



Arm RAN Acceleration Library

Revision:21.07

Reference Manual

Copyright © 2020-2021 Arm Limited (or its affiliates). All rights reserved.

Issue 00

102249_2107_00_en

1 Arm RAN Acceleration Library (ArmRAL) Reference Manual	1
1.1 About this book	1
1.2 Feedback	1
1.2.1 Feedback on this product	1
1.2.2 Feedback on content	2
1.3 Non-Confidential Proprietary Notice	2
1.4 Confidentiality Status	3
1.5 Product Status	3
1.6 Web Address	3
1.7 Progressive terminology commitment	3
1.8 Release Information	3
1.8.1 Document History	3
2 Get started with Arm RAN Acceleration Library (ArmRAL)	5
2.1 Prerequisites	5
2.2 Configure your build environment	6
2.3 Build Arm RAN Acceleration Library	6
2.4 Install Arm RAN Acceleration Library	8
2.5 Run the library tests	8
2.6 Run the benchmarks	9
2.7 Run the library examples	9
2.8 Documentation	10
2.9 Code coverage	10
2.10 Uninstall Arm RAN Acceleration Library	10
3 Link to Arm RAN Acceleration Library (ArmRAL)	11
3.1 Before you begin	11
3.2 Procedure	11
3.3 Example: Run 'fft_cf32_example.c'	12
3.4 Related information	13
4 Function Index	15
4.1 Functions	15
5 Function Documentation	17
5.1 Vector functions	17
5.1.1 Detailed Description	17
5.2 Matrix functions	18
5.2.1 Detailed Description	18
5.3 Lower PHY support functions	19
5.3.1 Detailed Description	19
5.4 Upper PHY support functions	20
5.4.1 Detailed Description	20

5.5 DU-RU IF support functions	21
5.5.1 Detailed Description	21
5.6 Vector Multiply	22
5.6.1 Detailed Description	22
5.6.2 Function Documentation	22
5.6.2.1 armral_cmplx_vecmul_f32()	22
5.6.2.2 armral_cmplx_vecmul_f32_2()	23
5.6.2.3 armral_cmplx_vecmul_i16()	24
5.6.2.4 armral_cmplx_vecmul_i16_2()	25
5.7 Vector Dot Product	27
5.7.1 Detailed Description	27
5.7.2 Function Documentation	27
5.7.2.1 armral_cmplx_vecdot_f32()	27
5.7.2.2 armral_cmplx_vecdot_f32_2()	28
5.7.2.3 armral_cmplx_vecdot_i16()	28
5.7.2.4 armral_cmplx_vecdot_i16_2()	29
5.7.2.5 armral_cmplx_vecdot_i16_2_32bit()	30
5.7.2.6 armral_cmplx_vecdot_i16_32bit()	30
5.8 Complex Matrix Multiplication	32
5.8.1 Detailed Description	32
5.8.2 Function Documentation	32
5.8.2.1 armral_cmplx_mat_mult_2x2_f32()	33
5.8.2.2 armral_cmplx_mat_mult_2x2_f32_iq()	33
5.8.2.3 armral_cmplx_mat_mult_4x4_f32()	34
5.8.2.4 armral_cmplx_mat_mult_4x4_f32_iq()	35
5.8.2.5 armral_cmplx_mat_mult_f32()	35
5.8.2.6 armral_cmplx_mat_mult_i16()	36
5.8.2.7 armral_cmplx_mat_mult_i16_32bit()	37
5.8.2.8 armral_solve_1x2_f32()	37
5.8.2.9 armral_solve_1x4_f32()	38
5.8.2.10 armral_solve_2x2_f32()	39
5.8.2.11 armral_solve_2x4_f32()	40
5.8.2.12 armral_solve_4x4_f32()	41
5.9 Complex Matrix Inversion	43
5.9.1 Detailed Description	43
5.9.2 Function Documentation	43
5.9.2.1 armral_cmplx_hermitian_mat_inverse_f32()	43
5.10 Sequence Generator	44
5.10.1 Detailed Description	44
5.10.2 Function Documentation	44
5.10.2.1 armral_seq_generator()	44

5.11 Modulation	46
5.11.1 Detailed Description	46
5.11.2 Function Documentation	46
5.11.2.1 armral_demodulation()	46
5.11.2.2 armral_modulation()	47
5.12 Correlation Coefficient	48
5.12.1 Detailed Description	48
5.12.2 Function Documentation	48
5.12.2.1 armral_corr_coeff_i16()	48
5.13 FIR filter	49
5.13.1 Detailed Description	49
5.13.2 Function Documentation	49
5.13.2.1 armral_fir_filter_cf32()	49
5.13.2.2 armral_fir_filter_cf32_decimate_2()	50
5.13.2.3 armral_fir_filter_cs16()	50
5.13.2.4 armral_fir_filter_cs16_decimate_2()	52
5.14 Mu-Law Compression	53
5.14.1 Detailed Description	53
5.14.2 Function Documentation	53
5.14.2.1 armral_mu_law_compr_8bit()	53
5.14.2.2 armral_mu_law_decompr_8bit()	53
5.15 Block Floating Point	55
5.15.1 Detailed Description	55
5.15.2 Function Documentation	55
5.15.2.1 armral_block_float_compr_12bit()	55
5.15.2.2 armral_block_float_compr_14bit()	56
5.15.2.3 armral_block_float_compr_8bit()	56
5.15.2.4 armral_block_float_compr_9bit()	57
5.15.2.5 armral_block_float_decompr_12bit()	57
5.15.2.6 armral_block_float_decompr_14bit()	58
5.15.2.7 armral_block_float_decompr_8bit()	58
5.15.2.8 armral_block_float_decompr_9bit()	59
5.16 CRC24	60
5.16.1 Detailed Description	60
5.16.2 Function Documentation	60
5.16.2.1 armral_crc24_a_be()	60
5.16.2.2 armral_crc24_a_le()	61
5.16.2.3 armral_crc24_b_be()	61
5.16.2.4 armral_crc24_b_le()	62
5.16.2.5 armral_crc24_c_be()	62
5.16.2.6 armral_crc24_c_le()	62

5.17 Polar Encoding	64
5.17.1 Detailed Description	64
5.17.2 Function Documentation	64
5.17.2.1 armral_polar_decoder()	64
5.17.2.2 armral_polar_encoder()	65
5.18 Fast Fourier Transforms (FFT)	66
5.18.1 Detailed Description	66
5.18.2 Typedef Documentation	66
5.18.2.1 armral_fft_plan_t	67
5.18.3 Enumeration Type Documentation	67
5.18.3.1 armral_fft_direction_t	67
5.18.4 Function Documentation	67
5.18.4.1 armral_fft_create_plan_cf32()	67
5.18.4.2 armral_fft_create_plan_cs16()	68
5.18.4.3 armral_fft_destroy_plan_cf32()	68
5.18.4.4 armral_fft_destroy_plan_cs16()	69
5.18.4.5 armral_fft_execute_cf32()	69
5.18.4.6 armral_fft_execute_cs16()	70
6 Data Structure Index	71
6.1 Data Structures	71
7 Data Structure Documentation	73
7.1 armral_cmplx_f32_t Struct Reference	73
7.1.1 Field Documentation	73
7.1.1.1 im	73
7.1.1.2 re	73
7.2 armral_cmplx_int16_t Struct Reference	74
7.2.1 Field Documentation	74
7.2.1.1 im	74
7.2.1.2 re	74
7.3 armral_compressed_data_12bit Struct Reference	74
7.3.1 Detailed Description	75
7.3.2 Field Documentation	75
7.3.2.1 exp	75
7.3.2.2 mantissa	75
7.4 armral_compressed_data_14bit Struct Reference	75
7.4.1 Detailed Description	75
7.4.2 Field Documentation	75
7.4.2.1 exp	76
7.4.2.2 mantissa	76
7.5 armral_compressed_data_8bit Struct Reference	76

7.5.1 Detailed Description	76
7.5.2 Field Documentation	76
7.5.2.1 exp	76
7.5.2.2 mantissa	77
7.6 armral_compressed_data_9bit Struct Reference	77
7.6.1 Detailed Description	77
7.6.2 Field Documentation	77
7.6.2.1 exp	77
7.6.2.2 mantissa	77
8 File Index	79
8.1 File List	79
9 File Documentation	81
9.1 armral.h File Reference	81
9.1.1 Macro Definition Documentation	84
9.1.1.1 ARMRAL_NUM_COMPLEX_SAMPLES	84
9.1.2 Enumeration Type Documentation	84
9.1.2.1 armral_fixed_point_index	84
9.1.2.2 armral_modulation_type	85
9.1.2.3 armral_status	85

Chapter 1

Arm RAN Acceleration Library (ArmRAL) Reference Manual

Copyright © 2020-2021 Arm Limited (or its affiliates). All rights reserved.

1.1 About this book

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

Arm RAN Acceleration Library contains a set of functions for accelerating telecommunications applications such as, but not limited to, 5G Radio Access Networks (RANs).

Arm RAN Acceleration Library is built as a static library, and must be linked in to any executable that needs to use the library. The source code can be built and modified by users to integrate with their components or clients. Header files are located in the `include` directory, the source code is located in the `src` directory, testing and benchmarking code is located in the `test` directory, and examples are located in the `examples` directory.

1.2 Feedback

1.2.1 Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

1.2.2 Feedback on content

If you have any comments on content, send an e-mail to errata@arm.com. Give:

- The title Arm RAN Acceleration Library Reference Manual.
- The number 102249_2107_00_en.
- If applicable, the relevant page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1.3 Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2020-2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

1.4 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.

1.5 Product Status

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

1.6 Web Address

<https://developer.arm.com>

1.7 Progressive terminology commitment

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

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact terms@arm.com.

1.8 Release Information

1.8.1 Document History

Issue	Date	Confidentiality	Change
2010-00	02 Oct 2020	Non-Confidential	New document for Arm RAN Acceleration Library v20.10
2101-00	08 Jan 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.01
2104-00	09 Apr 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.04
2107-00	09 Jul 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.07

Chapter 2

Get started with Arm RAN Acceleration Library (ArmRAL)

The Arm RAN Acceleration Library (ArmRAL) contains a set of functions for accelerating telecommunications applications such as, but not limited to, 5G Radio Access Networks (RANs).

2.1 Prerequisites

Arm RAN Acceleration Library runs on most AArch64 cores, however some parts of the library have more strict requirements:

- To use the Cyclic Redundancy Check (CRC) functions, you must run 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.

To build the library, you must have the following software already installed:

- A recent version of a C/C++ compiler, such as GCC. Arm RAN Acceleration Library has been tested with GCC 7.5.0, 8.2.0, 9.3.0, 10.2.0, and 11.1.0.
Note: If you are cross-compiling, you need a cross-toolchain compiler that targets AArch64. You can download open-source cross-toolchain builds of the GCC compiler on the [Arm Developer website](#). The variant to use for an AArch64 GNU/Linux target is 'aarch64-none-linux-gnu'.
- A recent version of CMake (version 3.0.0, or higher).

In addition to the preceding requirements:

- To run the benchmarks, you must have the Linux utility tool `perf` installed and a recent version of Python 3. Arm RAN Acceleration Library has been tested with Python 3.8.5.
- To build a local version of the documentation, you must have Doxygen installed. Arm RAN Acceleration Library has been tested with Doxygen version 1.8.13.
- To generate code coverage HTML pages, you must have `gcovr` installed. The library has been tested with `gcovr` version 4.2.

2.2 Configure your build 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 SVE2 version of the library, you must compile with GCC 11.1.0.

2.3 Build Arm RAN Acceleration Library

You must build the library before you can use it in your application development.

To build the library, use the following commands:

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

Substituting:

- `<build>` with a build directory name. The library is built in the specified directory.
- `[options]` with the CMake options to use to build the library.
- (Optional) `<install-dir>` with an installation directory name. When you install the library (see the next section), the library installs to the specified directory. If `<install-dir>` is not specified, the default is `/usr/local`.
- `<path>` with the path to the root directory of the library source.

Notes:

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

Common CMake `[options]` include:

- `-DCMAKE_INSTALL_PREFIX=<path>`
Specifies the base directory used to install the library. The library archive is installed to `<path>/lib` and headers are installed to `<path>/include`.
Default `<path>` is `/usr/local`.
- `-DCMAKE_BUILD_TYPE=Debug|Release`
Specifies the set of flags used to build the library. The default is `Release` which gives the optimal performance, however `Debug` might give a superior debugging experience. To optimize the performance of `Release` builds, assertions are disabled. Assertions are enabled in `Debug` builds.
Default is `Release`.
- `-DCMAKE_C_COMPILER=<name>`
Specifies the executable to use as the C compiler. If a compiler is not specified, the compiler used defaults to the contents of the `CC` environment variable. If neither are set, CMake attempts to use the generic system compiler `cc`. If `<name>` is not an absolute path, it must be findable in your current environment `PATH`.
- `-DCMAKE_CXX_COMPILER=<name>`
Specifies the executable to use as the C++ compiler. If a compiler is not specified, the compiler used defaults to the contents of the `CXX` environment variable. If neither are set, CMake attempts to use the generic system compiler `c++`. If `<name>` is not an absolute path, it must be findable in your current environment `PATH`.
- `-DBUILD_TESTING=On|Off`
Specifies whether to build (`On`), or not build, (`Off`) the correctness tests and benchmarking code for the library. `-DBUILD_TESTING=On` enables the `check` and `bench` targets described later. If after you build the library, you want to run the included tests and benchmarks, you must build your library with `-DBUILD_TESTING=On`.
Default is `Off`.
- `-DARMRAL_TEST_RUNNER=<command>`
Specifies a command that is used as a prefix before each test executable, such as where an emulator might be required. To see an example where `-DARMRAL_TEST_RUNNER` is used, see the "Run the library tests" section.
- `-DSTATIC_TESTING=On|Off`
Most C/C++ toolchains dynamically link to system libraries like `libc.so`, however this dynamic link is unsuitable or unsupported in some use cases. Setting `-DSTATIC_TESTING=On` forces the compiler to link the tests statically by appending the `-static` flag to the link line.
Default is `Off`.
- `-DBUILD_EXAMPLES=On|Off`
Specifies whether to build (`On`), or not build (`Off`), the examples in the examples folder. The example programs are simpler than the tests, and show how different parts of the library can be used. `-DBUILD_EXAMPLES=On` enables the `examples` and `run_examples` targets described later. If after you build the library, you want to run the included examples, you must build your library with `-DBUILD_EXAMPLES=On`.
Default is `Off`.
- `-DBUILD_SHARED_LIBS=On|Off`
Specifies whether to generate a shared library (`On`) or a static library (`Off`). To generate `libarmral.so`, use `-DBUILD_SHARED_LIBS=On`. To generate `libarmral.a`, use `-DBUILD_SHARED_LIBS=Off`.
Default is `Off`.

- `-DARMRAL_ENABLE_WERROR=On|Off`

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

Default is `Off`.

- `-DARMRAL_ENABLE_ASAN=On|Off`

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

Default is `Off`.

- `-DARMRAL_ENABLE_COVERAGE=On|Off`

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

Default is `Off`.

- `-DARMRAL_ARCH=<arch>`

Enable code that is optimized for `<arch>`. `<arch>` must be `NEON` or `SVE2`. To use `-DARMRAL_ARCH=SVE2`, you must use a compiler that supports `-march=armv8-a+sve2`.

Default is `NEON`.

2.4 Install Arm RAN Acceleration Library

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

1. Ensure you have write access for the installation directories:

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

2. Install the library. Run:

```
make install
```

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

2.5 Run the library tests

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

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

To build and run the tests, use:

```
make check
```

The tests test all the available functions in the library (building them if necessary, for example if you have not previously typed `make`). Testing times vary from system to system, but typically only take a few seconds.

If you are not developing on an AArch64 machine, or if you want to test the SVE2 version of the library on an AArch64 machine that does not support the SVE2 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:

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

2.6 Run the benchmarks

All the functions in the library contain benchmarking code that contains preset problem sizes.

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

To build and run the benchmarks, use:

```
make bench
```

Benchmark results are printed as JSON objects, which can then be collected or piped into other scripts for further processing.

2.7 Run the library 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 your library with the `-DBUILD_EXAMPLES=On` CMake option.

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

```
make run_examples
```

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

```
make examples
```

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

2.8 Documentation

A PDF version of the Arm RAN Acceleration Library Reference Manual is available online at:

<https://developer.arm.com/documentation/102249/2107>

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

To build the documentation, run:

```
make docs
```

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

2.9 Code coverage

You can generate information that describes how much of library is used by your application, or is covered by the included tests. To collect code coverage information, you must have built the library with `-DARMRAL_ENABLE_COVERAGE=On`. An example workflow might look like:

```
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 'Prerequisites' section.

2.10 Uninstall Arm RAN Acceleration Library

To uninstall the library, navigate to the library build directory that you previously ran `'make install'` in, and 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. To only remove the installed files (but not any directories) run:

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

Chapter 3

Link to Arm RAN Acceleration Library (ArmRAL)

This topic describes how to compile and link to Arm RAN Acceleration Library (ArmRAL), and demonstrates this procedure with an example (`fft_cf32_example.c`).

3.1 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 the library before trying to run any of the example programs. To build the library, use:

```
tar zxvf arm-ran-acceleration-library-21.07-aarch64.tar.gz
mkdir arm-ran-acceleration-library-21.07/build
cd arm-ran-acceleration-library-21.07/build
cmake ..
make -j
```

To learn more about how to build the library, see the **README** file in the tar.gz package.

3.2 Procedure

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

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

2. Run your binary:

```
./<binary>
```

3.3 Example: Run 'fft_cf32_example.c'

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

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

```
/*
   Arm RAN Acceleration Library
   Copyright 2020-2021 Arm Limited (or its affiliates).
   All rights reserved.
*/
#include "armral.h"

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

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

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

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

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

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

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

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

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

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

    example_fft_plan_and_execute(n);
}
```

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

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

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

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

An executable called `fft_cf32_example` is built.

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

```
./fft_cf32_example 5
```

which gives:

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

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

3.4 Related information

For more information, see the **README** file.

Chapter 4

Function Index

4.1 Functions

The functions that Arm RAN Acceleration Library supports are:

Vector functions	17
Vector Multiply	22
Vector Dot Product	27
Matrix functions	18
Complex Matrix Multiplication	32
Complex Matrix Inversion	43
Lower PHY support functions	19
Sequence Generator	44
Correlation Coefficient	48
FIR filter	49
Fast Fourier Transforms (FFT)	66
Upper PHY support functions	20
Modulation	46
CRC24	60
Polar Encoding	64
DU-RU IF support functions	21
Mu-Law Compression	53
Block Floating Point	55

Chapter 5

Function Documentation

5.1 Vector functions

Functions for working with vectors.

Modules

- [Vector Multiply](#)
Multiplies a complex vector by another complex vector and generates a complex result.
- [Vector Dot Product](#)
Computes the dot product of two complex vectors.

5.1.1 Detailed Description

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

- Vector elementwise multiplication (vector multiply)
- Vector dot product

5.2 Matrix functions

Functions for working with matrices.

Modules

- [Complex Matrix Multiplication](#)
Computes a matrix-by-matrix multiplication, storing the result in a destination matrix.
- [Complex Matrix Inversion](#)
Computes the inverse of a complex hermitian square matrix of size NxN.

5.2.1 Detailed Description

Functions are provided for working with matrices, including:

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

5.3 Lower PHY support functions

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

Modules

- [Sequence Generator](#)
Fills a pointer with a Gold Sequence of the specified length, generated from the specified seed.
- [Correlation Coefficient](#)
Calculates Pearson's Correlation Coefficient from a pair of complex vectors.
- [FIR filter](#)
FIR filter implemented for single-precision floating-point and 16-bit signed integers.
- [Fast Fourier Transforms \(FFT\)](#)
Computes the Discrete Fourier Transform (DFT) of a sequence (forwards transform), or the inverse (backwards transform).

5.3.1 Detailed Description

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.

5.4 Upper PHY support functions

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

Modules

- [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.
- [CRC24](#)
Computes a 24-bit Cyclic Redundancy Check (CRC) of an input buffer using carry-less multiplication and Barret reduction.
- [Polar Encoding](#)
Polar codes are used to encode the Uplink Control Information (UCI) over the PUCCH and PUSCH, and the Downlink Control Information (DCI) over the PDCCH, in downlink.

5.4.1 Detailed Description

The Lower PHY functions include support for:

- Digital modulation and demodulation, using QPSK, 16QAM, 64QAM, or 256QAM.
- 24-bit Cyclic Redundancy Check 24 (CRC24), both little-endian and big-endian, for the three 5G polynomials.
- Polar encoding and decoding.

5.5 DU-RU IF support functions

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

Modules

- [Mu-Law Compression](#)

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

- [Block Floating Point](#)

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

5.5.1 Detailed Description

The DU-RU IF functions include support for:

- Mu-Law (both compression and decompression).
- Block floating-point compression and decompression, in 8-bit, 9-bit, 12-bit, and 14-bit formats.

5.6 Vector Multiply

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

Functions

- `armral_status armral_cmplx_vecmul_i16` (int32_t n, const `armral_cmplx_int16_t` *a, const `armral_cmplx_int16_t` *b, `armral_cmplx_int16_t` *c)
- `armral_status armral_cmplx_vecmul_i16_2` (int32_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)
- `armral_status armral_cmplx_vecmul_f32` (int32_t n, const `armral_cmplx_f32_t` *a, const `armral_cmplx_f32_t` *b, `armral_cmplx_f32_t` *c)
- `armral_status armral_cmplx_vecmul_f32_2` (int32_t n, const float *a_re, const float *a_im, const float *b_re, const float *b_im, float *c_re, float *c_im)

5.6.1 Detailed Description

The complex arrays have a total of $2*n$ real values.
The following algorithm is used:

```
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];
}
```

5.6.2 Function Documentation

5.6.2.1 armral_cmplx_vecmul_f32()

```
armral_status armral_cmplx_vecmul_f32 (
    int32_t n,
    const armral_cmplx_f32_t * a,
    const armral_cmplx_f32_t * b,
    armral_cmplx_f32_t * c )
```

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

$$C_n = A_n B_n, \text{ where } 0 \leq n < N$$

where:

$$\begin{aligned} \text{Re}(C_n) &= \text{Re}(A_n)\text{Re}(B_n) - \text{Im}(A_n)\text{Im}(B_n) \\ \text{Im}(C_n) &= \text{Re}(A_n)\text{Im}(B_n) + \text{Im}(A_n)\text{Re}(B_n) \end{aligned}$$

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

$$\vec{x} = \{x_0, x_1, \dots, x_{N-1}\}$$

where:

$$x_i = (\text{Re}(x_i), \text{Im}(x_i)), 0 \leq i < N$$

Parameters

in	n	The number of samples in each vector
in	a	Points to the first input vector
in	b	Points to the second input vector
out	c	Points to the output vector

Returns

armral_status

5.6.2.2 armral_cmplx_vecmul_f32_2()

```
armral_status armral_cmplx_vecmul_f32_2 (
    int32_t n,
    const float * a_re,
    const float * a_im,
    const float * b_re,
    const float * b_im,
    float * c_re,
    float * c_im )
```

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

$$C_n = A_n B_n, \text{ where } 0 \leq n < N$$

where:

$$\begin{aligned} \text{Re}(C_n) &= \text{Re}(A_n)\text{Re}(B_n) - \text{Im}(A_n)\text{Im}(B_n) \\ \text{Im}(C_n) &= \text{Re}(A_n)\text{Im}(B_n) + \text{Im}(A_n)\text{Re}(B_n) \end{aligned}$$

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

For an array of complex values $\vec{x} = [x_i]_{i=0}^{N-1}$, we store these as:

$$\begin{aligned} \text{Re}(\vec{x}) &= [\text{Re}(x_i)]_{i=0}^{N-1} \\ \text{Im}(\vec{x}) &= [\text{Im}(x_i)]_{i=0}^{N-1} \end{aligned}$$

Parameters

in	n	The number of samples in each vector
in	a_re	Points to the real part of the first input vector
in	a_im	Points to the imaginary part of the first input vector
in	b_re	Points to the real part of the second input vector
in	b_im	Points to the imaginary part of the second input vector
out	c_re	Points to the real part of the output result
out	c_im	Points to the imaginary part of the output result

Returns

armral_status

5.6.2.3 armral_cmplx_vecmul_i16()

```
armral_status armral_cmplx_vecmul_i16 (
    int32_t n,
    const armral_cmplx_int16_t * a,
    const armral_cmplx_int16_t * b,
    armral_cmplx_int16_t * c )
```

This algorithm 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.

$$C_n = A_n B_n, \text{ where } 0 \leq n < N$$

where:

$$\begin{aligned} \text{Re}(C_n) &= \text{Re}(A_n)\text{Re}(B_n) - \text{Im}(A_n)\text{Im}(B_n) \\ \text{Im}(C_n) &= \text{Re}(A_n)\text{Im}(B_n) + \text{Im}(A_n)\text{Re}(B_n) \end{aligned}$$

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

$$\vec{x} = \{x_0, x_1, \dots, x_{N-1}\}$$

where:

$$x_i = (\text{Re}(x_i), \text{Im}(x_i)), 0 \leq i < N$$

Parameters

in	n	The number of samples in each vector
in	a	Points to the first input vector
in	b	Points to the second input vector
out	c	Points to the output vector

Returns

armral_status

5.6.2.4 armral_cmplx_vecmul_i16_2()

```
armral_status armral_cmplx_vecmul_i16_2 (
    int32_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 )
```

This algorithm 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.

$$C_n = A_n B_n, \text{ where } 0 \leq n < N$$

where:

$$\begin{aligned} \text{Re}(C_n) &= \text{Re}(A_n)\text{Re}(B_n) - \text{Im}(A_n)\text{Im}(B_n) \\ \text{Im}(C_n) &= \text{Re}(A_n)\text{Im}(B_n) + \text{Im}(A_n)\text{Re}(B_n) \end{aligned}$$

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

For an array of complex values $\vec{x} = [x_i]_{i=0}^{N-1}$, we store these as:

$$\begin{aligned} \text{Re}(\vec{x}) &= [\text{Re}(x_i)]_{i=0}^{N-1} \\ \text{Im}(\vec{x}) &= [\text{Im}(x_i)]_{i=0}^{N-1} \end{aligned}$$

Parameters

in	n	The number of samples in each vector
in	a_re	Points to the real part of the first input vector
in	a_im	Points to the imaginary part of the first input vector
in	b_re	Points to the real part of the second input vector
in	b_im	Points to the imaginary part of the second input vector
out	c_re	Points to the real part of the output result
out	c_im	Points to the imaginary part of the output result

Returns

armral_status

5.7 Vector Dot Product

Computes the dot product of two complex vectors.

Functions

- `armral_status armral_cmplx_vecdot_f32` (int32_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)
- `armral_status armral_cmplx_vecdot_f32_2` (int32_t n, const float *p_src_a_re, const float *p_src_a_im, const float *p_src_b_re, const float *p_src_b_im, float *p_src_c_re, float *p_src_c_im)
- `armral_status armral_cmplx_vecdot_i16` (int32_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)
- `armral_status armral_cmplx_vecdot_i16_2` (int32_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)
- `armral_status armral_cmplx_vecdot_i16_32bit` (int32_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)
- `armral_status armral_cmplx_vecdot_i16_2_32bit` (int32_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)

5.7.1 Detailed Description

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 following algorithm is used:

```
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];
}
```

5.7.2 Function Documentation

5.7.2.1 armral_cmplx_vecdot_f32()

```
armral_status armral_cmplx_vecdot_f32 (
    int32_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 )
```

This algorithm 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.

Parameters

in	n	The number of samples in each vector
in	p_src_a	Points to the first complex input vector
in	p_src_b	Points to the second complex input vector
out	p_src_c	Points to the output complex vector

Returns

armral_status

5.7.2.2 armral_cmplx_vecdot_f32_2()

```
armral_status armral_cmplx_vecdot_f32_2 (
    int32_t n,
    const float * p_src_a_re,
    const float * p_src_a_im,
    const float * p_src_b_re,
    const float * p_src_b_im,
    float * p_src_c_re,
    float * p_src_c_im )
```

This algorithm 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.

Parameters

in	n	The number of samples in each vector
in	p_src_a_re	Points to the real part of the first input vector
in	p_src_a_im	Points to the imaginary part of the first input vector
in	p_src_b_re	Points to the real part of the second input vector
in	p_src_b_im	Points to the imaginary part of the second input vector
out	p_src_c_re	Points to the real part of the output result
out	p_src_c_im	Points to the imaginary part of the output result

Returns

armral_status

5.7.2.3 armral_cmplx_vecdot_i16()

```
armral_status armral_cmplx_vecdot_i16 (
    int32_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 )
```

This algorithm 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.

Parameters

in	p_src_a	Points to the first input vector
in	p_src_b	Points to the second input vector
in	n	The number of samples in each vector
out	p_src_c	Points to the output complex result

Returns

armral_status

5.7.2.4 armral_cmplx_vecdot_i16_2()

```
armral_status armral_cmplx_vecdot_i16_2 (
    int32_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 )
```

This algorithm 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.

Parameters

in	p_src_a_re	Points to the real part of first input vector
in	p_src_a_im	Points to the imag part of first input vector
in	p_src_b_re	Points to the real part of second input vector
in	p_src_b_im	Points to the imag part of second input vector
in	n	The number of samples in each vector
out	p_src_c_re	Points to the real part of output complex result
out	p_src_c_im	Points to the imag part of output complex result

Returns

armral_status

5.7.2.5 armral_cmplx_vecdot_i16_2_32bit()

```
armral_status armral_cmplx_vecdot_i16_2_32bit (
    int32_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 )
```

This algorithm 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.

Parameters

in	n	The number of samples in each vector
in	p_src_a_re	Points to the real part of the first input vector
in	p_src_a_im	Points to the imaginary part of the first input vector
in	p_src_b_re	Points to the real part of the second input vector
in	p_src_b_im	Points to the imaginary part of the second input vector
out	p_src_c_re	Points to the real part of the output result
out	p_src_c_im	Points to the imaginary part of the output result

Returns

armral_status

5.7.2.6 armral_cmplx_vecdot_i16_32bit()

```
armral_status armral_cmplx_vecdot_i16_32bit (
    int32_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 )
```

This algorithm 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.

Parameters

in	n	The number of samples in each vector
in	p_src_a	Points to the first input vector
in	p_src_b	Points to the second input vector
out	p_src_c	Points to the output complex result

Returns

armral_status

5.8 Complex Matrix Multiplication

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

Functions

- `armral_status armral_cmplx_mat_mult_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)
- `armral_status armral_cmplx_mat_mult_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)
- `armral_status armral_cmplx_mat_mult_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)
- `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)
- `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)
- `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)
- `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)
- `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)
- `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)
- `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)
- `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)
- `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)

5.8.1 Detailed Description

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

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

5.8.2 Function Documentation

5.8.2.1 armral_cmplx_mat_mult_2x2_f32()

```
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 )
```

This algorithm performs an optimized product of two square 2x2 matrices. The algorithm assumes that matrix A (first matrix) is column-major before entering the `armral_cmplx_mat_mult_2x2_f32` function.

Matrix B (second matrix) is also considered to be column-major. The result of the product is a column-major matrix. In LTE and 5G, you can use the `armral_cmplx_mat_mult_2x2_f32` function in the equalization step in the formula:

$$\hat{x} = Gy$$

Equalization matrix G corresponds to the first input matrix (matrix A) of the function. It is assumed that matrix G is transposed during its computation so that it is presented column-major on input.

The second input matrix (matrix B) is formed by two 2x1 vectors (y vectors in the preceding formula) so that each row of B represents a 2x1 vector output from each antenna port, and each call to `armral_cmplx_mat_mult_2x2_f32` computes two distinct \hat{x} estimates.

Parameters

in	p_src_a	Points to the first input matrix
in	p_src_b	Points to the second input matrix
out	p_dst	Points to the output matrix

Returns

armral_status

5.8.2.2 armral_cmplx_mat_mult_2x2_f32_iq()

```
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 )
```

This algorithm performs an optimized product of two square 2x2 matrices whose complex elements have already been separated into real component and imaginary component arrays. The algorithm assumes that matrix A (first matrix) is column-major before entering the `armral_cmplx_mat_mult_2x2_f32_iq` function.

Matrix B (second matrix) is also considered to be column-major. The result of the product is a column-major matrix. In LTE and 5G, you can use the `armral_cmplx_mat_mult_2x2_f32_iq` function in the equalization step in the formula:

$$\hat{x} = Gy$$

Equalization matrix G corresponds to the first input matrix (matrix A) of the function. It is assumed that matrix G is transposed during its computation so that it is presented column-major on input.

The second input matrix (matrix B) is formed by two 2×1 vectors (y vectors in the preceding formula) so that each row of B represents a 2×1 vector output from each antenna port, and each call to `armral_cmplx_mat_mult_2x2_f32_iq` computes two distinct \hat{x} estimates.

Parameters

in	src_a_re	Points to the real part of the first input matrix
in	src_a_im	Points to the imag part of the first input matrix
in	src_b_re	Points to the real part of the second input matrix
in	src_b_im	Points to the imag part of the second input matrix
out	dst_re	Points to the real part of the output matrix
out	dst_im	Points to the imag part of the output matrix

Returns

`armral_status`

5.8.2.3 `armral_cmplx_mat_mult_4x4_f32()`

```
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 )
```

This algorithm performs an optimized product of two square 4×4 matrices. The algorithm assumes that matrix A (first matrix) is column-major before entering the `armral_cmplx_mat_mult_4x4_f32` function.

Matrix B (second matrix) is also considered to be column-major. The result of the product is a column-major matrix. In LTE and 5G, you can use the `armral_cmplx_mat_mult_4x4_f32` function in the equalization step in the formula:

$$\hat{x} = Gy$$

Equalization matrix G corresponds to the first input matrix (matrix A) of the function.

It is assumed that matrix G is transposed during its computation so that it is presented column-major on input.

The second input matrix (matrix B) is formed by four 4×1 vectors (y vectors in the preceding formula) so that each row of B represents a 4×1 vector output from each antenna port, and each call to `cmplx_mat_mult_4x4_f32` computes four distinct \hat{x} estimates.

Parameters

in	p_src_a	Points to the first input matrix
in	p_src_b	Points to the second input matrix
out	p_dst	Points to the output matrix

Returns

armral_status

5.8.2.4 armral_cmplx_mat_mult_4x4_f32_iq()

```
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 )
```

This algorithm performs an optimized product of two square 4x4 matrices whose complex elements have already been separated into real and imaginary component arrays. The algorithm assumes that matrix A (first matrix) is column-major before entering the `armral_cmplx_mat_mult_4x4_f32_iq` function. Matrix B (second matrix) is also considered to be column-major. The result of the product is a column-major matrix. In LTE and 5G, you can use the `armral_cmplx_mat_mult_4x4_f32_iq` function in the equalization step in the formula:

$$\hat{x} = Gy$$

Equalization matrix G corresponds to the first input matrix (matrix A) of the function. It is assumed that matrix G is transposed during its computation so that it is presented column-major on input. The second input matrix (matrix B) is formed by four 4x1 vectors (\underline{y} vectors in the preceding formula) so that each row of B represents a 4x1 vector output from each antenna port, and each call to `armral_cmplx_mat_mult_4x4_f32_iq` computes four distinct \hat{x} estimates.

Parameters

in	src_a_re	Points to the real part of the first input matrix
in	src_a_im	Points to the imag part of the first input matrix
in	src_b_re	Points to the real part of the second input matrix
in	src_b_im	Points to the imag part of the second input matrix
out	dst_re	Points to the real part of the output matrix
out	dst_im	Points to the imag part of the output matrix

Returns

armral_status

5.8.2.5 armral_cmplx_mat_mult_f32()

```
armral_status armral_cmplx_mat_mult_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 )

```

This algorithm performs the multiplication $A \times B$ for square matrices of float values, and assumes that matrices are stored in memory row-major.

Parameters

in	p_src_a	Points to the first input matrix
in	p_src_b	Points to the second input matrix
out	p_dst	Points to the output matrix
in	m	The number of rows in matrix A
in	n	The number of columns in matrix A
out	k	The number of rows in matrix B

Returns

armral_status

5.8.2.6 armral_cmplx_mat_mult_i16()

```

armral_status armral_cmplx_mat_mult_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 )

```

This algorithm performs the multiplication $A \times B$ for square matrices, and assumes that:

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

A 64-bit Q32.31 accumulator is used internally. If you do not need such a large range, consider using [armral_cmplx_mat_mult_i16_32bit](#) instead. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Parameters

in	p_src_a	Points to the first input matrix
in	p_src_b	Points to the second input matrix
out	p_dst	Points to the output matrix
in	m	The number of rows in matrix A
in	n	The number of columns in matrix A
out	k	The number of rows in matrix B

Returns

armral_status

5.8.2.7 armral_cmplx_mat_mult_i16_32bit()

```
armral_status armral_cmplx_mat_mult_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 )
```

This algorithm performs the multiplication $A \times B$ for square matrices, and assumes that:

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

A 32-bit Q0.31 saturating accumulator is used internally. If you need a larger range, consider using [armral_cmplx_mat_mult_i16](#) instead. To get a Q15 result, the final result is narrowed to 16 bits with saturation to get a Q15 result.

Parameters

in	p_src_a	Points to the first input matrix
in	p_src_b	Points to the second input matrix
out	p_dst	Points to the output matrix
in	m	The number of rows in matrix A
in	n	The number of columns in matrix A
out	k	The number of rows in matrix B

Returns

armral_status

5.8.2.8 armral_solve_1x2_f32()

```
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 )
```

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

$$\hat{x} = Gy$$

where y is a vector for the received signal, size corresponds to the number of antennae and \hat{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 2x4 matrix. I and Q components of G elements are assumed to be stored separated in memory.

Also, each coefficient of G ($G_{11}, G_{12}, G_{21}, G_{22}$) is assumed to be stored separated in memory locations set at `pGstride` 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 also exists for cases where the number of subcarriers per G matrix is equal to one.

Parameters

in	num_sub_carrier	The number of subcarrier to equalize
in	num_sc_per_g	The number of subcarriers per G matrix
in	p_y	Points to the input received signal
in	p_ystride	The stride between two Rx antennae
in	p_y_num_fract_bits	The number of fractional bits in y conversion
in	p_g_real	The real part of coefficient matrix G
in	p_g_imag	The imag part of coefficient matrix G
in	p_gstride	The stride between elements of G
out	p_x	Points to the output received signal
in	num_fract_bits_x	The number of fractional bits in x

Returns

`armral_status`

5.8.2.9 armral_solve_1x4_f32()

```
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 )
```

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

$$\hat{x} = Gy$$

where y is a vector for the received signal, size corresponds to the number of antennae and \hat{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 2x4 matrix. I and Q components of G elements are assumed to be stored separated in memory.

Also, each coefficient of G ($G_{11}, G_{12}, G_{21}, G_{22}$) is assumed to be stored separated in memory locations set at `pGstride` 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 also exists for cases where the number of subcarriers per G matrix is equal to one.

Parameters

in	num_sub_carrier	The number of subcarrier to equalize
in	num_sc_per_g	The number of subcarriers per G matrix
in	p_y	Points to the input received signal
in	p_ystride	The stride between two Rx antennae
in	p_y_num_fract_bits	The number of fractional bits in y conversion
in	p_g_real	The real part of coefficient matrix G
in	p_g_imag	The imag part of coefficient matrix G
in	p_gstride	The stride between elements of G
out	p_x	Points to the output received signal
in	num_fract_bits_x	The number of fractional bits in x

Returns

`armral_status`

5.8.2.10 `armral_solve_2x2_f32()`

```
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 )
```

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

$$\hat{x} = Gy$$

where y is a vector for the received signal, size corresponds to the number of antennae and \hat{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 2x2 matrix. I and Q components of G elements are assumed to be stored separated in memory.

Also, each coefficient of G ($G_{11}, G_{12}, G_{21}, G_{22}$) is assumed to be stored separated in memory locations set at `pGstride` 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 also exists for cases where the number of subcarriers per G matrix is equal to one.

Parameters

in	num_sub_carrier	The number of subcarriers to equalize
in	num_sc_per_g	The number of subcarriers per G matrix
in	p_y	Points to the input received signal
in	p_ystride	The stride between two Rx antennae
in	p_y_num_fract_bits	The number of fractional bits in y
in	p_g_real	The real part of coefficient matrix G
in	p_g_imag	The imag part of coefficient matrix G
in	p_gstride	The stride between elements of G
out	p_x	Points to the output received signal
in	p_xstride	The stride between two layers
in	num_fract_bits_x	The number of fractional bits in x

Returns

armral_status

5.8.2.11 armral_solve_2x4_f32()

```
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 )
```

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

$$\hat{x} = Gy$$

where y is a vector for the received signal, size corresponds to the number of antennae and \hat{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 2x4 matrix. I and Q components of G elements are assumed to be stored separated in memory.

Also, each coefficient of G ($G_{11}, G_{12}, G_{21}, G_{22}$) is assumed to be stored separated in memory locations set at `pGstride` 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 also exists for cases where the number of subcarriers per G matrix is equal to one.

Parameters

in	num_sub_carrier	The number of subcarrier to equalize
in	num_sc_per_g	The number of subcarriers per G matrix
in	p_y	Points to the input received signal
in	p_ystride	The stride between two Rx antennae
in	p_y_num_fract_bits	The number of fractional bits in y
in	p_g_real	The real part of coefficient matrix G
in	p_g_imag	The imag part of coefficient matrix G
in	p_gstride	The stride between elements of G
out	p_x	Points to the output received signal
in	p_xstride	The stride between two layers
in	num_frac_bits_x	The number of fractional bits in x

Returns

armral_status

5.8.2.12 armral_solve_4x4_f32()

```
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_frac_bits_x )
```

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

$$\hat{x} = Gy$$

where y is a vector for the received signal, size corresponds to the number of antennae and \hat{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 2x4 matrix. I and Q components of G elements are assumed to be stored separated in memory.

Also, each coefficient of G ($G_{11}, G_{12}, G_{21}, G_{22}$) is assumed to be stored separated in memory locations set at `pGstride` 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 also exists for cases where the number of subcarriers per G matrix is equal to one.

Parameters

in	num_sub_carrier	The number of subcarrier to equalize
in	num_sc_per_g	The number of subcarriers per G matrix
in	p_y	Points to the input received signal
in	p_ystride	The stride between two Rx antennae
in	p_y_num_fract_bits	The number of fractional bits in y
in	p_g_real	The real part of coefficient matrix G
in	p_g_imag	The imag part of coefficient matrix G
in	p_gstride	The stride between elements of G
out	p_x	Points to the output received signal
in	p_xstride	The stride between two layers
in	num_frac_bits_x	The number of fractional bits in x

Returns

armral_status

5.9 Complex Matrix Inversion

Computes the inverse of a complex hermitian square matrix of size NxN.

Functions

- `armral_status armral_cmplx_hermitian_mat_inverse_f32` (uint16_t size, uint16_t par, const `armral_cmplx_f32_t` *p_src, `armral_cmplx_f32_t` *p_dst)

5.9.1 Detailed Description

5.9.2 Function Documentation

5.9.2.1 armral_cmplx_hermitian_mat_inverse_f32()

```
armral_status armral_cmplx_hermitian_mat_inverse_f32 (
    uint16_t size,
    uint16_t par,
    const armral_cmplx_f32_t * p_src,
    armral_cmplx_f32_t * p_dst )
```

This algorithm computes the inverse of a complex hermitian square matrix of size NxN.

The supported dimensions are 2x2, 4x4, 8x8, and 16x16.

The input matrix, which is filled in row-major order, is made of complex float32_t elements in an interleaved form:

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

The output matrix follow the same assumptions of the input matrix.

This kernel enables the inversion of more than one matrix in parallel.
For parallel inversion, two options are supported: 2x[2x2] and 4x[4x4].

Parameters

in	size	The size of the input matrix
in	par	0: Process a single matrix 1: Process multiple matrices in parallel
in	p_src	Points to input matrix structure
out	p_dst	Points to the output matrix structure

Returns

armral_status

5.10 Sequence Generator

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

Functions

- `armral_status armral_seq_generator` (uint16_t sequence_len, uint32_t seed, uint8_t *p_dst)

5.10.1 Detailed Description

The sequence generator is the same generator that is described in 3GPP 36.211, Chapter 7.2.

5.10.2 Function Documentation

5.10.2.1 armral_seq_generator()

```
armral_status armral_seq_generator (
    uint16_t sequence_len,
    uint32_t seed,
    uint8_t * p_dst )
```

This algorithm 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 3GPP 36.211, Chapter 7.2. The generator uses two polynomials, $x^{(1)}$ and $x^{(2)}$, defined as:

$$\begin{aligned} x_{n+31}^{(1)} &= \left(x_{n+3}^{(1)} + x_n^{(1)} \right) \mod 2 \\ x_{n+31}^{(2)} &= \left(x_{n+3}^{(2)} + x_{n+2}^{(2)} + x_{n+1}^{(2)} + x_n^{(2)} \right) \mod 2 \end{aligned}$$

to generate the output sequence:

$$c_n = \left(x_{n+N_c}^{(1)} + x_{n+N_c}^{(2)} \right) \mod 2$$

where N_c is a constant with a value of 1600. The initialization for $x^{(1)}$ and $x^{(2)}$ satisfies the condition that:

$$\begin{aligned} x_0^{(1)} &= 1 \\ x_i^{(1)} &= 0, 1 \leq i \leq 30 \\ c^{(0)} &= \sum_{i=0}^{30} (x_i^{(2)} 2^i) \end{aligned}$$

The `cinit` parameter is provided as an input parameter for the algorithm, which is used to derive $x^{(2)}$. The algorithm generates $x^{(1)}$ and $x^{(2)}$, and skips the first 1600 bits.

Parameters

in	sequence_len	The length of the sequence in bits (cinit)
in	seed	The random sequence starting point
in	p_dst	Points to the output bits

Returns

armral_status

5.11 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.

Functions

- [armral_status armral_modulation](#) (uint32_t nbits, [armral_modulation_type](#) mod_type, const int8_t *p_src, [armral_cmplx_int16_t](#) *p_dst)
- [armral_status armral_demodulation](#) (uint32_t n_symbols, int16_t amp, uint16_t noise_power, [armral_modulation_type](#) mod_type, const [armral_cmplx_int16_t](#) *p_src, int8_t *p_dst)

5.11.1 Detailed Description

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).

5.11.2 Function Documentation

5.11.2.1 armral_demodulation()

```
armral_status armral_demodulation (
    uint32_t n_symbols,
    int16_t amp,
    uint16_t noise_power,
    armral_modulation_type mod_type,
    const armral_cmplx_int16_t * p_src,
    int8_t * p_dst )
```

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

The input sequence is assumed to be made of complex symbols $x = Re(x) + \jmath Im(x)$, whose components I and Q are 16 bits each (format Q2.13) and in an interleaved form:

$$\vec{x} = [x_i]_{i=0}^{N-1}$$

for complex symbols x_i .

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

The LLRs calculations are made approximately with thresholds method, to have similar performance of the raw calculation, but with a lower complexity.

All the constellations mapping follow the 3GPP TS 38.211 V15.2.0, Chapter 5.1 Modulation mapper.

Parameters

in	amp	The input signal average amplitude
in	pwr	The noise power
in	modType	The modulation type
in	pSrc	Points to input complex source (format Q2.13)
out	pDst	Points to the output byte seq

Returns

armral_status

5.11.2.2 armral_modulation()

```
armral_status armral_modulation (
    uint32_t nbits,
    armral_modulation_type mod_type,
    const int8_t * p_src,
    armral_cmplx_int16_t * p_dst )
```

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.

Parameters

in	nbits	The number of input modulated bits
in	mod_type	The type of modulation to perform
in	p_src	Points to input bit flow
out	p_dst	Points to output complex symbols (format Q2.13)

Returns

armral_status

5.12 Correlation Coefficient

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

Functions

- `armral_status armral_corr_coeff_i16` (int32_t n, const `armral_cmplx_int16_t` *p_src_a, const `armral_cmplx_int16_t` *p_src_b, `armral_cmplx_int16_t` *c)

5.12.1 Detailed Description

5.12.2 Function Documentation

5.12.2.1 `armral_corr_coeff_i16()`

```
armral_status armral_corr_coeff_i16 (
    int32_t n,
    const armral_cmplx_int16_t * p_src_a,
    const armral_cmplx_int16_t * p_src_b,
    armral_cmplx_int16_t * c )
```

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:

$$R_{xy} = \frac{\vec{x}\vec{y}^H - n\bar{x}\bar{y}}{\left(\left(|\vec{x}|^2 - n|\bar{x}|^2\right)\left(|\vec{y}|^2 - n|\bar{y}|^2\right)\right)^{\frac{1}{2}}}$$

where $\bar{x} = \frac{\sum_{i=0}^{n-1} x_i}{n}$ is the mean of the sequence \vec{x} , and similarly for \bar{y} .

Parameters

in	n	The number of complex samples in each vector
in	p_src_a	Points to the first input vector
in	p_src_b	Points to the second input vector
out	c	Points to result

Returns

`armral_status`

5.13 FIR filter

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

Functions

- `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)
- `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)
- `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)
- `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)

5.13.1 Detailed Description

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

$$y_n = b_0 x_{N-1} + b_1 x_{N-2} + \dots + b_{N-1} x_0 = \sum_{i=0}^{N-1} b_i x_{N-1-i}$$

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.

5.13.2 Function Documentation

5.13.2.1 `armral_fir_filter_cf32()`

```
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 )
```

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

The size of the input data 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.

Parameters

in	size	The number of complex samples in input
in	taps	The number of taps of the FIR filter
in	input	Points to the input samples buffer
in	coeffs	Points to the coefficients array
out	output	Points to the output array

Returns

armral_status

5.13.2.2 armral_fir_filter_cf32_decimate_2()

```
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 )
```

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

The size of the input data 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.

Parameters

in	size	The number of complex samples in input
in	taps	The number of taps of the FIR filter
in	input	Points to the input samples buffer
in	coeffs	Points to the coefficients array
out	output	Points to the output array

Returns

armral_status

5.13.2.3 armral_fir_filter_cs16()

```
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 )
```

Computes a complex signed 16-bit integer FIR filter.

The size of the input data 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.

Parameters

in	size	The number of complex samples in input
in	taps	The number of taps of the FIR filter
in	input	Points to the input samples buffer
in	coeffs	Points to the coefficients array
out	output	Points to the output array

Returns

armral_status

5.13.2.4 armral_fir_filter_cs16_decimate_2()

```
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 )
```

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

The size of the input data 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.

Parameters

in	size	The number of complex samples in input
in	taps	The number of taps of the FIR filter
in	input	Points to the input samples buffer
in	coeffs	Points to the coefficients array
out	output	Points to the output array

Returns

armral_status

5.14 Mu-Law Compression

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

Functions

- `armral_status armral_mu_law_compr_8bit` (uint32_t n_prb, const `armral_cmplx_int16_t` *src, `armral_compressed_data_8bit` *dst)
- `armral_status armral_mu_law_decompr_8bit` (uint32_t n_prb, const `armral_compressed_data_8bit` *src, `armral_cmplx_int16_t` *dst)

5.14.1 Detailed Description

5.14.2 Function Documentation

5.14.2.1 `armral_mu_law_compr_8bit()`

```
armral_status armral_mu_law_compr_8bit (
    uint32_t n_prb,
    const armral_cmplx_int16_t * src,
    armral_compressed_data_8bit * dst )
```

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 algorithm uses Mu=8 for implementation efficiency. The Mu-Law compression works 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 a shift value.

Parameters

in	<code>n_prb</code>	The number of input resource blocks
in	<code>src</code>	Points to the input complex samples sequence
out	<code>dst</code>	Points to the output 8-bit data and exponent

Returns

`armral_status`

5.14.2.2 `armral_mu_law_decompr_8bit()`

```
armral_status armral_mu_law_decompr_8bit (
    uint32_t n_prb,
```

```
const armral_compressed_data_8bit * src,  
armral_cmplx_int16_t * dst )
```

The Mu-Law decompression method is a logical reverse function of the compression method. The Mu-Law compression works 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 a shift value.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input 8-bit data and exponent
out	dst	Points to the output complex samples sequence

Returns

armral_status

5.15 Block Floating Point

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

Functions

- `armral_status armral_block_float_compr_8bit` (uint32_t n_prb, const `armral_cmplx_int16_t` *src, `armral_compressed_data_8bit` *dst)
Block floating-point compression to 8-bit.
- `armral_status armral_block_float_compr_9bit` (uint32_t n_prb, const `armral_cmplx_int16_t` *src, `armral_compressed_data_9bit` *dst)
Block floating point compression to 9-bit big-endian.
- `armral_status armral_block_float_compr_12bit` (uint32_t n_prb, const `armral_cmplx_int16_t` *src, `armral_compressed_data_12bit` *dst)
Block floating point compression to 12-bit big-endian.
- `armral_status armral_block_float_compr_14bit` (uint32_t n_prb, const `armral_cmplx_int16_t` *src, `armral_compressed_data_14bit` *dst)
Block floating point compression to 14-bit big-endian.
- `armral_status armral_block_float_decompr_8bit` (uint32_t n_prb, const `armral_compressed_data_8bit` *src, `armral_cmplx_int16_t` *dst)
Block floating-point decompression from 8 bit.
- `armral_status armral_block_float_decompr_9bit` (uint32_t n_prb, const `armral_compressed_data_9bit` *src, `armral_cmplx_int16_t` *dst)
Block floating point decompression from 9 bit big-endian.
- `armral_status armral_block_float_decompr_12bit` (uint32_t n_prb, const `armral_compressed_data_12bit` *src, `armral_cmplx_int16_t` *dst)
Block floating point decompression from 12 bit big-endian.
- `armral_status armral_block_float_decompr_14bit` (uint32_t n_prb, const `armral_compressed_data_14bit` *src, `armral_cmplx_int16_t` *dst)
Block floating point decompression from 14 bit big-endian.

5.15.1 Detailed Description

5.15.2 Function Documentation

5.15.2.1 `armral_block_float_compr_12bit()`

```
armral_status armral_block_float_compr_12bit (
    uint32_t n_prb,
    const armral_cmplx_int16_t * src,
    armral_compressed_data_12bit * dst )
```

The algorithm 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.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input complex samples sequence
out	dst	Points to the output 12-bit data and exponent

Returns

armral_status

5.15.2.2 armral_block_float_compr_14bit()

```
armral_status armral_block_float_compr_14bit (
    uint32_t n_prb,
    const armral_cmplx_int16_t * src,
    armral_compressed_data_14bit * dst )
```

The algorithm 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.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input complex samples sequence
out	dst	Points to the output 14-bit data and exponent

Returns

armral_status

5.15.2.3 armral_block_float_compr_8bit()

```
armral_status armral_block_float_compr_8bit (
    uint32_t n_prb,
    const armral_cmplx_int16_t * src,
    armral_compressed_data_8bit * dst )
```

The algorithm 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.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input complex samples sequence
out	dst	Points to the output 8-bit data and exponent

Returns

armral_status

5.15.2.4 armral_block_float_compr_9bit()

```
armral_status armral_block_float_compr_9bit (
    uint32_t n_prb,
    const armral_cmplx_int16_t * src,
    armral_compressed_data_9bit * dst )
```

The algorithm 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.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input complex samples sequence
out	dst	Points to the output 9-bit data and exponent

Returns

armral_status

5.15.2.5 armral_block_float_decompr_12bit()

```
armral_status armral_block_float_decompr_12bit (
    uint32_t n_prb,
    const armral_compressed_data_12bit * src,
    armral_cmplx_int16_t * dst )
```

The algorithm 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.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input compressed block sequence
out	dst	Points to the complex output sequence

Returns

armral_status

5.15.2.6 armral_block_float_decompr_14bit()

```
armral_status armral_block_float_decompr_14bit (
    uint32_t n_prb,
    const armral_compressed_data_14bit * src,
    armral_cmplx_int16_t * dst )
```

The algorithm 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.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input compressed block sequence
out	dst	Points to the complex output sequence

Returns

armral_status

5.15.2.7 armral_block_float_decompr_8bit()

```
armral_status armral_block_float_decompr_8bit (
    uint32_t n_prb,
    const armral_compressed_data_8bit * src,
    armral_cmplx_int16_t * dst )
```

The algorithm 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.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input compressed block sequence
out	dst	Points to the complex output sequence

Returns

armral_status

5.15.2.8 armral_block_float_decompr_9bit()

```
armral_status armral_block_float_decompr_9bit (
    uint32_t n_prb,
    const armral_compressed_data_9bit * src,
    armral_cmplx_int16_t * dst )
```

The algorithm 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.

Parameters

in	n_prb	The number of input resource blocks
in	src	Points to the input compressed block sequence
out	dst	Points to the complex output sequence

Returns

armral_status

5.16 CRC24

Computes a 24-bit Cyclic Redundancy Check (CRC) of an input buffer using carry-less multiplication and Barret reduction.

Functions

- `armral_status armral_crc24_a_le` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_a_be` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_b_le` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_b_be` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_c_le` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_c_be` (uint32_t size, const uint64_t *input, uint64_t *crc24)

5.16.1 Detailed Description

```
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
```

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

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

5.16.2 Function Documentation

5.16.2.1 `armral_crc24_a_be()`

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

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

Parameters

in	size	The number of bytes of the given buffer
in	input	Points to the input byte sequence
out	crc24	The computed CRC on 24 bit

Returns

armral_status

5.16.2.2 armral_crc24_a_le()

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

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

Parameters

in	size	The number of bytes of the given buffer
in	input	Points to the input byte sequence
out	crc24	The computed CRC on 24 bits

Returns

armral_status

5.16.2.3 armral_crc24_b_be()

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

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

Parameters

in	size	The number of bytes of the given buffer
in	input	Points to the input byte sequence
out	crc24	The computed CRC on 24 bit

Returns

armral_status

5.16.2.4 armral_crc24_b_le()

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

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

Parameters

in	size	The number of bytes of the given buffer
in	input	Points to the input byte sequence
out	crc24	The computed CRC on 24 bit

Returns

armral_status

5.16.2.5 armral_crc24_c_be()

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

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

Parameters

in	size	The number of bytes of the given buffer
in	input	Points to the input byte sequence
out	crc24	The computed CRC on 24 bit

Returns

armral_status

5.16.2.6 armral_crc24_c_le()

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

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

Parameters

in	size	The number of bytes of the given buffer
in	input	Points to the input byte sequence
out	crc24	The computed CRC on 24 bit

Returns

`armral_status`

5.17 Polar Encoding

Polar codes are used to encode the Uplink Control Information (UCI) over the PUCCH and PUSCH, and the Downlink Control Information (DCI) over the PDCCH, in downlink.

Functions

- [armral_status armral_polar_encoder](#) (uint16_t n, const uint32_t *p_u_seq_in, uint32_t *p_d_seq_out)
- [armral_status armral_polar_decoder](#) (uint16_t n, uint16_t k, uint16_t l, const int8_t *p_llr_in, uint32_t *p_u_seq_out)

5.17.1 Detailed Description

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 ($K \leq N$). 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 $[u] = [u(0), u(1), \dots, u(N-1)]$, if index i is included in the frozen bits set, then $u(i) = 0$. The input information bits are stored in the remaining entries. $[d] = [d(0), d(1), \dots, d(N-1)]$ is the vector of output encoded bits. $[G_N]$ is the channel transformation matrix ($N \times N$), obtained by recursively applying the Kronecker product from the basic kernel $G_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ to the order $n = \log_2(N)$.

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

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

5.17.2 Function Documentation

5.17.2.1 armral_polar_decoder()

```
armral_status armral_polar_decoder (
    uint16_t n,
    uint16_t k,
    uint16_t l,
    const int8_t * p_llr_in,
    uint32_t * p_u_seq_out )
```

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.

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). Not all list sizes are supported. Unsupported values of n or l will return `ARMRAL_ARGUMENT_ERROR`.

Parameters

in	n	The polar code length in bits, must be a power of 2
in	k	The number of real information bits (message + CRC) within the codeword of length n, where $k < n$
in	l	The list size to be used in decoding
in	p_llr_in	Points to the input sequence of LLR bytes
out	p_u_seq_out	Points to l decoded sequences, ordered by decreasing path metric, each of length n bits

Returns

armral_status

5.17.2.2 armral_polar_encoder()

```
armral_status armral_polar_encoder (
    uint16_t n,
    const uint32_t * p_u_seq_in,
    uint32_t * p_d_seq_out )
```

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

Parameters

in	n	The polar code length in bits, where n must be a power of 2
in	p_u_seq_in	Points to the input sequence [u] of bits [u(0), u(1), ..., u(N-1)]
out	p_d_seq_out	Points to the output encoded sequence [d] of bits [d(0), d(1), ..., d(N-1)]

Returns

armral_status

5.18 Fast Fourier Transforms (FFT)

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

Typedefs

- typedef struct [armral_fft_plan_t](#) [armral_fft_plan_t](#)

Enumerations

- enum [armral_fft_direction_t](#) { [ARMRAL_FFT_FORWARDS](#) = -1, [ARMRAL_FFT_BACKWARDS](#) = 1 }

Functions

- [armral_status armral_fft_create_plan_cf32](#) ([armral_fft_plan_t](#) **p, int n, [armral_fft_direction_t](#) dir)
Creates a plan to solve a complex fp32 FFT.
- [armral_status armral_fft_execute_cf32](#) (const [armral_fft_plan_t](#) *p, const [armral_cmplx_f32_t](#) *x, [armral_cmplx_f32_t](#) *y)
Performs a single FFT using the specified plan and arrays.
- [armral_status armral_fft_destroy_plan_cf32](#) ([armral_fft_plan_t](#) **p)
Destroys an FFT plan.
- [armral_status armral_fft_create_plan_cs16](#) ([armral_fft_plan_t](#) **p, int n, [armral_fft_direction_t](#) dir)
Creates a plan to solve a complex int16 (Q0.15 format) FFT.
- [armral_status armral_fft_execute_cs16](#) (const [armral_fft_plan_t](#) *p, const [armral_cmplx_int16_t](#) *x, [armral_cmplx_int16_t](#) *y)
Performs a single FFT using the specified plan and arrays.
- [armral_status armral_fft_destroy_plan_cs16](#) ([armral_fft_plan_t](#) **p)
Destroys an FFT plan.

5.18.1 Detailed Description

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

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

5.18.2 Typedef Documentation

5.18.2.1 armral_fft_plan_t

```
typedef struct armral_fft_plan_t armral_fft_plan_t
```

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

5.18.3 Enumeration Type Documentation

5.18.3.1 armral_fft_direction_t

```
enum armral_fft_direction_t
```

The direction of the FFT being computed. The direction is passed to [armral_fft_create_plan_cf32](#) and [armral_fft_create_plan_cs16](#).

Enumerator

ARMRAL_FFT_FORWARDS	Compute a forwards (non-inverse) FFT.
ARMRAL_FFT_BACKWARDS	Compute a backwards (inverse) FFT.

5.18.4 Function Documentation

5.18.4.1 armral_fft_create_plan_cf32()

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

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.

Parameters

in, out	p	A pointer to the resulting plan pointer. On output *p is a valid pointer, to be passed to armral_fft_execute_cf32 .
in	n	The problem size to be solved by this FFT plan.
out	dir	The direction to be solved by this FFT plan.

Returns

armral_status

5.18.4.2 armral_fft_create_plan_cs16()

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

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.

Parameters

in, out	p	A pointer to the resulting plan pointer. On output *p is a valid pointer, to be passed to <code>armral_fft_execute_cs16</code> .
in	n	The problem size to be solved by this FFT plan.
out	dir	The direction to be solved by this FFT plan.

Returns

armral_status

5.18.4.3 armral_fft_destroy_plan_cf32()

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

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`.

Parameters

in, out	p	A pointer to the FFT plan pointer. The pointer must be the value that is returned by an earlier call to <code>armral_fft_create_plan_cf32</code> . On function exit, the value that is pointed to is set to NULL.
---------	---	---

Returns

armral_status

5.18.4.4 armral_fft_destroy_plan_cs16()

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

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`.

Parameters

in, out	p	A pointer to the FFT plan pointer. The pointer must be the value that is returned by an earlier call to <code>armral_fft_create_plan_cs16</code> . On function exit, the value that is pointed to is set to <code>NULL</code> .
---------	---	---

Returns

armral_status

5.18.4.5 armral_fft_execute_cf32()

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

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

Parameters

in	p	A pointer to the FFT plan. The pointer is the value that is filled in by an earlier call to <code>armral_fft_create_plan_cf32</code> .
in	x	The input array for this FFT. The length must be the same as the value of <code>n</code> that was previously passed to <code>armral_fft_create_plan_cf32</code> .
out	y	The output array for this FFT. The length must be the same as the value of <code>n</code> that was previously passed to <code>armral_fft_create_plan_cf32</code> .

Returns

armral_status

5.18.4.6 armral_fft_execute_cs16()

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

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

Parameters

in	p	A pointer to the FFT plan. The pointer is the value that is filled in by an earlier call to <code>armral_fft_create_plan_cs16</code> .
in	x	The input array for this FFT. The length must be the same as the value of <code>n</code> that was previously passed to <code>armral_fft_create_plan_cs16</code> .
out	y	The output array for this FFT. The length must be the same as the value of <code>n</code> that was previously passed to <code>armral_fft_create_plan_cs16</code> .

Returns

`armral_status`

Chapter 6

Data Structure Index

6.1 Data Structures

The data structures that Arm RAN Acceleration Library supports are:

armral_cmplx_f32_t	
32-bit floating-point complex data type	73
armral_cmplx_int16_t	
16-bit signed integer complex data type	74
armral_compressed_data_12bit	
The structure for a 12-bit compressed block	74
armral_compressed_data_14bit	
The structure for a 14-bit compressed block	75
armral_compressed_data_8bit	
The structure for an 8-bit compressed block	76
armral_compressed_data_9bit	
The structure for a 9-bit compressed block	77

Chapter 7

Data Structure Documentation

7.1 armral_cmplx_f32_t Struct Reference

32-bit floating-point complex data type.

```
#include <armral.h>
```

Data Fields

- float [re](#)
32-bit real component
- float [im](#)
32-bit imaginary component

7.1.1 Field Documentation

7.1.1.1 im

```
float armral_cmplx_f32_t::im
```

7.1.1.2 re

```
float armral_cmplx_f32_t::re
```

The documentation for this struct was generated from the following file:

- [armral.h](#)

7.2 armral_cmplx_int16_t Struct Reference

16-bit signed integer complex data type.

```
#include <armral.h>
```

Data Fields

- [int16_t re](#)
16-bit real component
- [int16_t im](#)
16-bit imaginary component

7.2.1 Field Documentation

7.2.1.1 im

```
int16_t armral_cmplx_int16_t::im
```

7.2.1.2 re

```
int16_t armral_cmplx_int16_t::re
```

The documentation for this struct was generated from the following file:

- [armral.h](#)

7.3 armral_compressed_data_12bit Struct Reference

The structure for a 12-bit compressed block.

```
#include <armral.h>
```

Data Fields

- [int8_t exp](#)
Block exponent, in the range 0-4 (inclusive)
- [int8_t mantissa](#) [36]
Packed data, 12 bits per element.

7.3.1 Detailed Description

See [armral_block_float_compr_12bit](#) and [armral_block_float_decompr_12bit](#).

7.3.2 Field Documentation

7.3.2.1 exp

```
int8_t armral_compressed_data_12bit::exp
```

7.3.2.2 mantissa

```
int8_t armral_compressed_data_12bit::mantissa[36]
```

The documentation for this struct was generated from the following file:

- [armral.h](#)

7.4 armral_compressed_data_14bit Struct Reference

The structure for a 14-bit compressed block.

```
#include <armral.h>
```

Data Fields

- int8_t [exp](#)
Block exponent, in the range 0-2 (inclusive)
- int8_t [mantissa](#) [42]
Packed data, 14 bits per element.

7.4.1 Detailed Description

See [armral_block_float_compr_14bit](#) and [armral_block_float_decompr_14bit](#).

7.4.2 Field Documentation

7.4.2.1 exp

```
int8_t armral_compressed_data_14bit::exp
```

7.4.2.2 mantissa

```
int8_t armral_compressed_data_14bit::mantissa[42]
```

The documentation for this struct was generated from the following file:

- [armral.h](#)

7.5 armral_compressed_data_8bit Struct Reference

The structure for an 8-bit compressed block.

```
#include <armral.h>
```

Data Fields

- int8_t [exp](#)
Block exponent, in the range 0-8 (inclusive)
- int8_t [mantissa](#) [24]
Packed data, 8 bits per element.

7.5.1 Detailed Description

See [armral_block_float_compr_8bit](#) and [armral_block_float_decompr_8bit](#).

7.5.2 Field Documentation

7.5.2.1 exp

```
int8_t armral_compressed_data_8bit::exp
```

7.5.2.2 mantissa

```
int8_t armral_compressed_data_8bit::mantissa[24]
```

The documentation for this struct was generated from the following file:

- [armral.h](#)

7.6 armral_compressed_data_9bit Struct Reference

The structure for a 9-bit compressed block.

```
#include <armral.h>
```

Data Fields

- int8_t [exp](#)
Block exponent, in the range 0-7 (inclusive)
- int8_t [mantissa](#) [27]
Packed data, 9 bits per element.

7.6.1 Detailed Description

See [armral_block_float_compr_9bit](#) and [armral_block_float_decompr_9bit](#).

7.6.2 Field Documentation

7.6.2.1 exp

```
int8_t armral_compressed_data_9bit::exp
```

7.6.2.2 mantissa

```
int8_t armral_compressed_data_9bit::mantissa[27]
```

The documentation for this struct was generated from the following file:

- [armral.h](#)

Chapter 8

File Index

8.1 File List

The header files that Arm RAN Acceleration Library includes are:

armral.h	81
------------------------------------	----

Chapter 9

File Documentation

9.1 armral.h File Reference

```
#include <arm_neon.h>
#include <inttypes.h>
```

Data Structures

- struct [armral_cmplx_int16_t](#)
16-bit signed integer complex data type.
- struct [armral_cmplx_f32_t](#)
32-bit floating-point complex data type.
- struct [armral_compressed_data_8bit](#)
The structure for an 8-bit compressed block.
- struct [armral_compressed_data_9bit](#)
The structure for a 9-bit compressed block.
- struct [armral_compressed_data_12bit](#)
The structure for a 12-bit compressed block.
- struct [armral_compressed_data_14bit](#)
The structure for a 14-bit compressed block.

Macros

- #define [ARMRAL_NUM_COMPLEX_SAMPLES](#) 12

Typedefs

- typedef struct [armral_fft_plan_t](#) [armral_fft_plan_t](#)

Enumerations

- enum `armral_status` { `ARMRAL_SUCCESS` = 0, `ARMRAL_ARGUMENT_ERROR` = -1 }
- enum `armral_modulation_type` { `ARMRAL_MOD_QPSK` = 0, `ARMRAL_MOD_16QAM` = 1, `ARMRAL_MOD_64QAM` = 2, `ARMRAL_MOD_256QAM` = 3 }
- enum `armral_fixed_point_index` {
`ARMRAL_FIXED_POINT_INDEX_Q15` = 15, `ARMRAL_FIXED_POINT_INDEX_Q1_14` = 14, `ARMRAL_FIXED_POINT_INDEX_Q2_13` = 13, `ARMRAL_FIXED_POINT_INDEX_Q3_12` = 12,
`ARMRAL_FIXED_POINT_INDEX_Q4_11` = 11, `ARMRAL_FIXED_POINT_INDEX_Q5_10` = 10,
`ARMRAL_FIXED_POINT_INDEX_Q6_9` = 9, `ARMRAL_FIXED_POINT_INDEX_Q7_8` = 8,
`ARMRAL_FIXED_POINT_INDEX_Q8_7` = 7, `ARMRAL_FIXED_POINT_INDEX_Q9_6` = 6, `ARMRAL_FIXED_POINT_INDEX_Q10_5` = 5, `ARMRAL_FIXED_POINT_INDEX_Q11_4` = 4,
`ARMRAL_FIXED_POINT_INDEX_Q12_3` = 3, `ARMRAL_FIXED_POINT_INDEX_Q13_2` = 2, `ARMRAL_FIXED_POINT_INDEX_Q14_1` = 1, `ARMRAL_FIXED_POINT_INDEX_Q15_0` = 0 }
- enum `armral_fft_direction_t` { `ARMRAL_FFT_FORWARDS` = -1, `ARMRAL_FFT_BACKWARDS` = 1 }

Functions

- `armral_status armral_cmplx_vecmul_i16` (int32_t n, const `armral_cmplx_int16_t` *a, const `armral_cmplx_int16_t` *b, `armral_cmplx_int16_t` *c)
- `armral_status armral_cmplx_vecmul_i16_2` (int32_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)
- `armral_status armral_cmplx_vecmul_f32` (int32_t n, const `armral_cmplx_f32_t` *a, const `armral_cmplx_f32_t` *b, `armral_cmplx_f32_t` *c)
- `armral_status armral_cmplx_vecmul_f32_2` (int32_t n, const float *a_re, const float *a_im, const float *b_re, const float *b_im, float *c_re, float *c_im)
- `armral_status armral_cmplx_vecdot_f32` (int32_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)
- `armral_status armral_cmplx_vecdot_f32_2` (int32_t n, const float *p_src_a_re, const float *p_src_a_im, const float *p_src_b_re, const float *p_src_b_im, float *p_src_c_re, float *p_src_c_im)
- `armral_status armral_cmplx_vecdot_i16` (int32_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)
- `armral_status armral_cmplx_vecdot_i16_2` (int32_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)
- `armral_status armral_cmplx_vecdot_i16_32bit` (int32_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)
- `armral_status armral_cmplx_vecdot_i16_2_32bit` (int32_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)
- `armral_status armral_cmplx_mat_mult_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)
- `armral_status armral_cmplx_mat_mult_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)
- `armral_status armral_cmplx_mat_mult_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)
- `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)
- `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)
- `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)
- `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)

- [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)
- [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)
- [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)
- [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)
- [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)
- [armral_status armral_cmplx_hermitian_mat_inverse_f32](#) (uint16_t size, uint16_t par, const [armral_cmplx_f32_t](#) *p_src, [armral_cmplx_f32_t](#) *p_dst)
- [armral_status armral_seq_generator](#) (uint16_t sequence_len, uint32_t seed, uint8_t *p_dst)
- [armral_status armral_modulation](#) (uint32_t nbits, [armral_modulation_type](#) mod_type, const int8_t *p_src, [armral_cmplx_int16_t](#) *p_dst)
- [armral_status armral_demodulation](#) (uint32_t n_symbols, int16_t amp, uint16_t noise_power, [armral_modulation_type](#) mod_type, const [armral_cmplx_int16_t](#) *p_src, int8_t *p_dst)
- [armral_status armral_corr_coeff_i16](#) (int32_t n, const [armral_cmplx_int16_t](#) *p_src_a, const [armral_cmplx_int16_t](#) *p_src_b, [armral_cmplx_int16_t](#) *c)
- [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)
- [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)
- [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)
- [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)
- [armral_status armral_mu_law_compr_8bit](#) (uint32_t n_prb, const [armral_cmplx_int16_t](#) *src, [armral_compressed_data_8bit](#) *dst)
- [armral_status armral_mu_law_decompr_8bit](#) (uint32_t n_prb, const [armral_compressed_data_8bit](#) *src, [armral_cmplx_int16_t](#) *dst)
- [armral_status armral_block_float_compr_8bit](#) (uint32_t n_prb, const [armral_cmplx_int16_t](#) *src, [armral_compressed_data_8bit](#) *dst)
Block floating-point compression to 8-bit.
- [armral_status armral_block_float_compr_9bit](#) (uint32_t n_prb, const [armral_cmplx_int16_t](#) *src, [armral_compressed_data_9bit](#) *dst)
Block floating point compression to 9-bit big-endian.
- [armral_status armral_block_float_compr_12bit](#) (uint32_t n_prb, const [armral_cmplx_int16_t](#) *src, [armral_compressed_data_12bit](#) *dst)
Block floating point compression to 12-bit big-endian.
- [armral_status armral_block_float_compr_14bit](#) (uint32_t n_prb, const [armral_cmplx_int16_t](#) *src, [armral_compressed_data_14bit](#) *dst)
Block floating point compression to 14-bit big-endian.
- [armral_status armral_block_float_decompr_8bit](#) (uint32_t n_prb, const [armral_compressed_data_8bit](#) *src, [armral_cmplx_int16_t](#) *dst)
Block floating-point decompression from 8 bit.

- `armral_status armral_block_float_decompr_9bit` (uint32_t n_prb, const `armral_compressed_data_9bit` *src, `armral_cmplx_int16_t` *dst)
Block floating point decompression from 9 bit big-endian.
- `armral_status armral_block_float_decompr_12bit` (uint32_t n_prb, const `armral_compressed_data_12bit` *src, `armral_cmplx_int16_t` *dst)
Block floating point decompression from 12 bit big-endian.
- `armral_status armral_block_float_decompr_14bit` (uint32_t n_prb, const `armral_compressed_data_14bit` *src, `armral_cmplx_int16_t` *dst)
Block floating point decompression from 14 bit big-endian.
- `armral_status armral_crc24_a_le` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_a_be` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_b_le` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_b_be` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_c_le` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_crc24_c_be` (uint32_t size, const uint64_t *input, uint64_t *crc24)
- `armral_status armral_polar_encoder` (uint16_t n, const uint32_t *p_u_seq_in, uint32_t *p_d_seq_out)
- `armral_status armral_polar_decoder` (uint16_t n, uint16_t k, uint16_t l, const int8_t *p_llr_in, uint32_t *p_u_seq_out)
- `armral_status armral_fft_create_plan_cf32` (`armral_fft_plan_t` **p, int n, `armral_fft_direction_t` dir)
Creates a plan to solve a complex fp32 FFT.
- `armral_status armral_fft_execute_cf32` (const `armral_fft_plan_t` *p, const `armral_cmplx_f32_t` *x, `armral_cmplx_f32_t` *y)
Performs a single FFT using the specified plan and arrays.
- `armral_status armral_fft_destroy_plan_cf32` (`armral_fft_plan_t` **p)
Destroys an FFT plan.
- `armral_status armral_fft_create_plan_cs16` (`armral_fft_plan_t` **p, int n, `armral_fft_direction_t` dir)
Creates a plan to solve a complex int16 (Q0.15 format) FFT.
- `armral_status armral_fft_execute_cs16` (const `armral_fft_plan_t` *p, const `armral_cmplx_int16_t` *x, `armral_cmplx_int16_t` *y)
Performs a single FFT using the specified plan and arrays.
- `armral_status armral_fft_destroy_plan_cs16` (`armral_fft_plan_t` **p)
Destroys an FFT plan.

9.1.1 Macro Definition Documentation

9.1.1.1 ARMRAL_NUM_COMPLEX_SAMPLES

```
#define ARMRAL_NUM_COMPLEX_SAMPLES 12
```

The number of complex samples in each compressed block.

9.1.2 Enumeration Type Documentation

9.1.2.1 armral_fixed_point_index

```
enum armral_fixed_point_index
```

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

Enumerator

ARMRAL_FIXED_POINT_INDEX_Q15	1 sign bit, 0 integer bits, 15 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q1_14	1 sign bit, 1 integer bit, 14 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q2_13	1 sign bit, 2 integer bits, 13 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q3_12	1 sign bit, 3 integer bits, 12 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q4_11	1 sign bit, 4 integer bits, 11 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q5_10	1 sign bit, 5 integer bits, 10 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q6_9	1 sign bit, 6 integer bits, 9 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q7_8	1 sign bit, 7 integer bits, 8 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q8_7	1 sign bit, 8 integer bits, 7 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q9_6	1 sign bit, 9 integer bits, 6 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q10_5	1 sign bit, 10 integer bits, 5 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q11_4	1 sign bit, 11 integer bits, 4 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q12_3	1 sign bit, 12 integer bits, 3 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q13_2	1 sign bit, 13 integer bits, 2 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q14_1	1 sign bit, 14 integer bits, 1 fractional bit
ARMRAL_FIXED_POINT_INDEX_Q15_0	1 sign bit, 15 integer bits, 0 fractional bits

9.1.2.2 armral_modulation_type

enum [armral_modulation_type](#)

Formats that are supported by modulation and demodulation. See [armral_modulation](#) and [armral_demodulation](#).

Enumerator

ARMRAL_MOD_QPSK	QPSK, size 4 constellation, 2 bits per symbol.
ARMRAL_MOD_16QAM	16QAM, size 16 constellation, 4 bits per symbol
ARMRAL_MOD_64QAM	64QAM, size 64 constellation, 6 bits per symbol
ARMRAL_MOD_256QAM	256QAM, size 256 constellation, 8 bits per symbol

9.1.2.3 armral_status

enum [armral_status](#)

Error status returned by functions in the library.

Enumerator

ARMRAL_SUCCESS	No error.
ARMRAL_ARGUMENT_ERROR	One or more arguments are incorrect.

