

## Percepio Detect – Device Integration Guide

Version 2025.2, October 2025.

Percepio Detect is designed for systematic test monitoring and observability to provide instant insight on issues across multiple devices, e.g. in a test lab or during field testing.

Percepio Detect consists of four parts:

- **DFM:** The target-side C library of Percepio Detect and the focus of this document. This outputs “alerts”, i.e. machine-readable error reports with debugging data included.
- **Receiver:** Reads log files from the device (DFM) and saves the DFM data as alert files for ingestion by the Detect Server. See `readme-receiver.txt` for details.
- **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.
- **Client:** An integrated set of debugging tools for alert data, including Tracealyzer for viewing TraceRecorder traces and a core dump viewer. See `readme-client.txt` for details.

### Processor support

The target-side part of Percepio 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:

- Core dump support for Arm Cortex-M devices. This is based on gdb and CrashCatcher, with Percepio improvements for more efficient stack dumps.
- Tracealyzer support using Percepio [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.

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.

The TraceRecorder library has “kernelport” modules for each supported RTOS kernel. The “Bare Metal” kernelport is a minimal and portable variant without RTOS awareness, that can be extended for full Tracealyzer kernel trace support using the [Tracealyzer SDK](#).

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

- TraceRecorder:
  - `trcKernelPort.c`, `trcKernelPort.h`
  - `trcKernelPortConfig.h`
- DFM:
  - `dfmKernelPort.c`, `dfmKernelPort.h`

## Using the DFM Library

The device-side library, DFM, is provided as a C library under the Apache 2.0 license. This monitors the device, detects errors/anomalies and outputs “alert” messages on each detected issue.

DFM alerts are machine-readable error reports, containing metadata about the issue and debug data captured at the error, including a small core dump with the call-stack trace, as well as a TraceRecorder trace providing the most recent events. DFM provides several methods for detecting and reporting system anomalies:

- [Fault Exceptions](#) are captured and reported automatically, for Arm Cortex-M devices.
- [Stopwatches](#) offer latency profiling and capturing issues as response time anomalies.
- [The TaskMonitor](#) lets you profile CPU load and catch difficult bugs as CPU load anomalies.
- [Stack corruption](#) can be detected on e.g. stack smashing attacks and buffer overrun bugs.
- [Custom alerts](#) can be generated from your code, e.g. in user-defined error handlers.

For more details and examples, please refer to the [Percepio Demos repository on Github](#).

## 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. See **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, you may use a serial terminal program with logging capabilities or integrated logging capabilities in the IDE.

IAR EWARM users are recommended to use the integrated ITM logging support, as [described here](#).

Otherwise TeraTerm is a convenient solution on Windows, and gtkterm offers a similar experience on Linux (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. The Percepio Demos use 115200 baud by default.
- Run your embedded application and make sure the output is presented correctly.
- 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).

To use **gtkterm** on Linux:

- Install gtkterm. On Debian distributions using apt: “sudo apt install gtkterm”.
- Start gtkterm and ignore any warnings from the default configuration.
- Select “Configuration” -> “Port”. Select the right tty/COM port (often /dev/ttyACM0) and the right baud rate (usually 115200 baud).
- Enable “Configuration” -> “CR LF Auto”
- Select “Log” -> “To File...” and specify the log file name.

Optionally, save your configuration using “Configuration” -> “Save Configuration”. You can now start gtkterm with this setup by running “gtkterm -c <name>”

## 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. Also add either **dfmCoreDump-GCC.S** or **dfmCoreDump-IAR.S** to your project.

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. Select which [cloudport module](#) to use. This module specifies how to output the DFM data, for example via printf calls to a debug console (“Serial”) or as raw binary data over ITM/SWO (“ITM”).

2.4. Add the corresponding **dfmCloudPort.c** to your project, and the corresponding header files from **include** and **config** directories.

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.

### Step 3. Adding Core Dumps

The CrashCatcher library lets you collect core dumps, including registers, stack and other memory contents on Arm Cortex-M devices. CrashCatcher core dumps are provided as DFM alerts both on fault exceptions (e.g. on invalid memory access) and when calling `DFM_TRAP()` in your code. The resulting alert payloads are displayed using the integrated core dump viewer in the Detect Client.

To integrate CrashCatcher in your own Detect project, follow these steps:

3.1. Get the Percepio version of the CrashCatcher library from the [PercepioLibs folder](#).

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

3.3. If using GCC and Arm Cortex-M3 or higher, copy **CrashCatcher\_armv7m.S** into your project. For Cortex-M0 devices, use `CrashCatcher_armv6m.S` instead.

If using **IAR Embedded Workbench**, include the `.c` file **CrashCatcher\_armv7m.c** instead of the `.S` files.

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. Locate the interrupt vector table and replace all fault handlers (like `HardFault_Handler`) with **DFM\_Fault\_Handler**.

3.6. Review and update the `DFM_CFG` settings in **dfmCrashCatcherConfig.h**, in particular:

- **DFM\_CFG\_ADDR\_CHECK\_BEGIN**: Start address of RAM (i.e. where stacks are stored).
- **DFM\_CFG\_ADDR\_CHECK\_NEXT**: Often there is a reserved memory range after the RAM, that DFM must not include when dumping the stack. Then specify the first reserved address here, otherwise `0xFFFFFFFF`.

### Step 4. Adding TraceRecorder traces

[Percepio Tracealyzer](#) is an advanced visual analysis tool for event traces, that is included in the Percepio Detect 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.

4.1. 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.

- 4.2. Make sure to select the **RingBuffer** stream port.
- 4.3. Add a test call to DFM\_TRAP, [like in this example](#).
- 4.4. Run your application and capture the DFM output to a log file, like [explained here](#).
- 4.5. Open a terminal and run the Receiver tool on the log file, as described in percepio-receiver/readme-receiver.txt.
- 4.6. An alert including a trace payload (dfm\_trace.psfs) should now appear on the Server dashboard. Make sure that the Percepio Client is running and click the payload link. On first start, you will need to enter your Tracealyzer license key and then restart the application (i.e. click the link again).

**Notes:**

- Optionally, you may add [DFM\\_STACK\\_MARKER\(\)](#); at the start of each task. This stops the stack dump from including unnecessary data, minimizing the dump size and avoiding gdb warnings.
- If you don't see any output, many debuggers have a setting like **"Halt on Exception"** that halts execution at the first instruction of the fault handler. Resume execution to see the DFM output.
- By default, DFM core dumps include at most 300 bytes of the current stack. The max dump size is configurable in dfmCrashCatcherConfig.h.

## 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>.

## Appendix A - Custom Alerts, Payloads and Symptoms

As an alternative to DFM\_TRAP, you may 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.