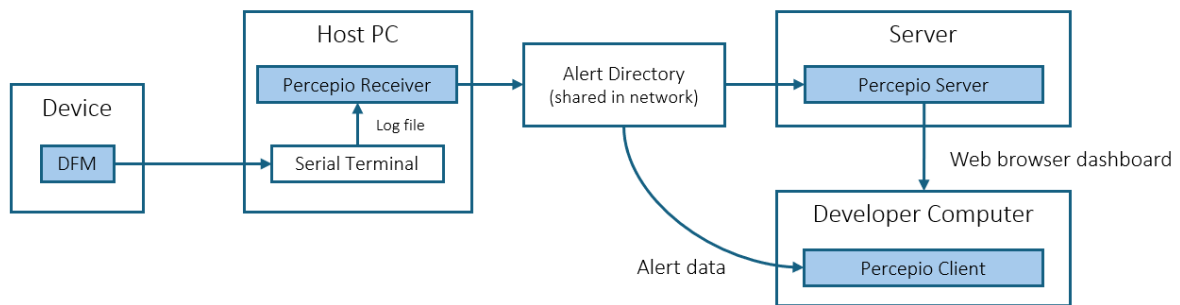


Percepio Detect – Device Integration Guide

Version 2024.3, December 2024.

Percepio Detect is designed for systematic test monitoring and observability for one or multiple devices to provide insight on crashes or other system anomalies. This can be used for multiple devices in a test lab, or used by an individual developer on a single device.



Percepio Detect consists of four parts:

- **Percepio DFM**, a C library for use in the device software and the focus of this document. DFM outputs "alert data" with diagnostic information. The same DFM library can also be used with Percepio DevAlert for observability in deployment. With Percepio Detect, the DFM data is typically sent to a local computer using a UART/serial port or similar, together with other device output and saved in a log file.
- **Percepio Receiver** (or "Receiver" for short). Reads the log files from the device (DFM), extracts the alert data and saves it as alert files for the Server.
See `percepio-receiver/readme-receiver.txt` to learn more.
- **Percepio Detect Server** (or "Server" for short). 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 `percepio-server/readme-server.txt` to learn more.
- **Percepio Detect Client** (or "Client" for short). 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 `percepio-client-window` and a Linux version (using Docker) found in `percepio-client-linux-docker`. See `readme-client.txt` in the respective directory to learn more.

Processor support

The target-side client of Percepio Detect, the DFM library, can be used with any embedded processor and is optimized to fit in 32-bit microcontrollers. The core parts of DFM are processor-agnostic and can be extended with more processor-specific modules for collecting specific diagnostic data, such as core dumps and event traces. At the time of writing, Percepio provides two such diagnostic modules:

- Core dump support for Arm Cortex-M devices, based on [CrashCatcher](#).
- Tracealyzer support using Percepio [TraceRecorder](#), supporting several processor families and extendible 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 makes it easy to integrate DFM with essentially any RTOS.

CrashCatcher has no RTOS dependencies, although Percepio has only tested it with FreeRTOS and bare metal applications so far.

For RTOS kernel tracing, the TraceRecorder library requires RTOS-dependent instrumentation. Percepio provides such support for various popular RTOSes.

You may also choose to integrate TraceRecorder using the “Bare Metal” option, without RTOS dependencies. This is found in the TraceRecorder/kernelports folder. This way, you can use all RTOS-independent logging features in TraceRecorder on any C/C++ software. This can also be extended for user-defined kernel tracing using the [Tracealyzer SDK](#).

For more advanced RTOSes like Zephyr, a pre-integrated DFM solution for your RTOS can be preferable. For example, Zephyr provides the kconfig system that can simplify the integration. DFM is included in the Zephyr repository, but that is not covered in this document. Contact support@percepio.com for updated information on Zephyr and other RTOS support.

Using the DFM Library

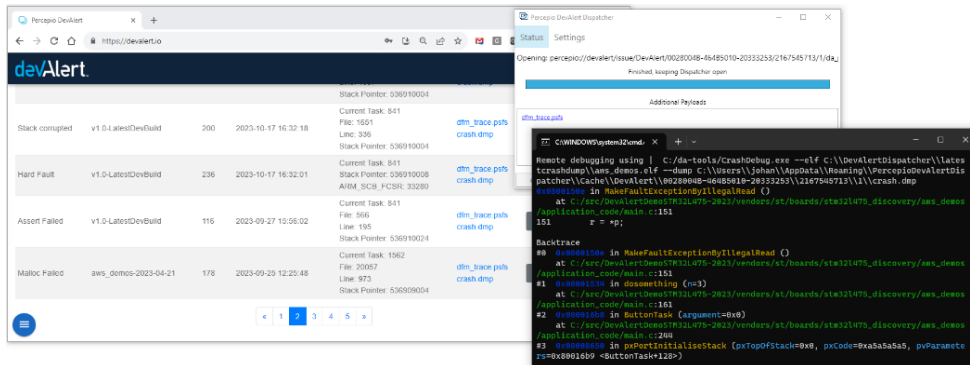
The device-side library, DFM, is provided as a C library under the Apache 2.0 license. DFM is intended to be called on errors and anomalies in the device and encodes the provided data into an “alert” packet for Percepio Detect or Percepio DevAlert. A code example is shown below.

```
#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 an arbitrary 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 alert types are defined in **dfmCodes.h**. This header file should be **generated** using the Server console and should not be modified manually to ensure the definitions stay in sync with the database definitions. You can define your own alert types in the Server console under Configuration -> Alert Types and then generate a new dfmCodes.h under Configuration -> Code Export.

The real power of Percepio Detect comes when including “payloads” such as core dumps, logs and system traces. To view the payloads, you simply click the links in the Server dashboard. This notifies the Percepio Payload Viewer Client on your local computer, that shows the payload data and in a suitable tool.



The **xDfmAlertAddSymptom** calls are used to create a “fingerprint” that characterizes the reported issue. The fingerprint is used to group the individual alerts into “issues” displayed in the Server dashboard. This way, if you get many alerts, you don’t need to inspect each alert individually but can consider all alerts within the same “issue” as repetitions of the same problem, assuming the provided fingerprint data is sufficient to accurately characterize the issue.

Each piece of fingerprint data is called a “symptom” and may include for example program counter, stack pointer, and selected variable values.

```
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);

    xDfmAlertAddPayload(xAlertHandle, logdata, logdatasize, "log.txt");
    xDfmAlertAddPayload(xAlertHandle, coredumpdata, coredumpsizesize, "coredump.bin");

    xDfmAlertEnd(xAlertHandle);
}
```

DFM also includes the **DFM_TRAP** macro that creates an Alert in a single line of code. This is provided as part of the Arm Cortex-M core dump support. This uses the NMI exception handler to call CrashCatcher. Here is an example from the demo project, where an example Alert is generated when a button is pressed.

```
DFM_TRAP(DFM_TYPE_MANUAL_TRACE, "Blue button pressed.", 0);
```

- Argument 1: The alert type, defined in dfmCodes.h.
- Argument 2: Alert description string, shown when selecting “Details” in dashboard.
- Argument 3: Restart flag. If 1, the device is also restarted. If 0, the execution continues.

The DFM library also includes support **stack integrity checking** using a GCC feature. This requires using a suitable build flag, e.g. *-fstack-protector-strong*. Currently the Server does not have this definition included by default, so you need to add an Alert Type for this purpose (Configuration -> Alert types) and update the definition of DFM_TYPE_STACK_CHK_FAILED in dfmCodes.h.

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. [An example is provided in the Stopwatch demo](#). The demo provide commands in the serial terminal like “poll on” to inspect the Stopwatch state, in particular the “high watermark” value. Type e.g. “load 4” to make the system exceed the expected duration and emit alerts. The lower value for “load”, the more frequent interrupts and higher runtimes.

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, but there are many alternatives that often can be configured in similar ways.

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.

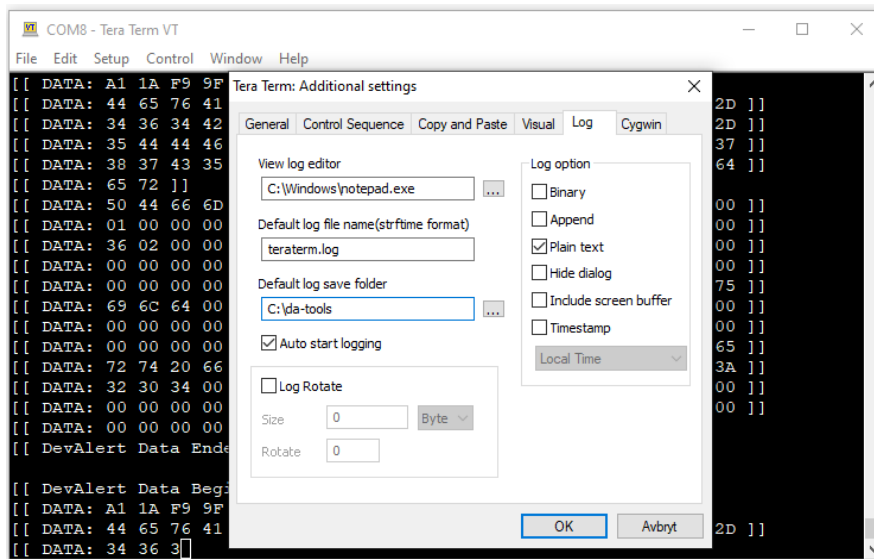
- **STM32U585 demo project:** The default baud rate is 1 MHz (1000000) and is set in console.c. Higher speeds are possible since using the fast VCOM of the STLINK v3 on this board, but occasional transmission errors have been noted if pushing the limit to 4 MHz.
- **STM32L475-Basic demo project:** The 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).

Step 2. DFM Library Integration

Namn	Senast ändrad	Typ	Storlek
cloudports	2023-11-07 10:27	Filmapp	
config	2023-11-07 10:27	Filmapp	
include	2023-11-07 10:27	Filmapp	
kernelports	2023-11-07 10:27	Filmapp	
storageports	2023-11-07 10:27	Filmapp	
dfm.c	2023-11-07 10:27	C Source	2 kB
dfmAlert.c	2023-11-07 10:27	C Source	18 kB
dfmCloud.c	2023-11-07 10:27	C Source	5 kB
dfmCrashCatcher.c	2023-11-07 10:27	C Source	10 kB
dfmEntry.c	2023-11-07 10:27	C Source	25 kB
dfmSession.c	2023-11-07 10:27	C Source	15 kB
dfmStorage.c	2023-11-07 10:27	C Source	6 kB
README.txt	2023-11-07 10:27	Textdokument	1 kB

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 /config/dfmConfig.h and update these settings:

DFM_CFG_PRINT(msg): Add a function call for printing to the serial port, e.g. printf(msg), puts(msg) or similar.

DFM_CFG_PRODUCTID: Should be 1 to match the “Default Product” in the Server.

DFM_CFG_ENABLE_DEBUG_PRINT: Set this to 1 to enable DFM error messages.
- 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

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. The Percepio Payload Viewer Client includes tools for viewing CrashCatcher core dumps.

Core dumps can be provided both on fault exceptions (e.g. on invalid memory access) and when calling DFM_TRAP() in your code. Note that CrashCatcher is intended for Arm Cortex-M devices only, but for other processors it is possible to integrate other core dump solutions in a similar way.

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 handler.
 - 3.5.3. Do the same for the NMI handler (this is used by **DFM_TRAP**).
- 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 to match your system (see the code documentation). The ADDR settings (BEGIN and NEXT) should match the memory range where the stacks are found. This is used to [avoid issues with reading outside the valid stack area](#).
- 3.7. Open config/dfmCrashCatcherConfig.h and set **DFM_CFG_CRASH_ADD_TRACE** to 0. We will enable the Tracealyzer traces in a later step.
- 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 output in the serial terminal.

Note: If your debugger halts execution on entering the DFM fault handler, this usually happens because of a debugger setting, **"Halt on Exception"** or similar. In Eclipse-based IDEs like STM32CubeIDE, this is typically found in the **Debug Configuration -> Startup**.
- 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 but also bare metal systems. The latter can be extended with custom instrumentation for any RTOS or similar C/C++ system 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 guides at <https://percepio.com/tracealyzer/gettingstarted/>, in line with the specific instructions below. Additional information is found in Tracealyzer User Manual (see Help menu), in particular the section "Creating and Loading Traces". This is the place to go if you want to configure a bare metal integration.

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

4.5. Open `config/dfmCrashCatcherConfig.h` and set **DFM_CFG_CRASH_ADD_TRACE** to 1.

4.6. 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.

4.7. 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 make 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");
```

It is recommended to pause or stop the tracing before calling `xTraceGetEventBuffer()` by calling `xTraceDisable()`.

To restart the tracing after the alert, call `xTraceEnable(TRC_START)`. This resets `TraceRecorder` and clears the trace buffer. 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.

If you want assistance with setting up Percepio Detect for your needs, or have general questions, feel free to contact support@percepio.com or sales@percepio.com.