NAC1080 software development guide

About this document

Scope and purpose
This document serves as an introduction to the NAC1080 build environment.

Intended audience
This document is intended for software engineers who want to develop firmware for NAC1080-based systems.

Table of contents

About this document ................................................................. 1
Table of contents ...................................................................... 1
1 Overview – software development ............................................. 3
2 Introduction ........................................................................... 5
  2.1 Installation ........................................................................ 5
  2.2 SDK content ...................................................................... 5
  2.3 Makefiles and build process ............................................... 6
3 Build .................................................................................... 7
  3.1 Eclipse ............................................................................. 10
4 Debugging ............................................................................. 15
  4.1 Lauterbach ........................................................................ 15
  4.2 SEGGER J-Link ................................................................. 15
    4.2.1 Ozone ......................................................................... 18
    4.2.2 Eclipse ........................................................................ 18
    4.2.3 Visual Studio Code ..................................................... 21
    4.2.4 J-Link Commander and command files...................... 22
  4.3 Flash breakpoints and HardFault handler ......................... 23
5 Memory ................................................................................. 24
  5.1 DPARAM .......................................................................... 24
  5.2 APARAM .......................................................................... 25
    5.2.1 Handler ....................................................................... 25
    5.2.2 Access privileges ........................................................ 28
    5.2.3 Miscellaneous ............................................................. 30
  5.3 RAM .................................................................................. 32
6 General-purpose input/output .................................................. 34
7 Timer ..................................................................................... 35
  7.1 System ticker ..................................................................... 35

User Manual Please read the sections "Important notice" and "Warnings" at the end of this document
www.infineon.com/nac1080
# Table of contents

7.2 System timer .................................................. 35  
7.2.1 Pulse-width modulation .................................. 36  
7.3 Standby timer .................................................. 36  
7.4 Real-time clock .................................................. 37  
8 Power-save mode ............................................... 38  
9 AES encryption .................................................... 40  
10 UART ............................................................. 43  
11 NAC1080 – ROM3 vs. ROM4 ............................... 45  
    Revision history ............................................... 47  
    Disclaimer ...................................................... 48
1 Overview – software development

This chapter gives a short overview of software development for the NAC1080. The NAC1080 is an IC suitable and optimized for nearfield communication (NFC) communication controlled actuation. It contains an NFC transceiver, an actuation interface, an Arm® Cortex®-M0 microcontroller and wired interfaces.

The chip can operate in two different power supply modes: active mode using an external 3 V power supply (e.g., battery), and passive mode where energy is harvested from the NFC field.

Memory types attached to the microcontroller comprise a ROM containing start-up code, NFC state machine and library functions, an NVM for customer firmware, and RAM.

In order to provide a versatile but secure communication path, the NAC1080 implements a proprietary extension to the NFC protocol. The NFC handler and the proprietary extension are located in the ROM and provide access to a mailbox for data exchange between an NFC reader, e.g., a smartphone, and the device.

To communicate with the device, an NFC reader can write a message to a mailbox located in RAM, and then trigger an action by calling one of 16 application-specific functions which can be implemented in non-volatile memory (NVM) to suit the needs of the device. These functions can write a response to the mailbox which in turn can then be read by the NFC reader.

Access to these functions and other features of the proprietary protocol can be enabled or denied in a parameter block, APARAM, in the NVM. With these configuration options, the application-specific firmware can adjust access rights and security according to the needs of the specific device.

Figure 1 NFC communication stack (example: smart lock)

For firmware development for the internal microcontroller, a software development kit (SDK) package is available. It is based on the GCC toolchain, using makefiles to manage the build process. The toolchain can be integrated in IDEs, e.g., Eclipse CDT. The SDK enables using the ROM library functions, and also provides a library with further functions that are linked into NVM.
To download the firmware into NVM and debug the application, the NAC1080 provides a serial wire debug (SWD) port. The SDK supports debug interfaces from Lauterbach and SEGGER (J-Link).

Smartphone apps are developed using the available toolchains for the smartphone architecture, e.g., Android Studio when developing for Android devices.
2 Introduction

The NAC1080 SDK enables the development of firmware for the NAC1080. It comprises a compiler, libraries for NAC1080 firmware, a sample project, a make utility, and support for debuggers.

The sample project is built based on a makefile. This makefile can be processed by calling `make` from the command line, or by configuring it as the build step in an IDE such as Eclipse CDT.

2.1 Installation

The SDK provides build tools for Windows hosts which compile C source code into a firmware image that can be downloaded to the NAC1080. For the download of the firmware image and for debugging, third-party tools are required (see chapter “Debugging”).

The SDK is provided as a .zip file. Please create a destination folder and unpack the .zip file inside this folder.

Note: When choosing the name for this folder, do not use spaces.

When unpacking the SDK, a directory structure is created with the build environment and a small sample project. The top folder contains a file `build_shell.bat`. Open it by double-clicking on it, then type: `make all`. When this command is completed, the toolchain is ready for use.

2.2 SDK content

The folder `license` contains information about the licensing terms of the open-source tools used in the SDK – mainly the GNU compiler collection and the GNU make utility. Please read these terms carefully.

The tools themselves are located in the folder `tools`, mainly in the subfolder `extern`. They are accompanied by scripts for the GNU tools, e.g., makefiles, and definitions for the Lauterbach Trace32 software in the subfolder `intern`.

More definitions and scripts can be found in the folders `tool_config` and `scripts`. Most of the files provide support for Lauterbach and SEGGER J-Link debug adapters. For Lauterbach Trace32 support, only a batch file from the root directory of the SDK must be started. To integrate the J-Link into a toolchain, usually a reference to the folder `tool_config\jlink` must be specified in the configuration of the toolchain, or the files found in this directory must be integrated into the J-Link installation, e.g., the NAC1080 device is merged into the driver's database.

How to use the debug adapters with the NAC1080 is described in chapter “Debugging”.

An example for configuring the J-Link debug adapter in the toolchain can be found in the `.vscode` folder. The JSON files contain configurations for building and debugging the firmware project in the `smack_sl` folder with Visual Studio Code (VSC). Just choose “Open Folder…” from VSC’s “File” menu and navigate to the SDK root folder, then VSC will open the `smack_sl` project. You may have to install a GDB plugin in order to work with J-Link’s GDB server.

The `smack_sl` folder holds a sample project which may be used as a start point for your own projects. The makefiles require that the source files (*.c) are located in the subfolder `src`, while the header files (*.h) are located in the folder `inc`. During the build process, a folder `build` with several subfolders is created in `smack_sl`.

A set of project files for Eclipse can be found in the folder `smack_sl`. To be able to debug the firmware within Eclipse, a GDB plugin needs to be installed and configured.

The folders `smack_lib` and `smack_rom` provide libraries located in the ROM of the NAC1080, or linked statically into the application targeted for NVM. The ROM library is not linked into the NVM firmware but defines entry points into the ROM code; therefore it must match the ROM version found in the NAC1080.

In the top-level directory of the SDK, there are several batch files and a makefile. To build a project from a command line, open `build_shell.bat`. This will open a command shell with an environment suitable for building the firmware, e.g., the path is set to include the make utility and the compiler. Here, you can start the build process through the top-level makefile by typing `make all` or cleanup the build directories with `make clean`. 
The file `t32_start_real.bat` starts the Lauterbach Trace32 software. It prepares scripts and passes them to Trace32, so that it comes up ready to flash the firmware image to the target and start the debug session.

### 2.3 Makefiles and build process

There is one central makefile in the top-level directory of the SDK which calls the makefile in the `smack_sl` folder. The preferred way to build the projects is through this top-level makefile. When using makefiles local to project folders, e.g., when starting a build from within Eclipse, a top-level `make clean` and `make all` should be done on a regular basis, and especially to build release versions.

For the makefile to operate, the environment must be set up correctly, e.g., the `PATH` environment variable must point to the compiler and build utilities. In the root folder of the SDK, there is a batch file `build_shell.bat`, which opens a command shell and adds all the required information to the environment. Within this command shell, the make utility can be started.

When creating new project folders, these have to be added to the project list defined in the top-level makefile. Please search for the line containing `NVM_PROJ_DIRS := smack_sl` and add the new projects to the variable `NVM_PROJ_DIRS`.

For Eclipse, there are project files in the folder `smack_sl` of the sample project. The project file also calls make through a separate batch file.

For VSC, there are configuration files within the `.vscode` folder which also call the make utility through a batch file that sets up the environment first before actually calling `make`. 
3 Build

The following figure shows the directory structure at the top level directory of the SDK, which was unpacked in the folder `smack`:

![Figure 2: SDK directory structure](image)

The directory with the example project is `smack_sl`. It contains the “C” source files of the projects and a makefile which controls the build process. The build result will be placed in a subdirectory `build\image` inside this folder.

The ROM library is found in the folder `smack_rom`. It provides header files with declarations of the functions provided by the ROM library together with an ELF file which provides symbol information to the linker during the build process. Functions of the ROM library are not linked into the NVM image of the user firmware but are called through a lookup table at their address inside the ROM. The subdirectories of `smack_rom` are described below.

The folder `smack_lib` contains the NVM library. Similar to the ROM library, the NVM library provides functions to aid in firmware development for the NAC1080; these functions are not present in the ROM but are linked into the NVM firmware image. The subdirectories of `smack_lib` are described below.

The folders `scripts`, `tool_config` and `tools` contain scripts and utilities needed to build the NVM image and help with debugging, e.g., compiler, make utility and Flash binaries for the debugger.

The folder `.vscode` holds the project files to work with the `smack_sl` project inside VSC. Project files for Eclipse (`project`, `.cproject`) are located inside the `smack_sl` project directory.

The licensing notes for the open-source tools used in the SDK are collected in the directory `license`.

In the following paragraphs, the directory structure of the SDK and the requirements for the build tools are presented. The SDK comprises a sample project `smack_sl` with a makefile and a linker script, which import all the files required. The easiest way to start a project is to copy this project and modify the source code.

**ROM library**

The files for the ROM library are located in a directory structure below the folder `smack_rom` as shown in Figure 3.
All directories named \textit{inc} hold header files with declarations of the ROM and CMSIS libraries. The header files for the ROM library can be found in the directory \texttt{smack\_rom/libs/smack\_lib/inc}, and the others contain CMSIS headers and system files. All these directories shall be added to the compiler’s include search path.

Starting with ROM version 4, the ROM library functions must be included with the header file \textit{rom\_lib.h} in the source files of the NVM user project:

\begin{verbatim}
#include "rom\_lib.h"
\end{verbatim}

This file provides wrapper macros to call the ROM library functions via a ROM-based table. All headers which define ROM library functions are already included in this file.

The ELF file \texttt{image\_rom.elf}, which provides symbol information of the ROM image to the linker, is located in the directory \texttt{smack\_rom\build\image}. The path of this file must be passed to the linker so that the linker can set the proper target addresses for calls of ROM library functions. Also, this file provides the symbols that mark the border between ROM-managed parts of the RAM. These symbols are mentioned in the chapter “RAM”.

\textbf{NVM library}

The NVM library provided with the SDK is a supplement to the ROM library. It is located in the directory \texttt{smack\_lib}, with a subdirectory layout as shown below:

\begin{verbatim}
\end{verbatim}

The \textit{inc} folder contains the header files with the declarations of the library functions. It must be added to the compiler’s include search path if the NVM library is used in a project.

The implementation of the functions is collected in the library file \texttt{libsmack.a} in the directory \texttt{smack\_lib\build\image}. The path to the library file must be added to the library search path of the linker,
and the library itself must be passed as an option to the linker as `smack` (without the leading `lib` and the extension `.a`).

This is an excerpt of the makefile in `smack_sl`. The directory with the NVM library is specified as a wildcard search, so new libraries which may be added in the future will also be found. Please note that here only arguments to a linker switch are specified; the command line with the proper switches for the linker is assembled in another makefile.

```make
# libraries to link
LINKER_LIBRARIES := smack
# ...and the directories to search for them
NVM_LIB_DIRS := $(sort $(wildcard $(REPO_ROOT_DIR)/smack_lib*))
NVM_LIB_BUILD_DIRS := $(addsuffix /build/image/, $(NVM_LIB_DIRS))
```

### Build shell

The SDK is simply unpacked into a directory; it is not installed through a software management system of the operating system and does not modify the environment, e.g., search path, of the host computer. Instead, the SDK provides scripts which set the environment in temporary shells or use dynamically generated absolute path specifications to call the build tools and the compiler.

The standard procedure to build a firmware image is by opening a command line and calling the `make` tool. The SDK provides a batch file `build_shell.bat` in the top-level directory, which opens a command line window. The batch file sets up the environment, e.g., adds the build tools to the `PATH` variable of the command line window. Now the `make` tool can be started on the command line to build the firmware.

The makefiles support the standard targets `all` to build a firmware image and `clean` to delete the firmware image and the intermediate files. Within the SDK, the main makefiles are usually named `Makefile`. They may include several other makefiles provided in the SDK, with different base names and the suffix `.mk`.

There is one makefile on the top-level directory of the SDK, which searches further for a set of project directories and calls `make` in these project directories. There, the local makefile is processed with the same target as given to the top-level makefile. This is the list of projects in the top-level makefile:

```make
NVM_PROJ_DIRS := smack_sl
```

In its directory, the top-level makefile, as provided in the SDK, searches for the project folder `smack_sl` of the sample project and calls a second instance of `make` within this directory which builds the target there. If further projects shall be built, their directories can be added to the line shown above, and the top-level makefile will call a second instance of `make` in each of the directories. As an example, if a project `smackfw_myproject` is added to the list, the line in the top-level makefile will look like this:

```make
NVM_PROJ_DIRS := smack_sl smackfw_myproject
```

The makefile in the top-level directory may have some dependencies to set up the SDK environment for the build tools, which the makefiles in the project directories do not have. Therefore, `make all` must be called from the build shell after unpacking the SDK. Even if calling `make` only from within the project directory as the Eclipse project file is doing, it is a good idea to call `make all` in the build shell in the SDK’s top-level directory from time to time.

`make clean` may not clean all files on the first call, as Windows sometimes reports files and directories to be deleted as busy, and rejects the operation. The files left over may cause problems during the next firmware build. Please repeat `make clean` a few times if this happens.
Visual Studio Code

In the .vscode directory, there is a set of project files which configure build and debug targets for the smack_sl sample project. The build process in this sample VSC configuration calls the make tool on the top-level directory of the SDK, with the help of a batch file that provides the SDK environment to a subshell. This is an excerpt of the tasks.json file:

```
"command": ".\run_task.bat",
"args": [
    "make all"
],
```

3.1 Eclipse

One of the IDEs that can be used to develop NAC1080 firmware is Eclipse CDT, the C/C++ edition of the Eclipse framework. In Eclipse, NAC1080 projects must be configured as external makefile projects. The SDK provides a batch file which, shall be used as the make tool.

The sample project smack_sl within the SDK also contains the sample Eclipse project files .project and .cproject. They are located in the directory of the smack_sl sample project. The scope of this project only covers this directory, and so does the build process when called from within Eclipse. Therefore, it is important to do a make all in the top-level directory of the SDK from a build shell after unpacking the SDK, and every time, when modifications of files outside the current directory are suspected.

Please note that all firmware projects must be located in a subfolder of the root directory of the SDK. In this subfolder, there must be a Makefile, and the header files must be located in a folder inc, and the source files in a folder src.

Copying an Eclipse project

An easy way to create a new Eclipse project for the NAC1080 is to copy the files .project and .cproject from the smack_sl folder to the directory with the new project. The settings for the build process are already configured in the .cproject file and do not need to be modified. In the .project file, you may want to edit the project's name. Look for this line and change “smack_sl” to the desired name:

```
<name>smack_sl</name>
```

After saving the file, the new project can be imported into an Eclipse workspace. Open the Import dialog with File→Import. In the dialog, select Existing Projects into Workspace from the category General and click the button Next to advance to the Import Projects dialog. There you can use the Browse button to search for the project directory, and Eclipse will search for the Eclipse project files underneath this directory. They can be selected or deselected for import; projects already present in the workspace are listed in gray. When the Finish button is clicked, the projects are finally imported into the workspace.

Creating a new Eclipse project

When creating a new project, it must reflect some requirements of the NAC1080 SDK. The project has to be configured as an external makefile project. Please do not use a managed make option for NAC1080 projects.

The following description assumes that the project directory exists and already contains the makefile and the source files. However, this is not required. The project folder can be created in this process, and usually, the source files are developed in Eclipse once the project has been added to the workspace. Depending on the project wizard selected, you may need to have a makefile ready.

To create a new Eclipse project, open the Template dialog in Eclipse with File→New→Makefile Project with Existing Code. Enter a project name, select the project directory, and pick the toolchain for the indexer that is
3 Build

the best match to the GCC used in the NAC1080 SDK. After clicking the Finish button, the project is added to the workspace but must be edited before it can be used.

![Import Existing Code](image)

**Figure 5  Import existing code**

All of the following settings must be configured for all of the configurations available in the Eclipse projects, or the build from within Eclipse will fail in some of the configurations. Usually, the configurations are named Debug and Release. In the example below, with a new Eclipse project created from the wizard Makefile Project with Existing Code, there is only one configuration Default.

First, the build settings must be configured to use the NAC1080 SDK. In the Project Explorer window, right-click the project just created, then select Properties at the bottom of the context menu. In the page C/C++ Build, make the adjustments described below and shown in the following screenshots.

A standard make tool will not provide the proper environment for the NAC1080 SDK. The Eclipse project must be configured to call a batch file which performs the proper calls to the SDK’s make tool, and to use the provided Makefile instead of generating own makefiles based on rules for toolchains other than the NAC1080 SDK (e.g., disable managed make). Uncheck the checkboxes Use default build command and Generate Makefiles automatically, delete any content in the field Build directory, and enter the following command in the field Build command:

```
${ProjDirPath}/../scripts/eclipse_eval_make_errorlevel.bat
```
In the second tab, Behavior, keep the default build targets *all* and *clean*. There is also a configuration option for parallel build. Disabling parallel builds helps to identify the context of error messages generated during the build process, while enabling parallel builds will speed up the build process but will mangle the output of the parallel build threads.
Eclipse helps you write the source code by managing a C/C++ index in the background with information about function prototypes and data structures. This index is the source for auto completion when typing the name of a member of a struct, or when jumping to the prototype of a function. To build this index, Eclipse must know where to find the header files.

To configure the paths to the include files, expand the category **C/C++ General** and select the dialog **Paths and Symbols**. There you can enter a path by clicking on the **Add** button. In the dialog that opens enter the path to one directory using Eclipse variables such as `${ProjDirPath}` where possible to make the Eclipse project independent of its root path. Select the checkboxes **Add to all configurations** and **Add to all languages**. Repeat this for all paths below:

\[
\begin{align*}
  &${ProjDirPath}/inc \\
  &${ProjDirPath}/../smack_rom/libs/smack_lib/inc \\
  &${ProjDirPath}/../smack_rom/libs/CMSIS/ifx/smack_series/inc \\
  &${ProjDirPath}/../smack_rom/libs/CMSIS/inc
\end{align*}
\]

When these settings are modified, it may be necessary to trigger an update of the index database manually with **Project→C/C++ Index→Rebuild**.
If the Eclipse version used does not provide the option *Makefile Project with Existing Code*, other templates can be used to create the project, preferably a Makefile project template, but also C/C++ project templates. In all cases, the project settings must be modified as mentioned above.

Figure 8  C/C++ General: Paths and Symbols

![Figure 8: C/C++ General: Paths and Symbols](image)
4  Debugging

The NAC1080 provides a two-wire SWD port to access the embedded Arm® Cortex®-M0 core.

4.1  Lauterbach

Lauterbach provides a set of debug adapters for different architectures. Most models suitable for an Arm® Cortex®-M0 should be able to connect to the NAC1080. The debugger IDE is named Trace32.

The NAC1080 SDK provides a set of scripts and batch files for instant use of a Lauterbach debugger. Simply start the batch file `t32_start_real.bat` to open a debug session.

The Lauterbach Trace32 software must be installed for the scripts to work. Please install the 64-bit version of the software, as the scripts may not fully support the 32-bit version.

Please do not use white spaces (blanks) in the paths to the Lauterbach software or the firmware projects, or the scripts may not work.

**Customization**

If you have installed the Trace32 software to a location other than the default, please edit the path in the `t32_start_real.bat` file, and also in the other batch files you intend to use. The path is specified in the batch file in a line like this:

```
if %OS%==64BIT set T32_DIR=c:\T32\bin\windows64
```

The script defines the firmware application to be loaded into the target. If the name of the project folder differs from the default `smack_sl`, please locate the following line in the `t32_start_real.bat` file and change it to the name of your project folder:

```
set SMACK_NVM_BUILD_DIR=%CD%\smack_sl\build
```

4.2  SEGGER J-Link

SEGGER provides debug solutions for different platforms, with the J-Link being a popular debug probe. Many development environments contain support for the J-Link, and in addition, SEGGER provides its own standalone debug solution, Ozone.

**J-Link Base vs. J-Link Plus**

The J-Link Plus comes with licenses to software features which are not available on a J-Link Base. According to the SEGGER website, a J-Link Base can be updated with those licenses at a later time.

Among other features, the J-Link Plus comprises licenses for Flash breakpoints and for the Ozone debugger. The Ozone license is needed only for debugging with the Ozone debugger, whereas Eclipse and VSC are accessing the J-Link through the GDB server, which is also working with the J-Link Base.

Before using Flash breakpoints, please read the chapter “Flash breakpoints and HardFault handler” and set the desired mode within the J-Link debugger configuration. Having Flash breakpoints enabled may render the chip unusable if the debug session is not exited cleanly.

**Customization**

If you have installed the J-Link software to a location other than the default, please edit the path in the environment you are using. For Eclipse and VSC, please follow the instructions in the respective chapters.
Customizing the device database

The J-Link software installs a database containing hundreds of devices that can be accessed out of the box. New devices such as the NAC1080, however, may require an extension to this database. There are several ways to apply this extension, depending on the tool to be used.

Some of the programs in the J-Link suite support a command line switch `-JLinkDevicesXMLPath`. After the switch, provide a path to a directory containing a file `JLinkDevices.xml` with definitions of the new device. For the NAC1080, search the directory `tool_config\jlink` and specify its path on the command line. This is the scheme that is used with Eclipse and VSC.

For programs where the `-JLinkDevicesXMLPath` option cannot be used, you may add the NAC1080 to the global XML file. This however requires administrator privileges – see below for how to achieve a similar goal as a standard user. See the J-Link documentation for reference. For the J-Link software (GDB server, J-Link Commander, etc.) and the Ozone debugger, these modifications have to be applied in their respective directories, as Ozone is installed to a different path than the standard J-Link tools.
4 Debugging

Locate the directories with the J-Link software and open the file JLinkDevices.xml with an editor. Search for the first device entry, and insert the NAC1080 entries just in front of it. The entries to be used for the NAC1080 can be taken from the XML file in the directory tool/config/jlink of the SDK. The top of the file then may look like this:

```
<DataBase>
<!-- Infineon Smack -->
<!-- Altera -->
<!-- Atmel -->

<ChipInfo Vendor="Infineon" Name="NAC1080" Core="JLINK_CORE_CORTEX_M0" WorkRAMAddr="0x00020800" WorkRAMSize="0x0800"/>
<FlashBankInfo Name="NVM CODE" BaseAddr="0x00010000" MaxSize="0xf000" Loader="image_ram.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" AlwaysPresent="1"/>
</Device>

<ChipInfo Vendor="Infineon" Name="NAC1081" Core="JLINK_CORE_CORTEX_M0" WorkRAMAddr="0x00020800" WorkRAMSize="0x0800"/>
<FlashBankInfo Name="NVM CODE" BaseAddr="0x00010000" MaxSize="0xf000" Loader="image_ram.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" AlwaysPresent="1"/>
</Device>

<ChipInfo Vendor="Infineon" Name="NGC1081" Core="JLINK_CORE_CORTEX_M0" WorkRAMAddr="0x00020800" WorkRAMSize="0x0800"/>
<FlashBankInfo Name="NVM CODE" BaseAddr="0x00010000" MaxSize="0xf000" Loader="image_ram.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" AlwaysPresent="1"/>
</Device>

<ChipInfo Vendor="Infineon" Name="Smack" Core="JLINK_CORE_CORTEX_M0" WorkRAMAddr="0x00020800" WorkRAMSize="0x0800"/>
<FlashBankInfo Name="NVM CODE" BaseAddr="0x00010000" MaxSize="0xf000" Loader="image_ram.elf" LoaderType="FLASH_ALGO_TYPE_OPEN" AlwaysPresent="1"/>
</Device>

<ChipInfo Vendor="Altera" Name="Cyclone V" Core="JLINK_CORE_CORTEX_A9" WorkRAMAddr="0xFFFF0000" WorkRAMSize="0x00010000"/>
<FlashBankInfo Name="QSPI Flash" BaseAddr="0x00000000" MaxSize="0x02000000" Loader="Devices/Altera/Cyclone_V/Altera_Cyclone_V_QSPI.elf" LoaderType="FLASH_ALGO_TYPE_OPEN"/>
</Device>

<ChipInfo Vendor="Atmel" Name="Cyclone V" Core="JLINK_CORE_CORTEX_A9" WorkRAMAddr="0xFFFF0000" WorkRAMSize="0x00010000"/>
<FlashBankInfo Name="QSPI Flash" BaseAddr="0x00000000" MaxSize="0x02000000" Loader="Devices/Atmel/Cyclone_V/Atmel_Cyclone_V_QSPI.elf" LoaderType="FLASH_ALGO_TYPE_OPEN"/>
</Device>

<!-- Ozone -->
```

In the device entries, there is an item Loader which specifies an ELF file that acts as a driver for the Flash memory inside the device. Copy the ELF files listed for the NAC1080 (e.g., image_ram.elf) from the SDK directory tool/config/jlink to the proper path, e.g., the location of the JLinkDevices.xml file.

If you do not have administrator privileges on your computer, locate the directories with the J-Link software and the Ozone debugger (if used) and copy them to a directory where you have access rights. Within these copies, apply the modifications to the JLinkDevices.xml files as described above, and start the J-Link programs out of these directories.
4 Debugging

4.2.1 Ozone

Ozone is the debugger of SEGGER and works with the license provided in the J-Link Plus. When starting Ozone, you will be asked for the connection settings, the device and the application to debug. For the application, browse for the ELF file that was generated during the build process, usually `image_nvm.elf`.

4.2.2 Eclipse

Eclipse is a popular framework for software development systems, and many systems build upon it. It can be expanded with plug-ins and comes in very different forms. To use it for developing NAC1080 firmware, make sure it has CDT support bundled.

Depending on the source of your Eclipse installation, it may already have a J-Link plug-in installed. If this is missing, search for it on Eclipse community sites. The plug-in is not provided by SEGGER, and SEGGER also does not provide support for it. After installation, there should be an option `GDB SEGGER J-Link Debugging` in the `Debug Configurations` dialog.

The plug-in connects to the GDB server `JLinkGDBServerCL.exe` of the J-Link software, whose path has to be configured in Eclipse and the J-Link plug-in. By default, the J-Link plug-in already has the path configured as a combination of two variables (“`${jlink_path}/${jlink_gdbserver}`”), and this should be kept there. Instead, configure the values of these variables in the preferences dialog accessible through `Window → Preferences`. In the category `MCU` there should be an entry `Global SEGGER J-Link Path`, which offers to configure the name of the folder and the executable of the GDB server.

![Eclipse Preferences: J-Link Path](image)

Figure 9 Eclipse Preferences: J-Link Path
4 Debugging

Now, a debug configuration can be added using the menu Run → Debug Configurations. When creating the debug configuration, the application has to be configured at the first tab, Main. Click Search Project… and pick the proper ELF file, or enter its path relative to the .project file of the Eclipse project.

![Debug Configuration](image)

**Figure 10** J-Link Plugin: Tab 1

On the second tab, Debugger, most of the defaults should be ready to use. Among other options, here the device name and the path to the JLinkDevices.xml file have to be given. In the field Other options the command line switches to the J-Link GDB server have to be configured. Here is an example:

```
-singlerun -strict -timeout 10000 -nogui -device NAC1080 -ir -nosilent -halt
-JLinkDevicesXMLPath ..\tool_config\jlink
```
On the third tab, **Startup**, configure whether to use Flash breakpoints, and set an initial breakpoint. Flash breakpoints come with a risk; please read **Flash breakpoints and HardFault handler** if you are considering enabling them.
4.2.3 Visual Studio Code

VSC connects to J-Link debugger through a plug-in provided by the community. If this is not yet installed, search for a J-Link plug-in. The plug-in starts the J-Link GDB server when a debug session is run.

In the NAC1080 SDK, there is a subdirectory .vscode with two JSON files used by VSC. In the file launch.json, some debugging sessions are already configured. In the file tasks.json, the build tasks are configured. With these files, you can use VSC immediately to build and debug firmware. Just chose Run → Start Debugging from the menu or press F5.

When the debug session starts, check the profile, e.g., on the upper left of the window. If it has started with a wrong profile, abort the debug session, switch to the desired profile and start a new session.
4 Debugging

4.2.4 J-Link Commander and command files

J-Link Commander is a utility installed with the J-Link software that allows the user to connect to a target, view the target’s state and write the NVM. It allows the user to specify a command file on the command line, which makes it suitable for NVM programming in production facilities.

The J-Link Commander's file name is JLink.exe, and it is installed in the standard SEGGER J-Link directory path, usually C:\Program Files (x86)\SEGGER\JLink. In the directory tool_config\jlink of the NAC1080 SDK, there is a command file flash_nvm.jlink with all the J-Link Commander commands required to write the application firmware to the NVM of the NAC1080. Just open a command line shell and enter the following commands (adjust the path to suit your directory layout):

```
cd <path of your tool_config\jlink directory>
“C:\Program Files (x86)\SEGGER\JLink\JLink.exe” -commandfile flash_nvm.jlink
```

Note: The JLinkDevices.xml in the directory of the JLink.exe file must have support for the NAC1080. If you have made a local copy of the J-Link installation to add the NAC1080 configuration, please call the JLink.exe with the path to your local directory.

The flash_nvm.jlink file simply lists the commands required to write a HEX file to NVM. Here is a short description of the commands:

```
device Smack
```

Select the proper device from the JLinkDevices.xml file. This sets the NVM layout and the proper scheme to write to NVM.

```
si SWD
```

Select SWD as the interface type to the target.

```
speed 4000
```

Set 4000 kHz clock speed on the SWD interface.

```
r
h
```

Put the target CPU into reset (“r”) and hold state (“h”). This prepares the NAC1080 for the next step.

```
loadfile ../../smack_sl/build/image/image_nvm.hex
```

Write the given file into NVM. If you are using a different path, please modify this line.

```
exit
```

Close the J-Link Commander and return to the command line.
4.3 Flash breakpoints and HardFault handler

Flash breakpoints are helpful in any of the debug environments; however, there is a caveat. The NAC1080 provides a limited number of hardware breakpoints, which may already be used by the IDE for single-stepping. Also, setting one breakpoint on a source code line may result in more than one breakpoint to be set in the assembler code, depending on the code generated from the source. In all these cases, a debug environment may use Flash breakpoints independently of the CPU breakpoint capabilities, and give a more flexible debugging experience.

Flash breakpoints are achieved by replacing the instruction at the breakpoint location with a BRK instruction. When the BRK instruction is executed, it generates an exception which the debugger can then handle, e.g., execute the original instruction instead and continue. However, if no debugger is connected to handle the BRK, the CPU will generate a HardFault.

On a HardFault, the ROM code, by default, executes a reset to restart the CPU and resolve the issue this way. However, when the condition for the HardFault still persists after the reset, the HardFault handler is executed again, and the CPU may get stuck in a boot loop. This will not only happen if the firmware code contains a serious issue but also if a BRK instruction is left from a debug session that could not be terminated in a clean manner.

As the debugger needs an amount of time to connect to the CPU after a reset, it cannot catch an early BRK instruction, and the CPU will issue a HardFault exception. As the HardFault handler initiates a reset of the CPU, a boot loop scenario is likely to happen, which may even lock out the debugger from connecting to the target at all. When working with Flash breakpoints, consider enabling a user HardFault handler through APARAM, which consists of a simple delay loop that gives you the time to start a debug session after a HardFault has happened and before the reset is executed.
This chapter describes the memory layout of the NAC1080. The memory comprises three different types of memory: ROM, NVM (Flash memory) and RAM.

**ROM**
ROM memory starts at address 0. It holds the vector table (including the reset vector) and the ROM library. The ROM library contains the start-up code, an NFC state machine with support for the proprietary NAC1080 protocol PROT_NAC, and a library of functions that can be used by firmware written to the NVM. The NFC state machine interacts with an NFC reader, with or without a customized firmware in NVM. It offers a mailbox for communication with the device. An NVM firmware can register itself to the NFC state machine, and can access the data in the mailbox in order to communicate with an NFC reader. The ROM library provides functions to interact with the NFC state machine and the peripherals of the chip.

**NVM**
The NVM is partitioned into three parts: DPARAM, APARAM and a firmware part. DPARAM holds configuration data defined during manufacturing and must not be modified. APARAM configures options of the ROM library and the interface to the NVM user firmware, and can be considered as a part of the user firmware. The format of this block is defined in a header file of the ROM library, and the source file with the definition of its content is usually part of the NVM firmware. The remainder of the NVM is available for the user firmware. The firmware sample provided in the SDK puts a version struct at the end of the NVM.

**RAM**
The RAM is divided into two parts of the same size. In both of the parts, the ROM library allocates some space at the beginning for its own purpose, e.g., variables and data buffers, which must not be modified by the NVM firmware. The boundaries of these areas may change with ROM versions and are made available to the linker of the NVM firmware through symbols provided by the ROM image.

### 5.1 DPARAM
DPARAM holds configuration data defined during manufacturing and must not be modified. The DPARAM block is located at address 00010000H and has a size of 400H bytes.
The base address and size depend on the ROM version. The ROM library provides functions to retrieve this information:

```c
const Dparams_t* dparam_pointer_get(void);
uint16_t dparam_size_get(void);
```

The fields in this block are described in the ROM library in the file `dparam.h`. The field `chip_uid` holds the chip's unique identifier, which is used in NFC. The fields `cal_stb_osc` and `cal_main_osc` hold the measured values of the standby and main oscillator frequencies and may be used to enhance the accuracy of timings.

### 5.2 APARAM

APARAM configures options of the ROM library and the interface to the NVM user firmware, as well as access rights of NFC requests.

The APARAM block is located at address 00010400H and has a size of 400H bytes.

The base address and size depend on the ROM version. The ROM library provides functions to retrieve this information:

```c
const Aparams_t* aparam_pointer_get(void);
uint16_t aparam_size_get(void);
```

The fields in this block are described in the ROM library in the file `aparam.h`. APARAM is intended to be configured as part of the NVM customer firmware. The firmware project shall contain a definition of this block, and the content shall be thoroughly examined. The example project in the SDK contains a file `sl_aparam.c`; however, this file is not configured for high security and is not intended to be used in commercial firmware without modifications.

#### 5.2.1 Handler

The exception vectors of the CPU point to handler functions inside the ROM. Many of them optionally call handler functions located in NVM, if configured in APARAM.

The vector table of the microcontroller holds entries for specific purposes such as reset handler or hard faults, and entries for processing interrupts of peripherals. Some of the interrupts are handled inside the ROM, e.g., reception of an NFC frame is handled inside the ROM code. For other interrupts, the ROM code offers an option to pass processing to a customized handler inside the NVM.

The ROM handlers that support nesting to NVM handler functions, when called, first read a function pointer located in APARAM. If the pointer directs to an address inside the NVM, the handler function in the NVM is called, otherwise the ROM handler continues. This way, the NVM handler replaces the ROM handler functionality.

If an NVM handler shall be called, a pointer to this function is configured for the respective field of the APARAM struct. To disable the call of an NVM handler, set the field to FFFFFFFFH, e.g., set the “erased” value of the NVM cell.

Handler functions in NVM are subroutines and take no argument, and return no value.
The ROM handler functions are similar to this pseudo code:

```c
void nvm_handler(void)
{
    fptr = aparams.nvm_handler;
    if (is_nvm_address(fptr))
    {
        fptr();
    }
    else
    {
        // serve interrupt in ROM
    }
}
```

An example of an NVM handler function can be found in the sample project in the SDK. In the file `sl_params.c` of the sample project, this handler is configured for the SysTick interrupt:

```c
void example_handler(void)
{
    example_counter++;
}
```
These entries in APARAM configure NVM handler functions:

<table>
<thead>
<tr>
<th>Function Pointer</th>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sense_adc_hand_addr</td>
<td>[0x503:0x500]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>timer0_hand_addr</td>
<td>[0x507:0x504]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>timer1_hand_addr</td>
<td>[0x50b:0x508]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>timer2_hand_addr</td>
<td>[0x50f:0x50c]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>timer3_hand_addr</td>
<td>[0x513:0x510]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio_evt_hand_addr</td>
<td>[0x517:0x514]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio_evt_gen_hand_addr</td>
<td>[0x51b:0x518]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>timer4_hand_addr</td>
<td>[0x51f:0x51c]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>timer5_hand_addr</td>
<td>[0x523:0x520]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>uart_hand_addr</td>
<td>[0x527:0x524]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>ssp_hand_addr</td>
<td>[0x52b:0x528]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>i2c_hand_addr</td>
<td>[0x52f:0x52c]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio0_hand_addr</td>
<td>[0x533:0x530]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio1_hand_addr</td>
<td>[0x537:0x534]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio2_hand_addr</td>
<td>[0x53b:0x538]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio3_hand_addr</td>
<td>[0x53f:0x53c]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio4_hand_addr</td>
<td>[0x543:0x540]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio5_hand_addr</td>
<td>[0x547:0x544]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio6_hand_addr</td>
<td>[0x54b:0x548]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>gpio7_hand_addr</td>
<td>[0x54f:0x54c]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>aes_hand_addr</td>
<td>[0x553:0x550]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>hard_fault_hand_addr</td>
<td>[0x557:0x554]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>stystick_hand_addr</td>
<td>[0x55b:0x558]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>wdt_hand_addr</td>
<td>[0x55f:0x55c]</td>
<td>absolute address of custom handler</td>
</tr>
<tr>
<td>nvm_hand_addr</td>
<td>[0x563:0x560]</td>
<td>absolute address of custom handler</td>
</tr>
</tbody>
</table>
5 Memory

5.2.2 Access privileges

The NAC1080 offers powerful access methods to its memory and peripherals through its proprietary NFC protocol enhancement. These access methods can be controlled and restricted by configuring access privileges in APARAM.

The NAC1080 offers a mailbox concept for communication between the device and a reader, e.g., a smartphone. For applications requiring a secure communication path, the access through the NFC interface can be restricted to just the mailbox functions. For special applications, the access privileges can be lowered.

Message disable

This field is a central control for direct access to the memory and hardware functionality inside the NAC1080. It is strongly recommended to disable access in production firmware unless intentionally used for the application:

```c
param32_t message_disable; /**< [0x45b:0x458] (32) <>0xffffffff: ignore external send message req */
```

The value which enables access is given by this macro defined in `nvm_params.h`:

```c
#define MESSAGE_AND_CALL_ENABLE       0xffffffff
```

All other values disable access. A new chip with an erased Flash has all access functions enabled. This setting controls access through NFC to the following functionality:

- NVM access (program, erase)
- GPIO control
- H-bridge control
- Call of function at a random address

Bypass mailbox

To write data to and read data from the mailbox, the proprietary enhancement to the NFC protocol offers a command to access the RAM area which defines the mailbox. By default, requests to addresses outside the mailbox are rejected. This can be changed by configuring the bypass mailbox setting:

```c
param32_t bypass_mailbox; /**< [0x44b:0x448] (32) 0xfca1fcb0 will bypass mailbox address check */
```

The value required to enable the bypass is defined in the following macro in `nvm_params.h`:

```c
#define BYPASS_MAILBOX        0xfca1fcb0
```

The exact value to enable the bypass depends on the ROM version. To disable the bypass, set the `bypass_mailbox` setting in APARAM to FFFFFFFF_H.

Address ranges

The mailbox mechanism offers functionality to read and write arbitrary locations within the address space of the CPU through a hardware abstraction layer (HAL). The address range is partitioned by the NFC protocol stack of the NAC1080, and for each partition, access can be separately granted or denied.
5 Memory

The NVM address range has a different scheme to control access than the other partitions, which offers control over each single page of the NVM.

The following table shows the address ranges of the partitions together with the associated field in APARAM.

<table>
<thead>
<tr>
<th>APARAM field</th>
<th>Start address</th>
<th>End address</th>
</tr>
</thead>
<tbody>
<tr>
<td>prot_rom1</td>
<td>00000000H</td>
<td>00001FFFH</td>
</tr>
<tr>
<td>prot_rom2</td>
<td>00002000H</td>
<td>00003FFFH</td>
</tr>
<tr>
<td>prot_ram1</td>
<td>00020000H</td>
<td>00021FFFH</td>
</tr>
<tr>
<td>prot_ram2</td>
<td>20000000H</td>
<td>20001FFFH</td>
</tr>
<tr>
<td>prot_hw1</td>
<td>2000A000H</td>
<td>2000C7FFH</td>
</tr>
<tr>
<td>prot_hw2</td>
<td>20010000H</td>
<td>20018BFFH</td>
</tr>
<tr>
<td>prot_hw3</td>
<td>40000000H</td>
<td>400003FFH</td>
</tr>
</tbody>
</table>

The values that are valid for these fields are defined in the following enumeration in nvm_params.h:

```c
enum nvm_prot_e
{
    nvm_prot_no_access  = 0x00,
    nvm_prot_read_only  = 0xf0,
    nvm_prot_write_only = 0x0f,
    nvm_prot_read_write = 0xff
};
```

For production firmware, it is recommended to disable access by setting all of the fields to nvm_prot_no_access (0) unless access shall be granted intentionally. Do not use values other than the ones listed in the enumeration to set the access protection fields. On a new chip with an erased NVM, the access to all partitions is enabled.

For the NVM access control, APARAM provides an array that can independently configure access protection for each NVM page:

```c
dparam_t    nvm_prot_sect[120];    /**< [0x5f7:0x580] (120*8) R/W protection of NVM pages
0..119    */
```

The access rights are defined in groups of two bits. Valid codes are:

<table>
<thead>
<tr>
<th>Protection type</th>
<th>Bit code</th>
</tr>
</thead>
<tbody>
<tr>
<td>No access</td>
<td>00B</td>
</tr>
<tr>
<td>Read only</td>
<td>10B</td>
</tr>
<tr>
<td>Write only</td>
<td>01B</td>
</tr>
<tr>
<td>Read/write</td>
<td>11B</td>
</tr>
</tbody>
</table>

Each entry of type `param_t` holds the access rights of four NVM pages.

The index of the array `nvm_prot_sect` selects a group of four NVM pages. Within this group, the lower bits select the NVM page at the lower address within these four pages.

For production firmware, it is recommended to disable access by setting all of the fields to 0 unless access shall be granted intentionally. On a new chip with an erased NVM, the access to all partitions is enabled.
Example: Calculate which bits configure access protection for the NVM page at address 0001D580H:

The NVM starts at address 00010000H, so the offset within the NVM is D580H.
The NVM page size is 128 bytes (80H bytes), so we calculate a page number within the NVM sector as 0580H/80H = 0BH, or 11D.
The index of the array defines the four-page group; in this example the index is 11D/4D = 2D.
The 2-bit group within the selected array member calculates as 11D mod 4D = 3D. To configure this page as read/write while all others are protected, the array member `nvm_prot_sect[11][2]` will be set to 11000000B, with the remainder of the array set to 0.

5.2.3 Miscellaneous

In the firmware, up to 16 user functions can be defined, which can be called by NFC. To call one of these functions, the NFC reader writes a CALL_APP_FUNCTION record with the function index to the mailbox and triggers it with a MESSAGE command. The NAC1080 then looks up the function pointer in APARAM and, if valid, calls the function:

```c
param_func_ptr_t app_prog[16]; /**< [0x447:0x408] (32 * 16) absolute address App function 0 through 15 */
```

To identify the device to a smartphone, a static NDEF type 2 tag can be stored inside the NVM. When a smartphone scans for NFC tags, it will read this NDEF tag and, depending on its content and the smartphone model, display it as text, open a web page or install an app which deals with the device. The NDEF tag is sent by the ROM-based NFC handler. The location of the NDEF tag is made known to the ROM in this APARAM entry:

```c
param_ptr_t    tag_type_2_ptr; /**< [0x57f:0x57c] (32) address of NFC tag information in NVM */
```

By default, the ROM start-up code enables the UART. The ROM can be instructed by an APARAM entry to leave the UART disabled, and the baud rate set by the ROM start-up code can be adjusted:

```c
param_t        default_uart_baudrate[4]; /**< [0x44f:0x44c] (24) default UART baudrate in baud */
param_t        disable_default_uart[4]; /**< [0x457:0x454] (32) 0x11223344 disable UART per ROM code */
```

The value to disable the UART is given in `nvm_params.h`:

```c
#define DISABLE_DEFAULT_UART 0x11223344   //!< 0x11223344 disable UART per ROM code
```

The SWD debug port can be disabled, if protection against reading back the NVM firmware is desired:

```c
#define DEBUGGER_OFF 0x5deb0ff5   //!< 0x5deb0ff5 will block debugger access
param32_t       kill_debugger; /**< [0x453:0x450] (32) 0x5deb0ff5 will block debugger access */
```
The GPIO pins are configured by the ROM start-up code if there are valid settings in APARAM. Each type of setting applies to all GPIO pins at the same time, e.g., if two GPIO pins are set to output in \texttt{gpio\_out\_en}, all others are set to input by the ROM start-up code.

On/off settings are configured in one word for all of the GPIO pins. The upper half of the word is a magic number, which indicates if the word is set intentionally and shall be considered by the ROM. The lower half configures the respective setting for the GPIO pins, with bit 0 targeting GPIO0. The magic number for the upper half of the APARAM word is given in \texttt{nvm\_param.h}, the APARAM entries in \texttt{aparam.h}:

```c
#define GPIO_VALUE_VALID 0xabba     //!< Default unique value for valid indication of GPIO params is gpio_value_valid

param32_t gpio_out_en;              /**< [0x483:0x480] (32) 0xabba + GPIO output enable */
param32_t gpio_in_en;               /**< [0x487:0x484] (32) 0xabba + GPIO input enable */
param32_t gpio_out_type;            /**< [0x48b:0x488] (32) 0xabba + GPIO output type */
param32_t gpio_pup_en;              /**< [0x48f:0x48c] (32) 0xabba + GPIO pull-up enable */
param32_t gpio_pdown_en;            /**< [0x493:0x490] (32) 0xabba + GPIO power-down enable */
param32_t gpio_out_value;           /**< [0x497:0x494] (32) 0xabba + GPIO output value */
```

Configuring the alternate functions of the GPIO pins requires more than one bit. Here, there is one APARAM entry for each of the GPIO pins, and one additional word as an indication that the configuration is valid and shall be set by the ROM start-up code for all of the pins. If this entry does not indicate a valid configuration, none of the alternate functions of the GPIO pins are configured. In the \texttt{gpio\_alt} members, the output alternative is encoded in bits 0...1, the input alternative in bits 2...4.

```c
#define GPIO_ALT_VALID 0xabbaabba   //!< default value 0xABBAABBA, indicates a valid alternate function config used by read_gpio_alt_nvm_config()

param32_t gpio_alt_valid;           /**< [0x49b:0x498] (32) GPIO Alt Valid */
param_t gpio_alt[16];               /**< [0x49f:0x49c] (32) GPIO Alt 0..3 */
```

The following entries configure the interrupt source for different interrupt vectors. The values to be written to the fields are identifiers for the interrupt source that generates the respective interrupt. When an interrupt is triggered, the ROM-based interrupt handler for this interrupt level is called, e.g., the handler that serves IRQ9. This handler looks up the interrupt source ID number in these entries, which defines the peripheral that is configured for this interrupt, e.g., PM0_IRQ_HAND for one of the system timer channels, then calls the appropriate ROM-based interrupt handler for this device. For most of the peripherals a user-defined handler

```c
```
can be configured, which is then executed as an alternative to the internal ROM-based handler, as described in chapter "Handler".

```c
param32_t evbus_handler1_source; /**< [0x4af:0x4ac] (32) 0x00 + custom source of evbus1 irq */
param32_t evbus_handler2_source; /**< [0x4b3:0x4b0] (32) 0x00 + custom source of evbus2 irq */
param32_t evbus_handler3_source; /**< [0x4b7:0x4b4] (32) 0x00 + custom source of evbus3 irq */
param32_t evbus_handler4_source; /**< [0x4bb:0x4b8] (32) 0x00 + custom source of evbus4 irq */
param32_t evbus_handler5_source; /**< [0x4bf:0x4bc] (32) 0x00 + custom source of evbus5 irq */
param32_t evbus_handler6_source; /**< [0x4c3:0x4c0] (32) 0x00 + custom source of evbus6 irq */
param32_t evbus_handler7_source; /**< [0x4c7:0x4c4] (32) 0x00 + custom source of evbus7 irq */
param32_t evbus_handler8_source; /**< [0x4cb:0x4c8] (32) 0x00 + custom source of evbus8 irq */
param32_t hp_irq9_cfg; /**< [0x4cf:0x4cc] (32) 0x00 + irq source of matrix irq */
param32_t hp_irq10_cfg; /**< [0x4d3:0x4d0] (32) 0x00 + irq source of matrix irq */
param32_t hp_irq11_cfg; /**< [0x4d7:0x4d4] (32) 0x00 + irq source of matrix irq */
param32_t hp_irq12_cfg; /**< [0x4db:0x4d8] (32) 0x00 + irq source of matrix irq */
param32_t hp_irq13_cfg; /**< [0x4df:0x4dc] (32) 0x00 + irq source of matrix irq */
param32_t hp_irq14_cfg; /**< [0x4e3:0x4e0] (32) 0x00 + irq source of matrix irq */
param32_t hp_irq9_col_cfg; /**< [0x4e7:0x4e4] (32) 0x00 + column config register */
param32_t hp_irq10_col_cfg; /**< [0x4eb:0x4e8] (32) 0x00 + column config register */
param32_t hp_irq11_col_cfg; /**< [0x4ef:0x4ec] (32) 0x00 + column config register */
param32_t hp_irq12_col_cfg; /**< [0x4f3:0x4f0] (32) 0x00 + column config register */
param32_t hp_irq13_col_cfg; /**< [0x4f7:0x4f4] (32) 0x00 + column config register */
param32_t hp_irq14_col_cfg; /**< [0x4fb:0x4f8] (32) 0x00 + column config register */
```

### 5.3 RAM

The RAM is divided into two parts of the same size of 8 kB.

The first block starts at address 00020000H.

At the beginning of this block, the ROM code allocates some space for its own purposes, e.g., a buffer which is used as a mailbox for data exchanged through NFC. This space must only be written to by the ROM library, with the exception of the mailbox.
5 Memory

The size of this area depends on the ROM version and typically is about 600\text{H} bytes. The end of this area is assigned to symbol \texttt{__romcode\_ram\_end\_}.

The stack must be located at the end of the first RAM block. Before starting a firmware in NVM, the ROM code checks the value of the initial stack pointer in the NVM vector table at address 10800\text{H} to validate if a user firmware exists in NVM.

The second block of RAM starts at address 20000000\text{H}.

At the beginning of this block, the ROM code allocates some space for its own purpose, e.g., buffers used for data exchanged through NFC. This space must only be written to by the ROM library.

The size of this area depends on the ROM version and typically is about 200\text{H} bytes. The end of this area is assigned to symbol \texttt{__romcode\_ram2\_end\_}. 

6 General-purpose input/output

The general-purpose inputs and outputs (GPIOs) of the NAC1080 are bidirectional pins. The output stages can be switched to open-drain operation. A GPIO port consists of:

- input stage
- output driver
- internal pull-up resistor
- internal pull-down resistor.

A GPIO port is configured by calling the function `single_gpio_iocfg()`. The ROM library also provides functions which can configure all GPIOs simultaneously from a set of parameters (`all_gpios_iocfg()`) or a configuration record preset in APARAM (`read_gpio_nvm_config()`).

The `single_gpio_iocfg()` takes six parameters, which separately enable the output and input stages, configure the output stage to push/pull or open drain, and enable internal pull-up and pull-down resistors for a given GPIO pin. The functions and parameters are described in the ROM library’s header files provided in the SDK. Please note that enabling the pull-up and pull-down resistors is not supported when open-drain mode is selected, for which an external pull-up is required.

The GPIO pins have IO multiplexers that switch the input and output signals from the GPIO registers to alternate functions, e.g., a UART controller. Please refer to the NAC1080 datasheet for a table of alternate functions available on the GPIO pins. These alternatives can be configured using the function `set_singlegpio_alt()`. The settings can also be configured from a preset record in APARAM with the function `read_gpio_alt_nvm_config()`.

Note: For GPIO pins used in a manner of half-duplex communications, they shall be configured as open-drain. Examples are single-wire (half-duplex) UART, SPI and I²C communication.
7 Timer

The NAC1080 comprises four different types of timer: the Arm® SysTick timer; a system timer with several 16-bit timers, one of them with PWM capabilities; a low-power standby timer; and an RTC timer that can still operate when only the standby power domain is powered.

The SysTick and system timers are clocked by the CPU clock while the standby timer and RTC are running on the low-power, low-frequency clock in the standby domain.

7.1 System ticker

The system ticker, also called SysTick, is a well-known 24-bit timer often found in Arm® microcontrollers.

The ROM library provides one function, `single_shot_systick()`, which inserts a delay based on the SysTick timer and uses the wait for interrupt (WFI) instruction to put the CPU into a low-power state until the timer interrupt – or any other interrupt – causes the CPU to continue. As the WFI instruction completes on any interrupt, the `single_shot_systick()` function may return before the desired delay has passed.

The NVM library provides a similar function `systick_singleshot_lib()`, which checks the SysTick’s status after the WFI instruction completes. The function returns when the SysTick timer generates an interrupt.

Please note that many of the ROM and NVM library functions are using the SysTick timer to put the CPU into a sleep mode whenever a delay is expected, e.g., when waiting for a device. The SysTick timer cannot be used in the user firmware across such function calls, as it is first reprogrammed and later stopped.

7.2 System timer

The system timer unit of the NAC1080 comprises six 16-bit timers that can be cascaded to form wider timers. Channel 0 supports the generation of PWM waveforms.

The system timer unit is clocked by the CPU clock.

The ROM and NVM libraries offer several functions to control the system timer unit. Most of them target one of the six timer channels, and some functions work with a cascaded timer pair that forms a 32-bit timer.

The first step when using a timer channel is to configure it by filling in a configuration struct and passing it to `sys_tim_chn_cfg()`. After setting the timer period with `set_sys_tim_chn_period()`, the timer can be started and stopped by calling `sys_tim_chn_control()`.

The library provides functions for single-shot delays that are using the WFI instruction to conserve power. Since the period of a 16-bit timer running on the CPU clock is short (about 2.3 ms), there is also a single-shot delay `sys_tim_singleshot_32()` that takes a 32-bit argument to achieve delays up to about 150 s.

The library also provides functions to set up and read a pair of cascaded timers. The two timers must be contiguous; when configuring the timer pair with the function `sys_tim_cyclic_cascaded()` it sets up the timer `channel` as the upper part of the timer pair, and the timer `channel - 1` as the lower part. When cascading timers, the period match of the lower timer is used as a clock of the higher timer. All six timers can be concatenated by configuring them with the function `sys_tim_chn_cfg()` to form a 96-bit timer.

The `sys_tim_cyclic_cascaded()` offers to configure periods of the upper and lower timer independently. The lower timer can be used as a prescaler, e.g., it can be set to wraparound after 1 ms. Then, the function `sys_tim_cyclic_cascaded_get_upper()` that reads the upper timer only returns milliseconds. If the period of the lower timer is set to its maximum value `FFFFH`, the timer pair can be regarded as one 32-bit timer. To read the combined counter value, the library provides the function `sys_tim_cyclic_cascaded_get_combined()`.

Note: Cascaded timers must complete one period before they properly switch to cascaded mode. After starting the timers, wait a few milliseconds, then read the current value and use it as a reference.
7 Timer

7.2.1 Pulse-width modulation

Timer 0 of the system timer block can be used to generate a PWM waveform. This signal can be routed to a GPIO pin.

To generate the PWM output, only a few steps are necessary:

- Configure a GPIO pin to output and set the alternate output function to PWM Out.
- Configure PWM function on system timer channel 0.
- Start the timer.

The configuration of the GPIO output and the alternate function is described in the chapter “General-purpose input/output”.

The ROM and NVM libraries provide an all-in-one PWM configuration function `sys_tim_pwm_config()`, which can configure the period and the duty cycle of the PWM. Both values are given as a number of clock ticks. As an alternative, the function `sys_tim_pwm_cfg()` allows the user to configure more details, but requires separate configuration of the timer configuration and period.

Finally, the timer has to be started. For PWM generation with timer 0, the library offers the special functions `sys_tim_pwm_start()` and `sys_tim_pwm_stop()`, which are wrappers to the more general function `sys_tim_chn_control()`.

This is an example for setting up and starting a PWM output:

```c
#define GPIO_PWM           1        // PWM output on GPIO1
#define PWM_PERIOD_TICKS   10000    // PWM period: max. 65535 (16 bits)
#define PWM_DUTY_TICKS     2000     // PWM duty: set to ~20% of period

// configure GPIO
set_singlegpio_alt(GPIO_PWM, 0, 3);
single_gpio_iocfg(true, false, true, false, false, GPIO_PWM);

// setup PWM...
sys_tim_pwm_config(PWM_PERIOD_TICKS, PWM_DUTY_TICKS);

// ...and start
sys_tim_pwm_start();
```

7.3 Standby timer

The standby timer unit is a 32-bit timer located in the standby domain of the NAC1080. It is clocked by the standby clock with a nominal frequency of 32768 Hz.

When started, the timer is running even in power-save mode when the CPU and the RAM are powered down, and can be used as an event to leave power-down mode. A capture register in the standby timer block stores the counter value when leaving power-save mode, enabling the CPU to determine the time spent in power-save mode if other wake-up sources besides the standby timer are configured.

The ROM library provides the function `config_stbtm()` to start and stop the timer as well as to configure the timer period. In a typical use case of the standby timer the time to be spent in power-save mode is configured with this function, then the CPU is put into power-save mode by calling `request_power_saving_mode()` with the standby timer selected as the wake-up source.

If the other wake-up sources, NFC or the WAKE_UP pin, are configured and trigger leaving power-save mode before the standby timer expires, the time spent in power-save mode can be determined by reading the capture register of the standby timer unit with the function `get_standby_time()`.
### 7.4 Real-time clock

The real-time clock (RTC) is a 32-bit timer located in the standby domain of the NAC1080. It is clocked by the standby clock and counts the number of seconds since it was enabled.

If the date and time is known when the RTC is started, the firmware can calculate the current date and time by adding the current RTC value. A possible use case is a battery-operated system, which is read out from time to time using a smartphone, where the smartphone app can send its current date and time each time it makes contact with the device.

Before the RTC can be used, it must be initialized by calling `rtc_init()`. For increased accuracy, this function also loads a calibration value into the RTC timer, which was measured during the production of the device and stored to DPARAM.

The RTC can then be started and stopped with `rtc_control()`. Please note that when the RTC is stopped, its counter resets to 0, and starts counting from 0 when enabled again.

The current time offset can be read with `rtc_get()`.
8 Power-save mode

The NAC1080 has different internal power domains, which enables power saving by switching off some of them. From a firmware perspective, the NAC1080 can be separated into a CPU and a standby power domain. The CPU power domain comprises the CPU core, the RAM, the NVM and the peripherals; however, some of them have their own power or clock domains and may be switched off even if the CPU is running. The CPU power domain being switched on is essential for firmware operation.

The standby power domain is enabled whenever there is a sufficient power supply to the device. It contains a state machine, which controls the basic operation of the NAC1080 as well as a standby timer and an RTC. The standby timer can generate a wake-up event after a configurable time span when the firmware decides to switch off its own power domain, and can restart the firmware by powering on the CPU again when the timer has expired. The RTC provides a monotonic time base, which can operate without interruption as long as power is supplied to the device, e.g., by a battery.

A further power domain exists in the NFC front end, which is powered by the radio-frequency (RF) field of an NFC reader device. With respect to the firmware, this power domain can generate a wake-up event when it is exposed to the RF field of the reader.

To save power in both battery and NFC-powered systems, strategies such as executing a WFI instruction can be used; an example is the `single_shot_systick()` function, which sets up the SysTick timer to generate a short delay. This stops the clock for parts of the CPU core until an interrupt is received, which saves some power. However, the CPU core and the memory systems are still powered and consuming energy.

For longer delays, where no action of the CPU is required and no I/O units such as UART and GPIO ports must be operating, the CPU power domain can be switched off. Then, only the standby power domain is still powered, and the device has its lowest power consumption. In this state, the NAC1080 can be restarted again by wake-up sources in the standby and NFC power domain:

- Standby timer expires.
- WAKE_UP pin reads the logical value configured as the wake-up event.
- Presence detection of an NFC field.

When the CPU power domain is switched on again after a wake-up event is received, the CPU is starting from a power-on reset. During power-save mode, the RAM is also disconnected from the power supply, and its content is lost. The firmware must determine the state mainly from reading the wake-up source with `get_wakeup_source()`, and other information it may be able to gather such as the count of the RTC.

The standby domain provides one 16-bit scratchpad register, which preserves its content throughout power-save mode. This register is used by the ROM code; in particular, bit 15 is modified while entering and leaving power-save mode. Several of the lower bits are not touched by the ROM code and can be used by the NVM firmware to pass some information between entering power-save mode and waking up again.

The power-save mode is activated by calling the function `request_power_saving_mode()`. When the power-save mode is entered, this function does not return, and the CPU power domain is switched off instead. The wake-up events that start the CPU again are configured by the function's arguments. In case the WAKE_UP pin is selected as a wake-up source, the polarity of the wake-up signal is also configured via a function argument.

When the CPU comes out of reset, the firmware can query the wake-up source with the function `get_wakeup_source()`. From the CPU perspective, this is a power-on reset, and may actually be a power-on event of the device. However, if `get_wakeup_source()` indicates one of the wake-up events that the firmware may eventually configure before entering power-save mode, the firmware may continue with a task other than it would on a real power-on reset of the device. So, the first thing to do in the firmware when its main function `_nvm_start()` is called is to distinguish between starting from power-save mode or from a real power-on reset.

To determine the time spent in power-save mode, start the standby timer before requesting to enter power-save mode. When the CPU comes out of reset, determine an eventual wake-up source with `get_wakeup_source()`. If the standby timer is reported, the time span in power-save mode was obviously the time that was configured for the standby timer to expire. If any other wake-up source is reported, determine the standby timer value at the time standby mode was left by calling `get_standby_time()`. Please remember that the
8 Power-save mode

timer runs on the low-speed clock (about 32.768 kHz) and that the value is captured when power-save mode is left and not when get_standby_time() is called.
9 AES encryption

The NAC1080 offers hardware-accelerated 128-bit AES encryption and decryption with a user-selected encryption key.

Once a 128-bit encryption key is loaded into the AES unit, 128-bit data blocks can be encrypted or decrypted with a minimum of software interaction. The ROM and NVM libraries provide functions which load the data from a source buffer into the AES unit, trigger the encryption or decryption, wait for completion, and copy the converted data to a destination buffer.

The AES unit operates with 32-bit registers, and the ROM library deals with words of 32-bit width. This minimizes the overhead when loading the 128-bit blocks, but has one drawback. The CPU is configured to run in little-endian mode, and when a word is loaded from RAM into a CPU register, the bytes are swapped. When dealing with byte streams, this may lead to unwanted results, and therefore the NVM library provides a similar set of functions that take pointers to unions holding byte arrays as an argument and convert between network and host byte order implicitly. Both sets of functions can be used mutually.

Please note that the pointers passed to the NVM library functions must still be aligned to word boundaries even if they are filled from byte arrays. For performance reasons, these functions are using word transfers too, even though they are slower than their ROM library counterparts as additional conversion steps and other issues are involved. If performance is a critical issue, set up the data flow through the system based on 32-bit words and use the ROM library functions.

The alignment to 32-bit words for the pointers passed to the NVM library functions is enforced by defining a union `aes_block_t` consisting of an array of bytes as well as an array of words with a total size of 128 bits. The buffers are passed to the functions as pointers, meaning that only a reference to the buffer is passed and data not copied by calling the functions but only once when the registers of the AES unit are read or written.

The ROM library functions also require pointers to word-aligned data buffers with a size of 128 bits to be passed. Here, no struct or union is defined, and the arguments are simply pointers to a `uint32_t`. It is the responsibility of the caller to ensure that the pointers target buffers of a size of at least four words.

Before data can be encrypted or decrypted, an encryption key with a length of 128 bits has to be loaded into the AES unit. This is achieved by the function `aes_load_key()`. If the key is available as a byte array, it can be directly passed to the function `aes_load_key_ba()`. This key is used for encryption as well as for decryption. There is no need to reload the key after a data block is encrypted or decrypted; however, it must be set again when one of the random number functions described below is executed.

Once the encryption key is loaded into the AES unit, a data block can be encrypted or decrypted with the function `calc_aes()`. The type of the operation is selected by the third argument of the function and can vary between the calls. When the function returns, the processed data is available in the buffer `result`. The buffer for the source data can be reused to accept the result. If the firmware shall operate on byte streams, the function `calc_aes_ba()` can be used instead to implicitly adjust the byte order.

The following code snippet shows the encryption of the plaintext block in `data` to a ciphertext in `encrypted`, followed by the decryption to the plaintext again, which would be performed by the receiver of the message. All data is stored in words. The assignment of the values to `key` and `data` just reads like the sample provided in the FIPS197 document; however, in the memory the byte order of each word is reversed due to the little-endian
configuration of the CPU. This must be considered when mixing byte and word access during the processing of the data, or when exchanging data with other devices that may use big-endian or network byte order.

```c
static void aes_rom(void)
{
    // sample key, data (plaintext) and encrypted string (ciphertext) taken from FIPS197
    static const uint32_t key[4] = {0x00010203, 0x04050607, 0x08090a0b, 0x0c0d0e0f};
    static const uint32_t data[4] = {0x00112233, 0x44556677, 0x8899aabb, 0xccddeeff};
    uint32_t encrypted[4];
    uint32_t decrypted[4];

    // load the key to be used in the following operations into AES unit
    aes_load_key(key);

    // encrypt the data block to a ciphertext
    calc_aes(encrypted, data, encrypt);
    // expected content in "encrypted": 0x69c4e0d8, 0x6a7b0430, 0xd8cdb780, 0x70b4c55a

    // now decrypt the ciphertext to plaintext again, result in "decrypted" shall match "data"
    calc_aes(decrypted, encrypted, decrypt);
}
```

The next code fragment implements the same functionality, but now operating on byte arrays instead of word arrays. The byte order in the CPU’s memory is the same as listed in the source code. With this solution, blocks within a byte stream can be encrypted and decrypted without caring about the local host byte order. As the functions `aes_load_key_ba()` and `calc_aes_ba()` also deal with word arrays internally, the pointer to the data buffers must be word-aligned, which is expressed in the data type `aes_block_t`, a union of a byte array and a word array. In the example, it is initialized via the byte array, and the AES library functions access the same data through the word array for optimized performance.

```c
static void aes_nvm(void)
{
    // sample key, data (plaintext) and encrypted string (ciphertext) taken from FIPS197
    static const aes_block_t key = {{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
        0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f}};
    static const aes_block_t data = {{0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88,
        0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff}};
    aes_block_t encrypted;
    aes_block_t decrypted;

    // load the key to be used in the following operations into AES unit
    aes_load_key_ba(key);

    // encrypt the data block to a ciphertext
    calc_aes_ba(encrypted, data, encrypt);
    // expected content in "encrypted": 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x0e, 0x3f,
    // 0xcd, 0xb7, 0x30, 0x70, 0xb4, 0xc5, 0x5a

    // now decrypt the ciphertext to plaintext again, result in "decrypted" shall match "data"
    calc_aes_ba(decrypted, encrypted, decrypt);
}
```
The libraries provide several functions to generate random numbers with the help of the AES unit. Please note that the functions load their own encryption keys on every call, overwriting a previously loaded key. When using the random number and encryption functions at the same time, the user encryption key must be loaded again before the next encryption operation whenever a random number was generated.

The function `generate_random_number()` assembles a data set from various sources, including some that return noise. This data set is then scrambled by the AES unit, generating a true random number.

Getting the noise to generate true random numbers costs some time. For applications where a pseudo random number is sufficient, the function `generate_random_number_fast()` provides a faster result. This function also uses inputs which give random results on some systems but will produce a predictable sequence of numbers in many environments. The function `generate_random_number_lib()` is placed between the other functions, being a bit faster than `generate_random_number()` but using lesser random inputs for each calculation.
10 UART

This chapter shows how to initialize the UART peripheral on the NAC1080. The UART supports transmission and reception of characters with a width of 5 to 8 bits, framed by a start bit and one or two stop bits. Parity is also supported. The UART does not provide handshake and modem control lines (e.g., CTS, RTS). If required for an application, these may be emulated by a software driver using standard GPIO pins.

The UART ports can be mapped to GPIO pins using the alternate GPIO functions described in the datasheet.

ROM library and APARAM

By default, the UART is enabled and connected to GPIO pins by the ROM start-up code. The UART handler of the ROM reads data received by the UART and processes and responds to it as defined by the Smack communication and message handler. The default behavior can be controlled in the APARAM block located in the NVM, which is usually built with the NVM firmware.

To disable the initialization of the UART by the ROM, set the field `disable_default_uart` to the disable value. If the initialization of UART by the ROM shall be used, the baud rate can be set in the field `default_uart_baudrate`.

A customized interrupt handler for the UART is called, if the field `uart_hand_addr` points to an address inside the NVM. If no NVM interrupt handler is given, e.g., the field holds the empty value of FFFFFFFFH, the UART interrupts will be processed by the message handler of the ROM, and the received data will be interpreted as NAC1080 message frames.

Initialization of the UART

The initialization of the UART comprises the following steps:

- Initialize the UART to a known state (still disabled).
- Configure the GPIO pins to be used.
- Configure the UART interrupt.
- Enable the UART.

For the GPIO pins, both the alternate pin functions as well as the GPIO characteristics must be configured. The following code snippet shows an example for the initialization sequence.
```c
#define UART_TX_PIN     0
#define UART_RX_PIN     2
#define UART_BAUDRATE   115200
#define UART_IRQ        14

// Start of the application program
void _nvm_start(void)
{
    /*lint -esym(550,dummy) variable dummy not accessed */
    uint8_t  dummy __attribute__((unused));

    // prototypes of ROM library functions used in this project (see header files for details):
    // void init_uart(bool parity_en, parity_t even_odd, bool two_stop_bits, bool fifo_en,
    // uint8_t word_length, bool stick_parity)
    // void set_uart_baudrate(uint32_t baudrate)
    // void configure_uart_irq(uint8_t irq_num, uint8_t rx_fifo_level, uint8_t tx_fifo_level,
    // bool rx_int, bool tx_int)
    // void set_uart_control(bool enable, bool rx_enable, bool tx_enable)
    // void set_singlegpio_alt(uint8_t gpio, uint8_t ain_en, uint8_t outsel)
    // uint8_t single_gpio_iocfg(const bool out_enable, const bool in_enable, const bool outtype, const bool pup, const bool pdown, uint8_t gpio)

    // initialize UART (will still be paused)
    init_uart(false, even, false, true, eight, false);
    set_uart_baudrate(UART_BAUDRATE);

    // setup GPIO pins: set alternative function, configure I/O driver
    set_singlegpio_alt(UART_RX_PIN, 2, 0);
    dummy = single_gpio_iocfg(false, true, true, true, false, UART_RX_PIN);
    set_singlegpio_alt(UART_TX_PIN, 0, 2);
    dummy = single_gpio_iocfg(true, false, true, false, false, UART_TX_PIN);

    // configure interrupt
    configure_uart_irq(UART_IRQ, 0, 4, false, false);

    // finally, start UART
    set_uart_control(true, true, true);

    // Now, the UART is up and running. The dand_handler within the ROM will handle PROT_NAC frames received through the UART.

    // background task is just an endless
    while (true)
    {
        asm("WFI");
    }
}
```
The concept behind the different ROM and SDK versions for the NAC1080 are the same, but they differ in some details.

The SDK provides a build system for the NAC1080, which contains files that define the ROM library interface for a given ROM version, and that also define symbols which are used when linking the NVM user firmware. This means that the NVM firmware built with one specific SDK runs on one ROM version but may fail on others. This is especially true for ROM versions 3 and earlier. In ROM version 4 the ROM library interface provides updates for enhanced compatibility to future ROM versions with the aim that NVM firmware images built for ROM4 will also run on the next ROM version if some basic rules are followed. To avoid side effects, always use the appropriate SDK release for chips with a specific ROM version.

Usually, it is quite easy to start a project based on ROM version 3, port it to ROM4, and use automated merge features provided by source code repositories such as Git to keep the branches up to date during further development. The set of functions in the ROM library is very similar, sometimes having changes in names for functions, enums and macros. In rare cases, the values behind enums may have changed. When passing parameters to functions, do use the provided enums and macros instead of numeric values or general macros like `TRUE` and `FALSE`.

Header files

The ROM library interface of ROM4 requires inclusion of the header file `rom_lib.h`, which includes all other header files of the ROM library. In the source files of the NVM user firmware, only this file must be included. In ROM3, the file `rom_lib.h` does not exist, and the various header files have to be included separately when declarations from these files are used.

The header file `rom_lib.h` was introduced in ROM4 to implement indirect jumps to the functions of the ROM library. The header file contains macros which look up the function address in a table with a fixed base address, then call the function via this pointer. If the base address of the function changes in the next ROM version, the firmware will still be able to call it via the pointer but will fail to do so if a direct jump with static linkage was used instead.

From ROM3 to ROM4, many enums and macros have been changed to more distinctive names. The source code of the NVM firmware must reflect these changes. This is one example in `sys_tim_drv.h`, with common names in the enum in ROM3:

```c
enum control_t {
    start,    //!< start
    stop      //!< stop
};
```

In ROM4, a prefix was added to get more unique names, similar to a namespace in C++:

```c
enum sys_tim_control_E {
    sys_tim_start,   //!< start
    sys_tim_stop     //!< stop
};
```

Mailbox address

The address and size of the mailbox may vary between ROM versions. It is good practice to read the actual values during run time instead of using fixed values or linking statically against one specific ROM image.
For the NVM firmware, the ROM library provides the functions `get_mailbox_address()` and `get_mailbox_size()`. When accessing the mailbox from the NVM firmware, a pointer to the mailbox shall be declared, with the return value of `get_mailbox_address()` assigned to it. Then, this pointer can be used to read and write the mailbox. The same is true when accessing the mailbox from an NFC reader such as a smartphone. When the NFC protocol stack on the smartphone has established the communication to the NAC1080, the app switches the NAC1080 to the proprietary protocol state `PROT_NAC` by sending the appropriate NFC frames. In this state, the base address and the size of the mailbox can be retrieved with the functions `transmit_mailbox_addr` and `transmit_mailbox_size`. Both of them are accessed through `MESSAGE request`. After the parameters have been retrieved, the app can write and read the mailbox content using the frames `WRITE_WORD` and `READ_WORD` with the appropriate base address.

**APARAM**

Many of the fields in the APARAM struct have changed their types from ROM3 to ROM4, and the assignments of values to them has to be modified in the NVM firmware projects as well. A C source file with definitions of the APARAM content can be found in the `smack_sl` example project; if the compiler reports errors when migrating an NVM project from one ROM version to another, refer to the file `smack_sl\src\sl_aparam.c` as a template. Most of the modifications combine arrays with four elements of type `param_t` (8-bit unsigned integer) to one `param32_t` (32-bit unsigned integer). Examples are generic 32-bit values like `default_uart_baudrate`, and 32-bit markers like `disable_default_uart` where a declaration as one single entry makes assignments of values to one `param32_t` more obvious than assignments of a 4-byte array (as was required by the toolchain in previous ROM versions).

Further modifications were made to the declarations of access rights. The arrays `prot_addr1` and `prot_addr2` of ROM3's APARAM were split into several fields with names that give a hint of the memory range they apply to, e.g., `prot_rom1` and `prot_ram2`. The different members `nvm_prot_sect0[8]`, `nvm_prot_sect1[8]`, etc. were combined to an array `nvm_prot_sect[120]` with the index denoting a block of four pages.

**NVM library**

The NVM library is a supplement to the ROM library with functions linked to the NVM firmware into NVM address space at build time of the NVM firmware image. When delivered as part of an SDK targeting a specific ROM version, the NVM library is also built for this ROM version. It uses the same interface to the ROM library as the NVM firmware does.

With new versions of the SDK, the NVM library may provide new functions or enhancements and fixes to existing functions. In some cases, this may require adjustments in the source code of the NVM project.

**Chip markings**

<table>
<thead>
<tr>
<th>ROM version</th>
<th>Chip marking</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3.0</td>
<td>A21000</td>
</tr>
<tr>
<td>V4.0</td>
<td>A21004</td>
</tr>
</tbody>
</table>
## Revision history

<table>
<thead>
<tr>
<th>Document version</th>
<th>Date of release</th>
<th>Description of changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1.0</td>
<td>2022-10-25</td>
<td>• First release</td>
</tr>
</tbody>
</table>