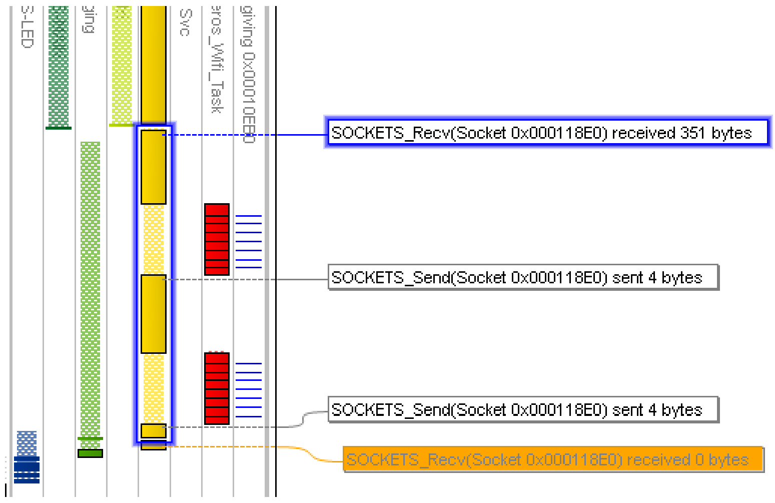
Adding Extensions to the Percepio Trace Recorder Library

Percepio Application Note PA-027, 2019-01-03

By default, the Tracealyzer Recorder Library allows for tracing two kinds of information: RTOS kernel events such as context-switches and RTOS API calls, and application events such as debug logging, state changes and variable values. The latter are logged as "User Events" by inserting calls to e.g. vTracePrint().

Since v4.2.9 there is a third way of tracing your application, using *recorder extensions.* These self-contained modules makes it easy to add tracing of important function calls, such as middleware APIs, driver APIs, or any other important functions you like to trace. An example screenshot from Tracealyzer is shown below, where the "Secure Socket" API in Amazon FreeRTOS has been traced using the extension system.


SOCKETS_Recv(Socket 0x000118E0) received 351 bytes

SOCKETS_Send(Socket 0x000118E0) sent 4 bytes

SOCKETS_Send(Socket 0x000118E0) sent 4 bytes

SOCKETS_Recv(Socket 0x000118E0) received 0 bytes

This does not require any code changes in targeted functions. Only a single #include line needs to be added in application source code files using the traced API. The rest is handled automatically by the preprocessor and trace recorder, based on the definitions in the extension.

Unlike the "User Event" concept, an extension is intended for systematic tracing of API function calls and can take advantage of all powerful features in Tracealyzer. This since extensions has two components, a header file with target-side definitions and also an XML file with host-side metadata. The XML file tells Tracealyzer how to interpret the events, e.g. as an I/O operation. This way, Tracealyzer understands the meaning of the events and includes them in relevant views, such as the I/O Plot.

Extensions are self-contained and easy to integrate, which makes them convenient for distribution. Embedded software vendors can thus develop and provide custom Tracealyzer extensions for their customers, without having to integrate instrumentation code in their existing code base. Moreover, extensions from different providers can be mixed freely within the same system, without causing conflicts.

This document describes how to use an existing extension, e.g. those included for Amazon FreeRTOS, and explain how the extension systems works in general. It does not describe how to make your own custom extensions. Also note that the extension system currently (as of v4.2.9) only supports streaming mode at the moment, not snapshot mode. We aim to address these limitations in the near future.

https://percepio.com

Quick Guide for the Amazon FreeRTOS Extensions

This assumes you have integrated the trace recorder library as described in the Tracealyzer User Manual, configured for streaming mode, and that you are using Tracealyzer v4.2.9 or later.

Tracealyzer includes two example extensions for Amazon FreeRTOS, allowing for tracing calls to some of the functions in aws_secure_sockets.h (SOCKETS_Connect, SOCKETS_Send, SOCKETS_Recv) and aws_wifi.h (WIFI_On, WIFI_Off, WIFI_ConnectAP). These extension files are already in the right directories.

To enable this tracing for Amazon FreeRTOS, follow these steps:

1. Open include/trcExtensions.h and set TRC_CFG_EXTENSION_COUNT to 2.
2. Search for all files containing #include "aws_secure_sockets.h" or #include "aws_wifi.h."
3. In these files, add #include "trcExtensions.h" as illustrated below, *except for* in the .c files containing the targeted functions, i.e. aws_secure_sockets.c and aws_wifi.c. Do not change those.

```
#include ...
#include "aws_secure_sockets.h"   /* The traced API */
#include "trcExtensions.h"        /* Should be after, or last */
```

4. Recompile your code and run a Tracealyzer recording in streaming mode. The new function call events should now appear in the trace.
5. In case you get compiler errors about "duplicate definitions" or similar, you have most likely included trcExtensions.h also in aws_secure_sockets.c or aws_wifi.c (or in a header file they include). Remove those includes in that case. See "How it Works" for an explaination.

The include of "trcExtensions.h" can remain in the code base, if desired. This since the trace wrapper functions are available even if the tracing is disabled, but are then only calling the original function and will probably be eliminated by compiler optimizations.

Using Extensions in General

This is a more general guide, that should be used if you wish to use other extensions than the included examples for Amazon FreeRTOS. The Amazon FreeRTOS extensions are however used as example.

This assumes you have integrated the trace recorder library as described in the Tracealyzer User Manual, configured for streaming mode, and that you are using Tracealyzer v4.2.9 or later.

An extension consists of two files, an extension header file (e.g. "aws_secure_sockets.tzext.h") and an XML file for the Tracealyzer host application (e.g. "aws_secure_sockets-v1.0.0.xml"). The header files for the Amazon FreeRTOS extensions are found in the recorder's include directory, and the corresponding XML files are found in the "Tracealyzer 4/cfg" directory.

The central point of the extension system is the header file trcExtensions.h, that may include up to four extensions header files, as well as the overall settings for the extension system.

Note that the limitation to four (4) extensions does not limit the number of functions that can be traced, as each extension can cover an arbitrary number of functions.

To enable one or several predefined extensions, follow these steps:

1. Update trcExtensions.h with respect to the following definitions:

1.1. TRC_CFG_EXTENSION_COUNT: The number of extensions to enable. This is 0 by default, indicating that no extensions are enabled. Setting this to 2 will enable extension 1 and 2, i.e. those defined in TRC_EXT1_HEADER and TRC_EXT2_HEADER. Maximum value is 4.

1.2. TRC_CFG_EXT_MAX_NAME_LEN: The number of bytes to reserve for each extension name in the trace header. It is important that this is large enough to accommodate the longest extension name in use. The extension name is given by the …NAME definition in the extension header.

1.3. For each extension you like to use, update:

1.3.1. TRC_EXT<n>_HEADER: The extension header file name for the *n*th extension you want to include, for example "aws_secure_sockets.tzext.h".

1.3.2. TRC_EXT<n>_PREFIX: The Extension Prefix of the *n*th extension header file you want to include. This is the first part of the definition names in the extension header file (e.g. TRC_EXT_SOCKETS).

2. Add #include "trcExtensions.h" in all .c files calling the API. This must be placed *after* the #include lines of the traced API(s). We recommend to always place the #include “trcExtensions.h” line as the last #include in the file.

```
#include ...
#include "aws_secure_sockets.h"  /* The traced API */
#include "trcExtensions.h"       /* Should be after, or last */
```

3. Verify that the extension’s corresponding XML file is found in the "Tracealyzer 4/cfg" folder. Also double-check that the file name matches the NAME and VERSION definitions in extension header.

4. Recompile your code, run a Tracealyzer session and the calls now appear in the trace.

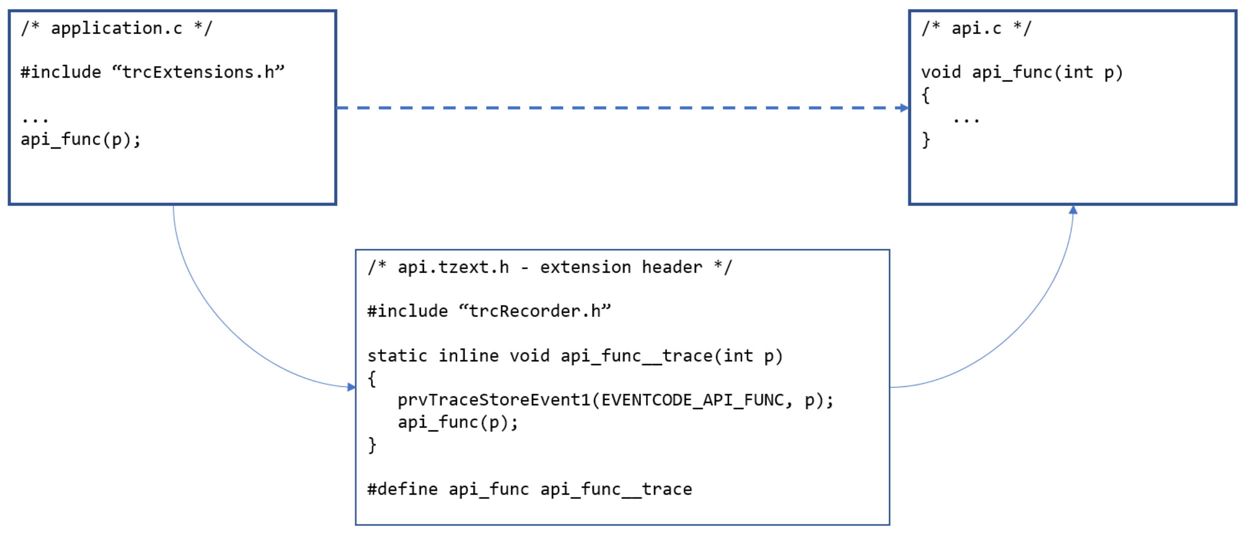
The include of “trcExtensions.h” can remain in the code base, if desired. This since the trace wrapper functions are available even if the tracing is disabled, but are then only calling the original function and will probably be eliminated by compiler optimizations.

How it Works

An extension is typically made for a specific API, i.e. a group of related functions, such as a device driver or middleware library, or a specific module of your application. As mentioned, each extension is defined by two files; the extension header file that defines the target-side tracing, and the host-side XML file with metadata that defines how Tracealyzer should interpret the data.

The automatic function tracing relies on preprocessor definitions in the extension header file, that redefines the selected function names to instead refer to a corresponding *trace wrapper function*. In the below illustration, the dashed line shows how the code appears to work, i.e. a direct call from *application.c* to *api.c*. The thinner solid lines shows what actually happens. By including “trcExtensions.h”, the compiler’s preprocessor will change all references of *api_func* to the corresponding trace wrapper function, *api_func__trace*. This is a small inline function defined in the extension header file, with the same declaration as the original function but with a suffix (“__trace”) added to the name.

The trace wrapper function calls the trace recorder library to store an event about the call, and also the original function. Any return value is propagated back, which makes it functionally transparent.



```
/* application.c */

#include “trcExtensions.h”

...
api_func(p);
```

```
/* api.c */

void api_func(int p)
{
    ...
}
```

```
/* api.tzext.h - extension header */

#include “trcRecorder.h”

static inline void api_func__trace(int p)
{
   prvTraceStoreEvent1(EVENTCODE_API_FUNC, p);
   api_func(p);
}

#define api_func api_func__trace
```

This works for regular function calls and also for function pointers/callbacks, since the initialization of the function pointer is also affected by the redefinition. And since the traced function is not modified, this even works with binary libraries and assembly functions.

The include of “trcExtensions.h” can remain in the code base, if desired. This since the trace wrapper functions are available even if the tracing is disabled, but are then only calling the original function and will probably be eliminated by compiler optimizations.

The extension system adds metadata to the trace header about the extensions used, such as the extension names and version numbers. This allows Tracealyzer to find the right XML files, with additional information such as symbolic event names and semantic information about the events, e.g. specifying that an event is an I/O operation, or a memory allocation. Tracealyzer expects to find the XML files in the Tracealyzer 4/cfg directory and the XML file name is expected to follow this following pattern:

```
<NAME>-v<VERSION_MAJOR>.<VERSION_MINOR>.<VERSION_PATCH>.xml
```

This versioning allows you to create multiple version of an extension and ensure that Tracealyzer uses the right XML definitions, even if viewing old traces recorded with earlier versions of the API/extension.

Extensions from different providers can be mixed in any order and combination, without conflicting event codes. This since the event codes are not hard-coded absolute values, like for the kernel events, but instead relative to a base code of each extension. The base code is set automatically by the preprocessor in compile time, based on the previously included extensions, and is included in the trace header so Tracealyzer can resolve them.

Known Limitations

The main limitation of this approach is that the function call redefinitions only works when the caller and callee are in different source files. It is possible to trace calls within the same file, but this may requires additional code changes, as described in the below section.

The reason for this, is that you must not include the extension header ("trcExtensions.h") in the .c file containing the body of a traced function, as the function body name is then redefined as well. This will make the compiler report errors about multiple definitions of the trace wrapper function, e.g. "myTracedFunction__trace". If you get this error, either remove the #include "trcExtension.h" from the particular file, or see the below section for how function call tracing can be performed within the same file.

A more general limitation is the tracing overhead. Since this is based on code instrumentation, there is a certain overhead of each traced call, typically only some microsecond per event, assuming a 32-bit MCU. However, if you trace too many function calls, this may noticeably affect the performance of your system. We therefore recommend to only trace the most important API calls to limit the tracing overhead.

Tracing Calls within the Same File

To avoid "multiple definition" errors caused by accidentally redefining the function body names, you need to disable the function name redefinitions in the files implementing traced functions.

This is done by adding #undef lines for each affected function, directly after the include of "trcExtensions.h", as shown below. This removes the problem, but also removes the automatic tracing of any calls to this function within the same file. Thus, to trace calls to this function from within the same file, you need to change the local calls manually to call the trace wrapper functions explicitly. This can remain in the code base, if desired. This since the trace wrapper functions are available even if the tracing is disabled but then only calling the original function, so they will probably be eliminated by compiler optimizations.

```
#include "trcExtensions.h"  /* Defines the trace wrapper etc. */
#undef myTracedFunction     /* Disable the function redefinition */
...
void myTracedFunction(int arg) /* Function body */
...
myTracedFunction__trace(42);   /* Traced call */
```

An alternative solution is to copy the function name redefinitions from the extension header file and insert them again after the corresponding function body.

```
#include "trcExtensions.h"  /* Defines the trace wrapper etc. */
#undef myTracedFunction     /* Disable the function redefinition */
...
void myTracedFunction(int arg) /* Function body */
...
#define myTracedFunction myTracedFunction__trace /* Reenable the tracing */
...
myTracedFunction(42);   /* Traced call */
```

Learning More

If you have not tried Tracealyzer before, visit https://percepio.com/download and sign up for an evaluation. An example trace in included, so you can begin exploring the capabilities of Tracealyzer directly.

Also make sure to check out our RTOS Debug Portal, with many articles and hands-on examples.

In case you have any questions, don't hesitate to contact us at support@percepio.com.