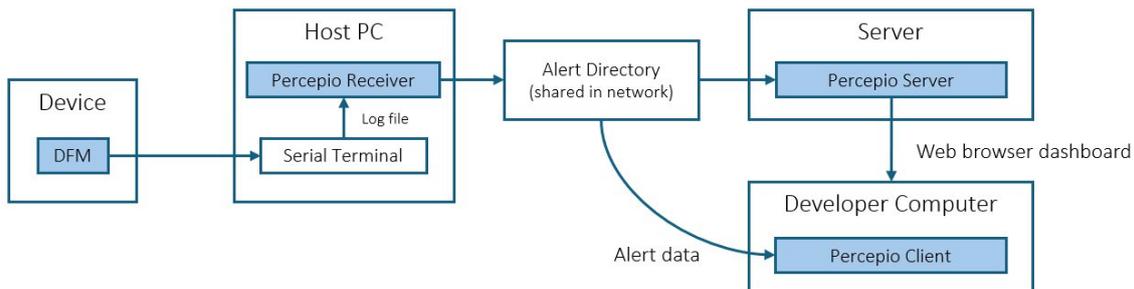


## Perceptio Detect – Device Integration Guide

Version 2025.1, March 2025.

Perceptio Detect is designed for systematic test monitoring and observability to provide instant insight on crashes or other system anomalies across multiple devices, e.g. in a test lab or during field testing. A typical multi-user setup for inhouse test monitoring is shown below, but you may also run the whole solution on a single computer.



Perceptio Detect consists of four parts:

- **Perceptio DFM:** The target-side C library and the focus of this document. DFM outputs "alerts" with debugging data. The same DFM library can also be used with Perceptio DevAlert for observability in deployment. With Perceptio Detect, the DFM data is typically sent to a local computer using a UART/serial port or similar direct connection.
- **Perceptio Receiver:** Reads the log files from the device (DFM), extracts the alert data and saves it as alert files for the Server. See `readme-receiver.txt` for details.
- **Perceptio Detect Server:** Reads alert files from Receiver, presents a summary in the web browser (the "Dashboard"), and provides access to alert payloads (e.g. traces and core dumps) for deeper analysis and debugging. See `readme-server.txt` for details.
- **Perceptio Detect Client:** An integrated set of tools for debugging alerts, including Tracealyzer and tools for viewing core dumps. Runs on each user's computer and responds to clicks on payload links in the Dashboard. Two versions of the client are provided, a Windows version in `perceptio-client-window` and a Linux version (using Docker) found in `perceptio-client-linux-docker`. See `readme-client.txt` for details.

### Processor support

The target-side client of Perceptio Detect consists of the DFM library and supporting libraries that provide debug data ("payloads") for the DFM alerts, for example TraceRecorder. The core parts of the DFM library are independent of the processor used, but the supporting libraries usually have hardware dependencies. At the time of writing, Perceptio provides two such supporting libraries:

- Core dump support for Arm Cortex-M devices. This is based on gdb and CrashCatcher, with Perceptio improvements to allow very small core dumps, less than 600 bytes in the demo.
- Tracealyzer support using Perceptio [TraceRecorder](#), supporting several processor families and extendable for any processor and RTOS using the [Tracealyzer SDK](#).

## RTOS support

The DFM library has minimal RTOS dependencies, isolated in the DFM “kernelport” module. This module is very small and easy to adapt for any RTOS. However, the Detect solution also uses two other libraries for providing debugging data, TraceRecorder and CrashCatcher.

Like DFM, the TraceRecorder library also has a “kernel port” module, providing RTOS instrumentation. There are kernelports available for various popular RTOSes in the github repository.

The demo uses the “Bare Metal” kernel port, a minimal and portable variant without RTOS awareness. This allows for application-level tracing with various event logging functions and can also be extended for full Tracealyzer kernel trace support using the [Tracealyzer SDK](#).

To use Percepio Detect with a different RTOS, you need to replace the following files:

- TraceRecorder:
  - o trcKernelPort.c
  - o trcKernelPort.h
  - o trcKernelPortConfig.h
  - o trcKernelPortSnapshotConfig.h
  - o trcKernelPortStreamingConfig.h
- DFM:
  - o dfmKernelPort.c
  - o dfmKernelPort.h

Note: For more advanced software platforms like Zephyr or ESP-IDF, with integrated configuration and build systems, and integrated core dump support, an adapted version of Percepio Detect can be preferable to facilitate integration. Contact [support@percepio.com](mailto:support@percepio.com) for more information.

CrashCatcher has no RTOS dependencies. Percepio has only tested it with FreeRTOS and bare metal applications so far, but it should work for any RTOS on Arm Cortex-M devices.

## Using the DFM Library

The device-side library, DFM, is provided as a C library under the Apache 2.0 license. DFM is intended to be called on errors and anomalies in the device and encodes the provided data into an “alert” packet for Percepio Detect or Percepio DevAlert.

Fault exceptions (crashes) are reported automatically, assuming that the DFM fault handler is installed as described later in this document.

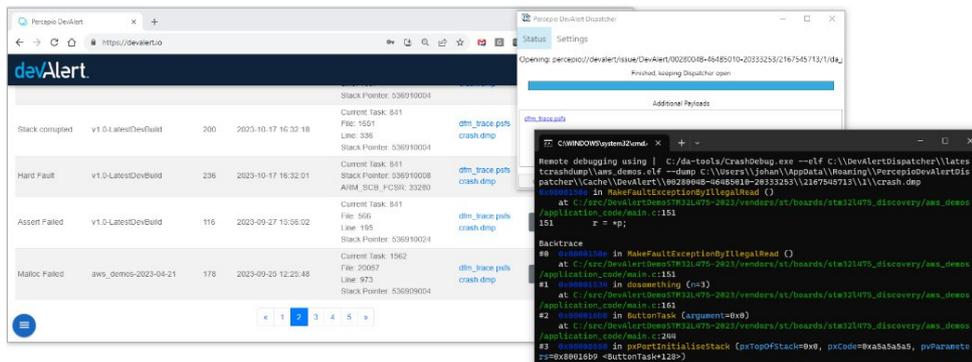
You may also create custom alerts from your code, during any kind of error handling. The easiest way is to call the **DFM\_TRAP macro**. The below example is from the STM32 demo, where an alert is created when pressing a button on the board.

DFM\_TRAP(DFM\_TYPE\_MANUAL\_TRACE, "Blue button pressed.", 0);

- Argument 1: The Alert Type ID, defined in dfmCodes.h.
- Argument 2: Alert description string, shown in “Details” in dashboard.
- Argument 3: Restart flag. If 1, the device is restarted by CrashCatcher after generating the alert. If 0, the execution continues without restart.

Alert Types are defined in the Detect dashboard (Configuration → Alert Types). There are some default types like DFM\_TYPE\_HARDFAULT, but you may also add your own alert types here. After adding new alert types, select Configuration → Code Export to generate a new dfmCode.h. This workflow ensures that your DFM definitions are in sync with the server database. Do not modify dfmCodes.h manually.

The real power of Percepio Detect comes from the “payloads”, such as core dumps and Tracealyzer traces, that allows for analyzing the cause of the alert. DFM\_TRAP adds such payloads automatically. A core dump payload is included by default. The TraceRecorder trace is also included if enabled in dfmCrashCatcherConfig.h.



To view the payloads, make sure that the Percepio Detect Client has been started on your local computer (contains the viewer tools) and then click the payload links in the Server dashboard.

The DFM library also includes support for **stack integrity checking** leveraging a GCC feature, the [-fs-tack-protector](#) flags. If you enable one of these build flags, e.g. `-fstack-protector-strong`, DFM will generate alerts if stack corruption is detected by the GCC-injected checks. Note that this GCC option may increase the footprint of your build.

Moreover, DFM includes a [Stopwatch feature](#) for generating automatic alerts (with a trace) if the time between `vDfmStopwatchBegin` and `vDfmStopwatchEnd` exceeds a specified limit. This can be used both for profiling purposes and for detecting multithreading issues. Key functions are:

- `dfmStopwatch_t* xDfmStopwatchCreate(const char* name, uint32_t expected_max);`

Initializes a stopwatch object and returns a handle used in following calls. This is only required once. By default, up to 4 stopwatch can be created, but the number can be changed in `dfmConfig.h` (`DFM_CFG_MAX_STOPWATCHES`).

The "expected\_max" argument is the threshold value in clock cycles. An alert is emitted if the time from `vDfmStopwatchBegin` to `vDfmStopwatchEnd` exceeds this value.

- `void vDfmStopwatchBegin(dfmStopwatch_t* sw);`

Starts the stopwatch. This function is very small, designed for minimal overhead.

- `void vDfmStopwatchEnd(dfmStopwatch_t* sw);`

Stops the stopwatch and checks the elapsed time since the last `vDfmStopwatchBegin` call on this stopwatch. An alert is emitted if the elapsed time exceeds the "expected\_max" limit provided to `xDfmStopwatchCreate()`.

Functions are also provided for printing statistics for the stopwatch, such as the high watermark. These can be used for basic profiling and to tune the "expected\_max" setting. See `dfmStopwatch.h` for details.

As an alternative to `DFM_TRAP`, you can also generate user-defined alerts with custom symptoms and/or payloads by calling the DFM Alert API. An example is shown below.

```
#include <dfm.h>
...
res = xDfmAlertBegin(DFM_TYPE_HARDFAULT, "Fault Exception", &xAlertHandle);
if (res == DFM_SUCCESS) {
    xDfmAlertAddSymptom(xAlertHandle, DFM_SYMPTOM_PC, stackframe->pc);
    xDfmAlertAddSymptom(xAlertHandle, DFM_SYMPTOM_STACKPTR, stackframe->sp);
    xDfmAlertEnd(xAlertHandle);
}
```

The first argument to `xDfmAlertBegin()` is the alert type, followed by a message string and finally a pointer where to store the resulting *alert handle*. The alert handle is then used in additional calls where more information is added to the alert. Finally, `xDfmAlertEnd` is called to finalize the alert. The data is then stored and/or transmitted, depending on the DFM configuration.

The `xDfmAlertAddSymptom` calls are used to create a "fingerprint" that characterizes the reported issue, provided as metadata in the alerts. This information is used to group identical alerts into "issues", displayed in the Server dashboard (Issue Overview). This provides deduplication, meaning you don't need to inspect each alert individually but can consider all alerts of the same "issue" as repetitions of the same problem.

Each piece of fingerprint data is called a “symptom” and may include for example program counter, stack pointer, and selected variable values. If using DFM\_TRAP, default symptoms are included automatically.

## DFM Device Integration

Note that Percepio provides demo projects for certain processors where these steps are already taken care of, but the following guide assumes an integration from scratch.

1. Make sure you can print text to a serial port or similar and receive it on the host computer in a terminal program, and log the data to a text file. If using Windows, a suitable configuration for TeraTerm is described in **Step 1. Serial Terminal Logging**.
2. Integrate the DFM library as described in **Step 2. DFM Library Integration**.
3. For Arm Cortex-M core dump support, you need CrashCatcher integrated with DFM. This is described in **Step 3. Collecting Core Dumps with CrashCatcher**.
4. To capture Tracealyzer traces in your alerts, integrate the TraceRecorder library in your project as described in **Step 4. Collecting System Traces with TraceRecorder**.

### Step 1. Serial Terminal Logging

To get the DFM data from the device to host, it is recommended to use a serial terminal program with logging capabilities.

On Windows, TeraTerm is a convenient solution. On Linux, gtkterm offers a similar experience (see below).

To configure **TeraTerm** to receive DFM data over a serial connection (COM port), select Setup -> Serial Port. Select the right COM port (usually the last if your device was recently plugged in) and the Speed.

Verify the UART baud rate in the demo project. If using the demo projects as starting point, the UART speed is either 115200 or 1000000.

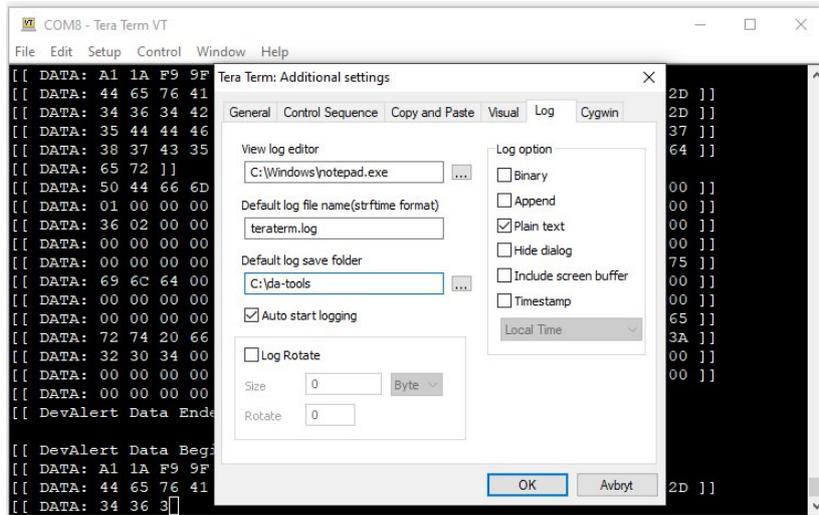
- **STM32U585 demo project:** The default baud rate is 1 MHz (1000000) and is set in console.c. Higher speeds are possible since using the fast VCOM of the STLINK v3 on this board, but occasional transmission errors have been noted if pushing the limit to 4 MHz.
- **STM32L475-Basic demo project:** The default baud rate is 115200, but can be increased in in main.c, in the Console\_UART\_Init function. 1 MHz should be safe. The maximum speed seems to be a bit over 2 MHz, but pushing the limit may cause occasional transmission errors.
- **STM32L475-Stopwatch demo project:** The default baud rate is 1 MHz (1000000), but can be increased in in main.c, in the Console\_UART\_Init function. The maximum speed seems to be a bit over 2 MHz, but pushing the limit may cause occasional transmission errors.

The other settings are usually fine as is.

Next, run your embedded application and make sure the output is presented correctly.

You can now start the logging by selecting File -> Log... and stop the logging by selecting File -> Show Log Dialog -> Close.

To start the TeraTerm logging automatically, select Setup -> Additional settings -> Log and configure the settings like below:



Default log save folder: Where to save the log file.

Auto start logging: Checked

Log Rotate: Unchecked

Append: Unchecked

Finally, select Setup -> Save setup and overwrite the default settings file (TERATERM.INI).

If using Linux and want to use **gtkterm** to receive the DFM data:

- Install gtkterm. On Debian distributions using apt: “sudo apt install gtkterm”.
- Start gtkterm and ignore any warnings from the default configuration.
- Select Configuration → Port. The STLINK Virtual COM port is usually found at /dev/ttyACM0. Select the right baud rate (115200 or 1000000, see last page).
- Enable Configuration → CR LF Auto
- Select Log → To File... and specify the log file name.
- Save your configuration using Configuration → Save Configuration, for example as “Detect”. You can now start gtkterm with this setup by running “gtkterm -c Detect”

## Step 2. DFM Library Integration

2.1. Copy the .c source code files from the DFM root folder into your project and ensure they are included in the build.

2.2. Copy all header files from the **include** and **config** directories to a suitable “include” directory where other header files for your project are found.

2.3. Add cloudports/Serial/dfmCloudPort.c to your project. This module specifies how to output the data.

2.4. Also copy the header files from `cloudports/Serial`, i.e., **include** and **config** directories, to the same “include” directory as in the previous step. The name “cloudport” is a bit misleading in this case, as DFM was originally designed for cloud upload with Percepio DevAlert.

2.5. Add `storageports/Dummy/dfmStoragePort.c` to your project. This module specifies how to store alert data on the device, but storage is not needed in this case. Also copy `trcStoragePort.h` from the local **include** directory to the same “include” directory as in the previous step.

2.6. Next, have a look in the **kernelport** directory. If there is kernelport module matching your RTOS, use that, otherwise select **Generic** kernelport. Add `dfmKernelPort.c` from the selected kernelport directory to your project. Copy the header files from the local **include** and **config** directories to the same “include” directory as in the previous step.

2.7. Open **dfmConfig.h** and update these settings:

- `DFM_CFG_PRINT(msg)`: Should provide a function call for printing to the serial port, e.g. `printf(msg)`, `puts(msg)` or similar.
- `DFM_CFG_PRODUCTID`: Set to 1 to match the “Default Product” in the Server.
- `DFM_CFG_ENABLE_DEBUG_PRINT`: Set this to 1 to enable error messages from the DFM library to be printed in the serial terminal.

2.8. Add a call to `xDfmInitializeForLocalUse()` in the startup, for example in the `main()` function, right after the initialization of the console serial output.

Note that `xDfmInitializeForLocalUse()` will apply special dummy values for the “Session ID” and “Device ID” alert fields that are replaced by the Receiver tool. In this case, the Session ID is based on a host-side timestamp, and the Device ID can be specified in the Receiver start script. If using other values for `DeviceID` or `SessionID`, they will remain.

### Step 3. Adding Core Dumps

DFM allows for saving device data as “payloads” when creating alerts. If you compare an Alert with an email, the payloads are like attachments. These payloads may contain any data and may provide data from other diagnostic libraries such as `CrashCatcher` and `TraceRecorder`.

`CrashCatcher` lets you collect core dumps, include registers, stack and other memory contents on Arm Cortex-M devices.

The Percepio Payload Viewer Client includes tools for viewing `CrashCatcher` core dumps. Percepio also provides an improved version of the `CrashCatcher` library that allows for very small core dumps relative to the current stack pointer. By default, only the top-most 300 bytes of the stack is saved. If the current stack depth is deeper, the earliest functions won’t be included but the recent call stack can still be displayed. The dump size is configurable in **dfmCrashCatcherConfig.h**.

Core dumps can be provided both on fault exceptions (e.g. on invalid memory access) and when calling `DFM_TRAP()` in your code.

The provided device demos already integrate the `CrashCatcher` library, but to integrate `CrashCatcher` from scratch in your own code, follow these steps:

3.1. Get the `CrashCatcher` library from one of the `device-integration/libraries` folder.

3.2. Copy **Core/src/CrashCatcher.c** into your project and make sure it is included in the build.

3.3. Copy **CrashCatcher\_armv7m.S** into your project, if using Arm Cortex-M3 or higher. For Cortex-M0 devices or other older devices, use **CrashCatcher\_armv6m.S** instead.

3.4. Copy **Core/src/CrashCatcherPriv.h** and **Include/CrashCatcher.h** to a suitable “include” directory in your project where your compiler will find them.

3.5. Next, we need to hook in the **CrashCatcher fault handler** so it is called on fault exceptions.

3.5.1. Locate the interrupt vector table in your startup code. In the demo projects, this is found in [startup\\_stm32l475xx.s](#) (or similar) in the ST directory.

3.5.2. Modify the fault exception vectors to call **DFM\_Fault\_Handler** instead of the original fault handlers (e.g. **HardFault\_Handler**, **BusFault\_Handler**, etc.).

3.5.3. Do the same for the NMI handler. This is needed by the **DFM\_TRAP()** macro.

3.6. Review and update the **DFM\_CFG** settings in **dfmCrashCatcherConfig.h**, in particular **DFM\_CFG\_ADDR\_CHECK\_BEGIN** and **DFM\_CFG\_ADDR\_CHECK\_NEXT**:

- **DFM\_CFG\_ADDR\_CHECK\_BEGIN**: The start address of RAM, where stacks are stored.

- **DFM\_CFG\_ADDR\_CHECK\_NEXT**: The start of the next address range, where NOT to read memory. For example, a reserved address range after the RAM.

Since the stack dumps start at the current stack pointer and normally span a fixed number of bytes upwards, this check is needed to avoid reading outside the valid address range if the stack pointer is near the end. Stack dumps are truncated at **ADDR\_CHECK\_NEXT - 1**.

3.7. Open **config/dfmCrashCatcherConfig.h** and set **DFM\_CFG\_CRASH\_ADD\_TRACE** to 0 to begin with. This disables collection of TraceRecorder traces for Tracealyzer. This can be enable later, once TraceRecorder has been integrated.

3.8. To test the core dump feature, add a call to **DFM\_TRAP()** in your code to trigger a core dump.

For example:

```
#include <dfm.h>
#include <dfmCrashCatcher.h>
...
DFM_TRAP(DFM_TYPE_ASSERT_FAILED, "My Core Dump", 0); // 0 = Don't restart.
```

3.9. Run your application and verify that you get DFM data in the serial terminal.

**Note:** If you don't see any output, it might be that execution is stopped by your debugger on entering fault handlers. In that case, try resuming the execution. Many debuggers have a setting like “**Halt on Exception**” or similar. We recommend disabling that.

3.10. Process the output log using **Percepio Receiver**, as described in **readme-receiver.txt**.

## Step 4. Adding TraceRecorder traces

[Percepio Tracealyzer](#) is an advanced visual analysis tool for event traces, that is included in the Percepio Payload Viewer Client. The trace data collection is done using the TraceRecorder library.

This can provide short traces (snapshots) as alert payloads. This way, you can collect detailed traces showing the software activity just before the issue was detected.

TraceRecorder supports several popular real-time operating systems such as FreeRTOS, Zephyr, ThreadX, PX5 as well as bare metal systems. The latter can be extended with custom instrumentation for any RTOS using the [Tracealyzer SDK](#).

4.1. To get started with Tracealyzer, you can sign up for a free evaluation license at <https://percepio.com/tracealyzer/download-tracealyzer/>

4.2. Follow the integration guide at <https://percepio.com/tracealyzer/gettingstarted/>, corresponding to your RTOS. For Bare Metal setups, an additional guide is provided in the device-integration folder found in the Percepio Detect package.

4.3. Make sure to select the **RingBuffer** stream port, which is designed for taking snapshots of the most recent trace data.

4.4. In `dfmCrashCatcherConfig.h`, set `DFM_CFG_CRASH_ADD_TRACE` to 1.

4.5. Run your application with a `DFM_TRAP` alert sometime after the `xTraceEnable` call (to collect some trace data) and make sure you get DFM data output.

4.6. Make sure the output is processed by Percepio Receiver as described in `percepio-receiver/readme-receiver.txt`, and that the alert shows up in the Server Dashboard. Make sure that the Percepio Client is running and click on the “.psfs” payload. This will display the trace using the Tracealyzer tool, included in the Client.

If you have not already installed your Tracealyzer license, you need to do that at the welcome screen and then restart the application.

If you want to implement custom alerts (rather than using `DFM_TRAP`), you can add the trace data as a DFM payload in the following way:

```
void* pvTraceData = (void*)0;
uint32_t ulTraceDataSize = 0;
...
xTraceDisable();
xTraceGetEventBuffer(&pvTraceData, &ulTraceDataSize);
xDfmAlertAddPayload(xAlertHandle, pvTraceData, ulTraceDataSize, "trace.psfs");
xTraceEnable(TRC_START);
```

It is recommended to pause or stop the tracing before transmitting the alert, by calling `xTraceDisable()`. New data must not be added to the trace buffer while sending or storing the alert as this is likely to cause issues.

To restart the tracing after the alert, call `xTraceEnable(TRC_START)`. This resets TraceRecorder and enables the tracing. Alternatively, use `xTracePause()` and `xTraceResume()`. This does not reset TraceRecorder, meaning that the traces can show events spanning over multiple alerts if they occur close in time. You may also consider adding a User Event (e.g. `xTracePrintf`) to mark the alert in the trace, just before calling `xTraceDisable()`. This may log any information of relevance.

## Learning More

The host-side setup for Percepio Detect is described in `readme.txt` in your Detect directory.

If you have questions or feedback, please contact Percepio at <https://percepio.com/contact-us>.