Arm® RAN Acceleration Library

Version 25.04

Reference Guide

Non-Confidential

Issue 00
102249_25.04_00_en

Arm® RAN Acceleration Library

Reference Guide

Release information

Document history

Issue	Date	Confidentiality	Change
2504-00	17 April 2025	Non-Confidential	Update for Arm RAN Acceleration Library v25.04
2501-00	23 January 2025	Non-Confidential	Update for Arm RAN Acceleration Library v25.01
2410-00	17 October 2024	Non-Confidential	Update for Arm RAN Acceleration Library v24.10
2407-00	18 July 2024	Non-Confidential	Update for Arm RAN Acceleration Library v24.07
2404-00	19 April 2024	Non-Confidential	Update for Arm RAN Acceleration Library v24.04
2401-00	19 January 2024	Non-Confidential	Update for Arm RAN Acceleration Library v24.01
2310-00	6 October 2023	Non-Confidential	Update for Arm RAN Acceleration Library v23.10
2307-00	7 July 2023	Non-Confidential	Update for Arm RAN Acceleration Library v23.07
2304-00	21 April 2023	Non-Confidential	Update for Arm RAN Acceleration Library v23.04
2301-00	27 January 2023	Non-Confidential	Update for Arm RAN Acceleration Library v23.01
2210-00	7 October 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.10
2207-00	15 July 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.07
2204-00	8 April 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.04

Issue	Date	Confidentiality	Change
2201-00	14 January 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.01
2110-00	8 October 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.10
2107-00	9 July 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.07
2104-00	9 April 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.04
2101-00	8 January 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.01
2010-00	2 October 2020	Non-Confidential	New document for Arm RAN Acceleration Library v20.10

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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

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...9
1.1 Conventions...9
1.2 Other information...10

2. Tutorials...11
2.1 Get started with Arm RAN Acceleration Library...11
2.2 Get started with Arm RAN Acceleration Library noisy channel simulation...18
2.3 Use Arm RAN Acceleration Library...24

3. Functions...29
3.1 Vector functions...29
3.1.1 Vector Multiply...29
3.1.2 Vector Dot Product...35
3.2 Matrix functions...41
3.2.1 Complex Matrix-Vector Multiplication...41
3.2.2 General Complex Matrix-Matrix Multiplication...54
3.2.3 Specific-Sized Complex Matrix-Matrix Multiplication...60
3.2.4 Channel Matrix-Matrix Multiplication...65
3.2.5 Complex Matrix Inversion...74
3.2.6 Complex Matrix Pseudo-Inverse...80
3.2.7 SVD of a Single Complex Matrix...83
3.3 Lower PHY support functions...87
3.3.1 Sequence Generator...87
3.3.2 Correlation Coefficient...88
3.3.3 FIR Filter...89
3.3.4 Fast Fourier Transforms (FFT)...94
3.3.5 Scrambling...100
3.4 Upper PHY support functions...101
3.4.1 Modulation...102
3.4.2 Cyclic Redundancy Check (CRC)...104
3.4.3 Polar Encoding...113
3.4.4 Low-Density Parity Check (LDPC)...127
3.4.5 LTE Turbo...144

3.4.6 LTE Convolutional Coding....162
3.5 DU-RU IF support functions....166
3.5.1 Mu-Law Compression....166
3.5.2 Block Scaling Compression....172
3.5.3 Block Floating Point....177

4. Data Structures....185
4.1 armral_cmplx_f32_t....185
4.2 armral_cmplx_int16_t....185
4.3 armral_compressed_data_12bit....185
4.4 armral_compressed_data_14bit....186
4.5 armral_compressed_data_8bit....186
4.6 armral_compressed_data_9bit....186
4.7 armral_ldpc_base_graph_t....187

5. Macros....188
5.1 ARMRAL_NUM_COMPLEX_SAMPLES....188
5.2 ARMRAL_LDPC_DEFAULT_OPTIONS....188
5.3 ARMRAL_LDPC_CRC_NO....188
5.4 ARMRAL_LDPC_CRC_16....189
5.5 ARMRAL_LDPC_CRC_24A....189
5.6 ARMRAL_LDPC_CRC_24B....189
5.7 ARMRAL_LDPC_CRC_EVERY_ITER....190
5.8 ARMRAL_LDPC_CRC_END_ITER....190
5.9 ARMRAL_LDPC_FILLER_BITS_IMPLICIT....191
5.10 ARMRAL_LDPC_FILLER_BITS_EXPLICIT....191

6. Enumerations....192
6.1 armral_status....192
6.2 armral_modulation_type....192
6.3 armral_fixed_point_index....192
6.4 armral_polar_frozen_bit_type....193
6.5 armral_polar_ibil_type....194
6.6 armral_fft_direction_t....194
6.7 armral_ldpc_graph_t....194

7. Type Aliases....195

7.1 armral_fft_plan_t... 195

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

We recommend the following. If you do not follow these recommendations your system might not work.

Warning

Your system requires the following. If you do not follow these requirements your system will not work.

Danger

You are at risk of causing permanent damage to your system or your equipment, or harming yourself.

Note

This information is important and 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

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

Introducing ArmRAL

ArmRAL provides optimized signal processing and related maths functions for enabling 5G Radio Access Network (RAN) deployments. It leverages the efficient vector units available on Arm cores that support the Armv8-a architecture to accelerate 5G NR and LTE signal processing workloads, including:

- Matrix and vector arithmetic, such as matrix multiplication.
- Fast Fourier Transforms (FFTs).
- Digital modulation and demodulation.
- Cyclic Redundancy Check (CRC).
- Encoding and decoding schemes, including Polar, Low-Density Parity Check (LDPC), and Turbo.
- Compression and decompression.

You can download ArmRAL from https://gitlab.arm.com/networking/ral.

ArmRAL is built as a static library and must be linked in to any executable that needs to use the library. Users can build and modify the source code to integrate with their components or clients. The `include` directory contains the header files, the `src` directory contains the source code, the `test` directory contains the testing code, the `bench` directory contains the benchmarking code, and the `examples` directory contains the examples.

Before you begin

- 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, the Gold sequence generator, and the convolutional encoder, 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 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 ArmRAL. Navigate to the product directory and use the following commands:

```
mkdir build
cd build
cmake -DBUILD_TESTING=On -DBUILD_EXAMPLES=On ..
make
```

Notes:

- The -DBUILD_TESTING=On and -DBUILD_EXAMPLES=On options are required if you want to run the library tests and benchmarks (-DBUILD_TESTING) and examples (-DBUILD_EXAMPLES).
- 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.

Common CMake options include:

- -DCMAKE_INSTALL_PREFIX=<install-dir>

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

  Default 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_WEXTRA={On|Off}`

  Use (`On`), or do not use (`Off`), `-Wextra` to build the library and tests. `-Wextra` enables additional compiler warnings over the default `-Wall`. Disabled by default to aid compatibility with untested and future compiler releases.

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 memory errors, such as reading or writing past the end of an array. `-DARMRAL_ENABLE_ASAN=On` incurs some reduction in runtime performance.

  Default is `Off`.

- `-DARMRAL_ENABLE_EFENCE={On|Off}`

  Enable Electric Fence when building the library and tests. Electric Fence will cause tests to segmentation fault in the presence of memory errors, such as reading or writing past the end of an array. This option allows you to test executables running under a test runner such as QEMU.

  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 ArmRAL 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 ArmRAL

After you have built ArmRAL, 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 ArmRAL package includes tests for the available functions in the library.

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

To build and run the tests, use:

```
make check
```

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 ArmRAL contain benchmarking code that contains preset problem sizes.

Note: To run the benchmark tests, you must have built ArmRAL 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. Alternatively, the Makefile target:

```
make bench_excel_summary
```

will run the benchmarks and produce an Excel spreadsheet of the results, in addition to printing them as JSON objects. To install the required Python packages for this target, use:

```
pip install -r <path>/python/requirements.txt
```

where `<path>` is the path to the root directory of the library source.

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 ArmRAL 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 ArmRAL, and how to use the library in general, is available in **Use Arm RAN Acceleration Library**, see `docs/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 ArmRAL 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 ArmRAL 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 ArmRAL 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/102249/2504
```

If you have Doxygen installed on your system, you can build a local HTML version of the ArmRAL 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 ArmRAL

To uninstall ArmRAL:

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 Arm RAN Acceleration Library 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 four different coding schemes: Polar, Turbo, Low-Density Parity Check (LDPC), and tail biting convolutional 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 5.1 (for tail biting convolutional coding) and 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, `<code>` can be one of the supported coding schemes (`convolutional`, `ldpc`, `polar` or `turbo`). Set `<code>` to `modulation` for simulation without 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 <args>
  ```

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

  ```
  ./simulation/<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`, `ldpc_awgn` , or `convolutional_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
           [-b num_blocks] [-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>,
  "num_blocks": <num_blocks>,
  "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] [-f len_filler_bits]
```

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>,
  "len_filler_bits": <len_filler_bits>,
  "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

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 ArmRAL before you can use it in your application development, or to run the example programs.

To build the library, use:

```
git clone -b armral-25.04 https://git.gitlab.arm.com/networking/ral.git
mkdir ral/build
cd ral/build
cmake ..
make -j
```

- To use the ArmRAL 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 ArmRAL. 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 ArmRAL source code
- `<path/to/armral/build>` with the path to your build of ArmRAL

2. Run your binary:

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

Example: Run 'fft_cf32_example.c'

In this example, we use ArmRAL 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
    SPDX-FileCopyrightText: <text>Copyright 2020-2025 Arm Limited and/or its
    affiliates <open-source-office@arm.com></text>
    SPDX-License-Identifier: BSD-3-Clause
*/
#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 initialized. 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 ArmRAL source code
- `<path/to/armral/build>` with the path to your build of ArmRAL

Note: For this example, there is a requirement to link against libm (`-lm`). libm is used in several functions in ArmRAL, 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

ArmRAL 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:

```
./polar_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];
}
```

3.1.1.1 armral_cmplx_vecmul_i16

This function performs the element-wise complex multiplication between two complex input sequences, `A` and `B`, of the same length, (`N`).

The implementation uses saturating arithmetic. Intermediate operations are performed on 32-bit variables in Q31 format. To convert the final result back into Q15 format, the final result is right-shifted and narrowed to 16 bits.

```
C[n] = A[n] * B[n], where 0 ≤ n < N-1
```

where:

```
Re{C[n]} = Re{A[n]}*Re{B[n]} - Im{A[n]}*Im{B[n]}
Im{C[n]} = Re{A[n]}*Im{B[n]} + Im{A[n]}*Re{B[n]}
```

Both input and output arrays populate with int16_t elements in Q15 format, with interleaved real and imaginary components:

```
x = {x[0], x[1], ..., x[N-1]}
```

where:

```
x[i] = (Re(x[i]), Im(x[i])), 0 ≤ i < N
```

Syntax

Defined in `armral.h` on line `309`:

```
armral_status armral_cmplx_vecmul_i16(uint32_t n, const armral_cmplx_int16_t *a,
                                      const armral_cmplx_int16_t *b,
                                      armral_cmplx_int16_t *c);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the first input vector.

b

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the second input vector.

c

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output vector.

3.1.1.2 armral_cmplx_vecmul_i16_2

This function performs the element-wise complex multiplication between two complex [I and Q separated] input sequences, `A` and `B`, of the same length (`N`).

The implementation uses saturating arithmetic. Intermediate operations are performed on 32-bit variables in Q31 format. To convert the final result back into Q15 format, the final result is right-shifted and narrowed to 16 bits.

```
C[n] = A[n] * B[n], where 0 ≤ n < N-1
```

where:

```
Re{C[n]} = Re{A[n]}*Re{B[n]} - Im{A[n]}*Im{B[n]}
Im{C[n]} = Re{A[n]}*Im{B[n]} + Im{A[n]}*Re{B[n]}
```

Both input and output arrays populate with int16_t elements in Q15 format, with separate arrays for real and imaginary components:

```
Re(x) = {Re(x[0]), Re(x[1]), ..., Re(x[N-1])}
Im(x) = {Im(x[0]), Im(x[1]), ..., Im(x[N-1])}
```

Syntax

Defined in `armral.h` on line `350`:

```
armral_status armral_cmplx_vecmul_i16_2(uint32_t n, const int16_t *a_re,
                                        const int16_t *a_im,
                                        const int16_t *b_re,
                                        const int16_t *b_im, int16_t *c_re,
                                        int16_t *c_im);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

a_re

A read-only parameter of type `const int16_t *`.

Points to the real part of the first input vector.

a_im

A read-only parameter of type `const int16_t *`.

Points to the imaginary part of the first input vector.

b_re

A read-only parameter of type `const int16_t *`.

Points to the real part of the second input vector.

b_im

A read-only parameter of type `const int16_t *`.

Points to the imaginary part of the second input vector.

c_re

A write-only parameter of type `int16_t *`.

Points to the real part of the output result.

c_im

A write-only parameter of type `int16_t *`.

Points to the imaginary part of the output result.

3.1.1.3 armral_cmplx_vecmul_f32

This function performs the element-wise complex multiplication between two complex input sequences, `A` and `B`, of the same length (`N`).

```
C[n] = A[n] * B[n], where 0 ≤ n < N-1
```

where:

```
Re{C[n]} = Re{A[n]}*Re{B[n]} - Im{A[n]}*Im{B[n]}
Im{C[n]} = Re{A[n]}*Im{B[n]} + Im{A[n]}*Re{B[n]}
```

Both input and output arrays populate with 32-bit float elements, with interleaved real and imaginary components:

```
x = {x[0], x[1], ..., x[N-1]}
```

where:

```
x[i] = (Re(x[i]), Im(x[i])), 0 ≤ i < N
```

Syntax

Defined in `armral.h` on line `390`:

```
armral_status armral_cmplx_vecmul_f32(uint32_t n, const armral_cmplx_f32_t *a,
                                      const armral_cmplx_f32_t *b,
```

```
                                    armral_cmplx_f32_t *c);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

a

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the first input vector.

b

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the second input vector.

c

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output vector.

3.1.1.4 armral_cmplx_vecmul_f32_2

This function performs the element-wise complex multiplication between two complex [I and Q separated] input sequences, `A` and `B`, of the same length (`N`).

```
C[n] = A[n] * B[n], where 0 ≤ n < N-1
```

where:

```
Re{C[n]} = Re{A[n]}*Re{B[n]} - Im{A[n]}*Im{B[n]}
Im{C[n]} = Re{A[n]}*Im{B[n]} + Im{A[n]}*Re{B[n]}
```

Both input and output arrays populate with 32-bit float elements, with separate arrays for real and imaginary components:

```
Re(x) = {Re(x[0]), Re(x[1]), ..., Re(x[N-1])}
Im(x) = {Im(x[0]), Im(x[1]), ..., Im(x[N-1])}
```

Syntax

Defined in `armral.h` on line `427`:

```
armral_status armral_cmplx_vecmul_f32_2(uint32_t n, const float32_t *a_re,
                                        const float32_t *a_im,
                                        const float32_t *b_re,
                                        const float32_t *b_im, float32_t *c_re,
                                        float32_t *c_im);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`n`

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

`a_re`

A read-only parameter of type `const float32_t *`.

Points to the real part of the first input vector.

`a_im`

A read-only parameter of type `const float32_t *`.

Points to the imaginary part of the first input vector.

`b_re`

A read-only parameter of type `const float32_t *`.

Points to the real part of the second input vector.

`b_im`

A read-only parameter of type `const float32_t *`.

Points to the imaginary part of the second input vector.

`c_re`

A write-only parameter of type `float32_t *`.

Points to the real part of the output result.

`c_im`

A write-only parameter of type `float32_t *`.

Points to the imaginary part of the output result.

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];
}
```

3.1.2.1 armral_cmplx_vecdot_f32

This function computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed. Array elements are assumed to be complex float32 and with interleaved real and imaginary parts.

Syntax

Defined in `armral.h` on line `477`:

```
armral_status armral_cmplx_vecdot_f32(uint32_t n,
                                      const armral_cmplx_f32_t *p_src_a,
                                      const armral_cmplx_f32_t *p_src_b,
                                      armral_cmplx_f32_t *p_src_c);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`n`

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

`p_src_a`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the first complex input vector.

p_src_b

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the second complex input vector.

p_src_c

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output complex vector.

3.1.2.2 armral_cmplx_vecdot_f32_2

This function computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed. Array elements are assumed to be 32-bit floats, and separate arrays are used for the real and imaginary parts of the input data.

Syntax

Defined in `armral.h` on line `499`:

```
armral_status armral_cmplx_vecdot_f32_2(uint32_t n, const float32_t *p_src_a_re,
                                        const float32_t *p_src_a_im,
                                        const float32_t *p_src_b_re,
                                        const float32_t *p_src_b_im,
                                        float32_t *p_src_c_re,
                                        float32_t *p_src_c_im);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

p_src_a_re

A read-only parameter of type `const float32_t *`.

Points to the real part of the first input vector.

p_src_a_im

A read-only parameter of type `const float32_t *`.

Points to the imaginary part of the first input vector.

p_src_b_re

A read-only parameter of type `const float32_t *`.

Points to the real part of the second input vector.

p_src_b_im

A read-only parameter of type `const float32_t *`.

Points to the imaginary part of the second input vector.

p_src_c_re

A write-only parameter of type `float32_t *`.

Points to the real part of the output result.

p_src_c_im

A write-only parameter of type `float32_t *`.

Points to the imaginary part of the output result.

3.1.2.3 armral_cmplx_vecdot_i16

This function computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed. Array elements are assumed to be complex int16 in Q15 format and interleaved.

To avoid overflow issues input values are internally extended to 32-bit variables and all intermediate calculations results are stored in 64-bit internal variables. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `522`:

```
armral_status armral_cmplx_vecdot_i16(uint32_t n,
                                      const armral_cmplx_int16_t *p_src_a,
                                      const armral_cmplx_int16_t *p_src_b,
                                      armral_cmplx_int16_t *p_src_c);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the first input vector.

p_src_b

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the second input vector.

n

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

p_src_c

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output complex result.

3.1.2.4 armral_cmplx_vecdot_i16_2

This function computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed. Array elements are assumed to be int16 in Q15 format and separate arrays are used for real parts and imaginary parts of the input data.

To avoid overflow issues input values are internally extended to 32-bit variables and all intermediate calculations results are stored in 64-bit internal variables. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `547`:

```
armral_status armral_cmplx_vecdot_i16_2(uint32_t n, const int16_t *p_src_a_re,
                                        const int16_t *p_src_a_im,
                                        const int16_t *p_src_b_re,
                                        const int16_t *p_src_b_im,
                                        int16_t *p_src_c_re,
                                        int16_t *p_src_c_im);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p_src_a_re

A read-only parameter of type `const int16_t *`.

Points to the real part of first input vector.

p_src_a_im

A read-only parameter of type `const int16_t *`.

Points to the imag part of first input vector.

p_src_b_re

A read-only parameter of type `const int16_t *`.

Points to the real part of second input vector.

p_src_b_im

A read-only parameter of type `const int16_t *`.

Points to the imag part of second input vector.

n

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

p_src_c_re

A write-only parameter of type `int16_t *`.

Points to the real part of output complex result.

p_src_c_im

A write-only parameter of type `int16_t *`.

Points to the imag part of output complex result.

3.1.2.5 armral_cmplx_vecdot_i16_32bit

This function computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed. Array elements are assumed to be complex int16 in Q15 format and interleaved.

All intermediate calculations results are stored in 32-bit internal variables, saturating the value to prevent overflow. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `570`:

```
armral_status armral_cmplx_vecdot_i16_32bit(uint32_t n,
                                            const armral_cmplx_int16_t *p_src_a,
                                            const armral_cmplx_int16_t *p_src_b,
                                            armral_cmplx_int16_t *p_src_c);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the first input vector.

p_src_b

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the second input vector.

p_src_c

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output complex result.

3.1.2.6 armral_cmplx_vecdot_i16_2_32bit

This function computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed.

Array elements are assumed to be int16 in Q15 format and separate arrays are used for both the real parts and imaginary parts of the input data. All intermediate calculation results are stored in 32-bit internal variables, saturating the value to prevent overflow. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `596`:

```
armral_status armral_cmplx_vecdot_i16_2_32bit(
    uint32_t n, const int16_t *p_src_a_re, const int16_t *p_src_a_im,
    const int16_t *p_src_b_re, const int16_t *p_src_b_im, int16_t *p_src_c_re,
    int16_t *p_src_c_im);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of samples in each vector.

p_src_a_re

A read-only parameter of type `const int16_t *`.

Points to the real part of the first input vector.

p_src_a_im

A read-only parameter of type `const int16_t *`.

Points to the imaginary part of the first input vector.

p_src_b_re

A read-only parameter of type `const int16_t *`.

Points to the real part of the second input vector.

p_src_b_im

A read-only parameter of type `const int16_t *`.

Points to the imaginary part of the second input vector.

p_src_c_re

A write-only parameter of type `int16_t *`.

Points to the real part of the output result.

p_src_c_im

A write-only parameter of type `int16_t *`.

Points to the imaginary part of the output result.

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` functions 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.

3.2.1.1 armral_cmplx_mat_vec_mult_i16

This function performs the multiplication `y = A x` for matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex int16 in Q15 format.
- Matrices are stored in memory in row-major order.

A 64-bit Q32.31 accumulator is used internally. If you do not need such a large range, consider using armral_cmplx_mat_vec_mult_i16_32bit instead. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `634`:

```
armral_status armral_cmplx_mat_vec_mult_i16(uint16_t m, uint16_t n,
                                             const armral_cmplx_int16_t *p_src_a,
                                             const armral_cmplx_int16_t *p_src_x,
                                             armral_cmplx_int16_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`m`

A read-only parameter of type `uint16_t`.

The number of rows in matrix `A` and the length of the output vector `y`.

`n`

A read-only parameter of type `uint16_t`.

The number of columns in matrix `A` and the length of the input vector `x` .

`p_src_a`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input matrix `A`.

`p_src_x`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input vector `x`.

`p_dst`

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output vector `y`.

3.2.1.2 armral_cmplx_mat_vec_mult_batch_i16

This function performs matrix-vector multiplication for a batch of `M` -by- `N` matrices and length `N` input vectors. Each multiplication is of the form `y = A x` for a matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex int16 in Q15 format.

- Matrices are stored in memory in row-major order.

The matrix elements are interleaved such that all elements for a particular location within the matrix are stored together. This means that, for instance, the first `num_mats` complex numbers stored are the first element of each of the matrices in the batch. The offset to the next location in the same matrix is given by the `num_mats` batch size:

```
{Re(0), Im(0), Re(1), Im(1), Re(2), Im(2), ...}
```

The same layout is used for vector elements, except that the offset to the next vector element is `num_mats * num_vecs_per_mat`.

The total number of elements in the batch (`num_mats * num_vecs_per_mat`) must be a multiple of 12. The number of vectors per matrix must be either 1 or a multiple of 4.

A 64-bit Q32.31 accumulator is used internally. If you do not need such a large range, consider using armral_cmplx_mat_vec_mult_batch_i16_32bit instead. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `683`:

```
armral_status armral_cmplx_mat_vec_mult_batch_i16(
    uint16_t num_mats, uint16_t num_vecs_per_mat, uint16_t m, uint16_t n,
    const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_x,
    armral_cmplx_int16_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_mats`

A read-only parameter of type `uint16_t`.

The number of input matrices.

`num_vecs_per_mat`

A read-only parameter of type `uint16_t`.

The number of input and output vectors for each input matrix. The total number of input vectors is `num_mats * num_vecs_per_mat`. There are the same number of output vectors.

`m`

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in each matrix `A` and the length of each output vector `y`.

`n`

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in each matrix `A` and the length of each input vector `x`.

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input matrix `A`.

p_src_x

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input vector `x`.

p_dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output vector `y`.

3.2.1.3 armral_cmplx_mat_vec_mult_batch_i16_pa

This function performs matrix-vector multiplication for a batch of `M` -by- `N` matrices and length `N` input vectors, utilizing a "pointer array" storage layout for the input and output matrix batches. Each multiplication is of the form `y = A x` for a matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex int16 in Q15 format.
- Matrices are stored in memory in row-major order.

The `p_srcs_a` parameter is an array of pointers of length `M * N`. The value of `p_srcs_a[i]` is a pointer to the i-th element of the first matrix in the batch, as represented in row-major ordering, such that the i-th element of the j-th matrix in the batch is located at `p_srcs_a[i][j]`. For example, the j-th matrix in a batch of 2-by-2 matrices is formed as:

```
p_srcs_a[0][j]  p_srcs_a[1][j]
p_srcs_a[2][j]  p_srcs_a[3][j]
```

The input vector array `p_srcs_x` and output vector array `p_dsts` also point to an array of pointers, such that the i-th element of the j-th vector is located at `p_srcs_x[i][j]` (and similarly for `p_dsts`). For example, the j-th vector in a batch of vectors of length `2` is formed as:

```
p_srcs_x[0][j]
p_srcs_x[1][j]
```

representing an identical format to the input matrices.

The total number of elements in the batch (`num_mats * num_vecs_per_mat`) must be a multiple of 12. The number of vectors per matrix must be either 1 or a multiple of 4.

A 64-bit Q32.31 accumulator is used internally. If you do not need such a large range, consider using armral_cmplx_mat_vec_mult_batch_i16_32bit_pa instead. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `741`:

```
armral_status armral_cmplx_mat_vec_mult_batch_i16_pa(
    uint16_t num_mats, uint16_t num_vecs_per_mat, uint16_t m, uint16_t n,
    const armral_cmplx_int16_t **p_srcs_a,
    const armral_cmplx_int16_t **p_srcs_x, armral_cmplx_int16_t **p_dsts);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_mats`

A read-only parameter of type `uint16_t`.

The number of input matrices.

`num_vecs_per_mat`

A read-only parameter of type `uint16_t`.

The number of input and output vectors for each input matrix. The total number of input vectors is `num_mats * num_vecs_per_mat`. There are the same number of output vectors.

`m`

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in each matrix `A` and the length of each output vector `y`.

`n`

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in each matrix `A` and the length of each input vector `x`.

`p_srcs_a`

A read-only parameter of type `const armral_cmplx_int16_t **`.

Points to the input matrix structure.

`p_srcs_x`

A read-only parameter of type `const armral_cmplx_int16_t **`.

Points to the input vector structure.

`p_dsts`

A write-only parameter of type `armral_cmplx_int16_t **`.

Points to the output vector structure.

3.2.1.4 armral_cmplx_mat_vec_mult_i16_32bit

This function performs the multiplication `y = A x` for matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex int16 in Q15 format.
- Matrices are stored in memory in row-major order.

A 32-bit Q31 saturating accumulator is used internally. If you need a larger range, consider using armral_cmplx_mat_vec_mult_i16 instead. To get a Q15 result, the final result is narrowed to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `766`:

```
armral_status armral_cmplx_mat_vec_mult_i16_32bit(
    uint16_t m, uint16_t n, const armral_cmplx_int16_t *p_src_a,
    const armral_cmplx_int16_t *p_src_x, armral_cmplx_int16_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows in matrix `A` and the length of the output vector `y`.

n

A read-only parameter of type `uint16_t`.

The number of columns in matrix `A` and the length of each input vector `x`.

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input matrix `A`.

p_src_x

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input vector `x`.

p_dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output matrix `y`.

3.2.1.5 armral_cmplx_mat_vec_mult_batch_i16_32bit

This function performs matrix-vector multiplication for a batch of `M` -by- `N` matrices and length `N` input vectors. Each multiplication is of the form `y = A x` for a matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex int16 in Q15 format.
- Matrices are stored in memory in row-major order.

The matrix elements are interleaved such that all elements for a particular location within the matrix are stored together. This means that, for instance, the first `num_mats` complex numbers stored are the first element of each of the matrices in the batch. The offset to the next location in the same matrix is given by the `num_mats` batch size:

```
{Re(0), Im(0), Re(1), Im(1), Re(2), Im(2), ...}
```

The same layout is used for vector elements, except that the offset to the next vector element is `num_mats * num_vecs_per_mat`.

A 32-bit Q31 saturating accumulator is used internally. If you need a larger range, consider using armral_cmplx_mat_vec_mult_batch_i16 instead. To get a Q15 result, the final result is narrowed to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `809`:

```
armral_status armral_cmplx_mat_vec_mult_batch_i16_32bit(
    uint16_t num_mats, uint16_t num_vecs_per_mat, uint16_t m, uint16_t n,
    const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_x,
    armral_cmplx_int16_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_mats

A read-only parameter of type `uint16_t`.

The number of input matrices.

num_vecs_per_mat

A read-only parameter of type `uint16_t`.

The number of input and output vectors for each input matrix. The total number of input vectors is `num_mats * num_vecs_per_mat`. There are the same number of output vectors.

m

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in each matrix `A` and the length of each output vector `y`.

n

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in each matrix `A` and the length of each input vector `x`.

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input matrix `A`.

p_src_x

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input vector `x`.

p_dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output vector `y`.

3.2.1.6 armral_cmplx_mat_vec_mult_batch_i16_32bit_pa

This function performs matrix-vector multiplication for a batch of `M` -by- `N` matrices and length `N` input vectors, utilizing a "pointer array" storage layout for the input and output matrix batches. Each multiplication is of the form `y = A x` for a matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex int16 in Q15 format.
- Matrices are stored in memory in row-major order.

The `p_srcs_a` parameter is an array of pointers of length `M * N`. The value of `p_srcs_a[i]` is a pointer to the i-th element of the first matrix in the batch, as represented in row-major ordering, such that the i-th element of the j-th matrix in the batch is located at `p_srcs_a[i][j]`. For example, the j-th matrix in a batch of 2-by-2 matrices is formed as:

```
p_srcs_a[0][j]  p_srcs_a[1][j]
p_srcs_a[2][j]  p_srcs_a[3][j]
```

The input vector array `p_srcs_x` and output vector array `p_dsts` also point to an array of pointers, such that the i-th element of the j-th vector is located at `p_srcs_x[i][j]` (and similarly for `p_dsts`). For example, the j-th vector in a batch of vectors of length `2` is formed as:

```
p_srcs_x[0][j]
p_srcs_x[1][j]
```

representing an identical format to the input matrices.

A 32-bit Q31 saturating accumulator is used internally. If you need a larger range, consider using armral_cmplx_mat_vec_mult_batch_i16_pa instead. To get a Q15 result, the final result is narrowed to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `862`:

```
armral_status armral_cmplx_mat_vec_mult_batch_i16_32bit_pa(
    uint16_t num_mats, uint16_t num_vecs_per_mat, uint16_t m, uint16_t n,
    const armral_cmplx_int16_t **p_srcs_a,
    const armral_cmplx_int16_t **p_srcs_x, armral_cmplx_int16_t **p_dsts);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_mats`

A read-only parameter of type `uint16_t`.

The number of input matrices.

`num_vecs_per_mat`

A read-only parameter of type `uint16_t`.

The number of input and output vectors for each input matrix. The total number of input vectors is `num_mats * num_vecs_per_mat`. There are the same number of output vectors.

`m`

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in each matrix `A` and the length of each output vector `y`.

`n`

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in each matrix `A` and the length of each input vector `x`.

`p_srcs_a`

A read-only parameter of type `const armral_cmplx_int16_t **`.

Points to the input matrix structure.

`p_srcs_x`

A read-only parameter of type `const armral_cmplx_int16_t **`.

Points to the input vector structure.

`p_dsts`

A write-only parameter of type `armral_cmplx_int16_t **`.

Points to the output vector structure.

3.2.1.7 armral_cmplx_mat_vec_mult_f32

This function performs the multiplication `y = A x` for matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex 32-bit float values.
- Matrices are stored in memory in row-major order.

Syntax

Defined in `armral.h` on line `882`:

```
armral_status armral_cmplx_mat_vec_mult_f32(uint16_t m, uint16_t n,
                                            const armral_cmplx_f32_t *p_src_a,
                                            const armral_cmplx_f32_t *p_src_x,
                                            armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows in matrix `A` and the length of the output vector `y`.

n

A read-only parameter of type `uint16_t`.

The number of columns in matrix `A` and the length of the input vector `x` .

p_src_a

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix `A`.

p_src_x

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input vector `x`.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix `y`.

3.2.1.8 armral_cmplx_mat_vec_mult_batch_f32

This function performs matrix-vector multiplication for a batch of `M` -by- `N` matrices and length `N` input vectors. Each multiplication is of the form `y = A x` for a matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex 32-bit float values.
- Matrices are stored in memory in row-major order.

The matrix elements are interleaved such that all elements for a particular location within the matrix are stored together. This means that, for instance, the first `num_mats` complex numbers stored are the first element of each of the matrices in the batch. The offset to the next location in the same matrix is given by the `num_mats` batch size:

```
{Re(0), Im(0), Re(1), Im(1), Re(2), Im(2), ...}
```

The same layout is used for vector elements, except that the offset to the next vector element is `num_mats * num_vecs_per_mat`.

The total number of elements in the batch (`num_mats * num_vecs_per_mat`) must be a multiple of 12.

Syntax

Defined in `armral.h` on line `925`:

```
armral_status armral_cmplx_mat_vec_mult_batch_f32(
    uint16_t num_mats, uint16_t num_vecs_per_mat, uint16_t m, uint16_t n,
    const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_x,
    armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_mats`

A read-only parameter of type `uint16_t`.

The number of input matrices.

`num_vecs_per_mat`

A read-only parameter of type `uint16_t`.

The number of input and output vectors for each input matrix. The total number of input vectors is `num_mats * num_vecs_per_mat`. There are the same number of output vectors.

`m`

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in each matrix `A` and the length of each output vector `y`.

n

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in each matrix `A` and the length of each input vector `x`.

p_src_a

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix `A`.

p_src_x

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input vector `x`.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output vector `y`.

3.2.1.9 armral_cmplx_mat_vec_mult_batch_f32_pa

This function performs matrix-vector multiplication for a batch of `M` -by- `N` matrices and length `N` input vectors, utilizing a "pointer array" storage layout for the input and output matrix batches. Each multiplication is of the form `y = A x` for a matrix `A` and vectors `x` and `y`, and assumes that:

- Matrix and vector elements are complex 32-bit float values.
- Matrices are stored in memory in row-major order.

The `p_srcs_a` parameter is an array of pointers of length `M * N`. The value of `p_srcs_a[i]` is a pointer to the i-th element of the first matrix in the batch, as represented in row-major ordering, such that the i-th element of the j-th matrix in the batch is located at `p_srcs_a[i][j]`. For example, the j-th matrix in a batch of 2-by-2 matrices is formed as:

```
p_srcs_a[0][j]  p_srcs_a[1][j]
p_srcs_a[2][j]  p_srcs_a[3][j]
```

The input vector array `p_srcs_x` and output vector array `p_dsts` also point to an array of pointers, such that the i-th element of the j-th vector is located at `p_srcs_x[i][j]` (and similarly for `p_dsts`). For example, the j-th vector in a batch of vectors of length `2` is formed as:

```
p_srcs_x[0][j]
p_srcs_x[1][j]
```

representing an identical format to the input matrices.

Syntax

Defined in `armral.h` on line `973`:

```
armral_status armral_cmplx_mat_vec_mult_batch_f32_pa(
    uint16_t num_mats, uint16_t num_vecs_per_mat, uint16_t m, uint16_t n,
    const armral_cmplx_f32_t **p_srcs_a, const armral_cmplx_f32_t **p_srcs_x,
    armral_cmplx_f32_t **p_dsts);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_mats`

A read-only parameter of type `uint16_t`.

The number of input matrices.

`num_vecs_per_mat`

A read-only parameter of type `uint16_t`.

The number of input and output vectors for each input matrix. The total number of input vectors is `num_mats * num_vecs_per_mat`. There are the same number of output vectors.

`m`

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in each matrix `A` and the length of each output vector `y`.

`n`

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in each matrix `A` and the length of each input vector `x`.

`p_srcs_a`

A read-only parameter of type `const armral_cmplx_f32_t **`.

Points to the input matrix structure.

`p_srcs_x`

A read-only parameter of type `const armral_cmplx_f32_t **`.

Points to the input vector structure.

`p_dsts`

A write-only parameter of type `armral_cmplx_f32_t **`.

Points to the output vector structure.

3.2.2 General Complex Matrix-Matrix Multiplication

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

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

3.2.2.1 armral_cmplx_matmul_i16

This function performs the multiplication `C = A B` for matrices, and assumes that:

- Matrix elements are complex int16 in Q15 format.
- Matrices are stored in memory in row-major order.

A 64-bit Q32.31 accumulator is used internally. If you do not need such a large range, consider using armral_cmplx_matmul_i16_32bit instead. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `1012`:

```
armral_status armral_cmplx_matmul_i16(uint16_t m, uint16_t n, uint16_t k,
                                      const armral_cmplx_int16_t *p_src_a,
                                      const armral_cmplx_int16_t *p_src_b,
                                      armral_cmplx_int16_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in matrices `A` and `C`.

n

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in matrices `B` and `C`.

k

A read-only parameter of type `uint16_t`.

The number of columns (`K`) in matrix `A` and the number of rows in matrix `B`.

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the first input matrix `A`.

p_src_b

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the second input matrix `B`.

p_dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output matrix `C`.

3.2.2.2 armral_cmplx_matmul_i16_noalloc

This function performs the multiplication `C = A B` for matrices, and assumes that:

- Matrix elements are complex int16 in Q15 format.
- Matrices are stored in memory in row-major order.

A 64-bit Q32.31 accumulator is used internally. If you do not need such a large range, consider using armral_cmplx_matmul_i16_32bit instead. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

Note:

- The buffer must be at least `k * n * sizeof(armral_cmplx_int16_t)` bytes.

Syntax

Defined in `armral.h` on line `1045`:

```
armral_status
armral_cmplx_matmul_i16_noalloc(uint16_t m, uint16_t n, uint16_t k,
                                const armral_cmplx_int16_t *p_src_a,
                                const armral_cmplx_int16_t *p_src_b,
                                armral_cmplx_int16_t *p_dst, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in matrices `A` and `C`.

n

A read-only parameter of type `uint16_t`.

The number of columns (N) in matrices B and C.

k

A read-only parameter of type `uint16_t`.

The number of columns (K) in matrix A and the number of rows in matrix B.

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the first input matrix A.

p_src_b

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the second input matrix B.

p_dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output matrix C.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.2.2.3 armral_cmplx_matmul_i16_32bit

This function performs the multiplication `C = A B` for matrices, and assumes that:

- Matrix elements are complex int16 in Q15 format.
- Matrices are stored in memory in row-major order.

A 32-bit Q31 saturating accumulator is used internally. If you need a larger range, consider using armral_cmplx_matmul_i16 instead. To get a Q15 result, the final result is narrowed to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `1071`:

```
armral_status armral_cmplx_matmul_i16_32bit(uint16_t m, uint16_t n, uint16_t k,
                                            const armral_cmplx_int16_t *p_src_a,
                                            const armral_cmplx_int16_t *p_src_b,
                                            armral_cmplx_int16_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in matrices `A` and `C`.

n

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in matrices `B` and `C`.

k

A read-only parameter of type `uint16_t`.

The number of columns (`K`) in matrix `A` and the number of rows in matrix `B`.

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the first input matrix `A`.

p_src_b

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the second input matrix `B`.

p_dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output matrix `C`.

3.2.2.4 armral_cmplx_matmul_f32

This function performs the multiplication `C = A B` for matrices, and assumes that:

- Matrix elements are complex 32-bit floating point values.
- Matrices are stored in memory in row-major order.

Syntax

Defined in `armral.h` on line `1092`:

```
armral_status armral_cmplx_matmul_f32(uint16_t m, uint16_t n, uint16_t k,
                                      const armral_cmplx_f32_t *p_src_a,
                                      const armral_cmplx_f32_t *p_src_b,
                                      armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in matrices `A` and `C`.

n

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in matrices `B` and `C`.

k

A read-only parameter of type `uint16_t`.

The number of columns (`K`) in matrix `A` and the number of rows in matrix `B`.

p_src_a

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the first input matrix `A`.

p_src_b

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the second input matrix `B`.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix `C`.

3.2.2.5 armral_cmplx_matmul_aah_f32

This function performs a matrix multiplication of an input `M` -by- `N` matrix `A` with its conjugate transpose `A^H`:

```
C = A A^H
```

`C` is therefore `M` -by- `M`.

The input matrix `p_src_a` and output matrix `p_src_c` are stored contiguously in memory, in row-major order.

`p_src_a` and `p_dst_c` must not alias each other.

Syntax

Defined in `armral.h` on line `1120`:

```
armral_status armral_cmplx_matmul_aah_f32(uint16_t m, uint16_t n,
```

```
                                          const armral_cmplx_f32_t *p_src_a,
                                          armral_cmplx_f32_t *p_dst_c);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows (`M`) in the input matrix `A`, and the number of rows and columns in the output matrix `C`.

n

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in the input matrix `A`.

p_src_a

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix `A`.

p_dst_c

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix `C`.

3.2.2.6 armral_cmplx_matmul_ahb_f32

This function performs the multiplication of the conjugate transpose of `A` with the matrix `B`, to compute the matrix `C`. That is:

```
C = A^H B
```

Matrix `A` is `K` -by- `M`, `B` is `K` -by- `N`, and `C` is `M` -by- `N`. All matrices are stored contiguously in memory, in row-major order.

None of the arrays passed to this function are allowed to alias.

Syntax

Defined in `armral.h` on line `1147`:

```
armral_status armral_cmplx_matmul_ahb_f32(uint16_t m, uint16_t n, uint16_t k,
                                          const armral_cmplx_f32_t *p_src_a,
                                          const armral_cmplx_f32_t *p_src_b,
                                          armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of columns (`M`) in matrix `A` and the number of rows in matrix `C`.

n

A read-only parameter of type `uint16_t`.

The number of columns (`N`) in matrices `B` and `C`.

k

A read-only parameter of type `uint16_t`.

The number of rows (`K`) in matrices `A` and `B`.

p_src_a

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix `A`.

p_src_b

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix `B`.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix `C`.

3.2.3 Specific-Sized Complex Matrix-Matrix Multiplication

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

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

3.2.3.1 armral_cmplx_mat_mult_2x2_f32

This function performs an optimized product of two square 2-by-2 matrices. The function assumes matrices are stored in column-major order. In LTE and 5G, you can use the `armral_cmplx_mat_mult_2x2_f32` function in the equalization step in the formula:

```
x^ = G y
```

Equalization matrix `G` corresponds to the first input matrix (matrix `A`) of the function. The function assumes that matrix `G` is transposed during computation so that the matrix presents as column-major on input.

The second input matrix (matrix `B`) is formed by two 2-by-1 vectors (`y` vectors in the preceding formula) so that each row of `B` represents a 2-by-1 vector output from each antenna port, and each call to `armral_cmplx_mat_mult_2x2_f32` computes two distinct `x^` estimates.

Syntax

Defined in `armral.h` on line `1189`:

```
armral_status armral_cmplx_mat_mult_2x2_f32(const armral_cmplx_f32_t *p_src_a,
                                            const armral_cmplx_f32_t *p_src_b,
                                            armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`p_src_a`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the first input matrix `A`.

`p_src_b`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the second input matrix `B`.

`p_dst`

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix.

3.2.3.2 armral_cmplx_mat_mult_2x2_f32_iq

This function performs an optimized product of two square 2-by-2 matrices whose complex elements have already been separated into real component and imaginary component arrays. The function assumes that matrices are stored in column-major order. In LTE and 5G, you can use the `armral_cmplx_mat_mult_2x2_f32_iq` function in the equalization step in the formula:

```
x^ = G y
```

Equalization matrix `G` corresponds to the first input matrix (matrix `A`) of the function. The function assumes matrix `G` is transposed during computation so that the matrix presents as column-major on input. The second input matrix (matrix `B`) is formed by two 2-by-1 vectors (`y` vectors in the preceding formula) so that each row of `B` represents a 2-by-1 vector output from each antenna port, and each call to `armral_cmplx_mat_mult_2x2_f32_iq` computes two distinct `x^` estimates.

Syntax

Defined in `armral.h` on line `1221`:

```
armral_status armral_cmplx_mat_mult_2x2_f32_iq(const float32_t *src_a_re,
                                                const float32_t *src_a_im,
                                                const float32_t *src_b_re,
                                                const float32_t *src_b_im,
                                                float32_t *dst_re,
                                                float32_t *dst_im);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`src_a_re`

A read-only parameter of type `const float32_t *`.

Points to the real part of the first input matrix.

`src_a_im`

A read-only parameter of type `const float32_t *`.

Points to the imag part of the first input matrix.

`src_b_re`

A read-only parameter of type `const float32_t *`.

Points to the real part of the second input matrix.

`src_b_im`

A read-only parameter of type `const float32_t *`.

Points to the imag part of the second input matrix.

`dst_re`

A write-only parameter of type `float32_t *`.

Points to the real part of the output matrix.

`dst_im`

A write-only parameter of type `float32_t *`.

Points to the imag part of the output matrix.

3.2.3.3 armral_cmplx_mat_mult_4x4_f32

This function performs an optimized product of two square 4-by-4 matrices. The function assumes that matrices are stored in column-major order. In LTE and 5G, you can use the `armral_cmplx_mat_mult_4x4_f32` function in the equalization step in the formula:

```
x^ = G y
```

Equalization matrix `G` corresponds to the first input matrix (matrix `A`) of the function. The function assumes that matrix `G` is transposed during computation so that the matrix presents as column-major on input. The second input matrix (matrix `B`) is formed by four 4-by-1 vectors (`y` vectors in the preceding formula) so that each row of `B` represents a 4-by-1 vector output from each antenna port, and each call to `cmplx_mat_mult_4x4_f32` computes four distinct `x^` estimates.

Syntax

Defined in `armral.h` on line `1251`:

```
armral_status armral_cmplx_mat_mult_4x4_f32(const armral_cmplx_f32_t *p_src_a,
                                            const armral_cmplx_f32_t *p_src_b,
                                            armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`p_src_a`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the first input matrix `A`.

`p_src_b`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the second input matrix `B`.

`p_dst`

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix.

3.2.3.4 armral_cmplx_mat_mult_4x4_f32_iq

This function performs an optimized product of two square 4-by-4 matrices whose complex elements have already been separated into real and imaginary component arrays. The function assumes matrices are stored in column-major order. In LTE and 5G, you can use the `armral_cmplx_mat_mult_4x4_f32_iq` function in the equalization step in the formula:

```
x^ = G y
```

Equalization matrix `G` corresponds to the first input matrix (matrix `A`) of the function. The function assumes that matrix `G` is transposed during computation so that the matrix presents as column-major on input. The second input matrix (matrix `B`) is formed by four 4-by-1 vectors (`y` vectors in the preceding formula) so that each row of `B` represents a 4-by-1 vector output from each antenna port, and each call to `armral_cmplx_mat_mult_4x4_f32_iq` computes four distinct `x^` estimates.

Syntax

Defined in `armral.h` on line `1282`:

```
armral_status armral_cmplx_mat_mult_4x4_f32_iq(const float32_t *src_a_re,
                                               const float32_t *src_a_im,
                                               const float32_t *src_b_re,
                                               const float32_t *src_b_im,
                                               float32_t *dst_re,
                                               float32_t *dst_im);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

src_a_re

A read-only parameter of type `const float32_t *`.

Points to the real part of the first input matrix.

src_a_im

A read-only parameter of type `const float32_t *`.

Points to the imag part of the first input matrix.

src_b_re

A read-only parameter of type `const float32_t *`.

Points to the real part of the second input matrix.

src_b_im

A read-only parameter of type `const float32_t *`.

Points to the imag part of the second input matrix.

dst_re

A write-only parameter of type `float32_t *`.

Points to the real part of the output matrix.

dst_im

A write-only parameter of type `float32_t *`.

Points to the imag part of the output matrix.

3.2.4 Channel 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, these functions take an extra fixed-point type specifier.

3.2.4.1 armral_solve_2x2_f32

In LTE and 5G, you can use the `armral_solve_2x2_f32` function in the equalization step, as in the formula:

```
x^ = G y
```

where `y` is a vector for the received signal, size corresponds to the number of antennae and `x^` is the estimate of the transmitted signal, size corresponds to the number of layers. `G` is the equalization complex matrix and is assumed to be a 2-by-2 matrix. I and Q components of `G` elements are assumed to be stored separated in memory. Also, each coefficient of `G` (`Gxy`, for `x, y = {1, 2}`) is assumed to be stored separated in memory locations set at `p_gstride` one from the other.

The number of input signals is assumed to be a multiple of 12, and must be divisible by the number of subcarriers per `G` matrix.

For type 1 equalization, the number of subcarriers per `G` matrix must be four. For type 2 equalization, the number of subcarriers per `G` matrix must be six. An implementation is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1342`:

```
armral_status
armral_solve_2x2_f32(uint32_t num_sub_carrier, uint32_t num_sc_per_g,
                     const armral_cmplx_int16_t *p_y, uint32_t p_ystride,
                     const armral_fixed_point_index *p_y_num_fract_bits,
                     const float32_t *p_g_real, const float32_t *p_g_imag,
                     uint32_t p_gstride, armral_cmplx_int16_t *p_x,
                     uint32_t p_xstride,
                     armral_fixed_point_index num_fract_bits_x);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_sub_carrier`

A read-only parameter of type `uint32_t`.

The number of subcarriers to equalize.

`num_sc_per_g`

A read-only parameter of type `uint32_t`.

The number of subcarriers per `G` matrix.

`p_y`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input received signal.

`p_ystride`

A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

`p_y_num_fract_bits`

A read-only parameter of type `const armral_fixed_point_index *`.

The number of fractional bits in `y`.

`p_g_real`

A read-only parameter of type `const float32_t *`.

The real part of coefficient matrix `G`.

`p_g_imag`

A read-only parameter of type `const float32_t *`.

The imag part of coefficient matrix `G`.

`p_gstride`

A read-only parameter of type `uint32_t`.

The stride between elements of `G`.

`p_x`

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output received signal.

`p_xstride`

A read-only parameter of type `uint32_t`.

The stride between two layers.

num_fract_bits_x

A read-only parameter of type `armral_fixed_point_index`.

The number of fractional bits in `x`.

3.2.4.2 armral_solve_2x4_f32

In LTE and 5G, you can use the `armral_solve_2x4_f32` function in the equalization step, as in the formula:

```
x^ = G y
```

where `y` is a vector for the received signal, size corresponds to the number of antennae and `x^` is the estimate of the transmitted signal, size corresponds to the number of layers.

`G` is the equalization complex matrix and is assumed to be a 2-by-4 matrix. I and Q components of `G` elements are assumed to be stored separated in memory.

Also, each coefficient of `G` (`Gxy`, for `x = {1, 2}` and `y = {1, 2, 3, 4}`) is assumed to be stored separated in memory locations set at `p_gstride` one from the other.

The number of input signals is assumed to be a multiple of 12, and must be divisible by the number of subcarriers per `G` matrix.

For type 1 equalization, the number of subcarriers per `G` matrix must be four. For type 2 equalization, the number of subcarriers per `G` matrix must be six. An implementation is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1392`:

```
armral_status
armral_solve_2x4_f32(uint32_t num_sub_carrier, uint32_t num_sc_per_g,
                     const armral_cmplx_int16_t *p_y, uint32_t p_ystride,
                     const armral_fixed_point_index *p_y_num_fract_bits,
                     const float32_t *p_g_real, const float32_t *p_g_imag,
                     uint32_t p_gstride, armral_cmplx_int16_t *p_x,
                     uint32_t p_xstride,
                     armral_fixed_point_index num_fract_bits_x);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_sub_carrier

A read-only parameter of type `uint32_t`.

The number of subcarriers to equalize.

num_sc_per_g
A read-only parameter of type `uint32_t`.

The number of subcarriers per `G` matrix.

p_y
A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input received signal.

p_ystride
A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

p_y_num_fract_bits
A read-only parameter of type `const armral_fixed_point_index *`.

The number of fractional bits in `y`.

p_g_real
A read-only parameter of type `const float32_t *`.

The real part of coefficient matrix `G`.

p_g_imag
A read-only parameter of type `const float32_t *`.

The imag part of coefficient matrix `G`.

p_gstride
A read-only parameter of type `uint32_t`.

The stride between elements of `G`.

p_x
A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output received signal.

p_xstride
A read-only parameter of type `uint32_t`.

The stride between two layers.

num_fract_bits_x
A read-only parameter of type `armral_fixed_point_index`.

The number of fractional bits in `x`.

3.2.4.3 armral_solve_4x4_f32

In LTE and 5G, you can use the `armral_solve_4x4_f32` function in the equalization step, as in the formula:

```
x^ = G y
```

where `y` is a vector for the received signal, size corresponds to the number of antennae and `x^` is the estimate of the transmitted signal, size corresponds to the number of layers.

The input values for y are given in the Q15 fixed-point format. Each component of the vector may have a different number of fractional bits. The number of fractional bits per `y` component is passed in an array of the same length as `y`.

`G` is the equalization complex matrix and is assumed to be a 4-by-4 matrix. I and Q components of `G` elements are assumed to be stored separated in memory.

Also, each coefficient of `G` (`Gxy`, for `x, y = {1, 2, 3, 4}`) is assumed to be stored separated in memory locations set at `p_gstride` one from the other.

It is assumed that each component of the vectors `y` and `x` are stored in memory at `p_y_stride` and `p_x_stride` one from the other. It is also assumed that `p_gstride` is greater than or equal to the number of subcarriers divided by the number of subcarriers per `G`. `p_y_stride` and `p_x_stride` are assumed greater than or equal to the number of subcarriers. If these assumptions are violated, the results returned will be incorrect.

The number of input signals is assumed to be a multiple of 12, and must be divisible by the number of subcarriers per `G` matrix.

For type 1 equalization, the number of subcarriers per `G` matrix must be four. For type 2 equalization, the number of subcarriers per `G` matrix must be six. An implementation is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1454`:

```
armral_status
armral_solve_4x4_f32(uint32_t num_sub_carrier, uint32_t num_sc_per_g,
                     const armral_cmplx_int16_t *p_y, uint32_t p_ystride,
                     const armral_fixed_point_index *p_y_num_fract_bits,
                     const float32_t *p_g_real, const float32_t *p_g_imag,
                     uint32_t p_gstride, armral_cmplx_int16_t *p_x,
                     uint32_t p_xstride,
                     armral_fixed_point_index num_fract_bits_x);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_sub_carrier

A read-only parameter of type `uint32_t`.

The number of subcarriers to equalize.

num_sc_per_g

A read-only parameter of type `uint32_t`.

The number of subcarriers per `G` matrix.

p_y

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input received signal.

p_ystride

A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

p_y_num_fract_bits

A read-only parameter of type `const armral_fixed_point_index *`.

The number of fractional bits in `y`.

p_g_real

A read-only parameter of type `const float32_t *`.

The real part of coefficient matrix `G`.

p_g_imag

A read-only parameter of type `const float32_t *`.

The imag part of coefficient matrix `G`.

p_gstride

A read-only parameter of type `uint32_t`.

The stride between elements of `G`.

p_x

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output received signal.

p_xstride

A read-only parameter of type `uint32_t`.

The stride between two layers.

num_fract_bits_x

A read-only parameter of type `armral_fixed_point_index`.

The number of fractional bits in `x`.

3.2.4.4 armral_solve_1x4_f32

In LTE and 5G, you can use the `armral_solve_1x4_f32` function in the equalization step, as in the formula:

```
x^ = G y
```

where `y` is a vector for the received signal, size corresponds to the number of antennae and `x^` is the estimate of the transmitted signal, size corresponds to the number of layers.

`G` is the equalization complex matrix and is assumed to be a 1-by-4 matrix (i.e. a row vector). I and Q components of `G` elements are assumed to be stored separated in memory.

Also, each coefficient of `G` (`G1y`, for `y = {1, 2, 3, 4}`) is assumed to be stored separated in memory locations set at `p_gstride` one from the other.

The number of input signals is assumed to be a multiple of 12, and must be divisible by the number of subcarriers per `G` matrix. For type 1 equalization, the number of subcarriers per `G` matrix must be four. For type 2 equalization, the number of subcarriers per `G` matrix must be six. An implementation is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1503`:

```
armral_status
armral_solve_1x4_f32(uint32_t num_sub_carrier, uint32_t num_sc_per_g,
                     const armral_cmplx_int16_t *p_y, uint32_t p_ystride,
                     const armral_fixed_point_index *p_y_num_fract_bits,
                     const float32_t *p_g_real, const float32_t *p_g_imag,
                     uint32_t p_gstride, armral_cmplx_int16_t *p_x,
                     armral_fixed_point_index num_fract_bits_x);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_sub_carrier`

A read-only parameter of type `uint32_t`.

The number of subcarriers to equalize.

`num_sc_per_g`

A read-only parameter of type `uint32_t`.

The number of subcarriers per `G` matrix.

p_y

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input received signal.

p_ystride

A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

p_y_num_fract_bits

A read-only parameter of type `const armral_fixed_point_index *`.

The number of fractional bits in `y` conversion.

p_g_real

A read-only parameter of type `const float32_t *`.

The real part of coefficient matrix `G`.

p_g_imag

A read-only parameter of type `const float32_t *`.

The imag part of coefficient matrix `G`.

p_gstride

A read-only parameter of type `uint32_t`.

The stride between elements of `G`.

p_x

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output received signal.

num_fract_bits_x

A read-only parameter of type `armral_fixed_point_index`.

The number of fractional bits in `x`.

3.2.4.5 armral_solve_1x2_f32

In LTE and 5G, you can use the `armral_solve_1x2_f32` function in the equalization step, as in the formula:

```
x^ = G y
```

where `y` is a vector for the received signal, size corresponds to the number of antennae and `x^` is the estimate of the transmitted signal, size corresponds to the number of layers. `G` is the equalization complex matrix and is assumed to be a 1-by-2 matrix (i.e. a row vector). I and Q components of `G` elements are assumed to be stored separated in memory.

Also, each coefficient of `G` (`G11`, `G12`) is assumed to be stored separated in memory locations set at `p_gstride` one from the other.

The number of input signals is assumed to be a multiple of 12, and must be divisible by the number of subcarriers per `G` matrix.

For type 1 equalization, the number of subcarriers per `G` matrix must be four. For type 2 equalization, the number of subcarriers per `G` matrix must be six. An implementation is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1549`:

```
armral_status
armral_solve_1x2_f32(uint32_t num_sub_carrier, uint32_t num_sc_per_g,
                     const armral_cmplx_int16_t *p_y, uint32_t p_ystride,
                     const armral_fixed_point_index *p_y_num_fract_bits,
                     const float32_t *p_g_real, const float32_t *p_g_imag,
                     uint32_t p_gstride, armral_cmplx_int16_t *p_x,
                     armral_fixed_point_index num_fract_bits_x);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_sub_carrier`

A read-only parameter of type `uint32_t`.

The number of subcarriers to equalize.

`num_sc_per_g`

A read-only parameter of type `uint32_t`.

The number of subcarriers per `G` matrix.

`p_y`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input received signal.

`p_ystride`

A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

`p_y_num_fract_bits`

A read-only parameter of type `const armral_fixed_point_index *`.

The number of fractional bits in `y` conversion.

p_g_real

A read-only parameter of type `const float32_t *`.

The real part of coefficient matrix `G`.

p_g_imag

A read-only parameter of type `const float32_t *`.

The imag part of coefficient matrix `G`.

p_gstride

A read-only parameter of type `uint32_t`.

The stride between elements of `G`.

p_x

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output received signal.

num_fract_bits_x

A read-only parameter of type `armral_fixed_point_index`.

The number of fractional bits in `x`.

3.2.5 Complex Matrix Inversion

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

3.2.5.1 armral_cmplx_hermitian_mat_inverse_f32

This function computes the inverse of a single complex Hermitian square matrix of size `N` -by- `N`.

The supported dimensions are 2-by-2, 3-by-3, 4-by-4, 8-by-8, and 16-by-16.

The input and output matrices are filled in row-major order with complex `float32_t` elements.

Syntax

Defined in `armral.h` on line `1582`:

```
armral_status armral_cmplx_hermitian_mat_inverse_f32(
    uint32_t size, const armral_cmplx_f32_t *p_src, armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The size of the input matrix.

p_src

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix structure.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix structure.

3.2.5.2 armral_cmplx_mat_inverse_f32

This function computes the inverse of a single complex square matrix of size `N` -by- `N`.

The supported dimensions are 2-by-2, 3-by-3, 4-by-4, 8-by-8, and 16-by-16.

The input and output matrices are filled in row-major order with complex `float32_t` elements.

Syntax

Defined in `armral.h` on line `1599`:

```
armral_status armral_cmplx_mat_inverse_f32(uint32_t size,
                                           const armral_cmplx_f32_t *p_src,
                                           armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The size of the input matrix.

p_src

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix structure.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix structure.

3.2.5.3 armral_cmplx_hermitian_mat_inverse_batch_f32

This function computes the inverse of a batch of `M` complex Hermitian square matrices, each of size `N` -by- `N`.

The supported matrix dimensions are 2-by-2, 3-by-3, and 4-by-4.

The input and output matrices are filled in row-major order with complex `float32_t` elements, interleaved such that all elements for a particular location within the matrix are stored together. This means that, for instance, the first four complex numbers stored are the first element from each of the first four matrices in the batch. The offset to the next location in the same matrix is given by the `num_mats` batch size:

```
{Re(0), Im(0), Re(1), Im(1), ..., Re(M - 1), Im(M - 1)}
```

The number of matrices in a batch (`M`) must be a multiple of the matrix dimension. So, if `N = 2` then `M` must be a multiple of two, and if `N = 4` then `M` must be a multiple of four.

Syntax

Defined in `armral.h` on line `1628`:

```
armral_status
armral_cmplx_hermitian_mat_inverse_batch_f32(uint32_t num_mats, uint32_t size,
                                             const armral_cmplx_f32_t *p_src,
                                             armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_mats`

A read-only parameter of type `uint32_t`.

The number (`M`) of input and output matrices.

`size`

A read-only parameter of type `uint32_t`.

The size (`N`) of the input and output matrix.

`p_src`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix structure.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix structure.

3.2.5.4 armral_cmplx_mat_inverse_batch_f32

This function computes the inverse of a batch of `M` complex general square matrices, each of size `N` -by- `N`.

The supported matrix dimensions are 2-by-2, 3-by-3, and 4-by-4.

The input and output matrices are filled in row-major order with complex `float32_t` elements, interleaved such that all elements for a particular location within the matrix are stored together. This means that, for instance, the first four complex numbers stored are the first element from each of the first four matrices in the batch. The offset to the next location in the same matrix is given by the `num_mats` batch size:

```
{Re(0), Im(0), Re(1), Im(1), ..., Re(M - 1), Im(M - 1)}
```

Syntax

Defined in `armral.h` on line `1655`:

```
armral_status
armral_cmplx_mat_inverse_batch_f32(uint32_t num_mats, uint32_t size,
                                   const armral_cmplx_f32_t *p_src,
                                   armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_mats

A read-only parameter of type `uint32_t`.

The number (`M`) of input and output matrices.

size

A read-only parameter of type `uint32_t`.

The size (`N`) of the input and output matrix.

p_src

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix structure.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix structure.

3.2.5.5 armral_cmplx_hermitian_mat_inverse_batch_f32_pa

This function computes the inverse of a batch of `M` complex Hermitian square matrices, each of size `N` -by- `N`, utilizing a "pointer array" storage layout for the input and output matrix batches.

The supported matrix dimensions are 2-by-2, 3-by-3, and 4-by-4. The `p_srcs` parameter is an array of pointers of length `N` -by- `N`. The value of `p_srcs[i]` is a pointer to the i-th element of the first matrix in the batch, as represented in row-major ordering, such that the i-th element of the j-th matrix in the batch is located at `p_srcs[i][j]`. Similarly, the j-th matrix in a batch of 2-by-2 matrices is formed as:

```
p_srcs[0][j]  p_srcs[1][j]
p_srcs[2][j]  p_srcs[3][j]
```

The output array `p_dsts` points to an array of pointers, representing an identical format to the input.

The number of matrices in a batch (`M`) must be a multiple of the matrix dimension. So, if `N = 2` then `M` must be a multiple of two, and if `N = 4` then `M` must be a multiple of four.

Syntax

Defined in `armral.h` on line `1688`:

```
armral_status armral_cmplx_hermitian_mat_inverse_batch_f32_pa(
    uint32_t num_mats, uint32_t size, const armral_cmplx_f32_t **p_srcs,
    armral_cmplx_f32_t **p_dsts);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_mats

A read-only parameter of type `uint32_t`.

The number (`M`) of input and output matrices.

size

A read-only parameter of type `uint32_t`.

The size (`N`) of the input and output matrix.

p_srcs

A read-only parameter of type `const armral_cmplx_f32_t **`.

Points to the input matrix structure.

p_dsts

A write-only parameter of type `armral_cmplx_f32_t **`.

Points to the output matrix structure.

3.2.5.6 armral_cmplx_mat_inverse_batch_f32_pa

This function computes the inverse of a batch of `M` complex general square matrices, each of size `N` -by- `N`, utilizing a "pointer array" storage layout for the input and output matrix batches.

The supported matrix dimensions are 2-by-2, 3-by-3, and 4-by-4. The `p_srcs` parameter is an array of pointers of length `N` -by- `N`. The value of `p_srcs[i]` is a pointer to the i-th element of the first matrix in the batch, as represented in row-major ordering, such that the i-th element of the j-th matrix in the batch is located at `p_srcs[i][j]`. Similarly, the j-th matrix in a batch of 2-by-2 matrices is formed as:

```
p_srcs[0][j]  p_srcs[1][j]
p_srcs[2][j]  p_srcs[3][j]
```

The output array `p_dsts` points to an array of pointers, representing an identical format to the input.

Syntax

Defined in `armral.h` on line `1716`:

```
armral_status
armral_cmplx_mat_inverse_batch_f32_pa(uint32_t num_mats, uint32_t size,
                                      const armral_cmplx_f32_t **p_srcs,
                                      armral_cmplx_f32_t **p_dsts);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_mats

A read-only parameter of type `uint32_t`.

The number (`M`) of input and output matrices.

size

A read-only parameter of type `uint32_t`.

The size (`N`) of the input and output matrix.

p_srcs

A read-only parameter of type `const armral_cmplx_f32_t **`.

Points to the input matrix structure.

p_dsts

A write-only parameter of type `armral_cmplx_f32_t **`.

Points to the output matrix structure.

3.2.6 Complex Matrix Pseudo-Inverse

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

3.2.6.1 armral_cmplx_pseudo_inverse_direct_f32

Computes the regularized pseudo-inverse of a single matrix. The `N` -by- `M` regularized pseudo-inverse `C` of an `M` -by- `N` matrix `A` is defined as:

```
C = A^H * (A * A^H + λ * I)^-1
```

for `M <= N`, and is defined as:

```
C = (A^H * A + λ * I)^-1 * A^H
```

for `M > N`.

This function performs numerical matrix inversion using the Schur complement to compute the regularized pseudo-inverse of `A` directly from the appropriate expression.

Warning: This method is numerically unstable for matrices that are not very well conditioned.

The input matrix `p_src` and output matrix `p_dst` are stored contiguously in memory, in row-major order.

Note:

- If `m <= n` the number of rows `m` in the input matrix must be 1, 2, 3, 4, 8 or 16.
- If `m > n` the number of columns `n` in the input matrix must be 1, 2, 3, 4, 8 or 16.

Syntax

Defined in `armral.h` on line `1771`:

```
armral_status
armral_cmplx_pseudo_inverse_direct_f32(uint16_t m, uint16_t n, float32_t lambda,
                                       const armral_cmplx_f32_t *p_src,
                                       armral_cmplx_f32_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows in input matrix `A`.

n

A read-only parameter of type `uint16_t`.

The number of columns in input matrix `A`.

lambda

A read-only parameter of type `float32_t`.

The real scalar quantity `λ`.

p_src

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix `A`.

p_dst

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix `C`.

3.2.6.2 armral_cmplx_pseudo_inverse_direct_f32_noalloc

Non-allocating variant of armral_cmplx_pseudo_inverse_direct_f32 .

Computes the regularized pseudo-inverse of a single matrix. The `N` -by- `M` regularized pseudo-inverse `C` of an `M` -by- `N` matrix `A` is defined as:

```
C = A^H * (A * A^H + λ * I)^-1
```

for `M <= N`, and is defined as:

```
C = (A^H * A + λ * I)^-1 * A^H
```

for `M > N`.

This function performs numerical matrix inversion using the Schur complement to compute the regularized pseudo-inverse of `A` directly from the appropriate expression.

Warning: This method is numerically unstable for matrices that are not very well conditioned.

The input matrix `p_src` and output matrix `p_dst` are stored contiguously in memory, in row-major order.

Note:

- If `m <= n` the number of rows `m` in the input matrix must be 1, 2, 3, 4, 8 or 16.
- If `m > n` the number of columns `n` in the input matrix must be 1, 2, 3, 4, 8 or 16.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

Note:

- If `m <= n` the buffer must be at least `m * m * sizeof(armral_cmplx_f32_t) + 3` bytes.
- If `m > n` the buffer must be at least `n * n * sizeof(armral_cmplx_f32_t) + 3` bytes.

Syntax

Defined in `armral.h` on line `1828`:

```
armral_status armral_cmplx_pseudo_inverse_direct_f32_noalloc(
    uint16_t m, uint16_t n, float32_t lambda, const armral_cmplx_f32_t *p_src,
    armral_cmplx_f32_t *p_dst, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`m`

A read-only parameter of type `uint16_t`.

The number of rows in input matrix `A`.

`n`

A read-only parameter of type `uint16_t`.

The number of columns in input matrix `A`.

`lambda`

A read-only parameter of type `float32_t`.

The real scalar quantity `λ`.

`p_src`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input matrix `A`.

`p_dst`

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output matrix `C`.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.2.7 SVD of a Single Complex Matrix

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

3.2.7.1 armral_svd_cf32

This function performs the Singular Value Decomposition (SVD) of an `M` -by- `N` single complex matrix `A`, where `M ≥ N` and `A` is stored in column-major order. The SVD of `A` is a two-sided decomposition in the form `A = U Σ V^H`, with `U` an `M` -by- `M` single complex orthogonal matrix. Note that we only store the first `N` columns of `U` because there are at most `N` non-zero singular values. `V` is an `N` -by- `N` single complex orthogonal matrix, and `Σ` is an `M` -by- `N` real matrix. Entries `Σ_{i,i}`, `i < n` contain the singular values, and other entries in `Σ` are zero. We only store the singular values, not the full matrix `Σ`. The singular values `Σ_{i,i}` are stored in vector `s` for `0 ≤ i < N`. The matrices `U` and `V^H` are implicitly used in the algorithm, unless parameter `vect` is specified to be true, in which case the left and right singular vectors (respectively) are stored in `U` and `V^H` in column-major order. The singular vectors are therefore stored contiguously in `U`, and are non-contiguous in `V^H`. Note that it is `V^H` that is returned, not `V`.

There are different algorithms for an efficient SVD. The most appropriate is automatically selected depending on the size of the input matrix.

Syntax

Defined in `armral.h` on line `4880`:

```
armral_status armral_svd_cf32(bool vect, uint32_t m, uint32_t n,
                              armral_cmplx_f32_t *a, float32_t *s,
                              armral_cmplx_f32_t *u, armral_cmplx_f32_t *vt);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

vect

A read-only parameter of type `bool`.

If true, both the singular values and the singular vectors are computed, else only the singular values are computed.

m

A read-only parameter of type `uint32_t`.

The number of rows (`M`) in matrix `A`.

n

A read-only parameter of type `uint32_t`.

The number of columns (`N`) in matrix `A`.

a

A parameter of type `armral_cmplx_f32_t *`.

On entry contains the `M` -by- `N` matrix on which to perform the SVD. On exit contains the Householder reflectors used to perform the bidiagonalization of `A`.

s

A write-only parameter of type `float32_t *`.

The vector of singular values.

u

A write-only parameter of type `armral_cmplx_f32_t *`.

The left singular vectors, if required. If `vect` is true, `u` is populated with the left singular vectors in the SVD. Sufficient memory to store `M N float32_t` values is assumed to have been allocated before the call to this method.

vt

A write-only parameter of type `armral_cmplx_f32_t *`.

The right singular vectors, if required. If `vect` is true, `vt` is populated with the right singular vectors in the SVD. Sufficient memory to store `N N float32_t` values is assumed to have been allocated before the call to this method.

3.2.7.2 armral_svd_cf32_noalloc

Non-allocating variant of armral_svd_cf32.

This function performs the Singular Value Decomposition (SVD) of an `M` -by- `N` single complex matrix `A`, where `M ≥ N` and `A` is stored in column-major order. The SVD of `A` is a two-sided decomposition in the form `A = U Σ V^H`, with `U` an `M` -by- `M` single complex orthogonal matrix. Note that we only store the first `N` columns of `U` because there are at most `N` non-zero singular values. `V` is an `N` -by- `N` single complex orthogonal matrix, and `Σ` is an `M` -by- `N` real matrix. Entries `Σ_{i,i}`, `i < n` contain the singular values, and other entries in `Σ` are zero. We only store the singular values, not the full matrix `Σ`. The singular values `Σ_{i,i}` are stored in vector `s` for `0 ≤ i < N`. The matrices `U` and `V^H` are implicitly used in the algorithm, unless parameter `vect` is specified to be true, in which case the left and right singular vectors (respectively) are stored in `U` and `V^H` in column-major order. The singular vectors are therefore stored contiguously in `U`, and are non-contiguous in `V^H`. Note that it is `V^H` that is returned, not `V`.

There are different algorithms for an efficient SVD. The most appropriate is automatically selected depending on the size of the input matrix.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_svd_cf32_noalloc_buffer_size with identical inputs.

Syntax

Defined in `armral.h` on line `4940`:

```
armral_status armral_svd_cf32_noalloc(bool vect, uint32_t m, uint32_t n,
                                      armral_cmplx_f32_t *a, float32_t *s,
                                      armral_cmplx_f32_t *u,
                                      armral_cmplx_f32_t *vt, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`vect`

A read-only parameter of type `bool`.

If true, both the singular values and the singular vectors are computed, else only the singular values are computed.

`m`

A read-only parameter of type `uint32_t`.

The number of rows (`M`) in matrix `A`.

`n`

A read-only parameter of type `uint32_t`.

The number of columns (`N`) in matrix `A`.

`a`

A parameter of type `armral_cmplx_f32_t *`.

On entry contains the `M` -by- `N` matrix on which to perform the SVD. On exit contains the Householder reflectors used to perform the bidiagonalization of `A`.

`s`

A write-only parameter of type `float32_t *`.

The vector of singular values.

`u`

A write-only parameter of type `armral_cmplx_f32_t *`.

The left singular vectors, if required. If `vect` is true, `u` is populated with the left singular vectors in the SVD. Sufficient memory to store `M N float32_t` values is assumed to have been allocated before the call to this method.

vt

A write-only parameter of type `armral_cmplx_f32_t *`.

The right singular vectors, if required. If `vect` is true, `vt` is populated with the right singular vectors in the SVD. Sufficient memory to store `N N float32_t` values is assumed to have been allocated before the call to this method.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.2.7.3 armral_svd_cf32_noalloc_buffer_size

Calculates the required buffer size in bytes needed to perform a singular value decomposition (SVD) of an `M` -by- `N` input matrix `A`.

Syntax

Defined in `armral.h` on line `4956`:

```
uint32_t armral_svd_cf32_noalloc_buffer_size(bool vect, uint32_t m, uint32_t n);
```

Returns

The required buffer size in bytes.

Parameters

vect

A read-only parameter of type `bool`.

If true, both the singular values and the singular vectors are computed, else only the singular values are computed.

m

A read-only parameter of type `uint32_t`.

The number of rows (`M`) in matrix `A`.

n

A read-only parameter of type `uint32_t`.

The number of columns (`N`) in matrix `A`.

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.

3.3.1.1 armral_seq_generator

This function generates a pseudo-random sequence (Gold Sequence) that is used in 4G and 5G networks to scramble data of a specific channel or to generate a specific sequence (for example for Downlink Reference Signal generation).

The sequence generator is the same generator that is described in the 3GPP Technical Specification (TS) 36.211, Chapter 7.2. The generator uses two polynomials, `x1` and `x2`, defined as:

```
x1(n+31) = (x1(n+3) + x1(n)) mod 2
x2(n+31) = (x2(n+3) + x2(n+2) + x2(n+1) + x2(n)) mod 2
```

to generate the output sequence:

```
c(n) = (x1(n+Nc) + x2(n+Nc)) mod 2
```

where `Nc` is a constant with a value of 1600. The initialization for `x1` and `x2` satisfies the condition that:

```
x1(0) = 1
x1(i) = 0              for i=1,2,...,30
x2(i) = cinit(i) >> i   for i=0,1,...,30
```

The `cinit` parameter is provided as an input parameter for the algorithm, which is used to derive `x2`. The algorithm generates `x1` and `x2` and skips the first 1600 bits.

Syntax

Defined in `armral.h` on line `1885`:

```
armral_status armral_seq_generator(uint32_t sequence_len, uint32_t seed,
                                   uint8_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`sequence_len`

A read-only parameter of type `uint32_t`.

The length of the sequence in bits (cinit).

`seed`

A read-only parameter of type `uint32_t`.

The random sequence starting point.

`p_dst`

A write-only parameter of type `uint8_t *`.

Points to the output bits.

3.3.2 Correlation Coefficient

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

3.3.2.1 armral_corr_coeff_i16

Calculates Pearson's Correlation Coefficient from a pair of vectors of complex numbers in Q15 format with real component and imaginary component interleaved, with the result stored to a pointer to a single complex number.

Pearson's correlation coefficient is calculated using:

```
           SUM(x*conj(y)) - n*avg(x)*avg(y)
Rxy = ----------------------------------------------
      SQRT(SUM(x*conj(x)) - n*avg(x)*conj(avg(x)))
      * SQRT(SUM(y*conj(y)) - n*avg(y)*conj(avg(y)))
```

Warning: n must be less than or equal to INT32_MAX, the largest number representable in a 32-bit signed integer.

Syntax

Defined in `armral.h` on line `2004`:

```
armral_status armral_corr_coeff_i16(uint32_t n,
                                    const armral_cmplx_int16_t *p_src_a,
                                    const armral_cmplx_int16_t *p_src_b,
                                    armral_cmplx_int16_t *c);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of complex samples in each vector.

p_src_a

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the first input vector.

p_src_b

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the second input vector.

c

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the result.

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.

3.3.3.1 armral_fir_filter_cf32

Computes a complex floating-point single-precision FIR filter.

The `size` parameter, which is the length of the input array, must be a multiple of four. Both the input array and the coefficients array must be padded with zeros up to the next multiple of four.

Syntax

Defined in `armral.h` on line `2048`:

```
armral_status armral_fir_filter_cf32(uint32_t size, uint32_t taps,
                                     const armral_cmplx_f32_t *input,
                                     const armral_cmplx_f32_t *coeffs,
                                     armral_cmplx_f32_t *output);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of complex samples in input.

`taps`

A read-only parameter of type `uint32_t`.

The number of taps of the FIR filter.

`input`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input samples buffer.

`coeffs`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the coefficients array.

`output`

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output array.

3.3.3.2 armral_fir_filter_cf32_decimate_2

Computes a complex floating-point single-precision FIR filter with a decimation factor of two.

The `size` parameter, which is the length of the input array before decimation, must be a multiple of eight. The input array must be padded with zeros up to the next multiple of eight, and the coefficients array must be padded with zeros up to the next multiple of four.

Syntax

Defined in `armral.h` on line `2070`:

```
armral_status armral_fir_filter_cf32_decimate_2(
    uint32_t size, uint32_t taps, const armral_cmplx_f32_t *input,
    const armral_cmplx_f32_t *coeffs, armral_cmplx_f32_t *output);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of complex samples in input.

`taps`

A read-only parameter of type `uint32_t`.

The number of taps of the FIR filter.

`input`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the input samples buffer.

`coeffs`

A read-only parameter of type `const armral_cmplx_f32_t *`.

Points to the coefficients array.

`output`

A write-only parameter of type `armral_cmplx_f32_t *`.

Points to the output array.

3.3.3.3 armral_fir_filter_cs16

Computes a complex signed 16-bit integer FIR filter.

The `size` parameter, which is the length of the input array, must be a multiple of eight. Both the input array and the coefficients array must be padded with zeros up to the next multiple of eight.

Syntax

Defined in `armral.h` on line `2088`:

```
armral_status armral_fir_filter_cs16(uint32_t size, uint32_t taps,
                                     const armral_cmplx_int16_t *input,
                                     const armral_cmplx_int16_t *coeffs,
                                     armral_cmplx_int16_t *output);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of complex samples in input.

`taps`

A read-only parameter of type `uint32_t`.

The number of taps of the FIR filter.

`input`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input samples buffer.

`coeffs`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the coefficients array.

`output`

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output array.

3.3.3.4 armral_fir_filter_cs16_decimate_2

Computes a complex signed 16-bit integer FIR filter with a decimation factor of two.

The `size` parameter, which is the length of the input array before decimation, must be a multiple of eight. The input array must be padded with zeros up to the next multiple of eight, and the coefficients array must be padded with zeros up to the next multiple of four.

Syntax

Defined in `armral.h` on line `2110`:

```
armral_status armral_fir_filter_cs16_decimate_2(
    uint32_t size, uint32_t taps, const armral_cmplx_int16_t *input,
    const armral_cmplx_int16_t *coeffs, armral_cmplx_int16_t *output);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of complex samples in input.

`taps`

A read-only parameter of type `uint32_t`.

The number of taps of the FIR filter.

`input`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input samples buffer.

`coeffs`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the coefficients array.

`output`

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output array.

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 one of

- armral_fft_create_plan_T
- armral_fft_create_2d_plan_T

where T is the data type of the input and output arrays. T can be one of cf32 or cs16. cf32 means complex single precision floating point. cs16 means complex fixed point in Q15 format.

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);
```

3.3.4.1 armral_fft_create_plan_cf32

Creates a 1-dimensional fp32 FFT plan for complex data.

Fills the passed pointer with a pointer to the plan that is created. The plan that is created can then be used to solve problems with specified size and direction. It is efficient to create plans once and reuse them, rather than creating a plan for every execute call. For some inputs, creating FFT plans can incur a significant overhead.

To avoid memory leaks, call armral_fft_destroy_plan_cf32 when you no longer need this plan.

Note: This function supports input sizes `n <= 42012`.

Syntax

Defined in `armral.h` on line `3236`:

```
armral_status armral_fft_create_plan_cf32(armral_fft_plan_t **p, int n,
                                          armral_fft_direction_t dir);
```

Returns

An `armral_status` value that indicates success or failure

Parameters

p

A write-only parameter of type `armral_fft_plan_t **`.

A pointer to the resulting plan pointer. On output `*p` is a valid pointer, to be passed to armral_fft_execute_cf32.

n

A read-only parameter of type `int`.

The problem size to be solved by this FFT plan.

dir

A read-only parameter of type `armral_fft_direction_t`.

The direction to be solved by this FFT plan. Either `ARMRAL_FFT_FORWARDS` or `ARMRAL_FFT_BACKWARDS`.

3.3.4.2 armral_fft_create_2d_plan_cf32

Creates a 2-dimensional fp32 FFT plan for complex data.

Fills the passed pointer with a pointer to the plan that is created. The plan can then be used to solve 2-d problems with specified dimensions and direction. It is efficient to create plans once and reuse them, rather than creating a plan for every execute call. For some inputs, creating FFT plans can incur a significant overhead.

The input and output arrays passed to armral_fft_execute_cf32 should be in row-major format, i.e. the dimension of length n1 is contiguous, and the dimension of length n0 has a stride between successive elements of n1.

To avoid memory leaks, call armral_fft_destroy_plan_cf32 when the plan is no longer needed.

Syntax

Defined in `armral.h` on line `3264`:

```
armral_status armral_fft_create_2d_plan_cf32(armral_fft_plan_t **p, int n0,
                                             int n1,
                                             armral_fft_direction_t dir);
```

Returns

An `armral_status` value that indicates success or failure

Parameters

p

A write-only parameter of type `armral_fft_plan_t **`.

A pointer to the resulting plan pointer. On output `*p` is a valid pointer, to be passed to armral_fft_execute_cf32.

n0

A read-only parameter of type `int`.

The size of the first dimension.

n1

A read-only parameter of type `int`.

The size of the second dimension.

dir

A read-only parameter of type `armral_fft_direction_t`.

The direction to be solved by this FFT plan. Either `ARMRAL_FFT_FORWARDS` or `ARMRAL_FFT_BACKWARDS`.

3.3.4.3 armral_fft_execute_cf32

Performs a single FFT using the specified plan and arrays.

Uses the plan created by armral_fft_create_plan_cf32 or armral_fft_create_2d_plan_cf32 to perform the configured FFT with the arrays that are specified.

Syntax

Defined in `armral.h` on line `3282`:

```
armral_status armral_fft_execute_cf32(const armral_fft_plan_t *p,
                                      const armral_cmplx_f32_t *x,
                                      armral_cmplx_f32_t *y);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

A read-only parameter of type `const armral_fft_plan_t *`.

A pointer to the FFT plan.

x

A read-only parameter of type `const armral_cmplx_f32_t *`.

The input array for this FFT. The array must contain `n` elements for 1-d plans, and `n0 * n1` for 2-d plans.

y

A write-only parameter of type `armral_cmplx_f32_t *`.

The output array for this FFT. The array must contain `n` elements for 1-d plans, and `n0 * n1` for 2-d plans.

3.3.4.4 armral_fft_destroy_plan_cf32

Destroys an FFT plan.

The armral_fft_execute_cf32 function frees any associated memory, and sets `*p = NULL`, for a plan that was previously created by armral_fft_create_plan_cf32 or armral_fft_create_2d_plan_cf32.

Syntax

Defined in `armral.h` on line `3297`:

```
armral_status armral_fft_destroy_plan_cf32(armral_fft_plan_t **p);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

A parameter of type `armral_fft_plan_t **`.

A pointer to the FFT plan pointer. On function exit, `*p` is set to `NULL` .

3.3.4.5 armral_fft_create_plan_cs16

Creates a 1-dimensional int16 (Q15 format) FFT plan for complex data.

Fills the passed pointer with a pointer to the plan that is created. The plan that is created can then be used to solve problems with specified size and direction. It is efficient to create plans once and reuse them, rather than creating a plan for every execute call. For some inputs, creating FFT plans can incur a significant overhead.

To avoid memory leaks, call armral_fft_destroy_plan_cs16 when you no longer need this plan.

Note: This function supports input sizes `n <= 42012`.

Syntax

Defined in `armral.h` on line `3321`:

```
armral_status armral_fft_create_plan_cs16(armral_fft_plan_t **p, int n,
                                          armral_fft_direction_t dir);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

A write-only parameter of type `armral_fft_plan_t **`.

A pointer to the resulting plan pointer. On output `*p` is a valid pointer, to be passed to armral_fft_execute_cs16.

n

A read-only parameter of type `int`.

The problem size to be solved by this FFT plan.

dir

A read-only parameter of type `armral_fft_direction_t`.

The direction to be solved by this FFT plan. Either `ARMRAL_FFT_FORWARDS` or `ARMRAL_FFT_BACKWARDS`.

3.3.4.6 armral_fft_create_2d_plan_cs16

Creates a 2-dimensional int16 (Q15 format) FFT plan for complex data.

Fills the passed pointer with a pointer to the plan that is created. The plan can then be used to solve 2-d problems with specified dimensions and direction. It is efficient to create plans once and reuse them, rather than creating a plan for every execute call. For some inputs, creating FFT plans can incur a significant overhead.

The input and output arrays passed to armral_fft_execute_cs16 should be in row-major format, i.e. the dimension of length n1 is contiguous, and the dimension of length n0 has a stride between successive elements of n1.

To avoid memory leaks, call armral_fft_destroy_plan_cs16 when you no longer need this plan.

Syntax

Defined in `armral.h` on line `3349`:

```
armral_status armral_fft_create_2d_plan_cs16(armral_fft_plan_t **p, int n0,
                                             int n1,
                                             armral_fft_direction_t dir);
```

Returns

An `armral_status` value that indicates success or failure

Parameters

p

A write-only parameter of type `armral_fft_plan_t **`.

A pointer to the resulting plan pointer. On output `*p` is a valid pointer, to be passed to armral_fft_execute_cf32.

n0

A read-only parameter of type `int`.

The size of the first dimension.

n1

A read-only parameter of type `int`.

The size of the second dimension.

dir

A read-only parameter of type `armral_fft_direction_t`.

The direction to be solved by this FFT plan. Either `ARMRAL_FFT_FORWARDS` or `ARMRAL_FFT_BACKWARDS`.

3.3.4.7 armral_fft_execute_cs16

Performs a single FFT using the specified plan and arrays.

Uses the plan created by armral_fft_create_plan_cs16 or armral_fft_create_2d_plan_cs16 to perform the configured FFT with the arrays that are specified.

Syntax

Defined in `armral.h` on line `3367`:

```
armral_status armral_fft_execute_cs16(const armral_fft_plan_t *p,
                                      const armral_cmplx_int16_t *x,
                                      armral_cmplx_int16_t *y);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

A read-only parameter of type `const armral_fft_plan_t *`.

A pointer to the FFT plan.

x

A read-only parameter of type `const armral_cmplx_int16_t *`.

The input array for this FFT. The array must contain `n` elements for 1-d plans, and `n0 * n1` for 2-d plans.

y

A write-only parameter of type `armral_cmplx_int16_t *`.

The output array for this FFT. The array must contain `n` elements for 1-d plans, and `n0 * n1` for 2-d plans.

3.3.4.8 armral_fft_destroy_plan_cs16

Destroys an FFT plan.

The armral_fft_execute_cs16 function frees any associated memory, and sets `*p = NULL`, for a plan that was previously created by armral_fft_create_plan_cs16 or armral_fft_create_2d_plan_cs16.

Syntax

Defined in `armral.h` on line `3382`:

```
armral_status armral_fft_destroy_plan_cs16(armral_fft_plan_t **p);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

A parameter of type `armral_fft_plan_t **`.

A pointer to the FFT plan pointer. On function exit, `*p` is set to `NULL` .

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.

3.3.5.1 armral_scramble_code_block

This function generates a block of scrambled bits using a pseudo-random sequence according to the scrambler described in the 3GPP Technical Specification (TS) 38.211. For a codeword `b` with length `M` transmitted on the physical channel, the block of bits `b(0), ..., b(M - 1)` is scrambled according to:

```
s(i) = (b(i) + c(i)) mod 2
```

where `s(0), ..., s(M - 1)` are the scrambled bits and `c` is a pseudo-random scrambling sequence defined by a length-31 Gold sequence. Note that this function cannot be used to scramble a transport block, as defined in TS 38.212 section 7.1.2.

Syntax

Defined in `armral.h` on line `5002`:

```
armral_status armral_scramble_code_block(const uint8_t *src, const uint8_t *seq,
                                         uint32_t num_bits, uint8_t *dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

src

A read-only parameter of type `const uint8_t *`.

Points to the array of input bits.

seq

A read-only parameter of type `const uint8_t *`.

Points to the pseudo-random sequence. This is assumed to be a Gold sequence. The Gold sequence generator armral_seq_generator can be used to generate this.

num_bits

A read-only parameter of type `uint32_t`.

The number of input bits.

dst

A read-only parameter of type `uint8_t *`.

Points to the array of output bits. This contains enough bytes to store `num_bits` bits.

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).

3.4.1.1 armral_modulation

Performs modulation of a bitstream, outputs a series of Q2.13 fixed-point complex symbols.

The expected size of `p_dst` depends on the modulation type being used: QPSK, 16QAM, 64QAM, and 256QAM consume two, four, six, and eight bits per symbol, respectively.

Syntax

Defined in `armral.h` on line `1923`:

```
armral_status armral_modulation(uint32_t nbits, armral_modulation_type mod_type,
                                const uint8_t *p_src,
                                armral_cmplx_int16_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`nbits`

A read-only parameter of type `uint32_t`.

The number of input modulated bits.

mod_type

A read-only parameter of type `armral_modulation_type`.

The type of modulation to perform.

p_src

A read-only parameter of type `const uint8_t *`.

Points to input bit flow.

p_dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to output complex symbols (format Q2.13).

3.4.1.2 armral_demodulation

This function implements the soft-demodulation (or soft bit demapping) for QPSK, 16QAM, 64QAM, and 256QAM constellations.

For complex symbols `x_i`, the input sequence is assumed to be made of complex symbols `rx = rx_re + j * rx_im`, whose components I and Q are 16 bits each (format Q2.13) and in an interleaved form:

```
{Re(0), Im(0), Re(1), Im(1), ..., Re(N - 1), Im(N - 1)}
```

The output of the soft-demodulation algorithm is a sequence of log-likelihood ratio (LLR) `int8_t` values, which indicate the relative confidence of the demapping decision, component by component, instead of taking a hard decision and giving the bit value itself.

The LLRs are computed using a maximum likelihood approach. The LLR calculations use a threshold method to approximate the maximum likelihood. This reduces the time complexity required for the demodulation, and gives good estimates of the maximum likelihood when the noise is low or moderate on a channel. In order to keep the LLRs in a range of `int8_t`, scaling can be applied with the use of a unit of least precision (`ulp`).

All the constellation mappings follow those defined in the 3GPP Technical Specification (TS) 38.211 V15.2.0, Chapter 5.1.

Syntax

Defined in `armral.h` on line `1965`:

```
armral_status armral_demodulation(uint32_t n_symbols, uint16_t ulp,
                                  armral_modulation_type mod_type,
                                  const armral_cmplx_int16_t *p_src,
                                  int8_t *p_dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`n_symbols`

A read-only parameter of type `uint32_t`.

The number of complex symbols in the input.

`ulp`

A read-only parameter of type `uint16_t`.

The change in input amplitude corresponding to a unit change in the output LLRs (format Q2.13). The integer representation of `ulp` must lie in the range [2, 2^15].

`mod_type`

A read-only parameter of type `armral_modulation_type`.

The modulation type.

`p_src`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to input complex source (format Q2.13).

`p_dst`

A write-only parameter of type `int8_t *`.

Points to the output byte seq.

3.4.2 Cyclic Redundancy Check (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.

3.4.2.1 armral_crc24_a_le

Computes the CRC24 of an input buffer using the `CRC24A` polynomial. Blocks of 64 bits are interpreted using little-endian ordering.

Syntax

Defined in `armral.h` on line `2618`:

```
armral_status armral_crc24_a_le(uint32_t size, const uint64_t *input,
                                uint64_t *crc24);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc24`

A write-only parameter of type `uint64_t *`.

The computed 24-bit CRC result.

3.4.2.2 armral_crc24_a_be

Computes the CRC24 of an input buffer using the `CRC24A` polynomial. Blocks of 64 bits are interpreted using big-endian ordering.

Syntax

Defined in `armral.h` on line `2630`:

```
armral_status armral_crc24_a_be(uint32_t size, const uint64_t *input,
                                uint64_t *crc24);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

crc24

A write-only parameter of type `uint64_t *`.

The computed 24-bit CRC result.

3.4.2.3 armral_crc24_b_le

Computes the CRC24 of an input buffer using the `CRC24B` polynomial. Blocks of 64 bits are interpreted using little-endian ordering.

Syntax

Defined in `armral.h` on line `2642`:

```
armral_status armral_crc24_b_le(uint32_t size, const uint64_t *input,
                                uint64_t *crc24);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

crc24

A write-only parameter of type `uint64_t *`.

The computed 24-bit CRC result.

3.4.2.4 armral_crc24_b_be

Computes the CRC24 of an input buffer using the `CRC24B` polynomial. Blocks of 64 bits are interpreted using big-endian ordering.

Syntax

Defined in `armral.h` on line `2654`:

```
armral_status armral_crc24_b_be(uint32_t size, const uint64_t *input,
                                uint64_t *crc24);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc24`

A write-only parameter of type `uint64_t *`.

The computed 24-bit CRC result.

3.4.2.5 armral_crc24_c_le

Computes the CRC24 of an input buffer using the `CRC24C` polynomial. Blocks of 64 bits are interpreted using little-endian ordering.

Syntax

Defined in `armral.h` on line `2666`:

```
armral_status armral_crc24_c_le(uint32_t size, const uint64_t *input,
                                uint64_t *crc24);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc24`

A write-only parameter of type `uint64_t *`.

The computed 24-bit CRC result.

3.4.2.6 armral_crc24_c_be

Computes the CRC24 of an input buffer using the `CRC24C` polynomial. Blocks of 64 bits are interpreted using big-endian ordering.

Syntax

Defined in `armral.h` on line `2678`:

```
armral_status armral_crc24_c_be(uint32_t size, const uint64_t *input,
                                uint64_t *crc24);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc24`

A write-only parameter of type `uint64_t *`.

The computed 24-bit CRC result.

3.4.2.7 armral_crc16_le

Computes the CRC16 of an input buffer using the `CRC16` polynomial. Blocks of 64 bits are interpreted using little-endian ordering.

Syntax

Defined in `armral.h` on line `2690`:

```
armral_status armral_crc16_le(uint32_t size, const uint64_t *input,
                              uint64_t *crc16);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc16`

A write-only parameter of type `uint64_t *`.

The computed 16-bit CRC result.

3.4.2.8 armral_crc16_be

Computes the CRC16 of an input buffer using the `CRC16` polynomial. Blocks of 64 bits are interpreted using big-endian ordering.

Syntax

Defined in `armral.h` on line `2702`:

```
armral_status armral_crc16_be(uint32_t size, const uint64_t *input,
                              uint64_t *crc16);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc16`

A write-only parameter of type `uint64_t *`.

The computed 16-bit CRC result.

3.4.2.9 armral_crc11_le

Computes the CRC11 of an input buffer using the `CRC11` polynomial. Blocks of 64 bits are interpreted using little-endian ordering.

Syntax

Defined in `armral.h` on line `2714`:

```
armral_status armral_crc11_le(uint32_t size, const uint64_t *input,
                              uint64_t *crc11);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc11`

A write-only parameter of type `uint64_t *`.

The computed 11-bit CRC result.

3.4.2.10 armral_crc11_be

Computes the CRC11 of an input buffer using the `CRC11` polynomial. Blocks of 64 bits are interpreted using big-endian ordering.

Syntax

Defined in `armral.h` on line `2726`:

```
armral_status armral_crc11_be(uint32_t size, const uint64_t *input,
                              uint64_t *crc11);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc11`

A write-only parameter of type `uint64_t *`.

The computed 11-bit CRC result.

3.4.2.11 armral_crc6_le

Computes the CRC6 of an input buffer using the `CRC6` polynomial. Blocks of 64 bits are interpreted using little-endian ordering.

Syntax

Defined in `armral.h` on line `2738`:

```
armral_status armral_crc6_le(uint32_t size, const uint64_t *input,
                             uint64_t *crc6);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc6`

A write-only parameter of type `uint64_t *`.

The computed 6-bit CRC result.

3.4.2.12 armral_crc6_be

Computes the CRC6 of an input buffer using the `CRC6` polynomial. Blocks of 64 bits are interpreted using big-endian ordering.

Syntax

Defined in `armral.h` on line `2750`:

```
armral_status armral_crc6_be(uint32_t size, const uint64_t *input,
                             uint64_t *crc6);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

`input`

A read-only parameter of type `const uint64_t *`.

Points to the input byte sequence.

`crc6`

A write-only parameter of type `uint64_t *`.

The computed 6-bit CRC result.

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).

3.4.3.1 armral_polar_frozen_mask

Computes the `frozen` bits mask used for encoding and decoding a Polar code.

The mask is formatted as an array of `uint8_t` elements, where each byte element describes the corresponding bit index in the Polar-encoded message. After armral_polar_subchannel_interleave, the value of each bit in the interleaved message is set based on the corresponding byte index of the `frozen` mask. The exact behavior of possible values in the `frozen` mask is described by `armral_polar_frozen_bit_type`.

The `armral_polar_frozen_mask` function takes both the number of information bits and the number of parity bits separately, because the number of parity bits does not depend exactly on `K` or `E`, but also depends on if you are coding for the uplink or downlink. The downlink always has zero parity bits.

The values of the input parameters must satisfy `K + n_pc < N` and satisfy `K + n_pc < E`. The possible values of `n_pc` and `n_pc_wm` are described in section 6.3.1.3.1 of the 3GPP Technical Specification (TS) 38.212: `n_pc` must be either 0 or 3, `n_pc_wm` must be either 0 or 1, and `n_pc >= n_pc_wm` must also be true.

Syntax

Defined in `armral.h` on line `2824`:

```
armral_status armral_polar_frozen_mask(uint32_t n, uint32_t e, uint32_t k,
                                       uint32_t n_pc, uint32_t n_pc_wm,
                                       uint8_t *frozen);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`n`

A read-only parameter of type `uint32_t`.

The Polar code length in bits, must be a power of 2.

`e`

A read-only parameter of type `uint32_t`.

The encoded code length in bits, after rate-matching (either shortening, puncturing or repetition).

`k`

A read-only parameter of type `uint32_t`.

The number of information bits in the encoded message, the sum of the message and CRC bits (`K = A + L`).

`n_pc`

A read-only parameter of type `uint32_t`.

The number of parity bits in the encoded message.

`n_pc_wm`

A read-only parameter of type `uint32_t`.

The number of row-weight-selected parity bits in the encoded message. Must be either zero or one.

`frozen`

A write-only parameter of type `uint8_t *`.

The output `frozen` mask, length `n` bytes. As described by `armral_polar_frozen_bit_type`, elements corresponding to `frozen` bits are set to `0xFF`, elements corresponding to parity bits are set to `0x01`, and elements corresponding to information bits are set to `0x00` .

3.4.3.2 armral_polar_subchannel_interleave

The `armral_polar_subchannel_interleave` function performs subchannel allocation. To calculate the `u` bit array, as specified in section 5.3.1.2 of the 3GPP Technical Specification (TS) 38.212, the function interleaves the supplied input bit array `c` into a larger output bit array. `c` interleaves into positions where the `frozen` mask indicates an information bit is present.

For a particular underlying Polar code of length `N` bits (`N` must be a power of two between 32 and 1024 inclusive), the `frozen` mask must be an array of length `N` bytes. By the nature of Polar coding, `K' ≤ N` must be true.

Syntax

Defined in `armral.h` on line `2851`:

```
armral_status armral_polar_subchannel_interleave(uint32_t n, uint32_t kplus,
                                                 const uint8_t *frozen,
                                                 const uint8_t *c, uint8_t *u);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The Polar code size `N`.

kplus

A read-only parameter of type `uint32_t`.

The number of information bits plus the number of parity bits: `K' = K + n_pc`.

frozen

A read-only parameter of type `const uint8_t *`.

Points to the `frozen` bits mask given by armral_polar_frozen_mask.

c

A read-only parameter of type `const uint8_t *`.

The input codeword, of length `K` bits.

u

A write-only parameter of type `uint8_t *`.

The output codeword including `frozen` and parity bits, of length `N` bits.

3.4.3.3 armral_polar_subchannel_deinterleave

The `armral_polar_subchannel_deinterleave` function performs the inverse of subchannel allocation. To calculate the `c` bit array, as specified in section 5.3.1.2 of the 3GPP Technical Specification (TS) 38.212, the function deinterleaves the supplied input bit array `u` into a smaller output bit array. Bits stored in `u` are taken from `c` at indices where the `frozen` mask indicates an information bit is present. The bits at the remaining `frozen` mask bit indices are ignored.

For a particular underlying Polar code of length `N` bits (`N` must be a power of two between 32 and 1024 inclusive), the `frozen` mask must be an array of length `N` bytes. By the nature of Polar coding, `K ≤ N` must be true.

Syntax

Defined in `armral.h` on line `2878`:

```
armral_status armral_polar_subchannel_deinterleave(uint32_t k,
                                                   const uint8_t *frozen,
                                                   const uint8_t *u,
                                                   uint8_t *c);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

k

A read-only parameter of type `uint32_t`.

The number of information bits, not including the number of parity bits.

frozen

A read-only parameter of type `const uint8_t *`.

Points to the `frozen` bits mask given by armral_polar_frozen_mask.

u

A read-only parameter of type `const uint8_t *`.

The input decoded codeword, including `frozen` and parity bits, of length `N` bits.

c

A write-only parameter of type `uint8_t *`.

The output codeword, of length `K` bits.

3.4.3.4 armral_polar_encode_block

Encodes the specified sequence of `n` input bits using Polar encoding.

Syntax

Defined in `armral.h` on line `2894`:

```
armral_status armral_polar_encode_block(uint32_t n, const uint8_t *p_u_seq_in,
                                        uint8_t *p_d_seq_out);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The Polar code length in bits, where `n` must be a power of 2.

p_u_seq_in

A read-only parameter of type `const uint8_t *`.

Points to the input sequence `[u]` of bits `[u(0), u(1), ..., u(N-1)]`.

p_d_seq_out

A write-only parameter of type `uint8_t *`.

Points to the output encoded sequence `[d]` of bits `[d(0), d(1), ..., d(N-1)]`.

3.4.3.5 armral_polar_decode_block

Decodes `k` real information bits from a Polar-encoded message of length `n` , given as input as a sequence of 8-bit log-likelihood ratios. The number of information bits `k` itself is not needed for the `armral_polar_decode_block` function itself, since computing the `frozen` bits mask is handled elsewhere in armral_polar_frozen_mask.

If `l=1`, the decoder uses a Successive Cancellation (SC) method. If `l>1` , the decoder uses a Successive Cancellation List (SCL) method instead. `l` candidate codewords are maintained and returned, sorted by worsening path metric (in other words, the first returned value is the most likely to be correct). List sizes of 1, 2, 4 and 8 are supported. Unsupported values of `n` or `l` will return `ARMRAL_ARGUMENT_ERROR`.

Syntax

Defined in `armral.h` on line `2920`:

```
armral_status armral_polar_decode_block(uint32_t n, const uint8_t *frozen,
                                        uint32_t l, const int8_t *p_llr_in,
                                        uint8_t *p_u_seq_out);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The Polar code length in bits, must be a power of 2

frozen

A read-only parameter of type `const uint8_t *`.

Points to the `frozen` bits mask given by armral_polar_frozen_mask.

l

A read-only parameter of type `uint32_t`.

The list size to be used in decoding.

p_llr_in

A read-only parameter of type `const int8_t *`.

Points to the input sequence of LLR bytes.

p_u_seq_out

A write-only parameter of type `uint8_t *`.

Points to `l` decoded sequences, ordered by decreasing path metric, each of length `n` bits.

3.4.3.6 armral_polar_rate_matching

Matches the rate of the Polar encoded code block to the rate of the channel using sub-block interleaving, bit selection, and channel interleaving based on Downlink or Uplink direction. This is as described in the 3GPP Technical Specification (TS) 38.212 section 5.4.1.

The code rate of the code block is defined by the ratio of the rate-matched length `e` to the number of information bits in the message `k`. It is assumed that `e` is strictly greater than `k`. Given a rate-matched length and number of information bits, the code block length is determined as described in section 5.3.1 of TS 38.212.

Syntax

Defined in `armral.h` on line `2947`:

```
armral_status armral_polar_rate_matching(uint32_t n, uint32_t e, uint32_t k,
                                         armral_polar_ibil_type i_bil,
                                         const uint8_t *p_d_seq_in,
                                         uint8_t *p_f_seq_out);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of bits in the code block.

e

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message.

k

A read-only parameter of type `uint32_t`.

The number of information bits in the code block.

i_bil

A read-only parameter of type `armral_polar_ibil_type`.

Flag to enable/disable the interleaving of coded bits.

p_d_seq_in

A read-only parameter of type `const uint8_t *`.

Points to `n` bits representing the Polar encoded message.

p_f_seq_out

A write-only parameter of type `uint8_t *`.

Points to `e` bits representing the rate-matched message.

3.4.3.7 armral_polar_rate_matching_noalloc

Non-allocating variant of armral_polar_rate_matching.

Matches the rate of the Polar encoded code block to the rate of the channel using sub-block interleaving, bit selection, and channel interleaving based on Downlink or Uplink direction. This is as described in the 3GPP Technical Specification (TS) 38.212 section 5.4.1.

The code rate of the code block is defined by the ratio of the rate-matched length `e` to the number of information bits in the message `k`. It is assumed that `e` is strictly greater than `k`. Given a rate-matched length and number of information bits, the code block length is determined as described in section 5.3.1 of TS 38.212.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

When `i_bil = ARMRAL_POLAR_IBIL_DISABLE` the buffer must be at least `((n + 7) / 8) * sizeof(uint8_t)` bytes. When `i_bil = ARMRAL_POLAR_IBIL_ENABLE` the buffer must be at least `((n + e + 7) / 8) * sizeof(uint8_t)` bytes.

Syntax

Defined in `armral.h` on line `2986`:

```
armral_status armral_polar_rate_matching_noalloc(
    uint32_t n, uint32_t e, uint32_t k, armral_polar_ibil_type i_bil,
    const uint8_t *p_d_seq_in, uint8_t *p_f_seq_out, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of bits in the code block.

e

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message.

k

A read-only parameter of type `uint32_t`.

The number of information bits in the code block.

i_bil

A read-only parameter of type `armral_polar_ibil_type`.

Flag to enable/disable the interleaving of coded bits.

p_d_seq_in

A read-only parameter of type `const uint8_t *`.

Points to `n` bits representing the Polar encoded message.

p_f_seq_out

A write-only parameter of type `uint8_t *`.

Points to `e` bits representing the rate-matched message.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.3.8 armral_polar_rate_recovery

Recovers the log-likelihood ratios (LLRs) from demodulation to match the length of Polar code blocks using channel deinterleaving, bit recovery, and sub-block deinterleaving. These operations are the inverse of channel interleaving, bit selection, and sub-block interleaving used in Polar rate matching, which is described in the 3GPP Technical Specification (TS) 38.212 section 5.4.1.1.

The size of the code block is given in section 5.3.1 of TS 38.212, and is related to the ratio of the rate matched length `e` and information bits per code block `k` according to clause 5.4.1 of TS 38.212.

The code rate of the code block is the ratio of the rate-matched length `e` to the number of information bits in the message `k`.

Syntax

Defined in `armral.h` on line `3021`:

```
armral_status armral_polar_rate_recovery(uint32_t n, uint32_t e, uint32_t k,
                                         armral_polar_ibil_type i_bil,
                                         const int8_t *p_llr_in,
                                         int8_t *p_llr_out);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The number of bits in the code block. Defined in section 5.3.1 of TS 38.212.

e

A read-only parameter of type `uint32_t`.

The length of the rate-matched message. This is also the number of LLRs in the demodulated message.

k

A read-only parameter of type `uint32_t`.

The number of information bits in the message to recover. The ratio of `e/k` is the rate of the transmitted code block. `e` is assumed to be strictly greater than `k`.

i_bil

A read-only parameter of type `armral_polar_ibil_type`.

Flag to enable/disable the de-interleaving of coded bits.

p_llr_in

A read-only parameter of type `const int8_t *`.

Points to `e` 8-bit LLRs, which are assumed to be the output of demodulation.

p_llr_out

A write-only parameter of type `int8_t *`.

Points to `n` 8-bit rate-recovered LLRs. This output can be passed as input to Polar decoding.

3.4.3.9 armral_polar_rate_recovery_noalloc

Non-allocating variant of armral_polar_rate_recovery. This function recovers the log-likelihood ratios (LLRs) from demodulation to match the length of Polar code blocks using channel deinterleaving, bit recovery, and sub-block deinterleaving. These operations are the inverse of channel interleaving, bit selection, and sub-block interleaving used in Polar rate matching, which is described in the 3GPP Technical Specification (TS) 38.212 section 5.4.1.1.

The size of the code block is given in section 5.3.1 of TS 38.212, and is related to the ratio of the rate matched length `e` and information bits per code block `k` according to clause 5.4.1 of TS 38.212.

The code rate of the code block is the ratio of the rate-matched length `e` to the number of information bits in the message `k`.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least `(n + e) * sizeof(uint8_t)` bytes.

Syntax

Defined in `armral.h` on line `3064`:

```
armral_status armral_polar_rate_recovery_noalloc(
    uint32_t n, uint32_t e, uint32_t k, armral_polar_ibil_type i_bil,
    const int8_t *p_llr_in, int8_t *p_llr_out, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`n`

A read-only parameter of type `uint32_t`.

The number of bits in the code block. Defined in section 5.3.1 of TS 38.212.

`e`

A read-only parameter of type `uint32_t`.

The length of the rate-matched message. This is also the number of LLRs in the demodulated message.

`k`

A read-only parameter of type `uint32_t`.

The number of information bits in the message to recover. The ratio of `e/k` is the rate of the transmitted code block. `e` is assumed to be strictly greater than `k`.

`i_bil`

A read-only parameter of type `armral_polar_ibil_type`.

Flag to enable/disable the de-interleaving of coded bits.

`p_llr_in`

A read-only parameter of type `const int8_t *`.

Points to `e` 8-bit LLRs, which are assumed to be the output of demodulation.

p_llr_out

A write-only parameter of type `int8_t *`.

Points to `n` 8-bit rate-recovered LLRs. This output can be passed as input to Polar decoding.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.3.10 armral_polar_crc_attachment

Performs the Cyclic Redundancy Check (CRC) attachment described in section 5.2.1 of the 3GPP Technical Specification (TS) 38.212.

Syntax

Defined in `armral.h` on line `3079`:

```
armral_status armral_polar_crc_attachment(const uint8_t *data_in, uint32_t a,
                                          uint8_t *data_out);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

data_in

A read-only parameter of type `const uint8_t *`.

Points to the input sequence.

a

A read-only parameter of type `uint32_t`.

The length of the input sequence (`A`).

data_out

A write-only parameter of type `uint8_t *`.

Points to the output sequence of length `K = A + L`, where `L` is the length of the CRC generator polynomial.

3.4.3.11 armral_polar_crc_attachment_noalloc

Non-allocating variant of armral_polar_crc_attachment.

This function performs the Cyclic Redundancy Check (CRC) attachment described in section 5.2.1 of the 3GPP Technical Specification (TS) 38.212.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_polar_crc_attachment_noalloc_buffer_size with identical inputs.

Syntax

Defined in `armral.h` on line `3103`:

```
armral_status armral_polar_crc_attachment_noalloc(const uint8_t *data_in,
                                                  uint32_t a, uint8_t *data_out,
                                                  void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

data_in

A read-only parameter of type `const uint8_t *`.

Points to the input sequence.

a

A read-only parameter of type `uint32_t`.

The length of the input sequence (`A`).

data_out

A write-only parameter of type `uint8_t *`.

Points to the output sequence of length `K = A + L`, where `L` is the length of the CRC generator polynomial.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.3.12 armral_polar_crc_attachment_noalloc_buffer_size

Calculates the required buffer size in bytes for Polar Cyclic Redundancy Check (CRC) attachment for an input sequence of length `A`.

Syntax

Defined in `armral.h` on line `3114`:

```
uint32_t armral_polar_crc_attachment_noalloc_buffer_size(uint32_t a);
```

Returns

The required buffer size in bytes.

Parameters

`a`

A read-only parameter of type `uint32_t`.

The length of the input sequence (`A`).

3.4.3.13 armral_polar_crc_check

Calculates the Cyclic Redundancy Check (CRC) value of a Polar decoded code block of length `A`, as described in section 5.1.1 of the 3GPP Technical Specification (TS) 38.212, and checks if the computed CRC value matches the CRC value attached to that code block.

If the CRC values match the function returns true; otherwise it returns false.

Syntax

Defined in `armral.h` on line `3131`:

```
bool armral_polar_crc_check(const uint8_t *data_in, uint32_t k);
```

Returns

A boolean value that indicates success or failure.

Parameters

`data_in`

A read-only parameter of type `const uint8_t *`.

Points to the input sequence of length `A = K - L`, where `L` is the length of the CRC generator polynomial.

`k`

A read-only parameter of type `uint32_t`.

The length of the message in the input sequence (`K`).

3.4.3.14 armral_polar_crc_check_noalloc

Non-allocating variant of armral_polar_crc_check .

This function calculates the Cyclic Redundancy Check (CRC) value of a Polar decoded code block of length `A`, as described in section 5.1.1 of the 3GPP Technical Specification (TS) 38.212, and checks if the computed CRC value matches the CRC value attached to that code block.

If the CRC values match the function returns true; otherwise it returns false.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_polar_crc_check_noalloc_buffer_size with identical inputs.

Syntax

Defined in `armral.h` on line `3158`:

```
bool armral_polar_crc_check_noalloc(const uint8_t *data_in, uint32_t k,
                                    void *buffer);
```

Returns

A boolean value that indicates success or failure.

Parameters

`data_in`

A read-only parameter of type `const uint8_t *`.

Points to the input sequence of length `A = K - L`, where `L` is the length of the CRC generator polynomial.

`k`

A read-only parameter of type `uint32_t`.

The length of the message in the input sequence (`K`).

`buffer`

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.3.15 armral_polar_crc_check_noalloc_buffer_size

Calculates the required buffer size in bytes needed to perform a Cyclic Redundancy Check (CRC) on a Polar decoded message of length `K`.

Syntax

Defined in `armral.h` on line `3168`:

```
uint32_t armral_polar_crc_check_noalloc_buffer_size(uint32_t k);
```

Returns

The required buffer size in bytes.

Parameters

k

A read-only parameter of type `uint32_t`.

The length of the message in the input sequence (`K`).

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.

3.4.4.1 armral_ldpc_get_base_graph

Uses the identifier of a base graph to get the data structure that describes a base graph.

Syntax

Defined in `armral.h` on line `3570`:

```
const armral_ldpc_base_graph_t *
armral_ldpc_get_base_graph(armral_ldpc_graph_t bg);
```

Returns

A pointer to an LDPC base graph.

Parameters

bg

A read-only parameter of type `armral_ldpc_graph_t`.

Enum identifier of the base graph to get.

3.4.4.2 armral_ldpc_encode_block

Performs encoding using LDPC as laid out in the 3GPP Technical Specification (TS) 38.212. Encoding is performed for a single code block.

The length of the code block is determined from the lifting size `z` and base graph. For example, for base graph 1 the length of a code block is `68 * z` bits, and for base graph 2 the length of the code block is `52 * z` bits. The output from the encoding begins at the third column of the base graph. The first two columns are punctured, as per section 5.3.2 of TS 38.212. The number of encoded bits returned from this function is `66 * z` for base graph 1, and `50 * z` for base graph 2. The values of `z` are limited to those in table 5.3.2-1 in TS 38.212.

The number of information bits in a code block is determined by the lifting size and base graph. For base graph 1 the number of information bits per code block is `22 * z`. For base graph 2 the number of information bits per code block is `10 * z`. It is assumed that the correct number of input bits is passed into this function.

Syntax

Defined in `armral.h` on line `3615`:

```
armral_status armral_ldpc_encode_block(const uint8_t *data_in,
                                       armral_ldpc_graph_t bg, uint32_t z,
                                       uint32_t len_filler_bits,
                                       uint8_t *data_out);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

data_in

A read-only parameter of type `const uint8_t *`.

The information bits to encode. It is assumed that the number of bits stored in `data_in` fits into a single code block. The number of information bits is assumed to be `22 * z` for base graph 1, and `10 * z` for base graph 2.

bg

A read-only parameter of type `armral_ldpc_graph_t`.

Identifier for the base graph to use for encoding. TS 38.212 defines two base graphs in table 5.3.2-2 and 5.3.2-3. The base graph, in combination with the lifting size `z`, determines the block size and the graph to use for encoding the block.

`z`

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

`len_filler_bits`

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size.

`data_out`

A write-only parameter of type `uint8_t *`.

The codeword to be transmitted. `data_out` has the first two columns for the base graphs punctured, and contains the information and calculated parity bits after encoding.

3.4.4.3 armral_ldpc_encode_block_noalloc

Non-allocating variant of armral_ldpc_encode_block.

This function performs encoding using LDPC as laid out in the 3GPP Technical Specification (TS) 38.212. Encoding is performed for a single code block.

The length of the code block is determined from the lifting size `z` and base graph. For example, for base graph 1 the length of a code block is `68 * z` bits, and for base graph 2 the length of the code block is `52 * z` bits. The output from the encoding begins at the third column of the base graph. The first two columns are punctured, as per section 5.3.2 of TS 38.212. The number of encoded bits returned from this function is `66 * z` for base graph 1, and `50 * z` for base graph 2. The values of `z` are limited to those in table 5.3.2-1 in TS 38.212.

The number of information bits in a code block is determined by the lifting size and base graph. For base graph 1 the number of information bits per code block is `22 * z`. For base graph 2 the number of information bits per code block is `10 * z`. It is assumed that the correct number of input bits is passed into this function.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_ldpc_encode_block_noalloc_buffer_size . with identical inputs.

Syntax

Defined in `armral.h` on line `3672`:

```
armral_status armral_ldpc_encode_block_noalloc(const uint8_t *data_in,
                                               armral_ldpc_graph_t bg,
                                               uint32_t z,
                                               uint32_t len_filler_bits,
                                               uint8_t *data_out, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`data_in`

A read-only parameter of type `const uint8_t *`.

The information bits to encode. It is assumed that the number of bits stored in `data_in` fits into a single code block. The number of information bits is assumed to be `22 * z` for base graph 1, and `10 * z` for base graph 2.

`bg`

A read-only parameter of type `armral_ldpc_graph_t`.

Identifier for the base graph to use for encoding. TS 38.212 defines two base graphs in table 5.3.2-2 and 5.3.2-3. The base graph, in combination with the lifting size `z`, determines the block size and the graph to use for encoding the block.

`z`

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

`len_filler_bits`

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size.

`data_out`

A write-only parameter of type `uint8_t *`.

The codeword to be transmitted. `data_out` has the first two columns for the base graphs punctured, and contains the information and calculated parity bits after encoding.

`buffer`

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.4.4 armral_ldpc_encode_block_noalloc_buffer_size

Calculates the required buffer size in bytes to encode a single code block using LDPC for a given base graph and lifting size `z`.

Syntax

Defined in `armral.h` on line `3696`:

```
uint32_t armral_ldpc_encode_block_noalloc_buffer_size(armral_ldpc_graph_t bg,
                                                     uint32_t z,
                                                     uint32_t len_filler_bits);
```

Returns

The required buffer size in bytes.

Parameters

`bg`

A read-only parameter of type `armral_ldpc_graph_t`.

Identifier for the base graph to use for encoding. TS 38.212 defines two base graphs in table 5.3.2-2 and 5.3.2-3. The base graph, in combination with the lifting size `z`, determines the block size and the graph to use for encoding the block.

`z`

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

`len_filler_bits`

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size.

3.4.4.5 armral_ldpc_decode_block

Performs decoding of LDPC using a layered min-sum algorithm. This is an iterative algorithm which takes 8-bit log-likelihood ratios (LLRs) and calculates the most likely codeword by calculating updates using information available from the parity checks in the LDPC graph. LLRs are updated after evaluating checks in a 'layer', where a layer is assumed to contain the same number of checks as the lifting size `z`. There are 46 layers in base graph 1, and 42 layers in base graph 2. Decoding is performed for a single code block.

The `options` parameter can be either ARMRAL_LDPC_DEFAULT_OPTIONS, or a bitwise-or'd result of the below fields:

CRC Type:

- ARMRAL_LDPC_CRC_NO (default)
- ARMRAL_LDPC_CRC_16 (not implemented)
- ARMRAL_LDPC_CRC_24A (not implemented)
- ARMRAL_LDPC_CRC_24B

CRC Mode:

- ARMRAL_LDPC_CRC_EVERY_ITER (default)
- ARMRAL_LDPC_CRC_END_ITER

Filler Bits:

- ARMRAL_LDPC_FILLER_BITS_IMPLICIT
- ARMRAL_LDPC_FILLER_BITS_EXPLICIT (default).

CRC Type, CRC Mode, and Filler Bits are all mutually exclusive groups. If more than one option from a mutually exclusive group is set, `armral_ldpc_decode_block` returns `ARMRAL_ARGUMENT_ERROR`.

If ARMRAL_LDPC_CRC_NO (default) is set in `options`, no CRC check is performed. If ARMRAL_LDPC_CRC_24B is set in `options`, a CRC check will be performed using armral_crc24_b_be. The frequency of how often the CRC check is done is determined by ARMRAL_LDPC_CRC_EVERY_ITER (default) and ARMRAL_LDPC_CRC_END_ITER. If the CRC check succeeds, `armral_ldpc_decode_block` returns `ARMRAL_SUCCESS`. If the CRC check fails on the last iteration, `armral_ldpc_decode_block` returns `ARMRAL_FAIL`.

The LDPC decoder supports two input formats, either explicit filler bits or implicit filler bits. If ARMRAL_LDPC_FILLER_BITS_EXPLICIT (default) is set in `options`, the input LLRs must be in the format

```
[ message LLRs | filler LLRs | parity LLRs ]
```

If ARMRAL_LDPC_FILLER_BITS_IMPLICIT is set in `options`, the input LLRs must be in the format

```
[ message LLRs | parity LLRs ]
```

`ARMRAL_LDPC_FILLER_BITS_EXPLICIT` should be used if `llrs` was generated by armral_ldpc_rate_recovery.

Syntax

Defined in `armral.h` on line `3787`:

```
armral_status armral_ldpc_decode_block(uint32_t n, const int8_t *llrs,
                                       armral_ldpc_graph_t bg, uint32_t z,
                                       uint32_t len_filler_bits,
                                       uint8_t *data_out, uint32_t max_its,
                                       uint32_t options);
```

Returns

Returns `ARMRAL_SUCCESS` on success, `ARMRAL_ARGUMENT_ERROR` if an input parameter is incorrect, or `ARMRAL_FAIL` if the CRC check for convergence fails.

Parameters

n

A read-only parameter of type `uint32_t`.

The length of `llrs`.

llrs

A read-only parameter of type `const int8_t *`.

The initial LLRs to use in the decoding. This is typically the output after demodulation and rate recovery.

bg

A read-only parameter of type `armral_ldpc_graph_t`.

The type of base graph to use for the decoding.

z

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

len_filler_bits

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size. Filler bits are used to calculate CRC internally. This is assumed to be a multiple of 8.

data_out

A write-only parameter of type `uint8_t *`.

The decoded bits. These are of length `22 * z` for base graph 1 and `10 * z` for base graph 2. It is assumed that the array `data_out` is able to store this many bits.

max_its

A read-only parameter of type `uint32_t`.

The maximum number of iterations of the LDPC decoder. The algorithm may terminate after fewer iterations if the current candidate codeword passes all the parity checks, or if it satisfies the CRC check.

options

A read-only parameter of type `uint32_t`.

See the documentation above for a summary of available `options`. If you want to use the default options, set the options parameter to either `0` or `ARMRAL_LDPC_DEFAULT_OPTIONS`.

3.4.4.6 armral_ldpc_decode_block_noalloc

Non-allocating variant of armral_ldpc_decode_block.

Performs decoding of LDPC using a layered min-sum algorithm. This is an iterative algorithm which takes 8-bit log-likelihood ratios (LLRs) and calculates the most likely codeword by calculating updates using information available from the parity checks in the LDPC graph. LLRs are updated after evaluating checks in a 'layer', where a layer is assumed to contain the same number of checks as the lifting size `z`. There are 46 layers in base graph 1, and 42 layers in base graph 2. Decoding is performed for a single code block.

The `options` parameter can be either ARMRAL_LDPC_DEFAULT_OPTIONS, or a bitwise-or'd result of the below fields:

CRC Type:

- ARMRAL_LDPC_CRC_NO (default)
- ARMRAL_LDPC_CRC_16 (not implemented)
- ARMRAL_LDPC_CRC_24A (not implemented)
- ARMRAL_LDPC_CRC_24B

CRC Mode:

- ARMRAL_LDPC_CRC_EVERY_ITER (default)
- ARMRAL_LDPC_CRC_END_ITER

Filler Bits:

- ARMRAL_LDPC_FILLER_BITS_IMPLICIT
- ARMRAL_LDPC_FILLER_BITS_EXPLICIT (default).

CRC Type, CRC Mode, and Filler Bits are all mutually exclusive groups. If more than one option from a mutually exclusive group is set, `armral_ldpc_decode_block` returns `ARMRAL_ARGUMENT_ERROR`.

If ARMRAL_LDPC_CRC_NO (default) is set in `options`, no CRC check is performed.
If ARMRAL_LDPC_CRC_24B is set in `options`, a CRC check will be performed using armral_crc24_b_be. The frequency of how often the CRC check is done is determined by ARMRAL_LDPC_CRC_EVERY_ITER (default) and ARMRAL_LDPC_CRC_END_ITER. If the CRC check succeeds, `armral_ldpc_decode_block` returns `ARMRAL_SUCCESS`. If the CRC check fails on the last iteration, `armral_ldpc_decode_block` returns `ARMRAL_FAIL`.

The LDPC decoder supports two input formats, either explicit filler bits or implicit filler bits. If ARMRAL_LDPC_FILLER_BITS_EXPLICIT (default) is set in `options`, the input LLRs must be in the format

```
[ message LLRs | filler LLRs | parity LLRs ]
```

If ARMRAL_LDPC_FILLER_BITS_IMPLICIT is set in `options`, the input LLRs must be in the format

```
[ message LLRs | parity LLRs ]
```

`ARMRAL_LDPC_FILLER_BITS_EXPLICIT` should be used if `llrs` was generated by armral_ldpc_rate_recovery.

Syntax

Defined in `armral.h` on line `3884`:

```
armral_status armral_ldpc_decode_block_noalloc(
    uint32_t n, const int8_t *llrs, armral_ldpc_graph_t bg, uint32_t z,
    uint32_t len_filler_bits, uint8_t *data_out, uint32_t max_its,
    uint32_t options, void *buffer);
```

Returns

Returns `ARMRAL_SUCCESS` on success, `ARMRAL_ARGUMENT_ERROR` if an input parameter is incorrect, or `ARMRAL_FAIL` if the CRC check for convergence fails.

Parameters

`n`

A read-only parameter of type `uint32_t`.

The length of `llrs`.

`llrs`

A read-only parameter of type `const int8_t *`.

The initial LLRs to use in the decoding. This is typically the output after demodulation and rate recovery.

`bg`

A read-only parameter of type `armral_ldpc_graph_t`.

The type of base graph to use for the decoding.

`z`

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

`len_filler_bits`

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size. Filler bits are used to calculate CRC internally. This is assumed to be a multiple of 8.

`data_out`

A write-only parameter of type `uint8_t *`.

The decoded bits. These are of length `22 * z` for base graph 1 and `10 * z` for base graph 2. It is assumed that the array `data_out` is able to store this many bits.

max_its

A read-only parameter of type `uint32_t`.

The maximum number of iterations of the LDPC decoder. The algorithm may terminate after fewer iterations if the current candidate codeword passes all the parity checks, or if it satisfies the CRC check.

options

A read-only parameter of type `uint32_t`.

See the documentation above for a summary of available `options`. If you want to use the default options, set the options parameter to either `0` or `ARMRAL_LDPC_DEFAULT_OPTIONS`.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.4.7 armral_ldpc_decode_block_noalloc_buffer_size

Calculates the required buffer size in bytes needed to perform LDPC decoding of a single code block for a given base graph and lifting size `z`.

Syntax

Defined in `armral.h` on line `3902`:

```
uint32_t armral_ldpc_decode_block_noalloc_buffer_size(armral_ldpc_graph_t bg,
                                                      uint32_t z,
                                                      uint32_t max_its);
```

Returns

The required buffer size in bytes.

Parameters

bg

A read-only parameter of type `armral_ldpc_graph_t`.

The type of base graph to use for the decoding.

z

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

max_its

A read-only parameter of type `uint32_t`.

The maximum number of iterations of the LDPC decoder. The algorithm may terminate after fewer iterations if the current candidate codeword passes all the parity checks, or if it satisfies the CRC check.

3.4.4.8 armral_ldpc_rate_matching

Matches the rate of the code block encoded with LDPC code to the rate of the channel using bit selection and bit interleaving. This is as described in the 3GPP Technical Specification (TS) 38.212 section 5.4.2.

The input to the rate matching is assumed to be the output from LDPC encoding for a single code block. The output from rate matching is to be passed to demodulation.

The code rate for a given code block is the ratio of rate matched length `e` to the number of information bits per code block. The number of information bits is assumed to be `22 * z` for base graph 1, and `10 * z` for base graph 2, where `z` is the lifting size. It is assumed that `e` is strictly greater than the number of information bits in a code block. `e` must also be a multiple of the modulation order (i.e. the number of bits per modulation symbol).

Syntax

Defined in `armral.h` on line `3955`:

```
armral_status armral_ldpc_rate_matching(armral_ldpc_graph_t bg, uint32_t z,
                                         uint32_t e, uint32_t nref,
                                         uint32_t len_filler_bits, uint32_t k,
                                         uint32_t rv, armral_modulation_type mod,
                                         const uint8_t *src, uint8_t *dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

bg

A read-only parameter of type `armral_ldpc_graph_t`.

The type of base graph for which rate matching is to be performed.

z

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

e

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message. This is assumed to be a multiple of the number of bits per modulation symbol.

nref

A read-only parameter of type `uint32_t`.

The soft buffer size for limited buffer rate matching. `nref` is defined in TS 38.212 section 5.4.2.1.

len_filler_bits

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size.

k

A read-only parameter of type `uint32_t`.

Codeblock size, the number of bits to encode as per section 5.3.2 of TS 38.212.

rv

A read-only parameter of type `uint32_t`.

Redundancy version used in rate matching. Must be in the set `{0, 1, 2, 3}`. The effect of choosing different redundancy versions is described in table 5.4.2.1-2 of TS 38.212.

mod

A read-only parameter of type `armral_modulation_type`.

The type of modulation to perform. Required to perform bit-interleaving, as described in section 5.4.2 of TS 38.212.

src

A read-only parameter of type `const uint8_t *`.

Input array. This is assumed to be the output of LDPC encoding. This contains `66 * z` bits in the case that base graph 1 is used, and `50 * z` bits in the case that base graph 2 is used.

dst

A write-only parameter of type `uint8_t *`.

Contains `e` bits of data, which is the rate-matched data ready to be passed to modulation.

3.4.4.9 armral_ldpc_rate_matching_noalloc

Non-allocating variant of armral_ldpc_rate_matching.

This function matches the rate of the code block encoded with LDPC code to the rate of the channel using bit selection and bit interleaving. This is as described in the 3GPP Technical Specification (TS) 38.212 section 5.4.2.

The input to the rate matching is assumed to be the output from LDPC encoding for a single code block. The output from rate matching is to be passed to demodulation.

The code rate for a given code block is the ratio of rate matched length `e` to the number of information bits per code block. The number of information bits is assumed to be `22 * z` for base graph 1, and `10 * z` for base graph 2, where `z` is the lifting size. It is assumed that `e` is strictly greater than the number of information bits in a code block. `e` must also be a multiple of the modulation order (i.e. the number of bits per modulation symbol).

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least `((2 * z * 66) + e) * sizeof(uint8_t)` bytes.

Syntax

Defined in `armral.h` on line `4018`:

```
armral_status armral_ldpc_rate_matching_noalloc(
    armral_ldpc_graph_t bg, uint32_t z, uint32_t e, uint32_t nref,
    uint32_t len_filler_bits, uint32_t k, uint32_t rv,
    armral_modulation_type mod, const uint8_t *src, uint8_t *dst, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

bg

A read-only parameter of type `armral_ldpc_graph_t`.

The type of base graph for which rate matching is to be performed.

z

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

e

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message. This is assumed to be a multiple of the number of bits per modulation symbol.

nref

A read-only parameter of type `uint32_t`.

The soft buffer size for limited buffer rate matching. `nref` is defined in TS 38.212 section 5.4.2.1.

len_filler_bits

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size.

k

A read-only parameter of type `uint32_t`.

Codeblock size, the number of bits to encode as per section 5.3.2 of TS 38.212.

rv

A read-only parameter of type `uint32_t`.

Redundancy version used in rate matching. Must be in the set `{0, 1, 2, 3}`. The effect of choosing different redundancy versions is described in table 5.4.2.1-2 of TS 38.212.

mod

A read-only parameter of type `armral_modulation_type`.

The type of modulation to perform. Required to perform bit-interleaving, as described in section 5.4.2 of TS 38.212.

src

A read-only parameter of type `const uint8_t *`.

Input array. This is assumed to be the output of LDPC encoding. This contains `66 * z` bits in the case that base graph 1 is used, and `50 * z` bits in the case that base graph 2 is used.

dst

A write-only parameter of type `uint8_t *`.

Contains `e` bits of data, which is the rate-matched data ready to be passed to modulation.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.4.10 armral_ldpc_rate_recovery

Recovers the log-likelihood ratios (LLRs) from demodulation to match the length of an LDPC code block. This is the inverse of the operations for rate matching for LDPC described in the 3GPP Technical Specification (TS) 38.212 section 5.4.2. The input array is of length `e` bytes, where `e` is the rate-matched length of the code block. It is assumed that `e` is a multiple of the modulation order (i.e. the number of bits per modulation symbol).

The size of the code block is determined using the base graph and lifting size `z`. For base graph 1, the code block is of length `68 * z`. For base graph 2, the code block is of length `52 * z`. The output of the rate recovery will be of length `66 * z` for base graph 1, and `50 * z` for base graph 2, as it is assumed that the first two information columns are punctured.

The rate of the code block is the ratio of the rate matched length `e` and the number of information bits in the code block. The number of information bits in the code block is `22 * z` for base graph 1, and `10 * z` for base graph 2.

The output array also serves as an input array. It contains the current approximation to LLRs. The LLRs calculated from the rate-recovery are summed to existing LLRs in the output array.

Syntax

Defined in `armral.h` on line `4082`:

```
armral_status armral_ldpc_rate_recovery(armral_ldpc_graph_t bg, uint32_t z,
                                        uint32_t e, uint32_t nref,
                                        uint32_t len_filler_bits, uint32_t k,
                                        uint32_t rv, armral_modulation_type mod,
                                        const int8_t *src, int8_t *dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`bg`

A read-only parameter of type `armral_ldpc_graph_t`.

The type of base graph for which rate matching is to be performed.

`z`

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

`e`

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message. This is assumed to be a multiple of the number of bits per modulation symbol.

`nref`

A read-only parameter of type `uint32_t`.

The soft buffer size for limited buffer rate matching. `nref` is defined in TS 38.212 section 5.4.2.1.

`len_filler_bits`

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size.

`k`

A read-only parameter of type `uint32_t`.

Codeblock size, the number of bits to encode as per section 5.3.2 of TS 38.212.

`rv`

A read-only parameter of type `uint32_t`.

Redundancy version used in rate matching. Must be in the set `{0, 1, 2, 3}`. The effect of choosing different redundancy versions is described in table 5.4.2.1-2 of TS 38.212.

mod

A read-only parameter of type `armral_modulation_type`.

The type of modulation to perform. Required to perform bit-interleaving, as described in section 5.4.2 of TS 38.212.

src

A read-only parameter of type `const int8_t *`.

Input array of a total of `e` 8-bit LLRs. This is the output after demodulation.

dst

A parameter of type `int8_t *`.

On entry, contains the current approximation to LLRs. If no approximation of the LLRs is known, all entries must be set to zero. The array has length `66 * z` for base graph 1, and `50 * z` for base graph 2. On exit, updated rate-recovered 8-bit LLRs, which are ready to be passed to decoding.

3.4.4.11 armral_ldpc_rate_recovery_noalloc

Non-allocating variant of armral_ldpc_rate_recovery. This function recovers the log-likelihood ratios (LLRs) from demodulation to match the length of an LDPC code block. This is the inverse of the operations for rate matching for LDPC described in the 3GPP Technical Specification (TS) 38.212 section 5.4.2. The input array is of length `e` bytes, where `e` is the rate-matched length of the code block. It is assumed that `e` is a multiple of the modulation order (i.e. the number of bits per modulation symbol).

The size of the code block is determined using the base graph and lifting size `z`. For base graph 1, the code block is of length `68 * z`. For base graph 2, the code block is of length `52 * z`. The output of the rate recovery will be of length `66 * z` for base graph 1, and `50 * z` for base graph 2, as it is assumed that the first two information columns are punctured.

The rate of the code block is the ratio of the rate matched length `e` and the number of information bits in the code block. The number of information bits in the code block is `22 * z` for base graph 1, and `10 * z` for base graph 2.

The output array also serves as an input array. It contains the current approximation to LLRs. The LLRs calculated from the rate-recovery are summed to existing LLRs in the output array.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least `e * sizeof(uint8_t)` bytes.

Syntax

Defined in `armral.h` on line `4154`:

```
armral_status armral_ldpc_rate_recovery_noalloc(
    armral_ldpc_graph_t bg, uint32_t z, uint32_t e, uint32_t nref,
    uint32_t len_filler_bits, uint32_t k, uint32_t rv,
    armral_modulation_type mod, const int8_t *src, int8_t *dst, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

bg

A read-only parameter of type `armral_ldpc_graph_t`.

The type of base graph for which rate matching is to be performed.

z

A read-only parameter of type `uint32_t`.

The lifting size. Valid values of the lifting size are described in table 5.3.2-1 in TS 38.212.

e

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message. This is assumed to be a multiple of the number of bits per modulation symbol.

nref

A read-only parameter of type `uint32_t`.

The soft buffer size for limited buffer rate matching. `nref` is defined in TS 38.212 section 5.4.2.1.

len_filler_bits

A read-only parameter of type `uint32_t`.

The number of filler bits. As per TS 38.212, section 5.2.2, filler bits insertion is needed to ensure that the code block segments have a valid length and are a multiple of the lifting size.

k

A read-only parameter of type `uint32_t`.

Codeblock size, the number of bits to encode as per section 5.3.2 of TS 38.212.

rv

A read-only parameter of type `uint32_t`.

Redundancy version used in rate matching. Must be in the set `{0, 1, 2, 3}`. The effect of choosing different redundancy versions is described in table 5.4.2.1-2 of TS 38.212.

mod

A read-only parameter of type `armral_modulation_type`.

The type of modulation to perform. Required to perform bit-interleaving, as described in section 5.4.2 of TS 38.212.

src

A read-only parameter of type `const int8_t *`.

Input array of a total of `e` 8-bit LLRs. This is the output after demodulation.

dst

A parameter of type `int8_t *`.

On entry, contains the current approximation to LLRs. If no approximation of the LLRs is known, all entries must be set to zero. The array has length `66 * z` for base graph 1, and `50 * z` for base graph 2. On exit, updated rate-recovered 8-bit LLRs, which are ready to be passed to decoding.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

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.

3.4.5.1 armral_turbo_perm_idx_init

This function generates all permutation indices used in the permutation step of the LTE Turbo decoding. This function may be used to generate the indices before calling the decoding functions armral_turbo_decode_block and armral_turbo_decode_block_noalloc, which accept the generated buffer as input.

This function generates permutation indices for all permitted lengths of the input data passed to the decoder. After generating the indices, the same buffer may be reused independent of any changes to the input data, including the length `k`.

This function takes a pre-allocated buffer (`buffer`) to fill with permutation indices.

The buffer must be at least `1065744 * sizeof(uint16_t)` bytes.

Syntax

Defined in `armral.h` on line `4194`:

```
armral_status armral_turbo_perm_idx_init(uint16_t *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

buffer

A write-only parameter of type `uint16_t *`.

A pre-allocated buffer to hold the permutation indices.

3.4.5.2 armral_turbo_encode_block

This function implements the LTE Turbo encoding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input an array `src` of length `k` bits, where `k` must be one of the values defined in TS 36.212 Table 5.1.3-3. The outputs of the encoding are written into the three arrays `dst0`, `dst1`, and `dst2`, each of which contains `k + 4` bits of output. The encoding is performed for a single code block.

Syntax

Defined in `armral.h` on line `4220`:

```
armral_status armral_turbo_encode_block(const uint8_t *src, uint32_t k,
                                        uint8_t *dst0, uint8_t *dst1,
                                        uint8_t *dst2);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

src

A read-only parameter of type `const uint8_t *`.

Input data of length `k` bits.

k

A read-only parameter of type `uint32_t`.

Length of the input code block in bits.

dst0

A write-only parameter of type `uint8_t *`.

The systematic portion of the output of length `k + 4` bits. If `k + 4` is not on a byte boundary, the most significant bits of the final byte in this array contain the systematic bits.

dst1

A write-only parameter of type `uint8_t *`.

The parity portion of the output of length `k + 4` bits. If `k + 4` is not on a byte boundary, the most significant bits of the final byte in this array contain the parity bits.

dst2

A write-only parameter of type `uint8_t *`.

The interleaved parity portion of the output of length `k + 4` bits. If `k + 4` is not on a byte boundary, the most significant bits of the final byte in this array contain the interleaved bits.

3.4.5.3 armral_turbo_encode_block_noalloc

Non-allocating variant of armral_turbo_encode_block.

This function implements the LTE Turbo encoding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input an array `src` of length `k` bits, where `k` must be one of the values defined in TS 36.212 Table 5.1.3-3. The outputs of the encoding are written into the three arrays `dst0`, `dst1`, and `dst2`, each of which contains `k + 4` bits of output. The encoding is performed for a single code block.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least `(k / 8) * sizeof(uint8_t)` bytes.

Syntax

Defined in `armral.h` on line `4257`:

```
armral_status armral_turbo_encode_block_noalloc(const uint8_t *src, uint32_t k,
                                                 uint8_t *dst0, uint8_t *dst1,
                                                 uint8_t *dst2, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

src

A read-only parameter of type `const uint8_t *`.

Input data of length `k` bits.

k

A read-only parameter of type `uint32_t`.

Length of the input code block in bits.

dst0

A write-only parameter of type `uint8_t *`.

The systematic portion of the output of length `k + 4` bits. If `k + 4` is not on a byte boundary, the most significant bits of the final byte in this array contain the systematic bits.

dst1

A write-only parameter of type `uint8_t *`.

The parity portion of the output of length `k + 4` bits. If `k + 4` is not on a byte boundary, the most significant bits of the final byte in this array contain the parity bits.

dst2

A write-only parameter of type `uint8_t *`.

The interleaved parity portion of the output of length `k + 4` bits. If `k + 4` is not on a byte boundary, the most significant bits of the final byte in this array contain the interleaved bits.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.5.4 armral_turbo_decode_block

This function implements a maximum a posteriori (MAP) algorithm to decode the output of the LTE Turbo encoding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input three arrays `sys`, `par` and `itl`, each of length `k + 4` bytes where `k` must be one of the values defined in TS 36.212 Table 5.1.3-3. These three arrays contain the log-likelihood ratios (LLRs) of the systematic, parity and interleaved parity bits. The decoding is performed for a single code block.

The output is written into the array `dst`, which must contain enough bytes to store `k` bits. These are hard outputs (that is, either 0 or 1); the function does not return LLRs.

The function takes a parameter `max_iter`, which specifies the maximum number of iterations that the decoder will perform. The algorithm will terminate in fewer iterations if there is no change in the computed LLRs between consecutive iterations.

Note: The function is called in one of two ways:

- `perm_idxs` is pre-populated by calling armral_turbo_perm_idx_init before the first call to `armral_turbo_decode_block`. This initialization only happens once and the resulting permutation array can be reused in multiple calls to `armral_turbo_decode_block`.
- `perm_idxs` is a null pointer. In this case `armral_turbo_decode_block` will regenerate the permutation indices during every call.

Syntax

Defined in `armral.h` on line `4303`:

```
armral_status armral_turbo_decode_block(const int8_t *sys, const int8_t *par,
                                        const int8_t *itl, uint32_t k,
                                        uint8_t *dst, uint32_t max_iter,
                                        uint16_t *perm_idxs);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`sys`

A read-only parameter of type `const int8_t *`.

The systematic portion of the input of length `k + 4` bytes representing 8-bit log-likelihood ratios.

`par`

A read-only parameter of type `const int8_t *`.

The parity portion of the input of length `k + 4` bytes representing 8-bit log-likelihood ratios.

`itl`

A read-only parameter of type `const int8_t *`.

The interleaved portion of the input of length `k + 4` bytes representing 8-bit log-likelihood ratios.

`k`

A read-only parameter of type `uint32_t`.

Length of the output code block in bits.

`dst`

A write-only parameter of type `uint8_t *`.

Decoded output data of length `k` bits.

`max_iter`

A read-only parameter of type `uint32_t`.

Maximum number of decoding iterations to perform.

`perm_idxs`

A read-only parameter of type `uint16_t *`.

Either a buffer containing the permutation indices generated by an earlier call to armral_turbo_perm_idx_init, or NULL.

3.4.5.5 armral_turbo_decode_block_noalloc

Non-allocating variant of armral_turbo_decode_block.

This function implements a maximum a posteriori (MAP) algorithm to decode the output of the LTE Turbo encoding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input three arrays `sys`, `par` and `itl`, each of length `k + 4` bytes where `k` must be one of the values defined in TS 36.212 Table 5.1.3-3. These three arrays contain the log-likelihood ratios (LLRs) of the systematic, parity and interleaved parity bits. The decoding is performed for a single code block.

The output is written into the array `dst`, which must contain enough bytes to store `k` bits. These are hard outputs (that is, either 0 or 1); the function does not return LLRs.

The function takes a parameter `max_iter`, which specifies the maximum number of iterations that the decoder will perform. The algorithm will terminate in fewer iterations if there is no change in the computed LLRs between consecutive iterations.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_turbo_decode_block_noalloc_buffer_size with identical inputs.

Note: The function is called in one of two ways:

- `perm_idxs` is pre-populated by calling armral_turbo_perm_idx_init before the first call to `armral_turbo_decode_block_noalloc`. This initialization only happens once and the resulting permutation array can be reused in multiple calls to `armral_turbo_decode_block_noalloc`.
- `perm_idxs` is a null pointer. In this case `armral_turbo_decode_block_noalloc` will regenerate the permutation indices during every call.

Syntax

Defined in `armral.h` on line `4361`:

```
armral_status armral_turbo_decode_block_noalloc(
    const int8_t *sys, const int8_t *par, const int8_t *itl, uint32_t k,
    uint8_t *dst, uint32_t max_iter, uint16_t *perm_idxs, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

sys

A read-only parameter of type `const int8_t *`.

The systematic portion of the input of length `k + 4` bytes representing 8-bit log-likelihood ratios.

par

A read-only parameter of type `const int8_t *`.

The parity portion of the input of length `k + 4` bytes representing 8-bit log-likelihood ratios.

itl

A read-only parameter of type `const int8_t *`.

The interleaved portion of the input of length `k + 4` bytes representing 8-bit log-likelihood ratios.

k

A read-only parameter of type `uint32_t`.

Length of the output code block in bits.

dst

A write-only parameter of type `uint8_t *`.

Decoded output data of length `k` bits.

max_iter

A read-only parameter of type `uint32_t`.

Maximum number of decoding iterations to perform.

perm_idxs

A read-only parameter of type `uint16_t *`.

Either a buffer containing the permutation indices generated by an earlier call to armral_turbo_perm_idx_init, or NULL.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.5.6 armral_turbo_decode_block_noalloc_buffer_size

Calculates the required buffer size in bytes required to perform Turbo decoding of a single code block of length `k`.

Syntax

Defined in `armral.h` on line `4372`:

```
uint32_t armral_turbo_decode_block_noalloc_buffer_size(uint32_t k);
```

Returns

The required buffer size in bytes.

Parameters

k

A read-only parameter of type `uint32_t`.

Length of the output code block in bits.

3.4.5.7 armral_turbo_decode_batch

This function implements a maximum a posteriori (MAP) algorithm to decode the output of the LTE Turbo encoding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input three arrays `sys`, `par` and `itl`, each of length `num_blocks * (k + 4)` bytes where `k` must be one of the values defined in TS 36.212 Table 5.1.3-3. These three arrays each contain a batch of `num_block` log-likelihood ratios (LLRs) of the systematic, parity and interleaved parity bits respectively. The decoding is performed in batches of 8 code blocks, with any remaining blocks decoded individually, using armral_turbo_decode_block.

The input arrays for the batch must be uninterleaved and stored contiguously, such that element `ki` of block `bi` 's systematic data, for example, is located at `sys[bi * (k + 4) + ki]`.

The output is written into the array `dst`, which must contain enough bytes to store `num_blocks * k` bits. The output is also uninterleaved and stored contiguously, such that byte `ki` of block `bi` is located at `dst[bi * (k / 8) + ki]`. These are hard outputs (that is, either 0 or 1); the function does not return LLRs.

The function takes a parameter `max_iter`, which specifies the maximum number of iterations that the decoder will perform. The algorithm will terminate in fewer iterations if there is no change in the computed LLRs between consecutive iterations.

Note: This function is called in one of two ways:

- `perm_idxs` is pre-populated by calling armral_turbo_perm_idx_init before the first call to `armral_turbo_decode_batch`. This initialization only happens once and the resulting permutation array can be reused in multiple calls to `armral_turbo_decode_batch`.
- `perm_idxs` is a null pointer. In this case `armral_turbo_decode_batch` will regenerate the permutation indices during every call.

Syntax

Defined in `armral.h` on line `4430`:

```
armral_status armral_turbo_decode_batch(uint32_t num_blocks, const int8_t *sys,
                                        const int8_t *par, const int8_t *itl,
                                        uint32_t k, uint8_t *dst,
                                        uint32_t max_iter, uint16_t *perm_idxs);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_blocks

A read-only parameter of type `uint32_t`.

Number of blocks of data to decode in one call to this function.

sys

A read-only parameter of type `const int8_t *`.

The batched systematic portion of the input of length `num_blocks * (k + 4)` bytes representing a batch of 8-bit log-likelihood ratios.

par

A read-only parameter of type `const int8_t *`.

The batched parity portion of the input of length `num_blocks * (k + 4)` bytes representing a batch of 8-bit log-likelihood ratios.

itl

A read-only parameter of type `const int8_t *`.

The batched interleaved portion of the input of length `num_blocks * (k + 4)` bytes representing a batch of 8-bit log-likelihood ratios.

k

A read-only parameter of type `uint32_t`.

Length of one output code block in bits.

dst

A write-only parameter of type `uint8_t *`.

Batched decoded output data of length `num_blocks * k` bits.

max_iter

A read-only parameter of type `uint32_t`.

Maximum number of decoding iterations to perform.

perm_idxs

A read-only parameter of type `uint16_t *`.

Buffer containing the permutation indices for all `k` generated by an earlier call to armral_turbo_perm_idx_init .

3.4.5.8 armral_turbo_decode_batch_noalloc

Non-allocating variant of armral_turbo_decode_batch.

This function implements a maximum a posteriori (MAP) algorithm to decode the output of the LTE Turbo encoding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input three arrays `sys`, `par` and `itl`, each of length `num_blocks * (k + 4)` bytes where `k` must be one of the values defined in TS 36.212 Table 5.1.3-3. These three arrays each contain a batch of `num_block` log-likelihood ratios (LLRs) of the systematic, parity and interleaved parity bits respectively. The decoding is performed in batches of 8 code blocks, with any remaining blocks decoded individually, using armral_turbo_decode_block_noalloc.

The input arrays for the batch must be uninterleaved and stored contiguously, such that element `ki` of block `bi` 's systematic data, for example, is located at `sys[bi * (k + 4) + ki]`.

The output is written into the array `dst`, which must contain enough bytes to store `num_blocks * k` bits. The output is also uninterleaved and stored contiguously, such that byte `ki` of block `bi` is located at `dst[bi * (k / 8) + ki]`. These are hard outputs (that is, either 0 or 1); the function does not return LLRs.

The function takes a parameter `max_iter`, which specifies the maximum number of iterations that the decoder will perform. The algorithm will terminate in fewer iterations if there is no change in the computed LLRs between consecutive iterations.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_turbo_decode_block_noalloc_buffer_size with identical inputs.

Note: This function is called in one of two ways:

- `perm_idxs` is pre-populated by calling armral_turbo_perm_idx_init before the first call to `armral_turbo_decode_batch_noalloc`. This initialization only happens once and the resulting permutation array can be reused in multiple calls to `armral_turbo_decode_batch_noalloc`.
- `perm_idxs` is a null pointer. In this case `armral_turbo_decode_batch_noalloc` will regenerate the permutation indices during every call.

Syntax

Defined in `armral.h` on line `4502`:

```
armral_status
armral_turbo_decode_batch_noalloc(uint32_t num_blocks, const int8_t *sys,
                                  const int8_t *par, const int8_t *itl,
                                  uint32_t k, uint8_t *dst, uint32_t max_iter,
                                  uint16_t *perm_idxs, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_blocks

A read-only parameter of type `uint32_t`.

Number of blocks of data to decode in one call to this function.

sys

A read-only parameter of type `const int8_t *`.

The batched systematic portion of the input of length `num_blocks * (k + 4)` bytes representing a batch of 8-bit log-likelihood ratios.

par

A read-only parameter of type `const int8_t *`.

The batched parity portion of the input of length `num_blocks * (k + 4)` bytes representing a batch of 8-bit log-likelihood ratios.

itl

A read-only parameter of type `const int8_t *`.

The batched interleaved portion of the input of length `num_blocks * (k + 4)` bytes representing a batch of 8-bit log-likelihood ratios.

k

A read-only parameter of type `uint32_t`.

Length of one output code block in bits.

dst

A write-only parameter of type `uint8_t *`.

Batched decoded output data of length `num_blocks * k` bits.

max_iter

A read-only parameter of type `uint32_t`.

Maximum number of decoding iterations to perform.

perm_idxs

A read-only parameter of type `uint16_t *`.

Buffer containing the permutation indices for all `k` generated by an earlier call to armral_turbo_perm_idx_init .

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.5.9 armral_turbo_decode_batch_noalloc_buffer_size

Calculates the required buffer size in bytes required to perform Turbo decoding of a batch of 8 code blocks, each of length `k`.

Syntax

Defined in `armral.h` on line `4515`:

```
uint32_t armral_turbo_decode_batch_noalloc_buffer_size(uint32_t k);
```

Returns

The required buffer size in bytes.

Parameters

k

A read-only parameter of type `uint32_t`.

Length of one output code block in bits.

3.4.5.10 armral_turbo_rate_matching

Matches the rate of the Turbo encoded code block to the rate of the channel using sub-block interleaving, bit collection, and bit selection and pruning. This is as described in 3GPP Technical Specification (TS) 36.212 section 5.1.4.1.

Syntax

Defined in `armral.h` on line `4537`:

```
armral_status armral_turbo_rate_matching(uint32_t d, uint32_t e, uint32_t rv,
                                         const uint8_t *src0,
                                         const uint8_t *src1,
                                         const uint8_t *src2, uint8_t *dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

d

A read-only parameter of type `uint32_t`.

The number of bits in the encoded message.

e

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message.

rv

A read-only parameter of type `uint32_t`.

The redundancy version number for the transmission.

src0

A read-only parameter of type `const uint8_t *`.

Input array. Stores `d` bits, which are the systematic output of Turbo encoding.

src1

A read-only parameter of type `const uint8_t *`.

Input array. Stores `d` bits, which are the parity output of Turbo encoding.

src2

A read-only parameter of type `const uint8_t *`.

Input array. Stores `d` bits, which are the interleaved parity output of Turbo encoding.

dst

A write-only parameter of type `uint8_t *`.

Output array. Stores `e` bits, which is the output after sub-block interleaving, bit collection and pruning as per 3GPP technical specification 36.212 section 5.1.4.

3.4.5.11 armral_turbo_rate_matching_noalloc

Non-allocating variant of armral_turbo_rate_matching.

This function matches the rate of the Turbo encoded code block to the rate of the channel using sub-block interleaving, bit collection, and bit selection and pruning. This is as described in 3GPP Technical Specification (TS) 36.212 section 5.1.4.1.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_turbo_rate_matching_noalloc_buffer_size with identical inputs.

Syntax

Defined in `armral.h` on line `4572`:

```
armral_status armral_turbo_rate_matching_noalloc(
    uint32_t d, uint32_t e, uint32_t rv, const uint8_t *src0,
    const uint8_t *src1, const uint8_t *src2, uint8_t *dst, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

d

A read-only parameter of type `uint32_t`.

The number of bits in the encoded message.

e

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message.

rv

A read-only parameter of type `uint32_t`.

The redundancy version number for the transmission.

src0

A read-only parameter of type `const uint8_t *`.

Input array. Stores `d` bits, which are the systematic output of Turbo encoding.

src1

A read-only parameter of type `const uint8_t *`.

Input array. Stores `d` bits, which are the parity output of Turbo encoding.

src2

A read-only parameter of type `const uint8_t *`.

Input array. Stores `d` bits, which are the interleaved parity output of Turbo encoding.

dst

A write-only parameter of type `uint8_t *`.

Output array. Stores `e` bits, which is the output after sub-block interleaving, bit collection and pruning as per 3GPP technical specification 36.212 section 5.1.4.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.5.12 armral_turbo_rate_matching_noalloc_buffer_size

Calculates the required buffer size in bytes needed to perform Turbo encoding rate matching for an encoded message of length `d` and a rate-matched message of length `e`.

Syntax

Defined in `armral.h` on line `4586`:

```
uint32_t armral_turbo_rate_matching_noalloc_buffer_size(uint32_t d, uint32_t e,
                                                         uint32_t rv);
```

Returns

The required buffer size in bytes.

Parameters

d

A read-only parameter of type `uint32_t`.

The number of bits in the encoded message.

e

A read-only parameter of type `uint32_t`.

The number of bits in the rate-matched message.

rv

A read-only parameter of type `uint32_t`.

The redundancy version number for the transmission.

3.4.5.13 armral_turbo_rate_recovery

Recovers the log-likelihood ratios (LLRs) from demodulation to match the length of a Turbo encoded code block. This is the inverse of the operations for rate matching for Turbo described in the 3GPP Technical Specification (TS) 36.212 section 5.1.4.1.

The destination arrays `dst0`, `dst1`, and `dst2` also serve as input arrays. On input, they contain the current approximation to LLRs. The LLRs calculated from the rate-recovery are summed with existing LLRs in the destination arrays. The LLRs are expected to be zero the first time rate recovery is performed. Using the output from rate recovery as input for another call to rate recovery with a different redundancy version allows for data from multiple redundancy versions to be combined.

Syntax

Defined in `armral.h` on line `4628`:

```
armral_status armral_turbo_rate_recovery(uint32_t d, uint32_t e, uint32_t rv,
```

```
                                             const int8_t *src, int8_t *dst0,
                                             int8_t *dst1, int8_t *dst2);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`d`

A read-only parameter of type `uint32_t`.

The number of recovered 8-bits LLRs.

`e`

A read-only parameter of type `uint32_t`.

The number of demodulated 8-bit LLRs.

`rv`

A read-only parameter of type `uint32_t`.

The redundancy version number for the transmission

`src`

A read-only parameter of type `const int8_t *`.

Input array of a total of `e` 8-bit LLRs. This is the output after demodulation.

`dst0`

A parameter of type `int8_t *`.

On entry, contains the current approximation to LLRs for the systematic output of Turbo encoding. If no approximation of the LLRs is known, all entries must be set to zero. The array has length `d`. On exit, contains updated rate-recovered 8-bit LLRs, which are ready to be passed to decoding.

`dst1`

A parameter of type `int8_t *`.

On entry, contains the current approximation to LLRs for the parity output of Turbo encoding. If no approximation of the LLRs is known, all entries must be set to zero. The array has length `d`. On exit, contains updated rate-recovered 8-bit LLRs, which are ready to be passed to decoding.

`dst2`

A parameter of type `int8_t *`.

On entry, contains the current approximation to LLRs for the interleaved parity output of Turbo encoding. If no approximation of the LLRs is known, all entries must be set to zero. The array has length `d`. On exit, contains updated rate-recovered 8-bit LLRs, which are ready to be passed to decoding.

3.4.5.14 armral_turbo_rate_recovery_noalloc

Non-allocating variant of armral_turbo_rate_recovery.

This function recovers the log-likelihood ratios (LLRs) from demodulation to match the length of a Turbo encoded code block. This is the inverse of the operations for rate matching for Turbo described in the 3GPP Technical Specification (TS) 36.212 section 5.1.4.1.

The destination arrays `dst0`, `dst1`, and `dst2` also serve as input arrays. On input, they contain the current approximation to LLRs. The LLRs calculated from the rate-recovery are summed with existing LLRs in the destination arrays. The LLRs are expected to be zero the first time rate recovery is performed. Using the output from rate recovery as input for another call to rate recovery with a different redundancy version allows for data from multiple redundancy versions to be combined.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_turbo_rate_recovery_noalloc_buffer_size with identical inputs.

Syntax

Defined in `armral.h` on line `4681`:

```
armral_status armral_turbo_rate_recovery_noalloc(uint32_t d, uint32_t e,
                                                 uint32_t rv, const int8_t *src,
                                                 int8_t *dst0, int8_t *dst1,
                                                 int8_t *dst2, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`d`

A read-only parameter of type `uint32_t`.

The number of recovered 8-bits LLRs.

`e`

A read-only parameter of type `uint32_t`.

The number of demodulated 8-bit LLRs.

`rv`

A read-only parameter of type `uint32_t`.

The redundancy version number for the transmission

src

A read-only parameter of type `const int8_t *`.

Input array of a total of `e` 8-bit LLRs. This is the output after demodulation.

dst0

A parameter of type `int8_t *`.

On entry, contains the current approximation to LLRs for the systematic output of Turbo encoding. If no approximation of the LLRs is known, all entries must be set to zero. The array has length `d`. On exit, contains updated rate-recovered 8-bit LLRs, which are ready to be passed to decoding.

dst1

A parameter of type `int8_t *`.

On entry, contains the current approximation to LLRs for the parity output of Turbo encoding. If no approximation of the LLRs is known, all entries must be set to zero. The array has length `d`. On exit, contains updated rate-recovered 8-bit LLRs, which are ready to be passed to decoding.

dst2

A parameter of type `int8_t *`.

On entry, contains the current approximation to LLRs for the interleaved parity output of Turbo encoding. If no approximation of the LLRs is known, all entries must be set to zero. The array has length `d`. On exit, contains updated rate-recovered 8-bit LLRs, which are ready to be passed to decoding.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.5.15 armral_turbo_rate_recovery_noalloc_buffer_size

Calculates the required buffer size in bytes required to perform Turbo rate recovery for `d` recovered LLRs and `e` demodulated LLRs.

Syntax

Defined in `armral.h` on line `4695`:

```
uint32_t armral_turbo_rate_recovery_noalloc_buffer_size(uint32_t d, uint32_t e,
                                                         uint32_t rv);
```

Returns

The required buffer size in bytes.

Parameters

d

A read-only parameter of type `uint32_t`.

The number of recovered 8-bits LLRs.

e

A read-only parameter of type `uint32_t`.

The number of demodulated 8-bit LLRs.

rv

A read-only parameter of type `uint32_t`.

The redundancy version number for the transmission

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.

3.4.6.1 armral_tail_biting_convolutional_encode_block

This function implements the LTE tail biting convolutional encoding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input an array `src` of length `k` bits. The outputs of the encoding are written into the three arrays `dst0`, `dst1`, and `dst2` (the coding rate is equal to 1/3), each of which contains `k` bits. The constraint length of the encoder is 7, hence it makes use of a shift register of 6 bits. The generator polynomials are:

- `g0 = 1 0 1 1 0 1 1`
- `g1 = 1 1 1 1 0 0 1`
- `g2 = 1 1 1 0 1 0 1`

The encoding is performed for a single code block.

Syntax

Defined in `armral.h` on line `4737`:

```
armral_status armral_tail_biting_convolutional_encode_block(const uint8_t *src,
                                                            uint32_t k,
                                                            uint8_t *dst0,
                                                            uint8_t *dst1,
                                                            uint8_t *dst2);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`src`

A read-only parameter of type `const uint8_t *`.

Input data of length `k` bits.

`k`

A read-only parameter of type `uint32_t`.

Length of the input code block in bits. `k` is assumed to be greater or equal than 8.

`dst0`

A write-only parameter of type `uint8_t *`.

The first output stream of length `k` bits.

`dst1`

A write-only parameter of type `uint8_t *`.

The second output stream of length `k` bits.

`dst2`

A write-only parameter of type `uint8_t *`.

The third output stream of length `k` bits.

3.4.6.2 armral_tail_biting_convolutional_decode_block

This function implements the Wrap Around Viterbi Algorithm (WAVA) to decode the output of the LTE tail biting convolutional coding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input three arrays containing the log-likelihood ratios (LLRs) of the three encoded streams of bits. The decoding is performed for a single code block. WAVA is 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 output is written into the array `dst`, which must contain enough bytes to store `k` bits. These are hard outputs (that is, either 0 or 1).

Syntax

Defined in `armral.h` on line `4768`:

```
armral_status armral_tail_biting_convolutional_decode_block(
    const int8_t *src0, const int8_t *src1, const int8_t *src2, uint32_t k,
    uint32_t iter_max, uint8_t *dst);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`src0`

A read-only parameter of type `const int8_t *`.

The first input of length `k` bytes representing 8-bit log-likelihood ratios.

`src1`

A read-only parameter of type `const int8_t *`.

The second input of length `k` bytes representing 8-bit log-likelihood ratios.

`src2`

A read-only parameter of type `const int8_t *`.

The third input of length `k` bytes representing 8-bit log-likelihood ratios.

`k`

A read-only parameter of type `uint32_t`.

Length of the output code block in bits.

`iter_max`

A read-only parameter of type `uint32_t`.

Maximum number of iterations.

`dst`

A write-only parameter of type `uint8_t *`.

Decoded output data of length `k` bits.

3.4.6.3 armral_tail_biting_convolutional_decode_block_noalloc

Non-allocating variant of armral_tail_biting_convolutional_decode_block .

This function implements the Wrap Around Viterbi Algorithm (WAVA) to decode the output of the LTE tail biting convolutional coding scheme described in 3GPP Technical Specification (TS) 36.212 "Multiplexing and channel coding". It takes as input three arrays containing the log-likelihood ratios (LLRs) of the three encoded streams of bits. The decoding is performed for a single code block. WAVA is 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.

This function takes a pre-allocated buffer (`buffer`) to use internally. This variant will not call any system memory allocators.

The buffer must be at least as large as the number of bytes returned by calling armral_tail_biting_convolutional_decode_block_noalloc_buffer_size with identical inputs.

Syntax

Defined in `armral.h` on line `4806`:

```
armral_status armral_tail_biting_convolutional_decode_block_noalloc(
    const int8_t *src0, const int8_t *src1, const int8_t *src2, uint32_t k,
    uint32_t iter_max, uint8_t *dst, void *buffer);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

src0

A read-only parameter of type `const int8_t *`.

The first input of length `k` bytes representing 8-bit log-likelihood ratios.

src1

A read-only parameter of type `const int8_t *`.

The second input of length `k` bytes representing 8-bit log-likelihood ratios.

src2

A read-only parameter of type `const int8_t *`.

The third input of length `k` bytes representing 8-bit log-likelihood ratios.

k

A read-only parameter of type `uint32_t`.

Length of the output code block in bits.

iter_max

A read-only parameter of type `uint32_t`.

Maximum number of iterations.

dst

A write-only parameter of type `uint8_t *`.

Decoded output data of length `k` bits.

buffer

A read-only parameter of type `void *`.

Workspace buffer to be used internally.

3.4.6.4 armral_tail_biting_convolutional_decode_block_noalloc_buffer_size

Calculates the required buffer size in bytes to decode the output of the tail biting convolutional coding scheme for an output code block of length `k` bits.

Syntax

Defined in `armral.h` on line `4819`:

```
uint32_t armral_tail_biting_convolutional_decode_block_noalloc_buffer_size(
    uint32_t k, uint32_t iter_max);
```

Returns

The required buffer size in bytes.

Parameters

`k`

A read-only parameter of type `uint32_t`.

Length of the output code block in bits.

`iter_max`

A read-only parameter of type `uint32_t`.

Maximum number of iterations.

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.

3.5.1.1 armral_mu_law_compr_8bit

The Mu-Law compression method combines a bit-shift operation for dynamic range with a nonlinear piece-wise approximation of the original logarithmic Mu-Law. The Mu-Law compression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 12 complex output samples, each 8 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2142`:

```
armral_status armral_mu_law_compr_8bit(uint32_t n_prb,
                                       const armral_cmplx_int16_t *src,
                                       armral_compressed_data_8bit *dst,
                                       const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_8bit *`.

Points to the output 8-bit data and exponent.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.1.2 armral_mu_law_compr_9bit

The Mu-Law compression method combines a bit-shift operation for dynamic range with a nonlinear piece-wise approximation of the original logarithmic Mu-Law. The Mu-Law compression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 12 complex output samples, each 9 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2164`:

```
armral_status armral_mu_law_compr_9bit(uint32_t n_prb,
                                       const armral_cmplx_int16_t *src,
                                       armral_compressed_data_9bit *dst,
                                       const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_9bit *`.

Points to the output 9-bit data and shift.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.1.3 armral_mu_law_compr_14bit

The Mu-Law compression method combines a bit-shift operation for dynamic range with a nonlinear piece-wise approximation of the original logarithmic Mu-Law. The Mu-Law compression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 12 complex output samples, each 14 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2185`:

```
armral_status armral_mu_law_compr_14bit(uint32_t n_prb,
                                        const armral_cmplx_int16_t *src,
                                        armral_compressed_data_14bit *dst,
                                        const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_14bit *`.

Points to the output 14-bit data and shift.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.1.4 armral_mu_law_decompr_8bit

The Mu-Law decompression method is a logical reverse function of the compression method. The Mu-Law decompression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 8-bit complex resource elements. Each block taken as input is expanded into 12 complex output samples, each 16 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values.

Syntax

Defined in `armral.h` on line `2206`:

```
armral_status armral_mu_law_decompr_8bit(uint32_t n_prb,
                                         const armral_compressed_data_8bit *src,
                                         armral_cmplx_int16_t *dst,
                                         const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_8bit *`.

Points to the input 8-bit data and shift.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output complex samples sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.1.5 armral_mu_law_decompr_9bit

The Mu-Law decompression method is a logical reverse function of the compression method. The Mu-Law decompression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 9-bit complex resource elements. Each block taken as input is expanded into 12 complex output samples, each 16 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values.

Syntax

Defined in `armral.h` on line `2227`:

```
armral_status armral_mu_law_decompr_9bit(uint32_t n_prb,
                                         const armral_compressed_data_9bit *src,
                                         armral_cmplx_int16_t *dst,
                                         const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_9bit *`.

Points to the input 9-bit data and shift.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output complex samples sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.1.6 armral_mu_law_decompr_14bit

The Mu-Law decompression method is a logical reverse function of the compression method. The Mu-Law decompression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 14-bit complex resource elements. Each block taken as input is expanded into 12 complex output samples, each 16 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values.

Syntax

Defined in `armral.h` on line `2248`:

```
armral_status armral_mu_law_decompr_14bit(
    uint32_t n_prb, const armral_compressed_data_14bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_14bit *`.

Points to the input 14-bit data and shift.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output complex samples sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.2 Block Scaling Compression

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

3.5.2.1 armral_block_scaling_compr_8bit

The function operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 8-bit post-scaled samples and a common unsigned scaling factor.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2280`:

```
armral_status
armral_block_scaling_compr_8bit(uint32_t n_prb, const armral_cmplx_int16_t *src,
                                armral_compressed_data_8bit *dst,
                                const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_8bit *`.

Points to the output 8-bit data and a scaling factor.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.2.2 armral_block_scaling_compr_9bit

The function operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 9-bit post-scaled samples and a common unsigned scaling factor.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2301`:

```
armral_status
armral_block_scaling_compr_9bit(uint32_t n_prb, const armral_cmplx_int16_t *src,
                                armral_compressed_data_9bit *dst,
                                const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_9bit *`.

Points to the output 9-bit data and a scaling factor.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.2.3 armral_block_scaling_compr_14bit

The function operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 14-bit post-scaled samples and a common unsigned scaling factor.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2321`:

```
armral_status armral_block_scaling_compr_14bit(
    uint32_t n_prb, const armral_cmplx_int16_t *src,
    armral_compressed_data_14bit *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_14bit *`.

Points to the output 14-bit data and a scaling factor.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.2.4 armral_block_scaling_decompr_8bit

The function operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 8-bit complex post-scaled resource elements and an unsigned scaling factor. Each block taken as input is expanded into 12 16-bit complex samples.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2340`:

```
armral_status armral_block_scaling_decompr_8bit(
    uint32_t n_prb, const armral_compressed_data_8bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_8bit *`.

Points to the input 8-bit data and scaling factor.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output complex samples sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.2.5 armral_block_scaling_decompr_9bit

The function operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 9-bit complex post-scaled resource elements and an unsigned scaling factor. Each block taken as input is expanded into 12 16-bit complex samples.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2359`:

```
armral_status armral_block_scaling_decompr_9bit(
    uint32_t n_prb, const armral_compressed_data_9bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_9bit *`.

Points to the input 9-bit data and a scaling factor.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output complex samples sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.2.6 armral_block_scaling_decompr_14bit

The function operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 14-bit complex post-scaled resource elements and an unsigned scaling factor. Each block taken as input is expanded into 12 16-bit complex samples.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2379`:

```
armral_status armral_block_scaling_decompr_14bit(
    uint32_t n_prb, const armral_compressed_data_14bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_14bit *`.

Points to the input 14-bit data and a scaling factor.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the output complex samples sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.3 Block Floating Point

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

3.5.3.1 armral_block_float_compr_8bit

Block floating-point compression to 8-bit.

The function operates on a fixed block size of one Resource Block (RB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 8-bit samples and one unsigned exponent.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2410`:

```
armral_status armral_block_float_compr_8bit(uint32_t n_prb,
                                            const armral_cmplx_int16_t *src,
                                            armral_compressed_data_8bit *dst,
                                            const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_8bit *`.

Points to the output 8-bit data and exponent.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.3.2 armral_block_float_compr_9bit

Block floating point compression to 9-bit big-endian.

The function operates on a fixed block size of one Resource Block (RB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 9-bit big-endian samples and one unsigned exponent. Big-endian means that where data from a 9-bit element is split across multiple bytes, the most significant bits are stored in the output byte with lowest address, and remaining bits are stored in the high bits of the next output byte.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2435`:

```
armral_status armral_block_float_compr_9bit(uint32_t n_prb,
                                            const armral_cmplx_int16_t *src,
                                            armral_compressed_data_9bit *dst,
                                            const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_9bit *`.

Points to the output 9-bit data and exponent.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.3.3 armral_block_float_compr_12bit

Block floating point compression to 12-bit big-endian.

The function operates on a fixed block size of one Resource Block (RB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 12-bit big-endian samples and one unsigned exponent. Big-endian means that where data from a 12-bit element is split across multiple bytes, the most significant bits are stored in the output byte with lowest address, and remaining bits are stored in the high bits of the next output byte.

Syntax

Defined in `armral.h` on line `2457`:

```
armral_status armral_block_float_compr_12bit(uint32_t n_prb,
                                             const armral_cmplx_int16_t *src,
                                             armral_compressed_data_12bit *dst,
                                             const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_12bit *`.

Points to the output 12-bit data and exponent.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.3.4 armral_block_float_compr_14bit

Block floating point compression to 14-bit big-endian.

The function operates on a fixed block size of one Resource Block (RB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 14-bit big-endian samples and one unsigned exponent. Big-endian means that where data from a 14-bit element is split across multiple bytes, the most significant bits are stored in the output byte with lowest address, and remaining bits are stored in the high bits of the next output byte.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2482`:

```
armral_status armral_block_float_compr_14bit(uint32_t n_prb,
                                             const armral_cmplx_int16_t *src,
                                             armral_compressed_data_14bit *dst,
                                             const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_cmplx_int16_t *`.

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_14bit *`.

Points to the output 14-bit data and exponent.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.3.5 armral_block_float_decompr_8bit

Block floating-point decompression from 8 bit.

The function operates on a fixed block size of one Resource Block (RB). Each block consists of 12 8-bit complex resource elements and an unsigned exponent. Each block taken as input is expanded into 12 16-bit complex samples.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2504`:

```
armral_status armral_block_float_decompr_8bit(
    uint32_t n_prb, const armral_compressed_data_8bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_8bit *`.

Points to the input compressed block sequence.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the complex output sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.3.6 armral_block_float_decompr_9bit

Block floating point decompression from 9 bit big-endian.

The function operates on a fixed block size of one Resource Block (RB). Each block consists of 12 9-bit big-endian complex resource elements and an unsigned exponent. Each block taken as input is expanded into 12 16-bit complex samples. Big-endian here means that where data from a 9-bit element is split across multiple bytes, the most significant bits are stored in the output byte with lowest address, and remaining bits are stored in the high bits of the next output byte.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2528`:

```
armral_status armral_block_float_decompr_9bit(
    uint32_t n_prb, const armral_compressed_data_9bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_9bit *`.

Points to the input compressed block sequence.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the complex output sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.3.7 armral_block_float_decompr_12bit

Block floating point decompression from 12 bit big-endian.

The function operates on a fixed block size of one Resource Block (RB). Each block consists of 12 12-bit big-endian complex resource elements and an unsigned exponent. Each block taken as input is expanded into 12 16-bit complex samples. Big-endian here means that where data from a 12-bit element is split across multiple bytes, the most significant bits are stored in the output byte with lowest address, and remaining bits are stored in the high bits of the next output byte.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2552`:

```
armral_status armral_block_float_decompr_12bit(
    uint32_t n_prb, const armral_compressed_data_12bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_12bit *`.

Points to the input compressed block sequence.

dst

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the complex output sequence.

scale

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

3.5.3.8 armral_block_float_decompr_14bit

Block floating point decompression from 14 bit big-endian.

The function operates on a fixed block size of one Resource Block (RB). Each block consists of 12 14-bit big-endian complex resource elements and an unsigned exponent. Each block taken as input is expanded into 12 16-bit complex samples. Big-endian here means that where data from a 14-bit element is split across multiple bytes, the most significant bits are stored in the output byte with lowest address, and remaining bits are stored in the high bits of the next output byte.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2576`:

```
armral_status armral_block_float_decompr_14bit(
    uint32_t n_prb, const armral_compressed_data_14bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

`n_prb`

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

`src`

A read-only parameter of type `const armral_compressed_data_14bit *`.

Points to the input compressed block sequence.

`dst`

A write-only parameter of type `armral_cmplx_int16_t *`.

Points to the complex output sequence.

`scale`

A read-only parameter of type `const armral_cmplx_int16_t *`.

Phase compensation term to use, or NULL.

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 `198`:

```
typedef struct {
  float32_t re; ///< 32-bit real component.
  float32_t 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 `190`:

```
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 `236`:

```
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 `247`:

```
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 `214`:

```
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 `225`:

```
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 `3424`:

```
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 `206`:

```
#define ARMRAL_NUM_COMPLEX_SAMPLES 12
```

5.2 ARMRAL_LDPC_DEFAULT_OPTIONS

Use the default options when doing LDPC decoding.

Option used by armral_ldpc_decode_block.

Implies ARMRAL_LDPC_CRC_NO and ARMRAL_LDPC_FILLER_BITS_EXPLICIT.

Syntax

Defined in `armral.h` on line `3465`:

```
#define ARMRAL_LDPC_DEFAULT_OPTIONS 0
```

5.3 ARMRAL_LDPC_CRC_NO

The LDPC decoder result should not be checked.

Option used by armral_ldpc_decode_block.

If set, no CRC check is executed and, ARMRAL_LDPC_CRC_EVERY_ITER and ARMRAL_LDPC_CRC_END_ITER are ignored.

`ARMRAL_LDPC_CRC_NO` is set by default.

Syntax

Defined in `armral.h` on line `3477`:

```
#define ARMRAL_LDPC_CRC_NO (1 << 1)
```

5.4 ARMRAL_LDPC_CRC_16

Check for convergence of LDPC decoding using CRC16.

Option used by armral_ldpc_decode_block.

Not currently supported.

Syntax

Defined in `armral.h` on line `3486`:

```
#define ARMRAL_LDPC_CRC_16 (1 << 2)
```

5.5 ARMRAL_LDPC_CRC_24A

Check for convergence of LDPC decoding using CRC24_A.

Option used by armral_ldpc_decode_block.

Not currently supported.

Syntax

Defined in `armral.h` on line `3495`:

```
#define ARMRAL_LDPC_CRC_24A (1 << 3)
```

5.6 ARMRAL_LDPC_CRC_24B

Check for convergence of LDPC decoding using CRC24_B.

Option used by armral_ldpc_decode_block.

If set, a CRC check will be performed during the decoding. The frequency of how often the CRC check is done is determined by ARMRAL_LDPC_CRC_EVERY_ITER and ARMRAL_LDPC_CRC_END_ITER. If the CRC check succeeds, armral_ldpc_decode_block will return

`ARMRAL_SUCCESS`. If the CRC check fails on the last iteration, armral_ldpc_decode_block will return `ARMRAL_FAIL`.

Syntax

Defined in `armral.h` on line `3509`:

```
#define ARMRAL_LDPC_CRC_24B (1 << 4)
```

5.7 ARMRAL_LDPC_CRC_EVERY_ITER

Check for convergence on every iteration of LDPC decoding.

Option used by armral_ldpc_decode_block.

If set, and ARMRAL_LDPC_CRC_NO is not set. A CRC check will be run on every iteration. If the CRC check succeeds, armral_ldpc_decode_block will return with `ARMRAL_SUCCESS`. If the max number of iterations is exceeded without a successful CRC check, `armral_ldpc_decode_block` will return `ARMRAL_FAIL`.

`ARMRAL_LDPC_CRC_EVERY_ITER` is set by default.

Syntax

Defined in `armral.h` on line `3524`:

```
#define ARMRAL_LDPC_CRC_EVERY_ITER (1 << 5)
```

5.8 ARMRAL_LDPC_CRC_END_ITER

Check for convergence only on the last iteration of LDPC decoding.

Option used by armral_ldpc_decode_block.

If set, and ARMRAL_LDPC_CRC_NO is not set. A CRC check will be run after armral_ldpc_decode_block has reached the max number of iterations. If the CRC check succeeds, `armral_ldpc_decode_block` returns `ARMRAL_SUCCESS`. If the CRC check fails, `armral_ldpc_decode_block` returns `ARMRAL_FAIL`.

Syntax

Defined in `armral.h` on line `3536`:

```
#define ARMRAL_LDPC_CRC_END_ITER (1 << 6)
```

5.9 ARMRAL_LDPC_FILLER_BITS_IMPLICIT

Assumes the LDPC decoding input has implicit filler bits.

Option used by armral_ldpc_decode_block.

If set, `len_filler_bits` LLRs corresponding to the filler bits will be inserted between the LLRs for the message bits and the parity bits before attempting to decode.

Syntax

Defined in `armral.h` on line `3547`:

```
#define ARMRAL_LDPC_FILLER_BITS_IMPLICIT (1 << 7)
```

5.10 ARMRAL_LDPC_FILLER_BITS_EXPLICIT

Assumes the LDPC decoding input has explicit filler bits.

Option used by armral_ldpc_decode_block.

If set, it is assumed that `len_filler_bits` LLRs corresponding to the filler bits have been inserted between the LLRs for the message bits and the parity bits. This is the format of the output LLRs from armral_ldpc_rate_recovery.

`ARMRAL_LDPC_FILLER_BITS_EXPLICIT` is set by default.

Syntax

Defined in `armral.h` on line `3561`:

```
#define ARMRAL_LDPC_FILLER_BITS_EXPLICIT (1 << 8)
```

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 `107`:

```
typedef enum {
  ARMRAL_SUCCESS = 0,          ///< No error.
  ARMRAL_ARGUMENT_ERROR = -1,  ///< One or more arguments are incorrect.
  ARMRAL_FAIL = -2,            ///< A failure has been detected.
} 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 `117`:

```
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 `128`:

```
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 `173`:

```
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 `182`:

```
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.

Syntax

Defined in `armral.h` on line `3209`:

```
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 `3414`:

```
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.

Syntax

Defined in `armral.h` on line `3204`:

```
typedef struct armral_fft_plan_t armral_fft_plan_t;
```