Percepio Detect™ Demo Guide

Version 2025.1 on Linux hosts, Apr 14, 2025

Percepio Detect provides Continuous Observability for embedded software during integration testing (CI) and system testing. This lets you capture crashes, anomalies and reliability risks in an automated way, and gain detailed insight on the causes.

Modern embedded software systems are often quite complex and many things can go wrong in runtime. Not all issues can be found by unit testing and code reviews. There is often a long tail of more tricky runtime issues showing up in later stages.

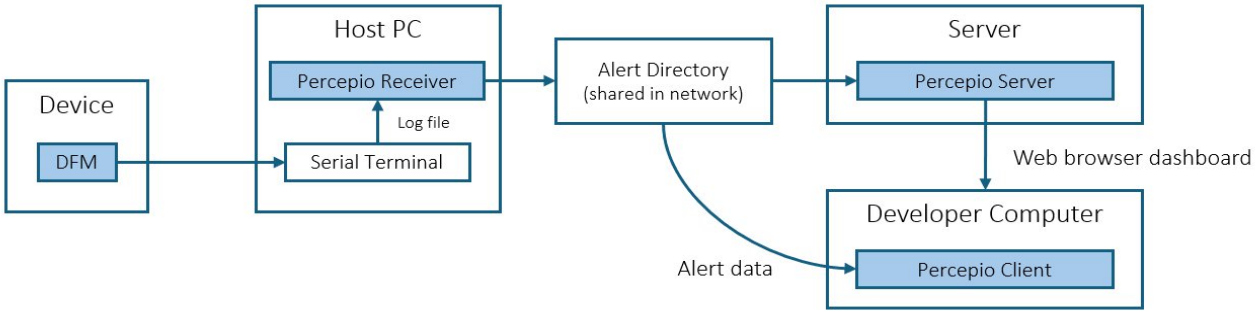
When issues occur outside of the debugging environment, you first need to reproduce the issue. This step can be very difficult and time-consuming, especially for sporadic issues.

Percepio Detect enables Continuous Observability, where monitoring of faults and anomalies is enabled early in the development, ”always on” and used throughout the development, testing, and optionally also in the field. This has the following benefits:

- Avoid the pains of issue reproduction. Turn ”nightmare bugs” into quick fixes by automatic capture of issues and debugging data at the first occurrence.
- Detect risks like ”near misses” and multi-threading issues early, reducing the risk of issues remaining in production code and the difficult debugging of such issues.
- Get a team dashboard on runtime issues with easy access to debugging data, so the right developers can start debugging right away - even remotely via company VPN.
- Debug on production boards without debug ports. A basic UART is sufficient.
- Monitor devices in the field, for example during field testing. The data can be saved on the device for later retrieval, enabling field use also for offline devices.
- Keep all sensitive device data and IP in your private network.

Solution Overview

The Percepio Detect solution consists of four parts, as described below.



- **Percepio DFM**: The main target-side component of Percepio Detect. This outputs "alerts" on faults and anomalies, including debugging data from other supporting libraries. In the demo, the data is written to the serial debug console in real time.

https://percepio.com

- **Detect Receiver**: Reads the device output, extracts the DFM data, converts it to the expected format and saves it as alert files for the Detect Server. This is a Python script and can be customized to accept any data encoding, e.g. Base64 or binary.
- **Detect Server**: Reads alert files from Receiver and presents a summary in the web browser (the "Dashboard"). Provides easy access to debugging data provided in the alerts, such as traces and core dumps.
- **Detect Client**: An integrated set of developer tools for debugging alerts, including Tracealyzer and tools for viewing core dumps. Runs on each user's computer to make it easy to debug reported issues.

Typical Setups

Percepio Detect is designed as a multi-user solution, where the server runs on a shared server in the internal developer network. However, the whole solution can be deployed on a single computer if desired, which is the setup used in the demo.

- **Single-user setup:** The Server, Client and Receiver runs on the same computer. Demo data is already provided in the test-data directory.
- **Multi-user setup:** The Server runs on a shared server. Each user runs the Client on their local development computer. The Receiver can run on any computer with access to the device output, for example a test computer with CI test runners.

Terminology

- Alert: A ”problem report” created by the DFM library on a device.
- Alert Type: The main type or reason for the alert, for example ”Hard Fault”.
- Symptom: A ”fingerprint” of an alert, such as the code location of a fault exception.
- Payload: Debugging data provided by an alert, e.g. traces and core dumps.
- Issue: A group of alerts with the same Symptoms and Alert Type.

Preparation Steps

- Make sure Docker Engine is installed. See https://docs.docker.com/engine/install/.
- Run "docker run hello-world" to see that Docker works as expected. It should download an image and start a Docker container that displays a message.
    - If you get a ”permission denied” error, you might need to add the ”docker” group to your user and reboot your computer.
    - You may also need to start the Docker service. See https://docs.docker.com/engine/daemon/start/

Running the Demo

After you have completed the preparation steps above, follow these steps to run the demo.

1. Extract the provided installation file (.tgz) to any suitable location on your computer.

2. Add your Detect license key in the server start script, **percepio-server.sh**. Locate the assignment of the LICENSE variable and update it like:

```
LICENSE="ABCD-ABCD-ABCD-ABCD"
```

Note that Tracealyzer and Detect Server have separate license keys, don’t mix them up!

3. Make sure the directory test-data/alert-files is accessible for writing also by “other” users. The Server runs as a different user and needs write access to the root of this folder.

```
chmod o+rw test-data/alert-files
```

4. Make sure the start scripts are executable by running:

```
chmod +x percepio-server/percepio-server.sh
```

```
chmod +x percepio-client/percepio-client.sh
```

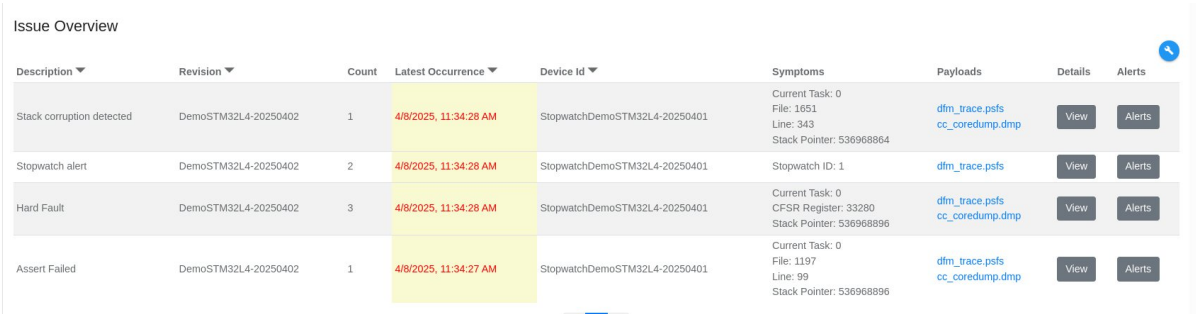
5. Enter the percepio-server folder and start the Server by running:

```
./percepio-server.sh start
```

6. Navigate to the percepio-client folder and start the Client by running:

```
./percepio-client.sh
```

7. Open http://127.0.0.1:8080 and check that you see the Server dashboard. It may take a few seconds for the Server to start up and load the demo data. The dashboard is updated every 5 seconds, but you may refresh the web browser manually for faster updates.



The "Issue Overview" shows a summary of all alerts in the Server database. Each row represents all alerts classified as the same ”Issue” since having identical Alert Type and Symptoms. This simplifies overview and analysis, in case of many reported alerts.

The most important columns are:

- **Revision:** shows the version of the device software that produced the alert.
- **Latest Occurrence:** shows the timestamp of the most recent alert for the Issue. The yellow highlighting indicate new alerts.
- **DeviceID**: An identifier of the device. This field can be overridden by the Receiver tool to specify a more descriptive name, for example including the current test suite name.
- **Symptoms**: The fingerprint of the issue, used for grouping Alerts into Issues.
- **Payloads:** Links to debugging data from the latest alert.
- **Details:** Shows additional information for the most recent alert, including the size of the payloads and the message string provided with the alert.
- **Alerts:** Shows all alerts of the same Issue. The Alerts page shows the complete list.

8. Lets start with classic crash debugging. Locate the ”Hard Fault” row and click the link to the provided core dump (”cc_coredump.dmp”). The core dump should now be displayed in the integrated core dump viewer, as shown below. (If not, see Troubleshooting below.)

```
=== Call Stack ===============
#0  0x08003e72 in MakeFaultExceptionByIllegalRead () at ../demo_alert.c:51
        r = <optimized out>
        p = 0x100000
#1  dosomething (n=n@entry=32) at ../demo_alert.c:61
No locals.
#2  0x08004550 in run_demo_command (
    buf=buf@entry=0x20003044 <demo_command> <error: Cannot access memory at addr
ess 0x20003044>) at ../main_baremetal.c:434
No locals.
#3  0x08004784 in main_superloop () at ../main_baremetal.c:212
        val = <optimized out>
        sum = 106
        count = 4
        stopwatch_alert_count = 2
#4  0x08004890 in main_baremetal () at ../main_baremetal.c:376
No locals.
#5  0x080041b6 in main ()
    at /home/johan/src/detect_basic_demo/STM32L475-Stopwatch/main.c:42
No locals.
Backtrace stopped: Cannot access memory at address 0x20017ffc


=== Source Code ==============
#0  0x08003e72 in MakeFaultExceptionByIllegalRead () at ../demo_alert.c:51
51          r = *p;
46          int r;
47          volatile unsigned int* p;
48
49          // Creating an invalid pointer)
50          p = (unsigned int*)0x00100000;
51          r = *p;
```

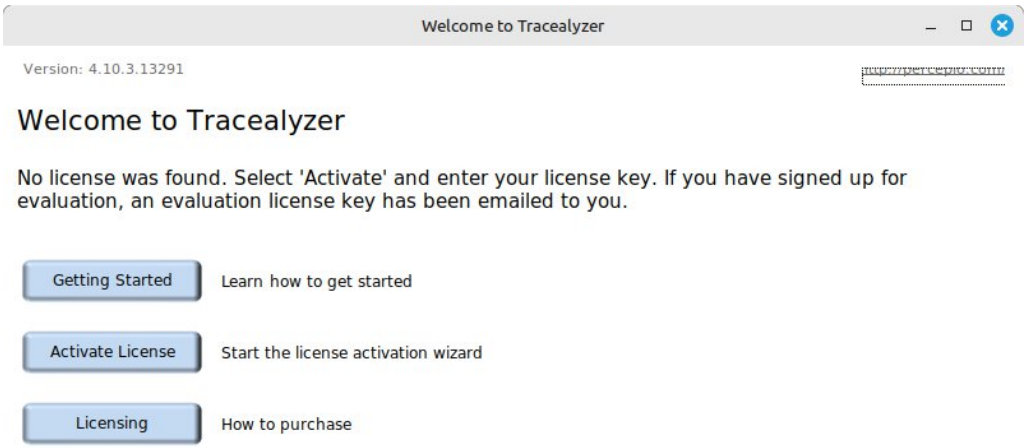
This alert was triggered by a hard fault exception in the device. The ”Call Stack” section shows the function that failed (the top one), the function arguments and the prior function calls in the current thread. The ”Source Code” section shows the source code at the fault location. The core dump viewer also shows other sections, like registers and disassembly.

The core dump solution is based on GDB and CrashCatcher, with Percepio improvements to enable more compact core dumps. The examples in the demo are only 556 bytes.

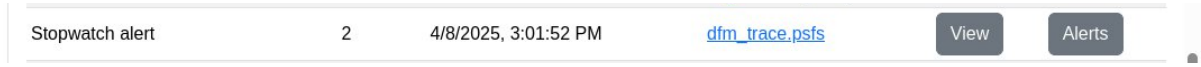
Core dumps are generated automatically on processor fault exceptions by the fault handler included in the DFM library, but can also be triggered by calling the DFM_TRAP() macro from your application-level fault handling code.

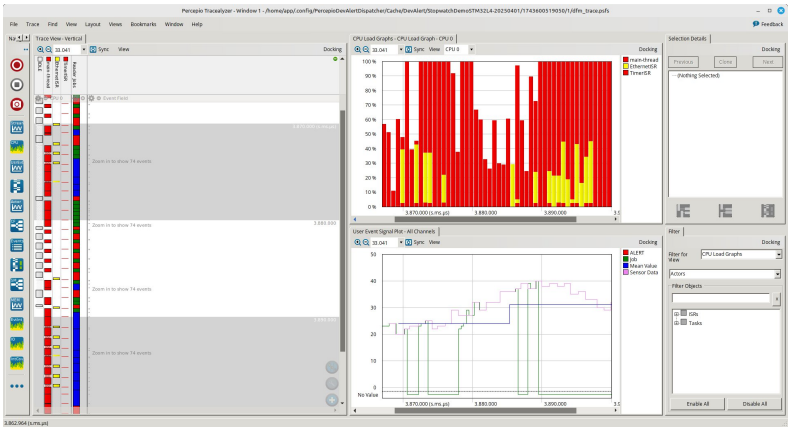
See also Fault Exceptions below for more information about this alert type.

9. Start Tracealyzer by clicking on a trace payloads (dfm_trace.psfs). On the first start, you need to enter your Tracealyzer license key. Select the option "Activate License", ”Activate key online” and enter your license key. Close the application.



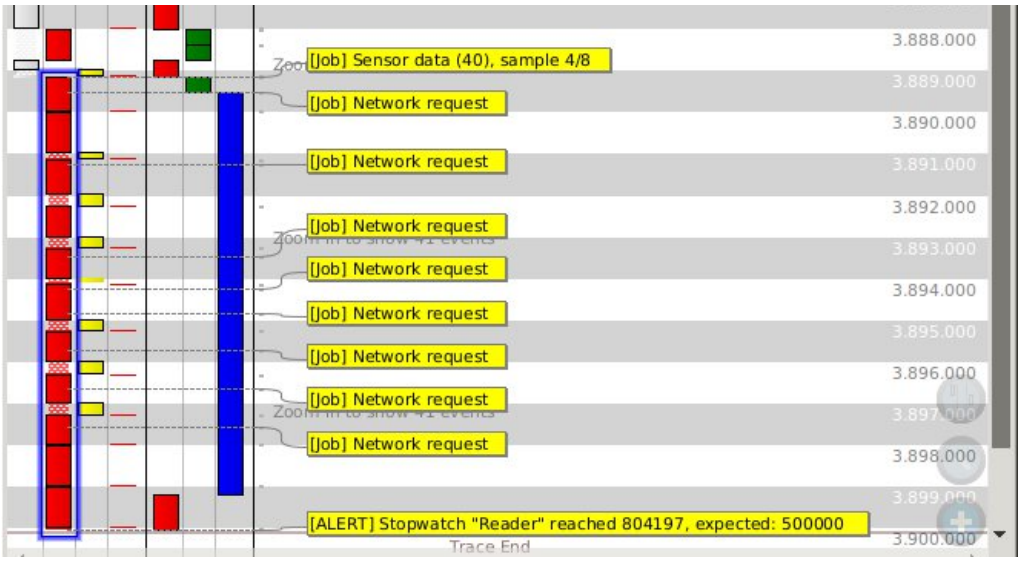
10. Now lets have a look at a Tracealyzer trace from the alerts. In the Dashboard, locate the ”Stopwatch alert” row and click the link.





This will open the trace in the Tracealyzer tool, included in the Detect Client. The trace data comes from the TraceRecorder library, that supports both RTOS kernel tracing and several kinds of application logging. The tracing is based on kernel instrumentation and logging calls. The events are written to a circular RAM buffer. On alerts, the trace buffer is included as a payload and shows the most recent events before the alert. Tracealyzer traces can be included with all alert types.

The trace from the ”Stopwatch alert” is generated by the Stopwatch monitoring feature in DFM, because Ethernet interrupts delayed the main thread longer than expected.



The yellow labels show ”User Events”, that are similar to printf calls but typically several orders of magnitude faster than printf calls over a slow UART. User event logging allows for adding additional details in the trace.

Initially you won’t see the yellow event labels in the trace view, because of the default zoom level. Zoom in a few steps using the ”magnifying glass” button in the upper left corner.

For clearer display, you can hide less relevant events using the Filter, found in the bottom right. Uncheck the category ”Notice Channels” to reduce the number of events displayed.

Note that the trace view uses a vertical time-line by default, where time flows downwards. The threads and interrupt handlers are shown in the left-most field, where each thread and interrupt handler (”actors” in Tracealyzer) is shown is a separate column.

In the middle, you see the ”Reader jobs” column. By default this is a collapsed into a single column, but has here been expanded using the small (+) button in the top. This column shows a TraceRecorder ”State Machine”, that is used to highlight the individual jobs executed in the thread. This can be used for tracing any kind of state transitions.

See also Stopwatch Monitoring below for more information about this alert type.

The Demo Alert Types

Hard Fault Alerts

Crashes are often detected by the processor, for example if an invalid memory address is used or random data is executed as code. This triggers a fault exception handler, that is it easiest form just restarts the device or contains a loop to halt the execution. The fault handler can also be used to log diagnostic information, but this is not always used to its full potential.

Percepio Detect can capture and report all types of Arm Cortex-M fault exceptions. This is enabled by installing the DFM fault handler (DFM_Fault_Handler) in the interrupt vector table. See also Step 8 above for an example.

Stopwatch Alerts

Stopwatch alerts are triggered when the time between two points in the code is longer than expected. This can be used to monitor response time requirements and to detect issues by their side-effect on timing. This is particularly useful on RTOS issues, like thread starvation.

Stopwatch alerts are implemented by inserting DFM function calls at the starting point and end point of the relevant sequence. The highest expected runtime is specified when initializing the stopwatch. If exceeded, an alert is emitted in ”End” function, but only if a new ”high watermark” has been found. This avoids redundant stopwatch alerts. Multiple stopwatches can be used in parallel and may span across different threads and interrupt handlers. To set the alert threshold, start with a high value (to avoid many alerts), run your tests and then inspect the High Watermark using the Stopwatch API.

You find the Stopwatch API in dfmStopwatch.h and this usage example in main_baremetal.c.

See also Step 10 above.

Stack Corruption Alerts

The DFM library includes support for GCC stack integrity checking. This requires using one of the *-fstack-protector* build flags. This adds a lightweight stack integrity check when returning from functions containing local buffers. If the stack is found to be corrupted, DFM will emit a ”Stack corruption detected” alert at the exit of the function. Note that this GCC feature increases the code size a bit due to the added checks.

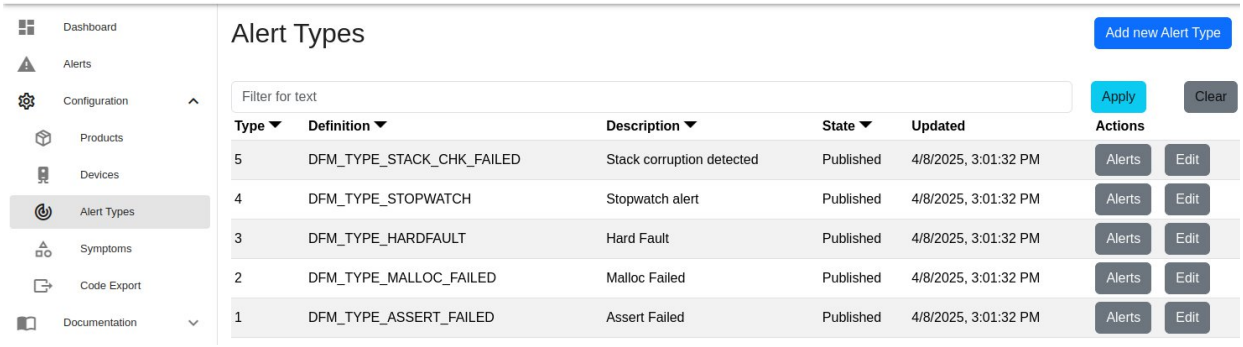
Assert Failed Alerts (User-Defined Alerts)

Assert statements are ”sanity checks” added in the code, often very useful for catching bugs in early phases. The demo project includes an assert macro (configASSERT) defined in trcConfig.h, that uses the DFM_TRAP macro to emit an alert. The DFM_TRAP macro saves a core dump with the stack trace, and optionally also a Tracealyzer trace. The DFM_TRAP macro can also be called directly from your application code for user-defined alerts.

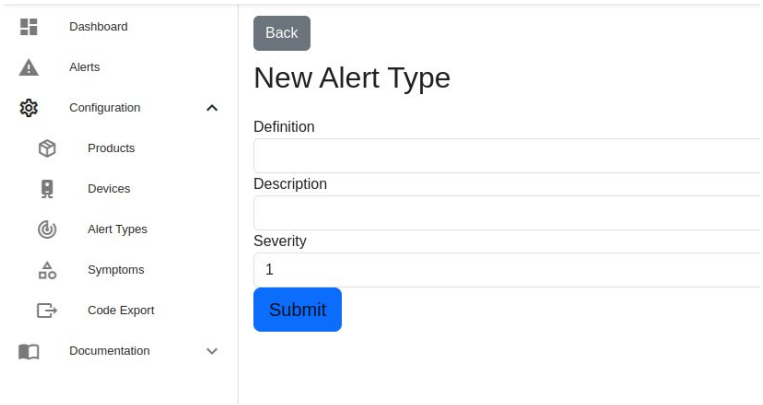
Adding New Alerts Types and Symptoms

You may add additional alert types and symptoms using the Server dashboard, under ”Configuration”. Their names are stored in the Server database and is represented by an integer ID in on the device side for efficiency reasons, specified in the DFM library.

Open the Alert Types page. Note the ”Status” column, where ”Published” means that the numeric ID is locked and can’t be deleted and reused with a different meaning. This avoids mismatching definitions in between firmware revisions.



To add a new entry, select ”Add new Alert Type”.



- Definition: The name of the numeric ID in DFM. Must start with ”DFM_TYPE_”.
- Description: The name displayed in the dashboard.
- Severity: This has no effect (not yet implemented).

After adding new entries, go to the ”Code Export” page to generate an updated dfmCodes.h. This workflow ensures matching definitions on device and server. You can then create alerts using your new Alert Type, for example like this:

```
DFM_TRAP(DFM_TYPE_ALERT_XYZ, ”Message”, 0)
```

The third parameter decides if DFM should restart the device (1) after creating the alert, or if to return and continue execution (0).

New Symptoms can be added in the same way on the Symptoms page, but is only needed if you create custom alerts using the DFM Alert API.

https://percepio.com

Troubleshooting

If you see this error message in the Client terminal window:

```
Authorization required, but no authorization protocol specified

Unhandled Exception: System.Exception: XOpenDisplay failed
```

This means that the Client (running in Docker) is not allowed to access the host display. In that case, please report this to Percepio together with your Linux setup. Then try the following alternative solution:

Install the xhost tool. On Debian/Ubuntu-based distributions, the xhost tool is found in the "x11-xserver-utils" package. Install this by “sudo apt-get install x11-xserver-utils”.

Then run

```
xhost +SI:localuser:$(whoami)
```

Close and restart the Client, click the payload link in the Dashboard again and verify that the debugging data is displayed. Add the command to the start script, before ”docker run”, or you may need to run the xhost command again after restarting your computer.

Learning More

More information on how to set up and customize Percepio Detect is found in readme.txt in root of the Percepio Detect package. You won’t need that to run the demo as-is, but make sure to read it before setting up your own customized solution.

If you have questions or feedback, please contact Percepio.