Arm® RAN Acceleration Library

Version 23.10

Reference Guide

Non-Confidential

Issue 00

102249_23.10_00_en

Arm® RAN Acceleration Library
Reference Guide

Release information

Document history

Issue	Date	Confidentiality	Change
2010-00	2 October 2020	Non-Confidential	New document for Arm RAN Acceleration Library v20.10
2101-00	8 January 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.01
2104-00	9 April 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.04
2107-00	9 July 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.07
2110-00	8 October 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.10
2201-00	14 January 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.01
2204-00	8 April 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.04
2207-00	15 July 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.07
2210-00	7 October 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.10
2301-00	27 January 2023	Non-Confidential	Update for Arm RAN Acceleration Library v23.01
2304-00	21 April 2023	Non-Confidential	Update for Arm RAN Acceleration Library v23.04
2307-00	7 July 2023	Non-Confidential	Update for Arm RAN Acceleration Library v23.07
2310-00	6 October 2023	Non-Confidential	Update for Arm RAN Acceleration Library v23.10

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

This book contains reference documentation for Arm RAN Acceleration Library (ArmRAL). The book was generated from the source code using Doxygen.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
Note	An important piece of information that needs your attention.
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. Tutorials

This section contains tutorials to help you use Arm RAN Acceleration Library.

2.1 Get started with Arm RAN Acceleration Library (ArmRAL)

Describes how to build, install, run tests and benchmarks, and uninstall Arm RAN Acceleration Library (ArmRAL).

Before you begin

If you have not already downloaded Arm RAN Acceleration library, visit https://developer.arm.com/solutions/infrastructure/developer-resources/5g/ran/download to download the source code.

- Ensure you have installed all the tools listed in the **Tools** section of the `RELEASE_NOTES.md` file.
- To use the Cyclic Redundancy Check (CRC) functions, you must run the library on a core that supports the AArch64 PMULL extension. If your machine supports the PMULL extension, pmull is listed under the **Features** list given in the `/proc/cpuinfo` file.

Build Arm RAN Acceleration Library (ArmRAL)

1. Configure your environment. If you have multiple compilers installed on your machine, you can set the `CC` and `CXX` environment variables to the path to the C compiler and C++ compiler that you want to use.

   If you are compiling natively on an AArch64-based machine, you must set suitable AArch64 native compilers. If you are cross-compiling for AArch64 on a machine that is based on a different architecture, you must set suitable AArch64 cross-compilers.

   Alternatively, your C and C++ compilers can be defined at build time using the `-DCMAKE_C_COMPILER` and `-DCMAKE_CXX_COMPILER` CMake options. You can read more about these options in the following section.

   Note: If you are building the SVE or SVE2 version of the library, you must compile with GCC 11.1.0 or newer.

2. Build Arm RAN Acceleration Library. Navigate to the unpacked product directory and use the following commands:

   ```
   mkdir <build>
   cd <build>
   cmake {options} -DBUILD_TESTING=On -DBUILD_EXAMPLES=On -
   DCMAKE_INSTALL_PREFIX=<install-dir> <path>
   make
   ```

   Substituting:

   - `<build>` with a build directory name. The library builds in the specified directory.

- `{options}` with the CMake options to use to build the library.
- (Optional) `<install-dir>` with an installation directory name. When you install Arm RAN Acceleration Library (see **Install Arm RAN Acceleration Library**), the library installs to the specified directory. If `<install-dir>` is not specified, the default is `/usr/local`.
- `<path>` with the path to the root directory of the library source.

Notes:

- The `-DBUILD_TESTING=On` and `-DBUILD_EXAMPLES=On` options are optional, but are required if you want to run the library tests (`-DBUILD_TESTING`) and benchmarks (`-DBUILD_EXAMPLES`).
- The `-DCMAKE_INSTALL_DIR=<install-dir>` option is optional and sets the install location (`<install-dir>`) for the library. The default location is `/usr/local`.
- By default, a static library is built. To build a dynamic or a static library use the `–DBUILD_SHARED_LIBS={On|Off}` option.
- By default, a Neon-optimized library is built. To specify which type of optimized library to build (Neon, SVE, or SVE2), use the `-DARMRAL_ARCH={NEON|SVE|SVE2}` option.

Other common CMake `{options}` include:

- `-DCMAKE_INSTALL_PREFIX=<path>`

  Specifies the base directory used to install the library. The library archive is installed to `<path>/lib` and headers are installed to `<path>/include`.

  Default `<path>` is `/usr/local`.

- `-DCMAKE_BUILD_TYPE={Debug|Release}`

  Specifies the set of flags used to build the library. The default is `Release` which gives the optimal performance, however `Debug` might give a superior debugging experience. To optimize the performance of **Release** builds, assertions are disabled. Assertions are enabled in **Debug** builds.

  Default is `Release`.

- `-DCMAKE_C_COMPILER=<name>`

  Specifies the executable to use as the C compiler. If a compiler is not specified, the compiler used defaults to the contents of the `CC` environment variable. If neither are set, CMake attempts to use the generic system compiler `cc`. If `<name>` is not an absolute path, it must be findable in your current environment `PATH`.

- `-DCMAKE_CXX_COMPILER=<name>`

  Specifies the executable to use as the C++ compiler. If a compiler is not specified, the compiler used defaults to the contents of the `CXX` environment variable. If neither are set, CMake attempts to use the generic system compiler `c++`. If `<name>` is not an absolute path, it must be findable in your current environment `PATH`.

- `-DBUILD_TESTING={On|Off}`

Specifies whether to build (`On`), or not build (`Off`), the correctness tests and benchmarking code for the library. `-DBUILD_TESTING=On` enables the `check` and `bench` targets described later. If after you build the library, you want to run the included tests and benchmarks, you must build your library with `-DBUILD_TESTING=On`.

Default is `Off`.

- `-DARMRAL_TEST_RUNNER=<command>`

  Specifies a command that is used as a prefix before each test executable, such as where an emulator might be required. To see an example where `-DARMRAL_TEST_RUNNER` is used, see the **Run the tests** section.

- `-DSTATIC_TESTING={On|Off}`

  Most C/C++ toolchains dynamically link to system libraries like `libc.so` , however this dynamic link is unsuitable or unsupported in some use cases. Setting `-DSTATIC_TESTING=On` forces the compiler to link the tests statically by appending the `-static` flag to the link line.

  Default is `Off`.

- `-DBUILD_EXAMPLES={On|Off}`

  Specifies whether to build (`On`), or not build (`Off`), the examples in the examples folder. The example programs are simpler than the tests, and show how different parts of the library can be used. `-DBUILD_EXAMPLES=On` enables the `examples` and `run_examples` targets described later. If after you build the library, you want to run the included examples, you must build your library with `-DBUILD_EXAMPLES=On`.

  Default is `Off`.

- `-DBUILD_SHARED_LIBS={On|Off}`

  Specifies whether to generate a shared library (`On`) or a static library (`Off`). To generate `libarmral.so`, use `-DBUILD_SHARED_LIBS=On`. To generate `libarmral.a`, use `-DBUILD_SHARED_LIBS=Off`.

  Default is `Off`.

- `-DARMRAL_ENABLE_WERROR={On|Off}`

  Use (`On`), or do not use (`Off`), `-Werror` to build the library and tests. `-Werror` converts any compiler warnings into errors. Disabled by default to aid compatibility with untested and future compiler releases.

  Default is `Off`.

- `-DARMRAL_ENABLE_ASAN={On|Off}`

  Enable AddressSanitizer when building the library and tests. AddressSanitizer adds extra runtime checks to enable you to catch errors, such as reads or writes off the end of arrays. `-DARMRAL_ENABLE_ASAN=On` incurs some reduction in runtime performance.

Default is `Off`.

- `-DARMRAL_ENABLE_COVERAGE={On|Off}`

  Enable (`On`), or disable (`Off`), code coverage instrumentation when building the library and tests. When analyzing code coverage, it can be useful to enable debug information (`-DCMAKE_BUILD_TYPE=Debug`) to ensure that compiler-optimized lines of code are not missed. For more information, see the **Code coverage** section.

  Default is `Off`.

- `-DARMRAL_ARCH={NEON|SVE|SVE2}`

  Enable code that is optimized for a specific architecture: `NEON`, `SVE`, or `SVE2`. To use `-DARMRAL_ARCH=SVE`, you must use a compiler that supports `-march=armv8-a+sve`. To use `-DARMRAL_ARCH=SVE2`, you must use a compiler that supports `-march=armv8-a+sve2`.

  Default is `NEON`.

- `-DARMRAL_SEMIHOSTING={On|Off}`

  Enable (`On`), or disable (`Off`), building Arm RAN Acceleration library with semihosting support enabled. When semihosting support is enabled, `--specs=rdimon.specs` is passed as an additional flag during compilation and `-lrdimon` is added to the link line for testing and benchmarking.

  Note: If you use `-DARMRAL_SEMIHOSTING=On` you must also use a compiler with the `aarch64-none-elf` target triple.

  Default is `Off`.

- `-DBUILD_SIMULATION={On|Off}`

  Enable (`On`), or disable (`Off`), building channel simulation programs. This allows you to simulate Additive White Gaussian Noise (AWGN) channels in order to quantify the quality of the forward error correction for a given encoding scheme and modulation scheme. For more information, please see the section called `Run the simulations`.

  Default is `On`.

Install Arm RAN Acceleration Library (ArmRAL)

After you have built Arm RAN Acceleration Library, you can install the library.

1. Ensure you have write access for the installation directories:
   - For a default installation, you must have write access for `/usr/local/lib/`, for the library, and `/usr/local/include/`, for the header files.
   - For a custom installation, you must have write access for `<install-dir>/lib/`, for the library, and `<install-dir>/include/`, for the header files.

2. Install the library. Run:

   ```
   make install
   ```

   An install creates an `install_manifest.txt` file in the library build directory. `install_manifest.txt` lists the installation locations for the library and the header files.

Run the tests

The Arm RAN Acceleration Library package includes tests for the available functions in the library.

Note: To run the library tests, you must have built Arm RAN Acceleration Library with the `-DBUILD_TESTING=On` CMake option.

To build and run the tests, use:

```
make check
```

The tests run and test the available functions in the library. Testing times vary from system to system, but typically only take a few seconds.

If you are not developing on an AArch64 machine, or if you want to test the SVE or SVE2 version of the library on an AArch64 machine that does not support the extension, you can use the `-DARMRAL_TEST_RUNNER` option to prefix each test executable invocation with a wrapper. Example wrappers include QEMU and Arm Instruction Emulator. For example, for QEMU you could configure the library to prefix the tests with `qemu-aarch64` using:

```
cmake .. -DBUILD_TESTING=On -DARMRAL_TEST_RUNNER=qemu-aarch64
make check
```

Run the benchmarks

All the functions in Arm RAN Acceleration Library contain benchmarking code that contains preset problem sizes.

Note: To run the benchmark tests, you must have built Arm RAN Acceleration Library with the `-DBUILD_TESTING=On` CMake option. You must also have the executable `perf` available on your system. This can be installed via your package manager.

To build and run the benchmarks, use:

```
make bench
```

Benchmark results print as JSON objects. To further process the results, you can collect the results to a file or pipe the results into other scripts.

Run the examples

The source for the example programs is available in the `examples` directory, found in the ArmRAL root directory.

Note: To compile and execute the example programs, you must have built Arm RAN Acceleration Library with the `-DBUILD_EXAMPLES=On` CMake option.

- To both build and run the example programs, use:

```
make run_examples
```

- To only build the example programs so that, for example, you can later choose which example programs to specifically run, use:

```
make examples
```

The built binaries can be found in the `examples` subdirectory of the build directory.

More information about the examples that are available in Arm RAN Acceleration Library, and how to use the library in general, is available in **Use Arm RAN Acceleration Library (ArmRAL)** (see `examples.md`).

Run the simulations

You can evaluate the quality of the error correction of the different encoding schemes against the signal-to-noise ratio using a set of noisy channel simulation programs. ArmRAL currently only supports zero-mean Additive White Gaussian Noise (AWGN) channel simulation.

Note: The simulation programs do not simulate a full codec, and are intended to be used to evaluate just the forward error correction properties of the encoding and decoding of a single code block. We do not consider channel properties. The source code for the simulations and documentation for their use are available in the `simulation` directory, found in the ArmRAL root directory.

Note: To compile and execute the simulation programs, you must have built Arm RAN Acceleration Library with the `-DBUILD_SIMULATION=On` CMake option. This option is set to `On` by default.

The following assumes that you are running commands from the build directory.

- To build all the simulation programs, use:

```
make simulation
```

The built binaries can be found in the `simulation` subdirectory of the build directory.

More information about the simulation programs that are available in Arm RAN Acceleration Library is available in `simulation/README.md`.

Code coverage

You can generate information that describes how much of the library is used by your application, or is covered by the included tests. To collect code coverage information, you must have built Arm RAN Acceleration Library with `-DARMRAL_ENABLE_COVERAGE=On`.

An example workflow could be:

```
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug -DBUILD_TESTING=On -DARMRAL_ENABLE_COVERAGE=On
make check
gcovr --html-details index.html -r ..
```

Here, the `-r ..` flag points `gcovr` to the ArmRAL source tree, rather than attempting to find the source in the build directory. The `gcovr` command generates a series of HTML pages, viewable with a web browser, that give information on the lines of code executed by the test suite.

To generate a plain-text summary about the lines of code executed by the test suite, use:

```
gcovr -r ..
```

If you run into an issue when running the `gcovr` command, you might need to update to a newer version of `gcovr`. To find out what versions of `gcovr` have been tested with ArmRAL, see the **Tools** section of the `RELEASE_NOTES.md` file.

Documentation

The Arm RAN Acceleration Library Reference Guide is available online at:

```
https://developer.arm.com/documentation/102249/2310
```

If you have Doxygen installed on your system, you can build a local HTML version of the Arm RAN Acceleration Library documentation using CMake.

To build the documentation, run:

```
make docs
```

The HTML builds and is output to `docs/html/`. To view the documentation, open the `index.html` file in a browser.

Uninstall Arm RAN Acceleration Library

To uninstall Arm RAN Acceleration Library:

1. Navigate to the library build directory (where you previously ran `make install`)
2. Run:

   ```
   make uninstall
   ```

   `make uninstall` removes all the files listed in `install_manifest.txt` and any empty directories. `make uninstall` also attempts to remove any directories which might have been created.

Note: To only remove the installed files (but not any directories), instead run:

```
cat install_manifest.txt | xargs rm
```

2.2 Get started with ArmRAL noisy channel simulation

Introduction

This directory contains utilities and programs that you can use to evaluate the error-correction performance of the coding schemes provided in Arm RAN Acceleration Library (ArmRAL). ArmRAL supports three different coding schemes: Polar, Turbo, and Low-Density Parity Check (LDPC) codes. In the presence of noise on a channel, it is expected that some messages may not be decoded perfectly. In the utilities provided we consider that noise on a channel is zero-mean Additive White Gaussian Noise (AWGN).

The remainder of this document is structured as follows. To start with you will find a mathematical description of the AWGN which is simulated. The definition of what is meant by bit and block error rates is then given, and we conclude with instructions for how to use the utilities contained in this folder.

Additive White Gaussian Noise (AWGN) Simulation

Noisy channels are simulated by adding noise to the symbols generated by the modulation routine. This simulates that a signal is sent over a noisy network. These noisy symbols are demodulated by the demodulation routine. In zero-mean AWGN simulations a zero-mean white Gaussian noise with prescribed standard deviation `r` is added to the symbols.

The simulation programs supplied as part of the ArmRAL package provide control over the Signal-to-Noise Ratio (SNR) expressed in decibels (dB), which is

```
SNR = 10 * log10(S / R)
```

where `R` is the noise power and `S` is the signal power. `S=1` is assumed.

The simulator samples noise with power (or mean squared amplitude) `R` from a normal distribution with zero-mean and standard deviation `r` equal to

```
r = sqrt(R / 2)
```

The simulator generates a Gaussian noise with standard deviation `r` and zero-mean using a linear congruential pseudo-random number generator. It is then converted to 16-bit fixed-point (Q2.13) format, with saturation. The noise is then applied to the amplitude and phase of the symbols generated by the modulation scheme (QAM-type). We then attempt to decode the noisy symbols.

The simulator runs a total of 10^7 trials in parallel over a maximum of 100 threads. During each trial the SNR starts at 0dB, which means `S=R=1`, and increases in steps of 0.5dB until convergence is reached. Convergence means that for all trials the bit error rate is lower than a hard coded threshold. This tolerance is 0 for `polar` and 1e-5 for `ldpc` and `turbo` codes.

The x-axis of the graphs which are plotted shows values of `Eb / N0`, which is the noise spectral density per energy per bit. This can be directly calculated from the SNR as

```
SNR = rho * Eb / N0
```

for spectral efficiency `rho`. To calculate the spectral efficiency, the modulation scheme and bandwidth of the channel must be known, and passed to the simulation program.

The simulation programs follow the description of coding and modulation schemes provided in 3GPP Technical Specification (TS) 36.12, Section 5.1.3 (for Turbo coding) and 3GPP TS 38.212, Section 5.3 (for Low-Density Parity Check (LDPC) and Polar coding). We make the following further assumptions:

1. There is no distinction of Uplink/Downlink when it comes to selecting the values for the parameters.
2. A transport block contains a single code block. Encoding and decoding is performed for a single code block only.
3. No Cyclic Redundancy Check (CRC) is performed.

The simulator computes the error rates in terms of bits or blocks by comparing the input bits of encoding and the output decoded bits. The input bits are generated randomly using a linear congruential generator.

The bit error rate is computed as the ratio of the number of incorrect bits `nb` and the product of the number of information bits per block `k` and the number of blocks.

```
ber = nb / (k * number_of_blocks)
```

The block error rate is computed as the ratio of the number of incorrectly decode blocks `nbl` and the number of blocks. An incorrectly decoded block is a block with at least one incorrectly decoded bit.

```
bler = nbl / number_of_blocks
```

Get started with simulation programs

Note: To compile and execute the simulation programs, you must have built ArmRAL with the –`DBUILD_SIMULATION=On` CMake option.

The following assumes that you are running commands from the build directory.

- To build all the simulation programs, use:

```
make simulation
```

The built binaries can be found in the `simulation` subdirectory of the build directory.

In the following, the coding scheme `<code>` must be one of `polar`, `turbo`, `ldpc`, or `modulation` for simulations without using a coding scheme.

- To build the AWGN channel simulation for a given coding scheme `<code>`, use:

```
make <code>_awgn
```

- To run the AWGN channel simulation for `<code>` with arguments `<args>`, use:

```
./simulation/<code>_awgn/<code>_awgn <args>
```

- To get a list of possible input arguments and associated documentation, use the same command without arguments:

```
./simulation/<code>_awgn/<code>_awgn
```

- Executing a simulation will write JSON output to stdout. The output contains information on the observed bit and block error rates for the input parameters, and varying `Eb / N0` ratios. This data can be plotted by making use of the Python scripts described in the section on drawing performance charts.

Modulation schemes

All simulators use modulation and demodulation, respectively, before and after adding noise to the channel.

The modulation scheme is not specific to the coding scheme. You can select the modulation scheme using the `-m` option associated with the `<mod_type>` parameter.

Valid `<mod_type>` parameters are:

```
0: QPSK
1: 16QAM
2: 64QAM
3: 256QAM
```

In order to get best error correction performance out of a simulation, the programs allow users to pass a scaling parameter to the simulator called `<demod_ulp>`. The simulator uses this parameter during demodulation to control the range of the generated log-likelihood ratios (LLRs). A default value for `<demod_ulp>` of 128 is used in the case that it is not specified. You will find that the best performance of decoding relies on a good choice of `<demod_ulp>`, and you are encouraged to provide a value for this parameter.

Simulation program for modulation

The program `modulation_awgn` simulates the transmission of data without performing any forward error correction. Data is modulated, then has additive white Gaussian noise (AWGN) added to it, before demodulation makes a hard decision. Errors in bits and blocks are counted from the hard decision made in demodulation. This output can be used to validate that the forward error correction schemes are working as expected.

You can run the `modulation` AWGN simulation with the following parameters:

```
modulation_awgn -k num_info_bits -m mod_type [-u demod_ulp]
```

For each value of the `Eb/N0` ratio used, a JSON record is written to stdout. The JSON record contains the following fields:

```
{
  "k": <num_info_bits>,
  "mod_type": <mod_type>,
  "ulp": <demod_ulp>,
  "Eb/N0": <eb_n0>,
  "snr": <snr>,
  "bler": <bler>,
  "ber": <ber>
}
```

Simulation programs for individual coding schemes

In this section, we give the definition of some parameters used in the programs associated with each coding scheme.

You can find more information in the help text of each program. To show the help text use

```
<sim_name> --help
```

where `<sim_name>` is one of `polar_awgn`, `turbo_awgn`, or `ldpc_awgn`. The help text of the programs gives more detailed descriptions on the parameters than you will find in the sections below. The information below helps you to run the simulation programs and understand their output.

You can run the `polar` coding Additive White Gaussian Noise (AWGN) simulation with the following parameters:

```
polar_awgn -k num_info_bits -e num_trans_bits
           -m mod_type -i i_bil [-u demod_ulp] [-l list_size]
```

For each value of the `Eb / N0` ratio used, a JSON record is written to stdout. The JSON record contains the following fields:

```
{
  "len": <codeword_length>,
  "e": <num_trans_bits>,
  "k": <num_info_bits>,
  "l": <list_size>,
```

```
  "mod_type": <mod_type>,
  "i_bil": <i_bil_type>
  "ulp": <demod_ulp>,
  "Eb/N0": <eb_n0>,
  "snr": <snr>,
  "bler": <bler>,
  "ber": <ber>
}
```

You can run the `turbo` coding Additive White Gaussian Noise (AWGN) simulation with the following parameters:

```
turbo_awgn -k num_bits -m mod_type -e num_matched_bits
           [-r rv] [-u demod_ulp] [-i iter_max]
```

For each value of the `Eb / N0` ratio used, a JSON record is written to stdout. The JSON record contains the following fields:

```
{
  "k": <num_bits>,
  "e": <num_matched_bits>,
  "mod_type": <mod_type>,
  "ulp": <demod_ulp>,
  "Eb/N0": <eb_n0>,
  "snr": <snr>,
  "bler": <bler>,
  "ber": <ber>
}
```

You can run the `LDPC` coding Additive White Gaussian Noise (AWGN) simulation with the following parameters:

```
ldpc_awgn -z lifting_size -b base_graph -m mod_type
          [-r redundancy_version] [-u demod_ulp]
```

For each value of the `Eb / N0` ratio used, a JSON record is written to stdout. The JSON record contains the following fields:

```
{
  "n": <input_length>,
  "bg": <base_graph>,
  "mod_type": <mod_type>,
  "rv ": <redundancy_version>,
  "Eb/N0": <eb_n0>,
  "snr": <snr>,
  "ulp": <demod_ulp>,
  "bler": <bler>,
  "ber": <ber>
}
```

You can run the `convolutional` coding Additive White Gaussian Noise (AWGN) simulation with the following parameters:

```
convolutional_awgn -k num_bits -m mod_type [-u demod_ulp] [-i iter_max]
```

For each value of the `Eb/N0` ratio used, a JSON record is written to stdout. The JSON record contains the following fields:

```
{
  "k": <num_bits>,
  "mod_type": <mod_type>,
  "iter_max": <iter_max>,
  "ulp": <demod_ulp>,
  "Eb/N0": <eb_n0>,
  "snr": <snr>,
  "bler": <bler>,
  "ber": <ber>
}
```

Drawing performance charts

The simulator allows users to evaluate the performance of a coding scheme. In the context of noisy channels, performance is evaluated in terms of output error rates for a given input `Eb / N0` ratio or signal-to-noise ratio `SNR`.

The simulation programs return both bit and block error rates in JSON-format along with other quantities of interest, like the modulation scheme or other code-specific parameters.

The performance is usually represented as a graph of error rates against the `Eb / N0` ratio.

Note: To plot the results of the simulation program, you may use a provided Python script (see the description below for example usage). Running these scripts requires a recent version of Python. ArmRAL has been tested with Python 3.8.5.

- To parse the output of the simulation programs and plot error rates against the `Eb / N0` ratio with arguments `<args>`, use:

```
./simulation/<code>_awgn/<code>_error_rate.py <args>
```

- To plot error rates against the `SNR` with arguments `<args>`, use:

```
./simulation/<code>_awgn/<code>_error_rate.py --x-unit snr <args>
```

- To get a list of possible input arguments and associated documentation for the Python script, use:

```
./simulation/<code>_awgn/<code>_error_rate.py --help
```

Drawing capacity charts

The simulator allows users to draw the data rates of each modulation and compare them to the capacity of the AWGN channel (the Shannon limit).

- To plot the rates against the `Eb / N0` ratio, use:

```
./simulation/capacity/capacity.py <args>
```

- To get a list of possible input arguments and associated documentation for the Python script, use:

  ```
  ./simulation/capacity/capacity.py --help
  ```

2.3 Use Arm RAN Acceleration Library (ArmRAL)

This topic describes how to compile and link your application code to Arm RAN Acceleration Library (ArmRAL).

Before you begin

- Ensure you have a recent version of a C/C++ compiler, such as GCC. See the Release Notes for a full list of supported GCC versions.

  If required, configure your environment. If you have multiple compilers installed on your machine, you can set the `CC` and `CXX` environment variables to the path to the C compiler and C ++ compiler that you want to use.

- You must build Arm RAN Acceleration Library before you can use it in your application development, or to run the example programs.

  To build the library, use:

  ```
  tar zxvf ral-armral-23.10.tar.gz
  mkdir ral-armral-23.10/build
  cd ral-armral-23.10/build
  cmake ..
  make -j
  ```

- To use the Arm RAN Acceleration Library functions in your application development, include the `armral.h` header file in your C or C++ source code.

  ```
  #include "armral.h"
  ```

Procedure

1. Build and link your program with Arm RAN Acceleration Library. For GCC, use:

   ```
   gcc -c -o <code-filename>.o <code-filename>.c -I <path/to/armral/source>/include
    -O2
   gcc -o <binary-filename> <code-filename>.o <path/to/armral/build>/libarmral.a -lm
   ```

   Substituting:

   - `<code-filename>` with the name of your own source code file
   - `<path/to/armral/source>` with the path to your copy of the Arm RAN Acceleration Library source code

- `<path/to/armral/build>` with the path to your build of Arm RAN Acceleration Library, as appropriate

2. Run your binary:

```
./<binary-filename>
```

Example: Run 'fft_cf32_example.c'

In this example, we use Arm RAN Acceleration Library to compute and solve a simple Fast Fourier Transform (FFT) problem.

The following source file can be found in the ArmRAL source directory under `examples/fft_cf32_example.c`:

```
/*
    Arm RAN Acceleration Library
    Copyright 2020-2023 Arm Limited and/or its affiliates <open-source-
office@arm.com>
*/
#include "armral.h"

#include <stdio.h>
#include <stdlib.h>

// This function shows how to create a plan and execute an FFT using the ArmRAL
// library
static void example_fft_plan_and_execute(int n) {
  armral_fft_plan_t *p;
  printf("Planning FFT of length %d\n", n);
  // In the planning, the direction of the FFT is indicated by the last
  // parameter, which is either -1 (for forwards) or 1 (for backwards)
  armral_fft_create_plan_cf32(&p, n, -1);

  // Create the data that is to be used in FFTs. The input array (x) needs to
  // be initialised. The output array (y) does not.
  armral_cmplx_f32_t *x =
      (armral_cmplx_f32_t *)malloc(n * sizeof(armral_cmplx_f32_t));
  armral_cmplx_f32_t *y =
      (armral_cmplx_f32_t *)malloc(n * sizeof(armral_cmplx_f32_t));
  for (int i = 0; i < n; ++i) {
    x[i] = (armral_cmplx_f32_t){(float)i, (float)-i};
    y[i] = (armral_cmplx_f32_t){0.F, 0.F};
  }

  printf("Input Data:\n");
  for (int i = 0; i < n; ++i) {
    printf("  (%f + %fi)\n", x[i].re, x[i].im);
  }
  printf("\n");

  // The FFTs are executed with different input and output data. The length
  // of the input and output arrays needs to be at least the same as that of
  // the length parameter with which the plan was created. No checks are
  // performed that this is the case in the library.
  printf("Performing FFT of length %d\n", n);
  armral_fft_execute_cf32(p, x, y);

  // A plan can be re-used to solve other FFTs, but once a plan is no longer
  // needed, it needs to be destroyed to avoid leaking memory.
  printf("Destroying plan for FFT of length %d\n", n);
  armral_fft_destroy_plan_cf32(&p);

  printf("Result:\n");
```

```
  for (int i = 0; i < n; ++i) {
    printf("  (%f + %fi)\n", y[i].re, y[i].im);
  }
  printf("\n");

  // Need to free the pointers to data. These are not owned by the FFT plan,
  // and it is the user's responsibility to manage the memory.
  free(x);
  free(y);
}

int main(int argc, char **argv) {
  if (argc < 2) {
    printf("Usage: %s len\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  int n = atoi(argv[1]);
  if (n < 1) {
    printf("Length parameter must be positive and non-zero\n");
    exit(EXIT_FAILURE);
  }

  example_fft_plan_and_execute(n);
}
```

1. To build and link the example program with GCC, use:

   ```
   gcc -c -o fft_cf32_example.o fft_cf32_example.c -I <path/to/armral/source>/
   include -O2
   gcc -o fft_cf32_example fft_cf32_example.o <path/to/armral/build>/libarmral.a -lm
   ```

   Substituting:

   - `<path/to/armral/source>` with the path to your copy of the Arm RAN Acceleration Library source code
   - `<path/to/armral/build>` with the path to your build of Arm RAN Acceleration Library, as appropriate

   Note: For this example, there is a requirement to link against libm (`-lm`). libm is used in several functions in Arm RAN Acceleration Library, and so might be required for your own programs.

   An executable called `fft_cf32_example` is built.

2. Run the `fft_cf32_example` executable. To input the length of FFT to compute, the example program takes the length as an argument. To run with the length of FFT set to 5, use:

   ```
   ./fft_cf32_example 5
   ```

   which gives:

   ```
   Planning FFT of length 5
   Input Data:
     (0.000000 + 0.000000i)
     (1.000000 + -1.000000i)
     (2.000000 + -2.000000i)
     (3.000000 + -3.000000i)
     (4.000000 + -4.000000i)
   ```

```
Performing FFT of length 5
Destroying plan for FFT of length 5
Result:
  (10.000000 + -10.000000i)
  (0.940955 + 5.940955i)
  (-1.687701 + 3.312299i)
  (-3.312299 + 1.687701i)
  (-5.940955 + -0.940955i)
```

Other examples: block-float, modulation, and polar examples

Arm RAN Acceleration Library also includes block-float, modulation, and polar examples. These example files can also be found in the `/examples/` directory.

In addition to the `fft_cf32_example.c` FFT example, the following examples are included:

- `block_float_9b_example.c`

  Fills a single Resource Block (RB) with a set of random numbers and uses the block floating-point compression API to compress the numbers into a 9-bit compressed format. `block_float_9b_example.c` then uses the decompression function to convert the numbers to their original format, then returns the numbers side-by-side for comparison.

  The example binary does not take an argument. For example, to run a compiled binary of the `block_float_9b_example.c`, called, `block_float_9b_example`, use:

  ```
  ./block_float_9b_example
  ```

- `modulation_example.c`

  Uses the modulation and demodulation API to simulate applying 256QAM modulation to an array of random input bits. To show that taking a hard-decision with no noise applied gives the original input, `modulation_example.c` then demodulates the data, before returning the values.

  The example binary does not take an argument. For example, to run a compiled binary of the `modulation_example.c`, called, `modulation_example`, use:

  ```
  ./modulation_example
  ```

- `polar_example.cpp`

  Uses the polar coding and modulation APIs to simulate a complete flow from an original input codeword to the final polar-decoded output. In particular, the Polar encoder and decoder are used, as well as the subchannel interleaving functionality. Example implementations of other parts of the coding process, such as sub-block interleaving and rate-matching, are also provided.

  The example binary takes three arguments, in the following order:

  1. The polar code size (`N`)
  2. The rate-matched codeword length (`E`)
  3. The number of information bits (`K`)

For example, to run a compiled binary of the `polar_example.cpp`, called, `polar_example`, with an input array of `N = 128`, `E = 100`, and `K = 35`, use:

```
./modulation_example 128 100 35
```

Each example can be run according to the **Procedure** described above, as demonstrated in the **Example: Run 'fft_cf32_example.c'** section.

3. Functions

This section describes the functions that are available in Arm RAN Acceleration Library.

3.1 Vector functions

Functions for working with vectors.

Functions are provided for working with arrays of 16-bit integers (Q15 format) and 32-bit floating-point numbers. In particular:

- Vector element-wise multiplication (vector multiply)
- Vector dot product

3.1.1 Vector Multiply

Multiplies a complex vector by another complex vector and generates a complex result.

The complex arrays have a total of `2*n` real values.

The vector multiplication algorithm is:

```
for (n = 0; n < numSamples; n++) {
    pDst[2n+0] = pSrcA[2n+0] * pSrcB[2n+0] - pSrcA[2n+1] * pSrcB[2n+1];
    pDst[2n+1] = pSrcA[2n+0] * pSrcB[2n+1] + pSrcA[2n+1] * pSrcB[2n+0];
}
```

3.1.2 Vector Dot Product

Computes the dot product of two complex vectors.

The vectors are multiplied element-by-element and then summed.

`pSrcA` points to the first complex input vector and `pSrcB` points to the second complex input vector. `n` specifies the number of complex samples. The data in each array is stored as armral_cmplx_f32_t elements, with separate arrays for real and imaginary components:

```
(real, imag, real, imag, ...)
```

Each array has a total of `n` complex values.

The dot product algorithm is:

```
real_result = 0;
```

```
imag_result = 0;
for (n = 0; n < numSamples; n++) {
   real_result += p_src_a[2n+0]*p_src_b[2n+0] - p_src_a[2n+1]*p_src_b[2n+1];
   imag_result += p_src_a[2n+0]*p_src_b[2n+1] + p_src_a[2n+1]*p_src_b[2n+0];
}
```

3.2 Matrix functions

Functions for working with matrices.

Functions are provided for working with matrices, including:

- Matrix-vector multiplication for 16-bit integer datatypes.
- Matrix-matrix multiplication. Supports both 16-bit integer and 32-bit floating-point datatypes. In addition, the `solve` routines support specifying a custom Q-format specifier for both input and output matrices, instead of assuming that the input is in Q15 format.
- Matrix inversion. Supports the 32-bit floating-point datatype.

3.2.1 Complex Matrix-Vector Multiplication

Computes a matrix-by-vector multiplication, storing the result in a destination vector.

The destination vector is only written to and can be uninitialized.

3.2.2 Complex Matrix-Matrix Multiplication

Computes a matrix-by-matrix multiplication, storing the result in a destination matrix.

The destination matrix is only written to and can be uninitialized.

To permit specifying different fixed-point formats for the input and output matrices, the `solve` routines take an extra fixed-point type specifier.

3.2.3 Complex Matrix Inversion

Computes the inverse of a complex Hermitian square matrix of size `N-by-N`.

3.2.4 Complex Matrix Pseudo-Inverse

Computes the regularized pseudo-inverse of a complex matrix of size `M-by-N`.

3.2.5 SVD decomposition of single complex matrix

The Singular Value Decomposition (SVD) is used for selecting orthogonal user equipment pairing in mMIMO channels.

3.3 Lower PHY support functions

Functions for working in the lower physical layer (lower PHY).

The Lower PHY functions include support for:

- A Gold Sequence generator
- A correlation coefficient of a pair of 16-bit integer arrays (in Q15 format)
- FIR filters. Supports both 16-bit integer and 32-bit floating-point datatypes. Support is provided for decimation factors of both one and two.
- Fast Fourier Transforms (FFTs). Supports both 16-bit integer and 32-bit floating-point datatypes.
- Scrambling of a bit sequence. Supports scrambling of data from individual code blocks, but not from transport blocks.

3.3.1 Sequence Generator

Fills a pointer with a Gold Sequence of the specified length, generated from the specified seed.

The sequence generator is the same generator that is described in the 3GPP Technical Specification (TS) 36.211, Chapter 7.2.

3.3.2 Correlation Coefficient

Calculates Pearson's Correlation Coefficient from a pair of complex vectors.

3.3.3 FIR filter

FIR filter implemented for single-precision floating-point and 16-bit signed integers.

For example, given an input array `x`, an output array `y`, and a set of coefficients `b`, the following is calculated:

```
y[n] = b[0] x[N-1] +
       b[1] x[N-2] +
       ... +
       b[N-1] x[0]
     =
```

The FIR coefficients are assumed to be reversed in memory, such that `b_N` above is the first coefficient in memory rather than the last.

3.3.4 Fast Fourier Transforms (FFT)

Computes the Discrete Fourier Transform (DFT) of a sequence (forwards transform), or the inverse (backwards transform).

FFT plans are represented by an opaque structure. To fill the plan structure, define a pointer to the structure and call armral_fft_create_plan_cf32 or armral_fft_create_plan_cs16. For example:

```
armral_fft_plan_t *plan;
armral_fft_create_plan_cf32(&plan, 32, ARMRAL_FFT_FORWARDS);
armral_fft_execute_cf32(plan, x, y);
armral_fft_destroy_plan_cf32(&plan);
```

3.3.5 Scrambling

Scrambles the input bits using the given pseudo-random sequence.

The scrambler can be applied for Physical Uplink Control Channels (PUCCH) formats 2, 3 and 4, as well as Physical Downlink Shared Channel (PDSCH), Physical Downlink Control Channel (PDCCH), and Physical Broadcast Channel (PBCH). The implementation here covers the scrambling described in 3GPP Technical Specification (TS) 38.211, sections 6.3.2.5.1, 6.3.2.6.1, 7.3.1.1, 7.3.2.3, and 7.3.3.1.

3.4 Upper PHY support functions

Functions for working in the upper physical layer (upper PHY).

The Upper PHY functions include support for:

- Digital modulation and demodulation, using QPSK, 16QAM, 64QAM, or 256QAM.
- Cyclic Redundancy Check (CRC), both little-endian and big-endian, for the six 5G polynomials (CRC24A, CRC24B, CRC24C, CRC16, CRC11, and CRC6).
- Polar encoding and decoding.
- Low-Density Parity Check (LDPC) encoding and decoding.
- LTE Turbo encoding and decoding.
- LTE tail biting convolutional encoding and decoding.
- Rate matching and rate recovery for Polar coding.
- Rate matching and rate recovery for LDPC coding.

3.4.1 Modulation

Performs modulation and demodulation of digital signals. Modulation takes a bitstream and outputs a series of Q2.13 fixed-point complex symbols. Demodulation takes Q2.13 fixed-point complex symbols and generates a series of log-likelihood ratios (LLRs), which can be used in Polar decoding.

The functions take as parameter the modulation type being used, namely either QPSK or QAM, see `armral_modulation_type`.

The number of complex samples needed for a given bitstream (and therefore the size of the memory buffer passed) depends on the modulation type being used: QPSK, 16QAM, 64QAM, and 256QAM correspond to two, four, six, and eight bits per symbol, respectively (log base-2 of the constellation size).

3.4.2 CRC

Computes a Cyclic Redundancy Check (CRC) of an input buffer using carry-less multiplication and Barret reduction.

```
CRC24A polynomial = x^24 + x^23 + x^18 + x^17 + x^14 + x^11 + x^10 + x^7 +
                    x^6 + x^5 + x^4 + x^3 + x + 1
CRC24B polynomial = x^24 + x^23 + x^6 + x^5 + x + 1
CRC24C polynomial = x^24 + x^23 + x^21 + x^20 + x^17 + x^15 + x^13 + x^12 +
                    x^8 + x^4 + x^2 + x + 1
CRC16 polynomial  = x^16 + x^12 + x^5 + 1
CRC11 polynomial  = x^11 + x^10 + x^9 + x^5 + 1
CRC6 polynomial   = x^6 + x^5 + 1
```

The input buffer is assumed to be padded to at least 8 bytes. If the input size is greater than 8 bytes, then padding to a multiple of 16 bytes (128 bits) is assumed.

Both little-endian and big-endian orderings are provided, using the `le` and `be` suffixes, respectively.

3.4.3 Polar encoding

In uplink, Polar codes are used to encode the Uplink Control Information (UCI) over the Physical Uplink Control Channel (PUCCH) and Physical Uplink Shared Channel (PUSCH). In downlink, Polar codes are used to encode the Downlink Control Information (DCI) over the Physical Downlink Control Channel (PDCCH).

By construction, Polar codes only allow code lengths that are powers of two (`N=2^n`). The number of input information bits, `K`, can take any arbitrary value up to the maximum value of `N` (`K<=N`). In particular, 5G NR restricts the usage of Polar codes length from `N`=32 bits to `N`=1024 bits. For `N`<32, other types of channel coding are performed.

Given the input sequence vector `[u] = [u(0), u(1), ..., u(N-1)]`, if index `i` is included in the `frozen` bits set, then `u(i) = 0`. The input information bits are stored in the remaining entries. `[d] = [d(0), d(1), ..., d(N-1)]` is the vector of output encoded bits. `[G_N]` is the channel transformation matrix (`N-by-N`), obtained by recursively applying the Kronecker product from the basic kernel `G_2 = |1 0; 1 1|` to the order `n = log2(N)`.

The output after encoding, `[d]`, is obtained by `[d] = [u]*[G_N]`.

For more information, refer to the 3GPP Technical Specification (TS) 38.212 V16.0.0 (2019-12).

3.4.4 Low-Density Parity Check (LDPC)

Performs encoding and decoding of data using Low-density Parity Check (LDPC) methods. The implementation is described in the 3GPP Technical Specification (TS) 38.212, in sections 5.2.2 and 5.3.2.

Encoding of a single block is supported. Depending on the rate matching applied to a signal, one of two base graphs are used when creating an LDPC encoding. Concepts of rate matching are not included, but the implementation provided does take the graph as input to be able to perform different encoding operations.

A base graph is described by a sparse matrix, in which each non-zero entry indicates the presence of a shifted identity matrix. The size of the matrix is denoted by `z` and depends on the size of the message to encode. `z` is referred to as the lifting size, and a lifting size belongs to a particular lifting set (indices from 0 to 7). The amount each identity matrix is shifted by depends on the lifting set index.

3.4.5 LTE Turbo

Performs encoding and decoding of data using LTE Turbo methods. The encoding scheme is defined in section 5.1.3.2 of the 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". The decoder implements a maximum a posteriori (MAP) algorithm and returns a hard decision (either 0 or 1) for each output bit. The encoding and decoding are performed for a single code block.

3.4.6 LTE convolutional coding

Performs encoding and decoding of data using LTE tail biting convolutional coding. The encoding scheme is defined in section 5.1.3.1 of the 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". The decoder implements the Wrap Around Viterbi Algorithm (WAVA) described in R. Y. Shao, Shu Lin and M. P. C. Fossorier, "Two decoding algorithms for tailbiting codes", in IEEE Transactions on Communications, vol. 51, no. 10, pp. 1658-1665, Oct. 2003. The encoding and decoding are performed for a single code block.

3.5 DU-RU IF support functions

Functions for working with Distributed Units (DUs) and Radio Units (RUs).

The DU-RU IF functions include support for:

- Mu-Law compression and decompression, in 8-bit, 9-bit, and 14-bit formats.
- Block floating-point compression and decompression, in 8-bit, 9-bit, and 14-bit formats.
- Block scaling compression and decompression, in 8-bit, 9-bit, and 14-bit formats.

3.5.1 Mu-Law Compression

The Mu-Law algorithm enables the compression of User Plane (UP) data over the fronthaul interface.

3.5.2 Block Scaling Compression

Implements algorithms for data compression and decompression using block scaling representation of complex samples.

3.5.3 Block Floating Point

Implements algorithms for data compression and decompression through block floating-point representation of complex samples.

4. Data Structures

This section describes the data structures that are available in Arm RAN Acceleration Library.

4.1 armral_cmplx_f32_t

32-bit floating-point complex data type.

Syntax

Defined in `armral.h` on line `195`:

```
typedef struct {
  float re; ///< 32-bit real component.
  float im; ///< 32-bit imaginary component.
} armral_cmplx_f32_t;
```

4.2 armral_cmplx_int16_t

16-bit signed integer complex data type.

Syntax

Defined in `armral.h` on line `187`:

```
typedef struct {
  int16_t re; ///< 16-bit real component.
  int16_t im; ///< 16-bit imaginary component.
} armral_cmplx_int16_t;
```

4.3 armral_compressed_data_12bit

The structure for a 12-bit compressed block.

See armral_block_float_compr_12bit and armral_block_float_decompr_12bit.

Syntax

Defined in `armral.h` on line `233`:

```
typedef struct {
  int8_t exp;           ///< Block exponent, in the range 0-4 (inclusive).
  int8_t mantissa[36]; ///< Packed data, 12 bits per element.
} armral_compressed_data_12bit;
```

4.4 armral_compressed_data_14bit

The structure for a 14-bit compressed block.

See armral_block_float_compr_14bit and armral_block_float_decompr_14bit.

Syntax

Defined in `armral.h` on line `244`:

```
typedef struct {
  int8_t exp;           ///< Block exponent, in the range 0-2 (inclusive).
  int8_t mantissa[42]; ///< Packed data, 14 bits per element.
} armral_compressed_data_14bit;
```

4.5 armral_compressed_data_8bit

The structure for an 8-bit compressed block.

See armral_block_float_compr_8bit and armral_block_float_decompr_8bit.

Syntax

Defined in `armral.h` on line `211`:

```
typedef struct {
  int8_t exp;           ///< Block exponent, in the range 0-8 (inclusive).
  int8_t mantissa[24]; ///< Packed data, 8 bits per element.
} armral_compressed_data_8bit;
```

4.6 armral_compressed_data_9bit

The structure for a 9-bit compressed block.

See armral_block_float_compr_9bit and armral_block_float_decompr_9bit.

Syntax

Defined in `armral.h` on line `222`:

```
typedef struct {
  int8_t exp;           ///< Block exponent, in the range 0-7 (inclusive).
  int8_t mantissa[27]; ///< Packed data, 9 bits per element.
} armral_compressed_data_9bit;
```

4.7 armral_ldpc_base_graph_t

Data structure required to store the data in a Low Density Parity Check (LDPC) base graph. The data of a base graph is stored in Compressed Sparse Row (CSR) format.

Syntax

Defined in `armral.h` on line `3222`:

```
typedef struct {
  ///
  /// The number of rows in the base graph.
  uint32_t nrows;

  /// The number of columns in the base graph which are associated with message
  /// bits. Punctured columns are included.
  uint32_t nmessage_bits;

  /// The number of block columns that are in the codeword. `ncodeword_bits` is
  /// the number of columns in the base graph minus the two punctured columns.
  uint32_t ncodeword_bits;

  /// The indices of the start of a row in the base graph, which you can use to
  /// index into the `col_inds` array to get the column indices of the non-zero
  /// entries in a row of the base graph.
  const uint32_t *row_start_inds;

  /// The indices of the non-zero columns in the base graph. Each of the entries
  /// in a row are stored contiguously. The start of a row is identified by
  /// indices stored in the `row_start_inds` array. For example, the start of
  /// row with index (zero-based) `2` is at index `row_start_inds[2]`.
  const uint32_t *col_inds;

  /// The shifts applied to the identity matrix to give the matrix at each
  /// non-zero column in the base graph. The shifts for all lifting sets are
  /// stored in this array. All shifts for one lifting set are stored before the
  /// next lifting set. This means that the shifts for lifting set with index
  /// (zero-based) `3`, and row with index `5` is at index
  /// `(row_start_inds[5] + 3) * 8`, where `8` is the number of lifting
  /// sets.
  const uint32_t *shifts;
} armral_ldpc_base_graph_t;
```

5. Macros

This section describes the macro definitions that are available in Arm RAN Acceleration Library.

5.1 ARMRAL_NUM_COMPLEX_SAMPLES

The number of complex samples in each compressed block.

Syntax

Defined in `armral.h` on line `203`:

```
#define ARMRAL_NUM_COMPLEX_SAMPLES 12
```

5.2 ARMRAL_LDPC_NO_CRC

A constant which can be passed to `armral_ldpc_decode_block` when the input code block has no CRC attached.

Syntax

Defined in `armral.h` on line `3260`:

```
#define ARMRAL_LDPC_NO_CRC 0
```

6. Enumerations

This section describes the enumeration definitions (`enum` in C/C++) that are available in Arm RAN Acceleration Library.

6.1 armral_status

Error status returned by functions in the library.

Syntax

Defined in `armral.h` on line `105`:

```
typedef enum {
  ARMRAL_SUCCESS = 0,          ///< No error.
  ARMRAL_ARGUMENT_ERROR = -1,  ///< One or more arguments are incorrect.
} armral_status;
```

6.2 armral_modulation_type

Formats that are supported by modulation and demodulation. See armral_modulation and armral_demodulation.

Syntax

Defined in `armral.h` on line `114`:

```
typedef enum {
  ARMRAL_MOD_QPSK = 0,   ///< QPSK, size 4 constellation, 2 bits per symbol.
  ARMRAL_MOD_16QAM = 1,  ///< 16QAM, size 16 constellation, 4 bits per symbol.
  ARMRAL_MOD_64QAM = 2,  ///< 64QAM, size 64 constellation, 6 bits per symbol.
  ARMRAL_MOD_256QAM = 3  ///< 256QAM, size 256 constellation, 8 bits per symbol.
} armral_modulation_type;
```

6.3 armral_fixed_point_index

Fixed-point format index `Q[integer_bits, fractional_bits]` for `int16_t`. For usage information, see the `armral_solve_*` functions.

Syntax

Defined in `armral.h` on line `125`:

```
typedef enum {
  /// 1 sign bit, 0 integer bits, 15 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q15 = 15,
  /// 1 sign bit, 1 integer bit, 14 fractional bits.
```

```
  ARMRAL_FIXED_POINT_INDEX_Q1_14 = 14,
  /// 1 sign bit, 2 integer bits, 13 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q2_13 = 13,
  /// 1 sign bit, 3 integer bits, 12 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q3_12 = 12,
  /// 1 sign bit, 4 integer bits, 11 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q4_11 = 11,
  /// 1 sign bit, 5 integer bits, 10 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q5_10 = 10,
  /// 1 sign bit, 6 integer bits, 9 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q6_9 = 9,
  /// 1 sign bit, 7 integer bits, 8 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q7_8 = 8,
  /// 1 sign bit, 8 integer bits, 7 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q8_7 = 7,
  /// 1 sign bit, 9 integer bits, 6 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q9_6 = 6,
  /// 1 sign bit, 10 integer bits, 5 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q10_5 = 5,
  /// 1 sign bit, 11 integer bits, 4 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q11_4 = 4,
  /// 1 sign bit, 12 integer bits, 3 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q12_3 = 3,
  /// 1 sign bit, 13 integer bits, 2 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q13_2 = 2,
  /// 1 sign bit, 14 integer bits, 1 fractional bit.
  ARMRAL_FIXED_POINT_INDEX_Q14_1 = 1,
  /// 1 sign bit, 15 integer bits, 0 fractional bits.
  ARMRAL_FIXED_POINT_INDEX_Q15_0 = 0
} armral_fixed_point_index;
```

6.4 armral_polar_frozen_bit_type

Defines the values that can be stored in the output `frozen` mask that is created by armral_polar_frozen_mask. For a given input bit array, each index `i` in the `frozen` mask describes the corresponding bit index `i` in the array. Each entry describes the origin of the bit at the point of output from armral_polar_encode_block, in particular whether the origin of the bit was an information bit (present in the original codeword), a parity bit (calculated from the codeword bits), or a `frozen` bit (set to zero).

Syntax

Defined in `armral.h` on line `170`:

```
typedef enum {
  ARMRAL_POLAR_INFO_BIT = 0,    ///< Information bit.
  ARMRAL_POLAR_PARITY_BIT = 1,  ///< Parity bit.
  ARMRAL_POLAR_FROZEN_BIT = 255 ///< Frozen bit (set to zero).
} armral_polar_frozen_bit_type;
```

6.5 armral_polar_ibil_type

Enable or disable the interleaving of coded bits in Polar rate matching.

Syntax

Defined in `armral.h` on line `179`:

```
typedef enum {
  ARMRAL_POLAR_IBIL_DISABLE = 0, ///< Downlink direction
  ARMRAL_POLAR_IBIL_ENABLE = 1,  ///< Uplink direction
} armral_polar_ibil_type;
```

6.6 armral_fft_direction_t

The direction of the FFT being computed. The direction is passed to armral_fft_create_plan_cf32 and armral_fft_create_plan_cs16.

Syntax

Defined in `armral.h` on line `3061`:

```
typedef enum {
  ARMRAL_FFT_FORWARDS = -1, ///< Compute a forwards (non-inverse) FFT.
  ARMRAL_FFT_BACKWARDS = 1, ///< Compute a backwards (inverse) FFT.
} armral_fft_direction_t;
```

6.7 armral_ldpc_graph_t

Identifies the base graph to use in LDPC encoding and decoding. The base graphs are defined in tables 5.3.2-2 and 5.3.2-3 in the 3GPP Technical Specification (TS) 38.212.

Syntax

Defined in `armral.h` on line `3212`:

```
typedef enum {
  LDPC_BASE_GRAPH_1, ///< Identifier for LDPC base graph 1.
  LDPC_BASE_GRAPH_2  ///< Identifier for LDPC base graph 2.
} armral_ldpc_graph_t;
```

7. Type Aliases

This section describes the type aliases (`typedef` in C/C++) that are available in Arm RAN Acceleration Library.

7.1 armral_fft_plan_t

The opaque structure to an FFT plan. You must fill an FFT plan before you use it. To fill an FFT plan, call armral_fft_create_plan_cf32 or armral_fft_create_plan_cs16.

Syntax

Defined in `armral.h` on line `3054`:

```
typedef struct armral_fft_plan_t armral_fft_plan_t;
```