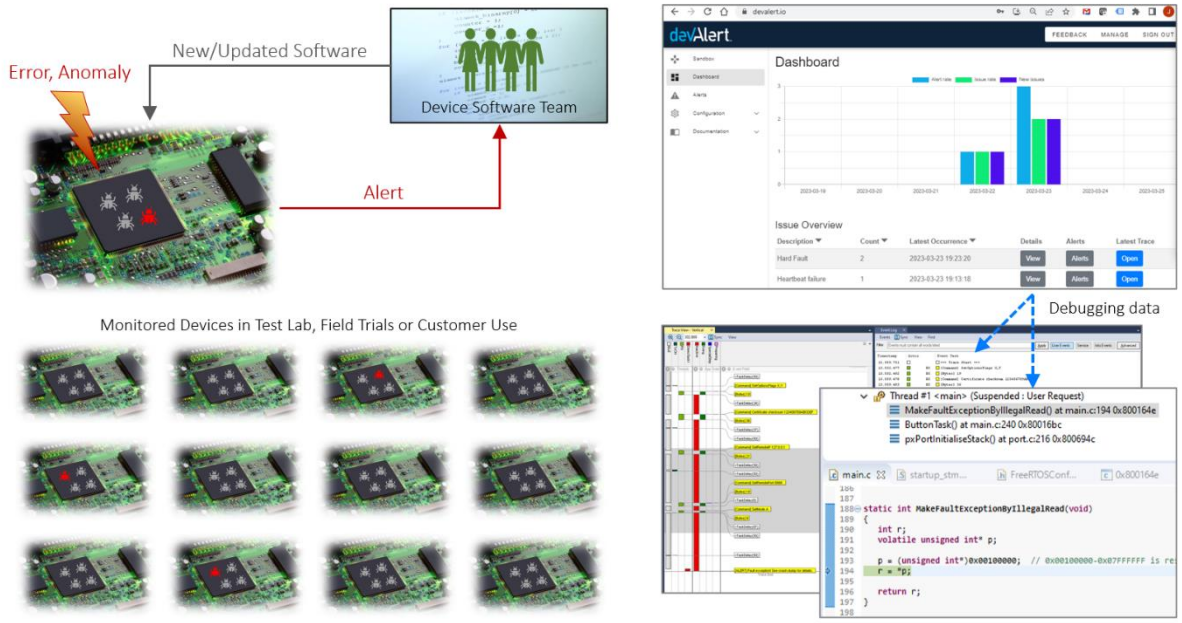
Getting Started with DevAlert 2.0

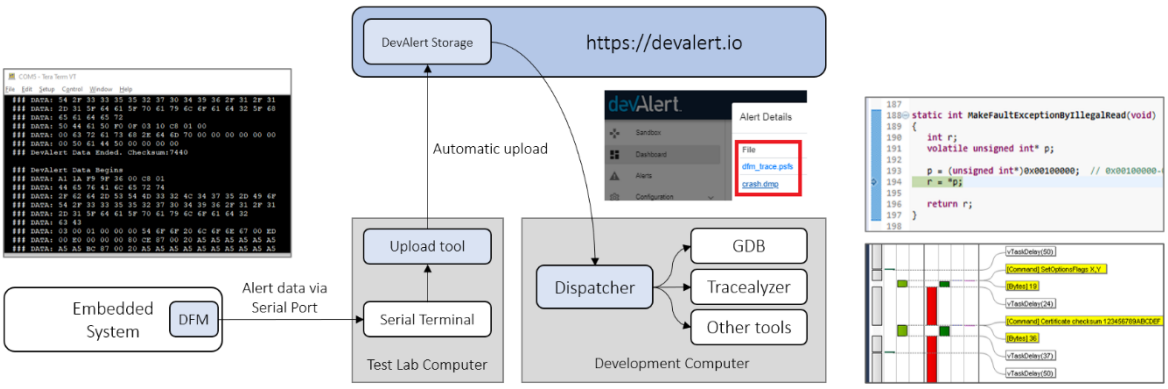
Percepio DevAlert is a cloud-based observability solution for edge devices and embedded software, that lets you detect and analyze issues remotely during system testing, in field trials or in customer use. If your device software would encounter an unexpected situation or misbehave for other reasons, you will be notified right away and get all information needed to quickly solve the problem.

DevAlert uses a novel hybrid desktop/cloud architecture where all device data is easily available in the web browser while also letting you provide your own private data storage to ensure data control and privacy. And you can hook in your familiar desktop tools for remote debugging in a secure way.



DevAlert gives your team members a shared dashboard with remote access to diagnostic data, automatically collected in the devices on system errors and other anomalies. This may include core dumps, event traces and any other device data of interest.

This article explains how to set up Percepio DevAlert for monitoring local devices connected using serial port connections. This is useful as an initial setup for getting started with DevAlert and is also applicable for monitoring system testing on multiple devices in a test lab.



https://percepio.com
Technical support: support@percepio.com

DevAlert relies on a target-side client library, DFM, that is added in your embedded application. This is responsible for encoding reports from your application into “alert” packets for DevAlert. Note that the DFM library is passive until its API is called from your code, e.g., in an error handler. What happens next depend on your configuration. The alert is either stored on the device and/or uploaded to cloud storage, depending on what “cloudport” and “storageport” modules that are used. The cloudport module defines how to output the data and the storageport defines how to read and write the data to a local storage, such as internal flash in the device.

In this article, we will set up a solution where the alert data is transmitted from the DFM client in your embedded system to your desktop computer using the serial port, e.g. via a UART or a local network connection. The DFM library includes a “cloudport” for serial port transfer where the data is printed as Hex strings, received by a serial terminal application on a host computer and then piped to a provided “upload tool” (devalerthttps) that uploads to your DevAlert evaluation account storage.

In a production device, you may instead upload the data directly from the device to your own cloud storage, for example using MQTT over Wi-Fi, or simply store the DevAlert data on the device for later retrieval via a temporary connection to host computer.

DevAlert is designed with data privacy in mind and allows for using customer-hosted storage solutions, for example an Amazon S3 bucket in your own AWS account. This way, the diagnostic payload data never leaves your private domain. However, in this example we are using the Percepio-hosted storage provided with the evaluation account to simplify the setup. This way, you don’t need to configure a cloud-side integration.

This guide is targeting Arm Cortex-M devices running an RTOS such as FreeRTOS, or a bare metal platform, but the DevAlert client (DFM) can easily be adapted for other RTOSes.

This guide is not intended for Linux-based devices. While the DevAlert architecture is capable of supporting Linux systems and Linux support is planned for the future, the present solution has not yet been verified for use with Linux.

Processor support

The target-side client of DevAlert, 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 adding diagnostic data in the alerts, 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.

The provided Core Dump solution (CrashCatcher) is specific for Arm Cortex-M, but this part can be excluded or replaced with another core dump solution. CrashCatcher officially only supports ARMv6-M and ARMv7-M processors (i.e. up to Cortex-M7), but we have used it successfully also on newer ARMv8-M processors like Arm Cortex-M33.

If using a different processor family, you may exclude CrashCatcher and instead integrate a different core dump solution. Such are sometimes provided by the processor vendor, e.g., intended for serial port output or flash storage, for example espcoredump from Espressif. To create an alert with other

types of core dump data, use the function xDfmAlertAddPayload() and configure the Dispatcher tool to launch a suitable tool or script to display the provided data, as demonstrated in this document.

RTOS support

The DFM library can be used with any RTOS assuming minor changes. The only RTOS dependency in the DFM library is a single function for getting the name of the current task. This is only needed for including the currently running task in the CrashCatcher integration (dfmCrashCatcher.c). This can easily be removed or replaced with a suitable function matching your RTOS.

The TraceRecorder library requires RTOS-dependent instrumentation to record kernel events, but you may use the "Bare Metal" option to integrate TraceRecorder without RTOS dependencies. This allows for logging several types of events at application level. You may also extend this with your own kernel or API instrumentation using the Tracealyzer SDK.

Note that the DevAlert client DFM includes support for Zephyr RTOS, although this is not yet fully covered by this guide. DFM is available in the Zephyr repository as a submodule and in the manifest since Zephyr 3.5.0, so you configure it easily using the kconfig system. A proper guide for Zephyr users will follow. Feel free to contact support@percepio.com if you have questions or need assistance with your integration.

Using the DevAlert Target-side Client (DFM)

The target-side client of DevAlert, 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 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 DfmAlertHandle_t pointer where the resulting alert handle will be stored. This is then used in additional calls where more information is added. Finally you call xDfmAlertEnd where the data is stored and/or transmitted, depending on the DFM configuration.

The alert types are defined in dfmCodes.h. This header file is generated using the DevAlert console and should not be modified manually, since the definitions must reflect the cloud service database. You can define your own alert types in the DevAlert console under Configuration -> Alert Types and then update dfmCodes.h in Configuration -> Code Export.

The xDfmAlertAddSymptom calls creates a "fingerprint" that characterizes the reported issue, for example using important processor registers and other information. Each piece of fingerprint data is called a "symptom". The fingerprint is used to group the individual alerts into "issues" displayed in the DevAlert dashboard. This way, you don't need to inspect each individual alert but can consider all alerts within the same "issue" as repetitions of the same problem, assuming you have a sufficiently detailed fingerprint. We recommend including at least the program counter and the stack pointer.

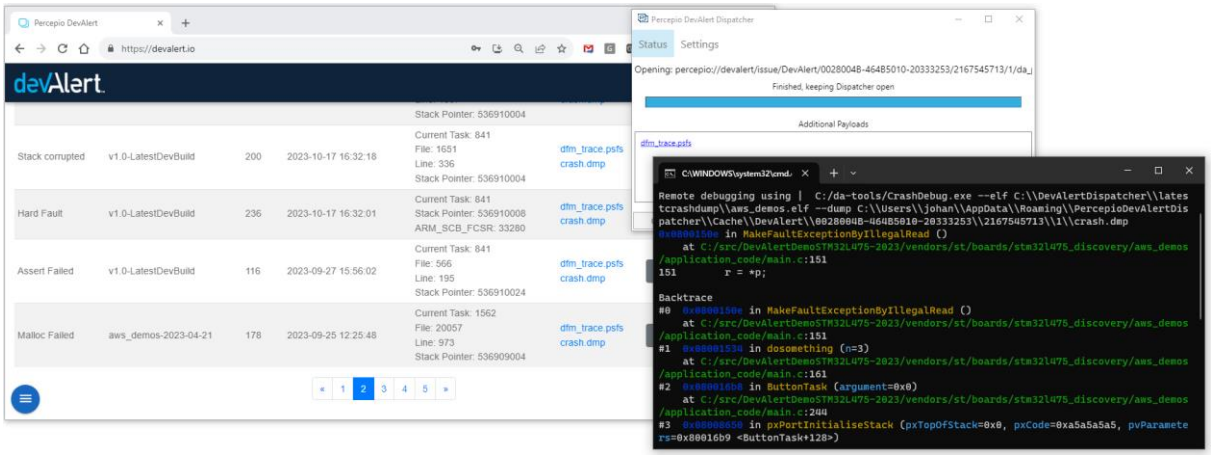
The real power of DevAlert comes when including “payloads” such as core dumps, system traces and other device data. This is done using xDfmAlertAddPayload, as demonstrated below.

```
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, coredumpsize, "coredump.bin");

    xDfmAlertEnd(xAlertHandle);
}
```

To view such diagnostic payloads after receiving the alert, you simply click the links in the DevAlert dashboard. This starts the Dispatcher tool on your local computer, that downloads the payload data and launches it in a suitable desktop tool. In the below example, we selected “crash.dmp” from a Hard Fault alert, and Dispatcher tool launched a GDB script to display the selected core dump.



DFM also includes a macro for creating a standard Alert with a single line of code, DFM_TRAP. This is provided as part of the Arm Cortex-M core dump support and is intended for critical errors. The default behavior is to switch to the NMI interrupt handler, save a core dump and restart the device. It is however easy to define similar alert macros with different behaviors. In the example below, from dfmCrashCatcher.c, the DFM_TRAP macro is used to report stack corruption detected using a GCC compiler feature. To enable this check, use the build flag *-fstack-protector-strong*.

```
/* Called by gcc when stack corruption is detected, if using gcc option -fsta
void __stack_chk_fail(void) {
    DFM_TRAP(DFM_TYPE_STACK_CHK_FAILED, "Stack corruption detected");
}
```

DevAlert Device Integration

To get started with DevAlert and DFM, follow the following steps. Some require a bit more instructions and have their own subsection after this guide. The first steps (1-11) will let you upload a test data file and see this in you DevAlert account. Then we will set up the device integration using the DFM library with data transfer using a serial port. Finally, we extend the solution with CrashCatcher core dumps and Tracealyzer traces for improved remote debugging support.

1. Sign up for an evaluation account at https://devalert.io/auth/signup.
2. Open the welcome email to get your temporary password.
3. Sign in at https://devalert.io/auth/login and update your password.
4. At the welcome screen, select “Activate Service”.
5. Install Python (if you don’t already have it) from https://www.python.org and make sure it is accessible from your terminal. You may need to add it to your PATH environment variable.
6. Download the DevAlert upload tool “devalerthttps” from https://percepio.com/downloads/devalert-eval-upload-serial.zip and extract the contents to a new directory, e.g. “devalert-tools”.
7. Open a terminal and enter your “devalert-tools” directory.

   Run “devalerthttps configure” to configure the upload. The username is the email used when registering your DevAlert evaluation account and the password is DevAlert account password. Note that the username/password authentication in devalerthttps is only used for evaluation accounts in order to simplify the setup. Production accounts uses mutual TLS authentication.

```
C:\da-tools>devalerthttps configure
Enter username (email), the same used to log in to https://devalert.io
Username [johan.kraft@percepio.com]: johan.kraft@percepio.com
Enter password, the same used to log in to https://devalert.io
Password [press ENTER to keep current]:
Username and password are valid
Test successful

C:\da-tools>
```

8. Check that devalerthttps reports “Test successful”. This usually takes about 10 seconds.
9. To test the upload tool, run the following command in your “devalert-tool” directory. This will upload an example alert from the provided file example.log.

   python devalertserial-eval.py --file example.log | devalerthttps.exe store-trace

```
C:\Users\johan>cd C:\da-tools

C:\da-tools>python devalertserial-eval.py --file example.log | devalerthttps.exe store-trace
Calling POST, subject: 'DevAlert/0028004B-464B5010-20333253/1408680610/1/1-1_da_header', url:
1.amazonaws.com/eval/2021-12-21/trace?key=DevAlert/0028004B-464B5010-20333253/1408680610/1/1-
rs
Calling POST, subject: 'DevAlert/0028004B-464B5010-20333253/1408680610/1/1-1_da_payload1_head
.eu-west-1.amazonaws.com/eval/2021-12-21/trace?key=DevAlert/0028004B-464B5010-20333253/140868
ntent with 48 chars
Calling POST, subject: 'DevAlert/0028004B-464B5010-20333253/1408680610/1/1-1_da_payload1', ur
t-1.amazonaws.com/eval/2021-12-21/trace?key=DevAlert/0028004B-464B5010-20333253/1408680610/1/
chars
```

   This should result in three uploaded data chunks. It may take a few seconds for each upload. Then close the upload tool by pressing Ctrl-C.
10. Check the dashboard at https://devalert.io. The test alert should appear after a few seconds and should include a payload called “demo_payload.txt”.
11. Setup the Dispatcher tool to view the provided test data, as described in the section **DevAlert Dispatcher - Basic Setup** below.

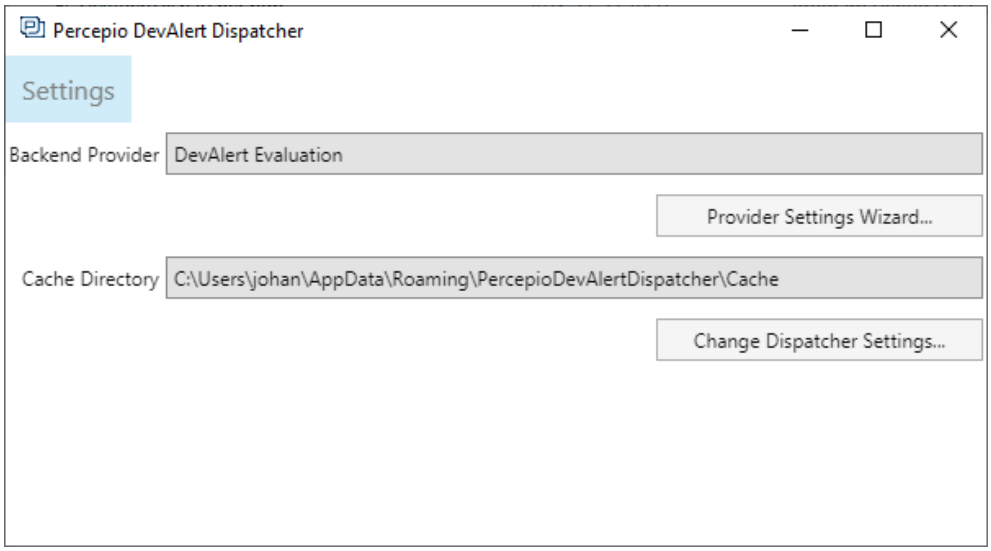
Step 11. DevAlert Dispatcher – Basic Setup

To access a payload from the DevAlert dashboard, we need to download and configure DevAlert Dispatcher. This is a special downloader tool for DevAlert alert payloads that runs on your local computer when you click on the payload links in the dashboard. This is the key to the unique data control and privacy in DevAlert, as the payloads can be stored separately, outside the DevAlert service, and don't need to pass through Percepio servers. For evaluation accounts the storage is hosted by Percepio for simplicity, but it is still separated from the core DevAlert service.

Dispatcher also loads the downloaded data into the right desktop tool, according to you configuration, meaning all payloads are accessible with a single click.

11.1. Sign in to your DevAlert account and download the latest version of the Dispatcher tool from https://devalert.io/dispatcher.

11.2. Extract all files to your "devalert-tools" directory, locate the the DevAlertDispatcher executable and start it manually. The Dispatcher tool normally starts in a few seconds although it can take a bit longer the first time. You should see the main screen like depicted below.



11.3. Select "Provider Setting Wizard…" and press "Next".

11.4. Enter your username (email) and password. Select "Test" to verify the credentials. Next.

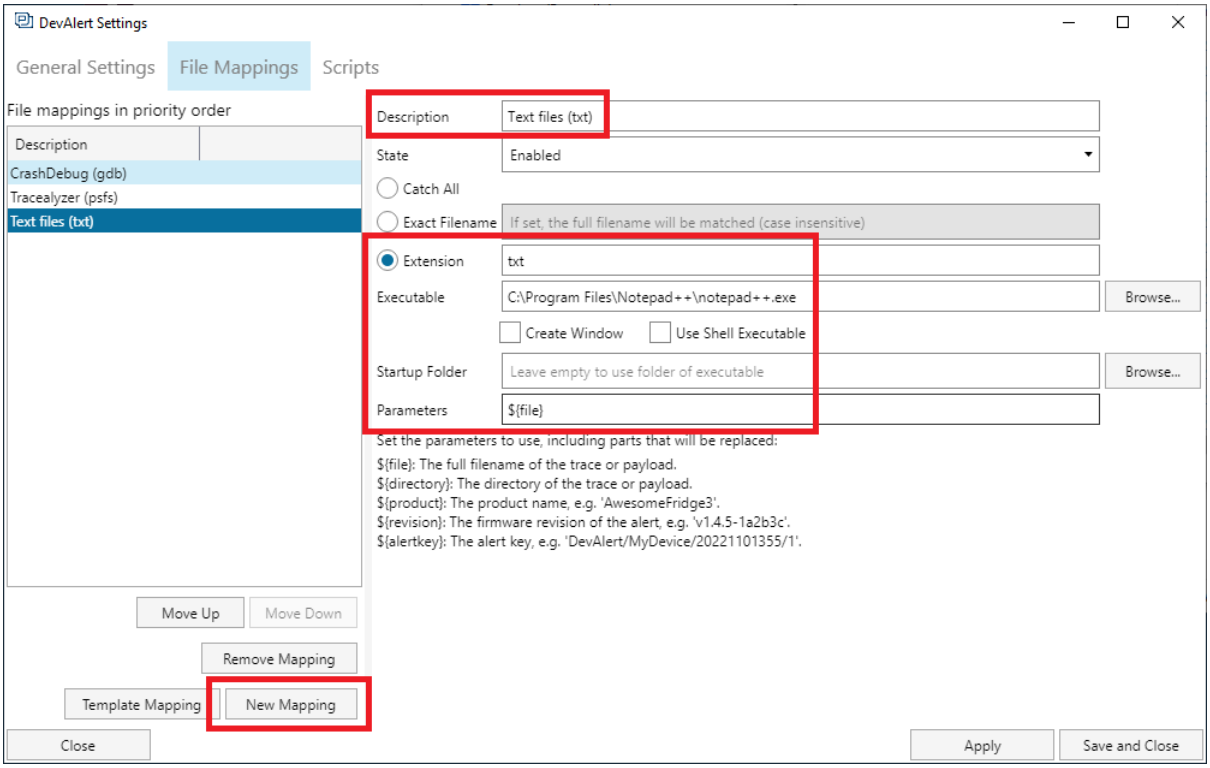
11.5. As "Backend provider" select "DevAlert Evaluation". Next.

11.6. Dispatcher will ask you're your username (email) and password a second time. This is for the backend configuration. Select "Test" to verify the credentials. Next, and finish the wizard.

11.7. On the main Dispatcher screen, select "Change Dispatcher Settings…".

11.8. Verify that "Download Link Status" reads "Handling Percepio links from DevAlert." This is the status of the web browser integration, where Dispatcher is registered as a protocol handler. If the status is different, click "Enable Download Links" to register the tool manually.

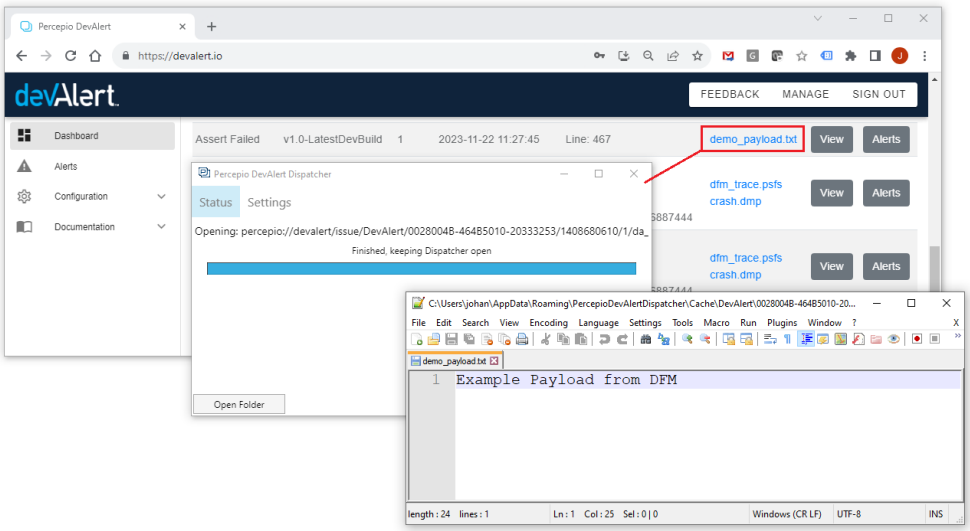
11.9. Select File Mappings and select "New Mapping". We need to create a mapping for text files (.txt) to view the example payload.



11.10. Select “Extension” and enter “txt”. Add the path to your preferred text editor and enter ${file} under parameters. This is the path to payload file once downloaded.

11.11. Save and Close. Close Dispatcher.

11.12. Open your DevAlert dashboard at https://devalert.io. Click on the “demo_payload.txt” link in the example alert. This downloads the selected payload and launches the data in the right tool, according to your Dispatcher configuration.



Congratulations, you should now have the devalerthttps upload and Dispatcher tool working!

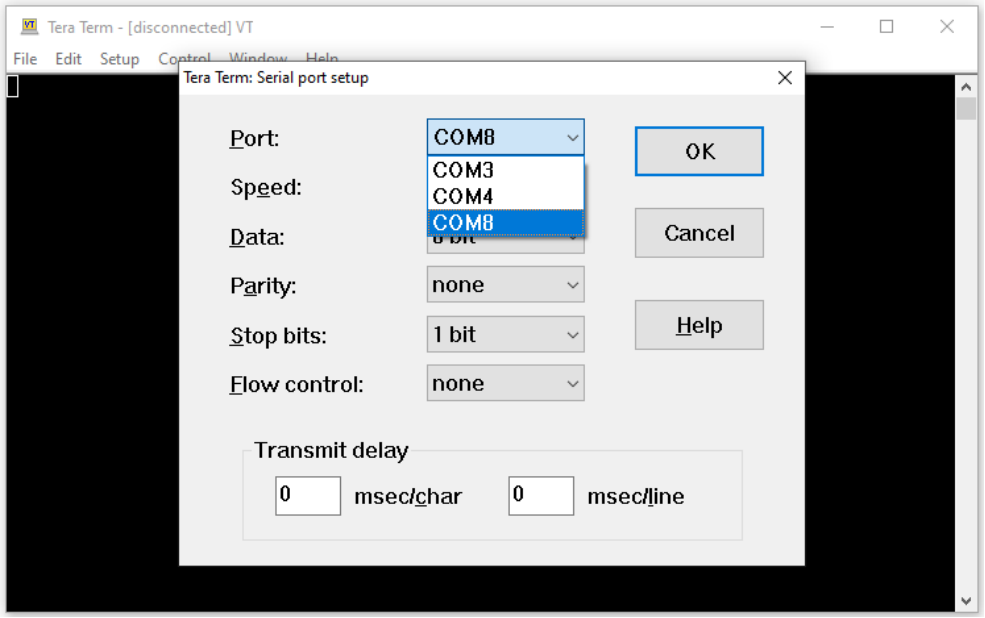
The next step is to set up your own alerts on your own device. This requires adding the DevAlert client (DFM) in your embedded system. Follow the steps below.

12. Find or create a suitable demo project for your target processor in your development environment. The demo application can be “anything”, but it is recommended to use a public code example, e.g. from your RTOS- or processor vendor. This since parts of the memory contents will be uploaded to Percepio-hosted evaluation storage used in this example.
13. Make sure you can print text to a serial port or similar and receive it on the host computer in a terminal program. The terminal program needs to support logging the device output to a file. For Windows users TeraTerm is recommended and used as example in this guide. A suitable configuration of TeraTerm is described in **TeraTerm Setup** below.
14. Integrate the DFM client as described in **DFM Library Integration** below.
15. For Arm Cortex-M core dump support, you need CrashCatcher, CrashDebug and GDB. This is described in **Core Dumps with CrashCatcher** below.
16. To capture Tracealyzer traces in your alerts, integrate the TraceRecorder library in your project as described in **Adding Tracealyzer Support** below.

Step 13. TeraTerm Setup

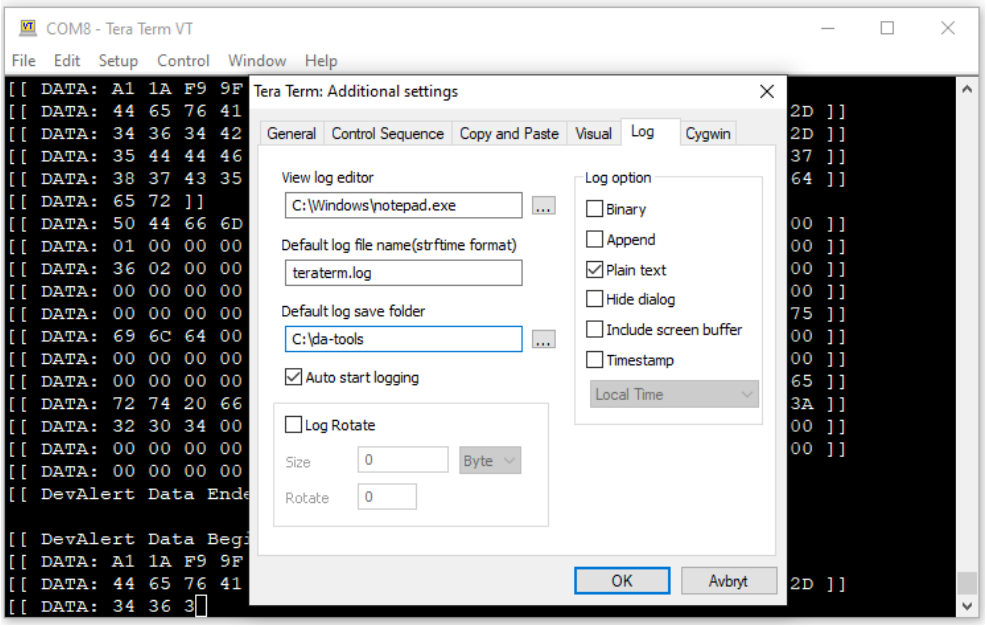
We have found TeraTerm a very convenient serial terminal program for receiving and logging serial port data on Windows, but there are many alternatives that often can be configured in similar ways. If you prefer a different serial terminal application, configure that in a similar way such that the output is logged to a file in the “devalert-tools” directory.

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 Speed. The other settings are usually fine as is.



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

Select Setup -> Additional settings -> Log and configure the settings like below:



Default log save folder: your Devalert tools folder

Auto start logging: Checked

Log Rotate: Unchecked

Append: Unchecked

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

Close Teraterm and continue with Step 14 below.

Step 14. DFM Library Integration

14.1. Once you have signed up for a DevAlert evaluation account and logged in at https://devalert.io, download the DFM library source code from https://devalert.io/dfm/files.

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

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

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

14.4. Add cloudports/Serial/dfmCloudPort.c to your project. This module specifies how to output the data. Also copy the header files from the local **include** and **config** directories to the same “include” directory as in the previous step.

14.5. 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.

14.6. 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.

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

14.8. Add a call to xDfmInitialize() in the startup. Place the call in the main() function at a point where your cloudport function (DFM_CFG_PRINT) is ready to use. In the current version of DFM this needs two callback functions, described below.

14.8.1. xGetUniqueSessionID - Should provide the a “session identifier”. This should ideally be a unique string that is never repeated for a particular device, for example a reboot counter that is stored in flash memory or a wallclock timestamp if available. For an evaluation setup, you may consider using a random number. But this is NOT recommended for production devices this may lead to alerts being considered as identical duplicates and rejected.

14.8.2. xGetDeviceName - The second callback should provide the device name. For evaluation purposes you may use a constant string here, like “Device123”. For production devices this should use a unique serial number for the particular device.

Both callbacks should write the ID to the provided buffer (cBuffer) and the length of the string to *pulBytesWritten. See the code example below.

```
DfmResult_t myGetSessionID(char cBuffer[], uint32_t ulSize, uint32_t* pulBytesWritten);
DfmResult_t myGetDeviceName(char cBuffer[], uint32_t ulSize, uint32_t* pulBytesWritten);

...

if (xDfmInitialize(myGetSessionID, myGetDeviceName) == DFM_FAIL)
{
     configPRINTF(("Failed to initialize DFM\r\n"));
}

...
```

```
DfmResult_t myGetSessionID(char cBuffer[], uint32_t ulSize, uint32_t* pulBytesWritten)
{
     uint32_t nBytes = snprintf(cBuffer, ulSize, "%08X", <SessionID> );
     if (nBytes > 0) {
          *pulBytesWritten = nBytes;
          return DFM_SUCCESS;
     }
     return DFM_FAIL;
}

DfmResult_t myGetDeviceName(char cBuffer[], uint32_t ulSize, uint32_t* pulBytesWritten)
{
    uint32_t nBytes = snprintf(cBuffer, ulSize, "MyDevice");
    if (nBytes > 0){
        *pulBytesWritten = nBytes;
        return DFM_SUCCESS;
    }
    return DFM_FAIL;
}
```

14.9. After the call to xDfmInitialize(), add a test alert like in the following code example.

```
#include <dfm.h>

DfmAlertHandle_t xAlertHandle = 0;
char* payloadString = "My own payload from DFM";

...

if (xDfmAlertBegin(DFM_TYPE_ASSERT_FAILED, "Demo alert", &xAlertHandle) == DFM_SUCCESS) {
    xDfmAlertAddSymptom(xAlertHandle, DFM_SYMPTOM_LINE, __LINE__);
    xDfmAlertAddPayload(xAlertHandle,
                        payloadString, strlen(payloadString), "my_payload.txt");
    xDfmAlertEnd(xAlertHandle);
}
```

14.10. Build and run your project and make sure you see DFM data in the serial console.

14.11. Configure your serial terminal program to save the data to a log file (e.g. “mydevice.log”) and upload the data like in the previous example:

```
python devalertserial-eval.py --file teraterm.log | devalerthttps.exe store-trace
```

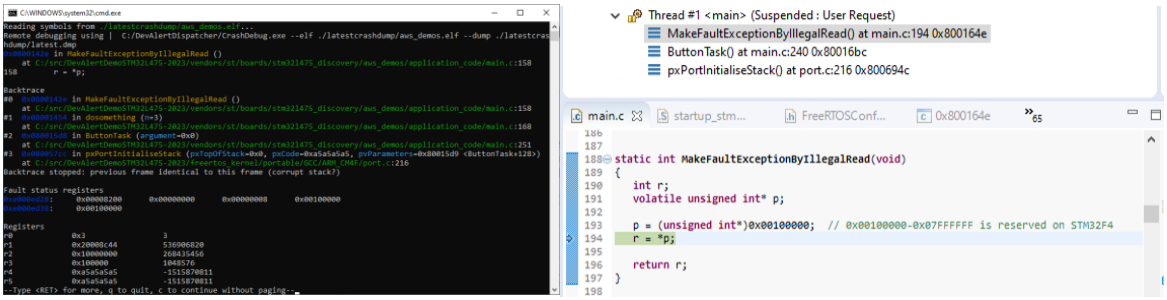
You should now see your own alert in the DevAlert dashboard.

Congratulations, you now have full chain working from device to dashboard!

Next, if using an Arm Cortex-M device, continue with step 15 where we include CrashCatcher to collect core dumps for source code debugging. Otherwise jump to step 16.

Step 15. Core Dumps with CrashCatcher on Arm Cortex-M devices (optional)

Core dumps include registers, stack and other memory contents. They are created on the target side using the CrashCatcher library and loaded into GDB using the CrashDebug tool. This way you can inspecting the system state in a GDB client and also load the core dump in for example the Eclipse GDB-based debugger, or in other debugging tools using GDB. In this way, you can view core dumps from remote devices using your familiar debugger using all its' provided views for inspecting the memory, special registers and more.



Core dumps can be provided both on fault exceptions (e.g. on invalid memory access) and when calling DFM_TRAP() in your code. The integration between DFM and CrashCatcher is provided by dfmCrashCatcher.c so you don't need to define these alerts yourself. Just follow the steps below. 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.

15.1. Get the CrashCatcher library from https://github.com/adamgreen/CrashCatcher.

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

15.3. Copy the assembly file **CrashCatcher_armv7m.S** into your project, if using Arm Cortex-M3 or higher (also for Arm Cortex-M33 and other Armv8-m devices). For Cortex-M0 devices, use CrashCatcher_armv6m.S instead.

15.4. Copy the following header files to a suitable "include" directory in your project where your compiler will find them:

- Core/src/CrashCatcherPriv.h
- Include/CrashCatcher.h

15.5. Open and edit the .S file as described in **CrashCatcher Fault Exception Handlers** below.

15.6. Download CrashDebug from https://github.com/adamgreen/CrashDebug/tree/master/bins and save all files in your **devalert-tools** directory, i.e. next to the Dispatcher tool.

15.7. Make sure you have GDB for Arm Cortex-M devices. If using a GCC-based development tool you should already have this (arm-none-eabi-gdb) on your computer, otherwise download the Arm GNU toolchain from https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads.

15.8. Copy the **arm-none-eabi-gdb** executable to your **devalert-tools** directory.

15.9. Open the **template** subdirectory provided with the Dispatcher tool. Select **crash_debug.bat** if using Windows or **crash_debug.sh** if using Linux and copy this to the **devalert-tools** directory.

15.10. Start Dispatcher and select "Change Dispatcher Settings…".

15.11. Select File Mappings -> New Mapping.

15.12. Select Extension and enter **dmp** as file name extension to match.

15.13. For Executable, select your crash_debug script in the **devalert-tools** folder.

15.14. Check the setting “Create Window”.

15.15. Under Parameters, add the following three parameters (in this order):

- The path to your project’s ELF file (the binary from your compiler). As a first test you can hardcode this to your local build output directory. Later on, you may consider using the variable ${revision} when configuring the ELF path. This provides the DFM setting DFM_CFG_FIRMWARE_VERSION (see dfmConfig.h) from the alert metadata.
- ${file} – This provides the path to the downloaded payload file, in this case the CrachCatcher core dump file.
- --gdb – Add this optional parameter to make the GDB window remain open. This way you can run GDB commands to inspect the core dump in further detail. If omitted, the script will instead create a text file with the GDB output and display that in the default text editor.

15.16. Save the File Mapping and close Dispatcher.

15.17. Open config/dfmCrashCatcherConfig.h and set **DFM_CFG_CRASH_ADD_TRACE** to 0. We will enable the Tracealyzer traces in a later step.

15.18. Open **CrashCatcherPriv.h** and locate the setting **CRASH_CATCHER_STACK_WORD_COUNT**. CrashCatcher sets up its own stack for the exception handling and this needs to be sufficiently large for the DFM_CFG_PRINT function. Increase this to 250-300 to be on the safe side.

15.19. To test the core dump feature, add a call to **DFM_TRAP()** in your code to trigger a core dump and restart. For example:

```
#include <dfm.h>
#include <dfmCrashCatcher.h>
...
DFM_TRAP(DFM_TYPE_ASSERT_FAILED, "My Core Dump");
```

15.20. Run your application and inspect the serial terminal output carefully to ensure it is complete, i.e., that each data block ends with a checksum. Start the upload tool if not already running and your new alert should appear in the dashboard, with the core dump as a “dmp” payload.

15.21. Click on the “dmp” payload and Dispatcher will start your GDB script to show the core dump.

Congratulations, you should now have core dump support in DevAlert!

You may also load core dumps into your IDE debugger, assuming it used GDB. See **Appendix A** for an example using STM32CubeIDE. This approach works in most Eclipse-based IDEs and other IDEs based on GNU tools.

Next, continue with Step 16 below where we add Tracealyzer traces in your alerts.

Step 15.5. CrashCatcher Fault Exception Handlers

Note that CrashCatcher defines a hard fault exception handler in the assembly file (**CrashCatcher_armv7m.S** or **CrashCatcher_armv6m.S**). By default only HardFault_Handler is defined but we need to extend this a bit, as shown below.

```
.global HardFault_Handler
.type HardFault_Handler, %function

.global UsageFault_Handler
.type UsageFault_Handler, %function

.global BusFault_Handler
.type BusFault_Handler, %function

.global NMI_Handler
.type NMI_Handler, %function

.thumb_func
HardFault_Handler:
UsageFault_Handler:
BusFault_Handler:
NMI_Handler:       /* Used by DFM_TRAP() */
```

Locate the references to “HardFault_Handler” and add “NMI_Handler” in the same way to enable the DFM_TRAP() macro. If you have enabled other fault exceptions like BusFault or UsageFault, add them as well. Finally, make sure that the CrashCatcher exception handers are used instead of the default handlers in the interrupt vector table.

The size of the core dump can be configured and may include multiple memory ranges. By default, the CrashCatcher integration in DFM stores the last 300 bytes relative to the stack pointer. If using an RTOS, this will be the stack pointer of the currently running task. 300 bytes is usually sufficient to see the most recent function calls and GDB can show incomplete stacks where the earliest calls are missing. You may however configure the stack capture size in dfmCrashCatcherConfig.h. To include additional memory ranges, see the “CRASH_MEM_REGION” settings in dfmCrashCatcher.h.

Next, continue with step 15.6 above.

Step 16. Adding Tracealyzer Support (optional)

Percepio Tracealyzer is a visualization tool for event traces that simplifies debugging, especially for more complex issues in multithreaded RTOS applications. This can be used within DevAlert by integrating the TraceRecorder library and providing the trace data as an alert payload. This way, you can collect traces on system errors and anomalies showing the timeline of events just before the alert was created.

Tracealyzer supports a number of common real-time operating systems using different tracing libraries, e.g. LTTng for Linux and Percepio TraceRecorder for MCU-class RTOSes. The DevAlert client library, DFM, has so far only been tested with the Percepio TraceRecorder. This includes support for FreeRTOS, SafeRTOS, Zephyr, ThreadX, PX5 and bare metal systems. The latter can be extended with custom instrumentation for any RTOS or similar C/C++ system using the Tracealyzer SDK.

16.1. To get started with Tracealyzer, sign up for a free evaluation license at https://percepio.com/tracealyzer/download-tracealyzer/

16.2. 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.

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

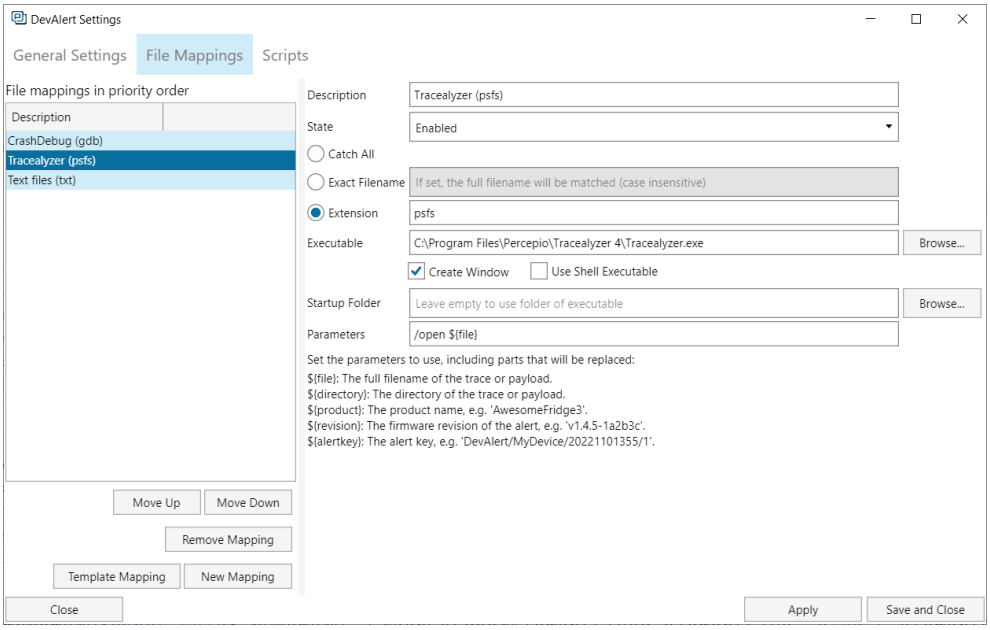
16.4. In your alerts, add the trace data as a payload in the following way:

```
void* pvTraceData = (void*)0;
uint32_t ulTraceDataSize = 0;
...
xTraceDisable();
xTraceGetEventBuffer(&pvTraceData, &ulTraceDataSize);
xDfmAlertAddPayload(xAlertHandle, pvTraceDatar, ulTraceDataSize, “dfm_trace.psfs”);
```

Note that it is recommended to call xTraceDisable() before calling xTraceGetEventBuffer() if interrupts are enabled at the point of the alert. If the device isn’t restarting after the alert, you may add xTraceEnable(TRC_START); after the xDfmAlertEnd() to resume the tracing.

You may also consider adding a User Event (e.g. xTracePrintF) to mark the alert in the trace. You may add multiple such events or add additional parameters to log any information of relevance. Make sure add these calls before calling xTraceDisable() or xTraceGetEventBuffer().

16.5. In Dispatcher, add a File Mapping for Tracealyzer, with the parameter “/open ${file}” as demonstrated below.



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

16.7. Run your application with the **DFM_TRAP** alert (see Step 15.19) and make sure it is uploaded.

16.8. You should now see an alert with a “dfm_trace.psfs” payload. Click it and Dispatcher should start Tracealyzer and load the trace.

Congratulations, you should now have Tracealyzer support enabled in DevAlert. To learn more about Tracealyzer, see e.g. https://percepio.com/tracealyzer/ and https://percepio.com/category/handson/.

If Tracealyzer doesn’t start in 10 seconds or so, check if you have another open Tracealyzer instance and close that first.

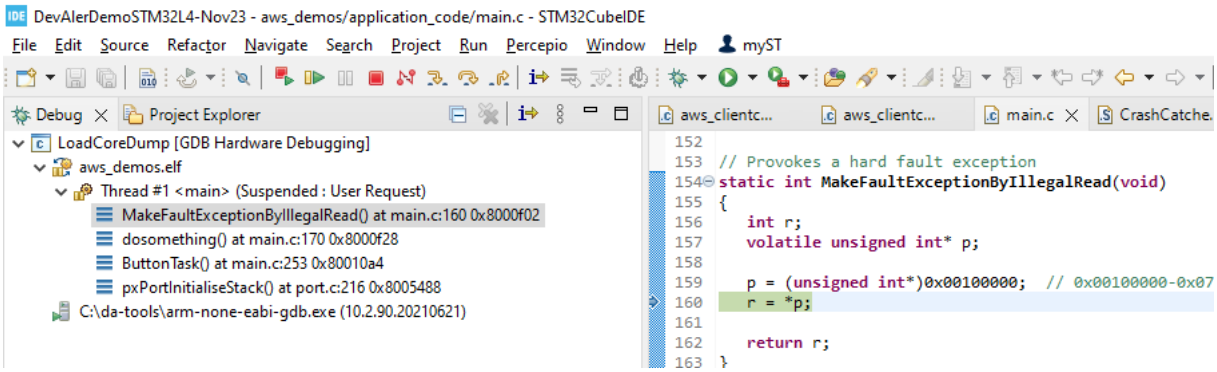
You have now completed the DevAlert getting started guide and have a DevAlert setup for using serial port transfer to your evaluation account. To use DevAlert in other ways, for example uploading the alerts directly from the device to cloud, you need a production account and other cloudport and storageport modules. This may require developing custom modules to get your desired setup. This is not covered by this guide but there are example modules included in the DFM repository. Note the modules under kernelports/<RTOS>. For example kernelports/FreeRTOS/AWS_MQTT, which is intended MQTT upload to AWS IoT Core. This allows for storing the data in your own AWS account.

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

Appendix A. Loading Core Dumps into STM32CubeIDE

Once a core dump has been downloaded using DevAlert, you may consider loading it into your IDE debugger instead of only viewing it in the GDB window. This way you can leverage all you IDE debugger views, just like if debugging a local device that is halted on a break point.

This approach can be used with most IDEs that rely on GNU tools like GCC and GDB.

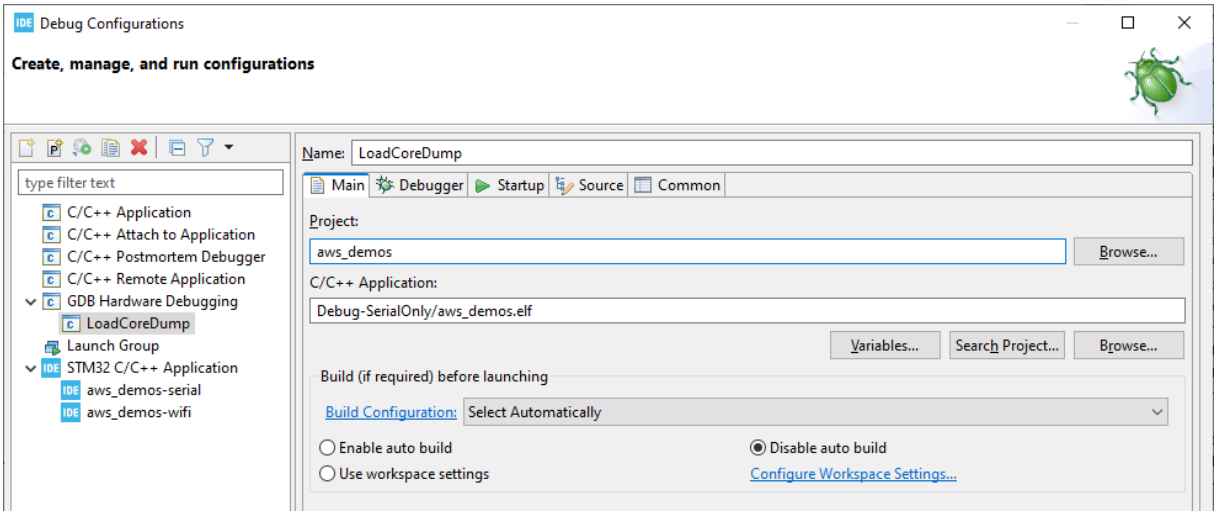


A.1. Extend your crash_debug script with the following commands to copy the core dump file to a known location. The ELF file is also copied in this case, although that isn't strictly necessary. Note that the below example is for Windows.

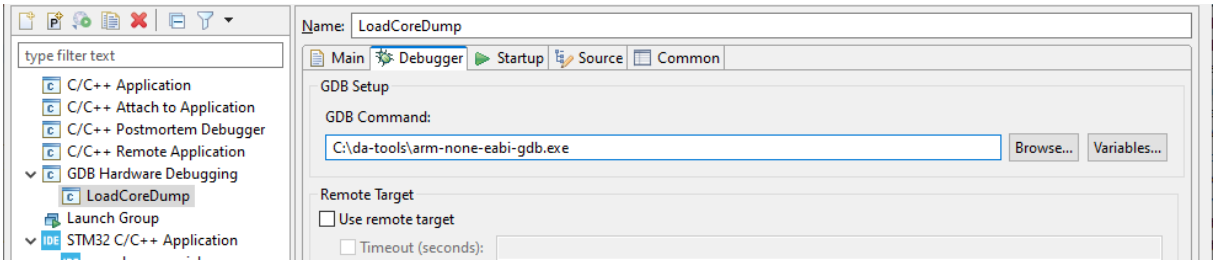
```
copy /y %1 latest.elf

copy /y %2 latest.dmp
```

A.2. Open Debug Configuration and add a new entry, of the type "GDB Hardware Debugging" and call it "LoadCoreDump" or similar. On the first page, it is recommended to check "Disable auto build".



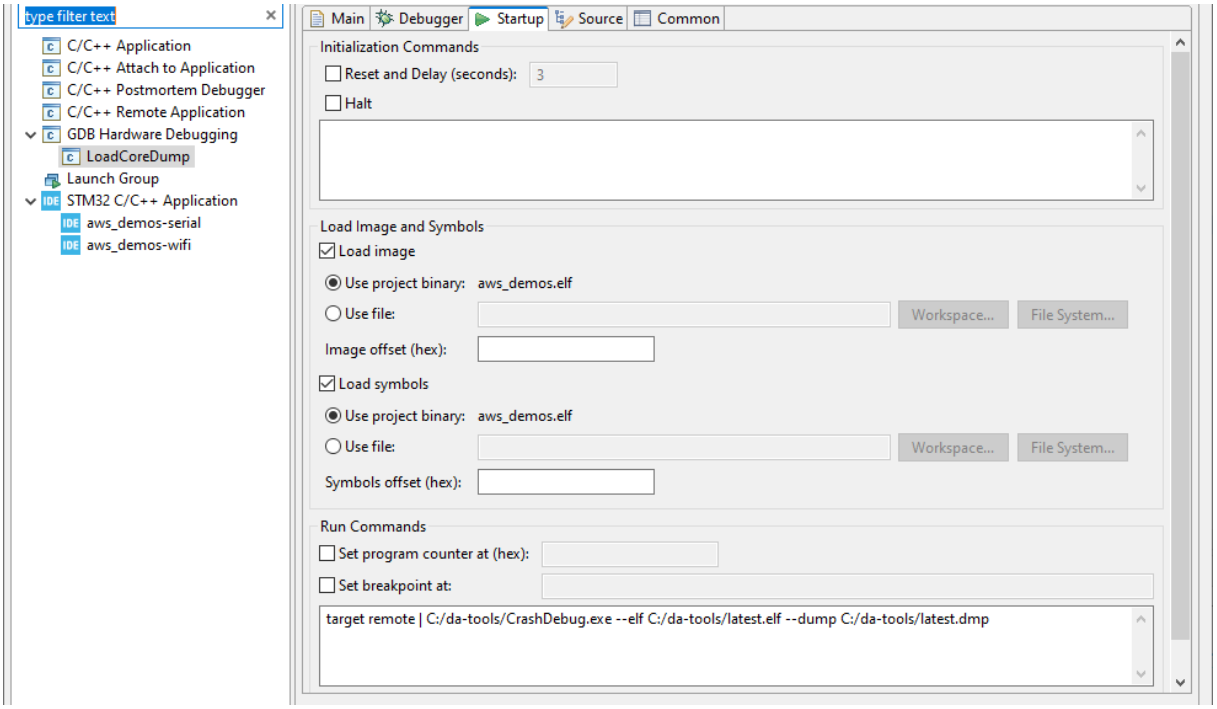
A.3. On the Debugger page, as GBD Command enter the path to your gdb executable in your devalert-tools directory, e.g., C:\da-tools\arm-none-eabi-gbd.exe.



A.4. On the Startup page, check Load Image and Load Symbols. Then enter the following under Run Commands:

```
target remote | C:/da-tools/CrashDebug.exe --elf C:/da-tools/latest.elf --dump C:/da-tools/latest.dmp
```

Make sure to update the file paths to match your “devalert-tools” directory. Note that CrashDebug.exe requires forward slashes in the paths or double backslashes on Windows.



A.5. Save the Debug Configuration.

A.6. Download a core dump payload from the DevAlert dashboard (it will show the latest downloaded).

A.7. Start your new Debug Configuration in STM32CubeIDE.