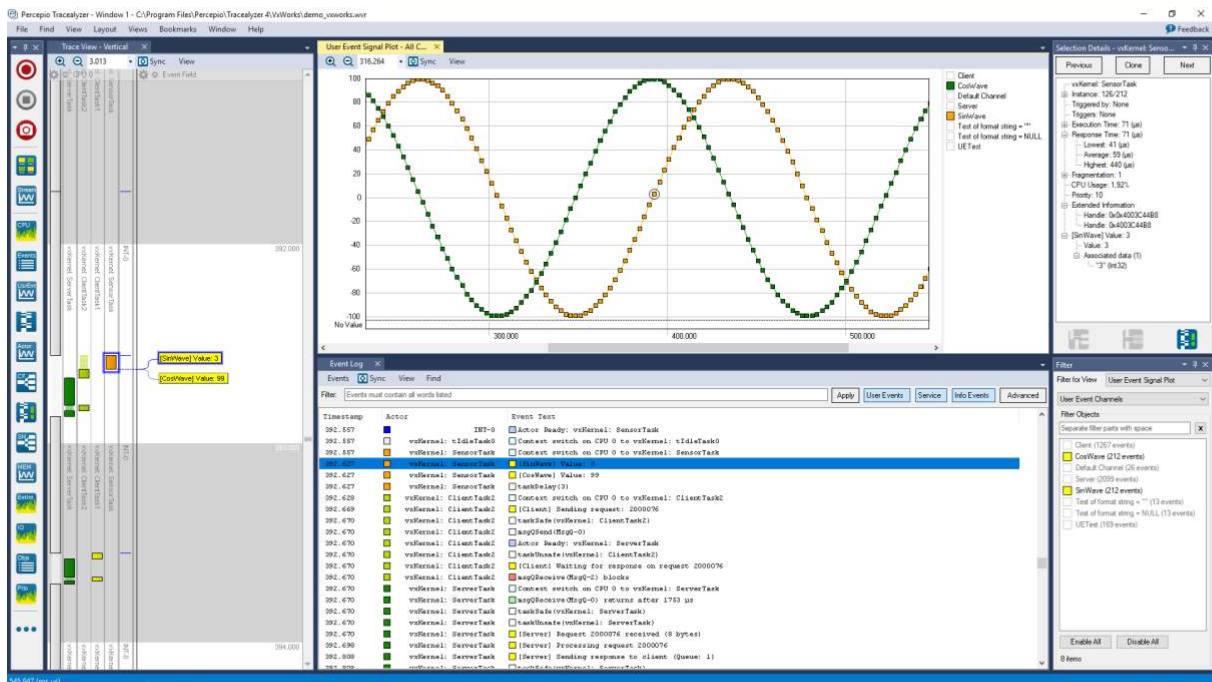


Figure 4: User Event Signal Plot

Integrated memory usage view: Tracealyzer provides a graph showing dynamic memory allocation over time, i.e., malloc() and free() calls, as illustrated in figure 5 below. This allows you to spot memory leaks and excessive memory usage. Underneath the graph you can see a more detailed memory allocation view, where addresses are shown and malloc() calls can be matched with their

corresponding free() calls to isolate all remaining allocations, which makes it easier to pinpoint memory leaks. Since both memory views are connected to the trace view, you can easily find and analyze the context of any issues found.

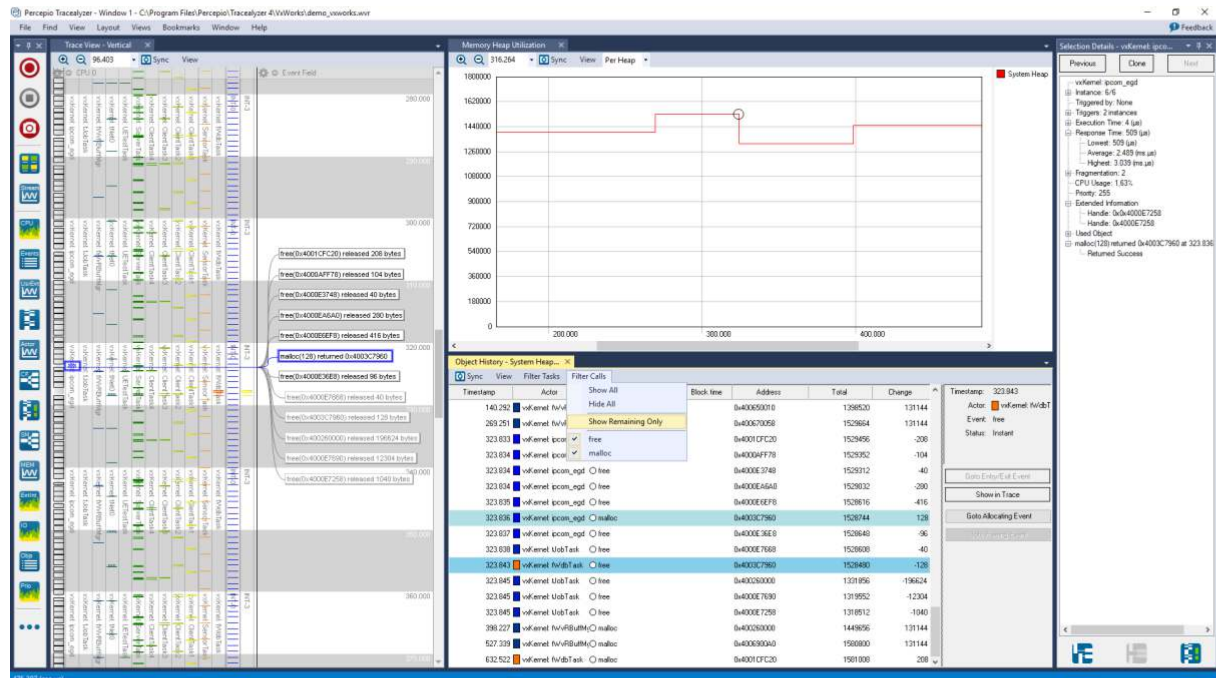


Figure 5: Memory Heap Utilization

For further information, contact support@percepio.com. We are happy to offer an online demonstration and help you get started using your own tools and software. The percepio.com web site has a lot of material about getting the most out of Tracealyzer, for both beginners and more experienced users.