arm

Arm® RAN Acceleration Library

Version 22.04

Reference Guide

Arm® RAN Acceleration Library
Reference Guide

Release information

Document history

Issue	Date	Confidentiality	Change
2010-00	2 October 2020	Non-Confidential	New document for Arm RAN Acceleration Library v20.10
2101-00	8 January 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.01
2104-00	9 April 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.04
2107-00	9 July 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.07
2110-00	8 October 2021	Non-Confidential	Update for Arm RAN Acceleration Library v21.10
2201-00	14 January 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.01
2204-00	8 April 2022	Non-Confidential	Update for Arm RAN Acceleration Library v22.04

Proprietary Notice

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

1 Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

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

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

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

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

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

1.2 Other information

See the Arm website for other relevant information.

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

2 Tutorials

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

2.1 Get started with Arm RAN Acceleration Library (ArmRAL)

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

Before you begin

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

- Ensure you have installed all the tools listed in the *Tools* section of the `RELEASE_NOTES.md` file.

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

Build Arm RAN Acceleration Library (ArmRAL)

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

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

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

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

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

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

   Substituting:

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

- `{options}` with the CMake options to use to build the library.

- (Optional) `<install-dir>` with an installation directory name. When you install Arm RAN Acceleration Library (see *Install Arm RAN Acceleration Library*), the library installs to the specified directory. If `<install-dir>` is not specified, the default is `/usr/local`.

- `<path>` with the path to the root directory of the library source.

Notes:

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

- The `-DCMAKE_INSTALL_DIR=<install-dir>` option is optional and sets the install location (`<install-dir>`) for the library. The default location is `/usr/local`.

- By default, a static library is built. To build a dynamic or a static library use the `-DBUILD_SHARED_LIBS={On|Off}` option.

- By default, a Neon-optimized library is built. To specify which type of optimized library to build (Neon, SVE, or SVE2), use the `-DARMRAL_ARCH={NEON|SVE|SVE2}` option.

Other common CMake `{options}` include:

- `-DCMAKE_INSTALL_PREFIX=<path>`

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

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

- `-DCMAKE_BUILD_TYPE={Debug|Release}`

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

  Default is `Release`.

- `-DCMAKE_C_COMPILER=<name>`

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

- `-DCMAKE_CXX_COMPILER=<name>`

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

- `-DBUILD_TESTING={On|Off}`

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

  Default is `Off`.

- `-DARMRAL_TEST_RUNNER=<command>`

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

- `-DSTATIC_TESTING={On|Off}`

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

  Default is `Off`.

- `-DBUILD_EXAMPLES={On|Off}`

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

  Default is `Off`.

- `-DBUILD_SHARED_LIBS={On|Off}`

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

  Default is `Off`.

- `-DARMRAL_ENABLE_WERROR={On|Off}`

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

  Default is `Off`.

- `-DARMRAL_ENABLE_ASAN={On|Off}`

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

  Default is `Off`.

- `-DARMRAL_ENABLE_COVERAGE={On|Off}`

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

  Default is `Off`.

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

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

  Default is `NEON`.

- `-DARMRAL_SEMIHOSTING={On|Off}`

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

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

  Default is `Off`.

Install Arm RAN Acceleration Library (ArmRAL)

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

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

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

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

2. Install the library. Run:

```
make install
```

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

Run the tests

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

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

To build and run the tests, use:

```
make check
```

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

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

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

Run the benchmarks

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

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

To build and run the benchmarks, use:

```
make bench
```

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

Run the examples

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

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

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

```
make run_examples
```

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

```
make examples
```

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

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

Code coverage

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

An example workflow could be:

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

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

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

```
gcovr -r ..
```

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

Documentation

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

```
https://developer.arm.com/documentation/102249/2204
```

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

To build the documentation, run:

```
make docs
```

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

Uninstall Arm RAN Acceleration Library

Describes how to uninstall Arm RAN Acceleration Library.

To uninstall Arm RAN Acceleration Library:

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

```
make uninstall
```

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

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

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

2.2 Use Arm RAN Acceleration Library (ArmRAL)

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

Before you begin

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

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

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

  To build the library, use:

```
tar zxvf arm-ran-acceleration-library-22.04-aarch64.tar.gz
```

```
mkdir arm-ran-acceleration-library-22.04/build
cd arm-ran-acceleration-library-22.04/build
cmake ..
make -j
```

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

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

Procedure

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

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

Substituting:

- `<code-filename>` with the name of your own source code file

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

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

2. Run your binary:

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

Example: Run 'fft_cf32_example.c'

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

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

```
/*
    Arm RAN Acceleration Library
    Copyright 2020-2022 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/source>/
    include -O2
    gcc -o fft_cf32_example fft_cf32_example.o <path/to/armral/build>/libarmral.a -lm
    ```

    Substituting:

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

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

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

    An executable called `fft_cf32_example` is built.

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

```
./fft_cf32_example 5
```

which gives:

```
Planning FFT of length 5
Input Data:
  (0.000000 + 0.000000i)
  (1.000000 + -1.000000i)
  (2.000000 + -2.000000i)
  (3.000000 + -3.000000i)
  (4.000000 + -4.000000i)
Performing FFT of length 5
Destroying plan for FFT of length 5
Result:
  (10.000000 + -10.000000i)
  (0.940955 + 5.940955i)
  (-1.687701 + 3.312299i)
  (-3.312299 + 1.687701i)
  (-5.940955 + -0.940955i)
```

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

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

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

* `block_float_9b_example.c`

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

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

  ```
  ./block_float_9b_example
  ```

* `modulation_example.c`

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

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

  ```
  ./modulation_example
  ```

- `polar_example.cpp`

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

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

  1. The polar code size (N)

  2. The rate-matched codeword length (E)

  3. The number of information bits (K)

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

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

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

3 Functions

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

3.1 Vector functions

Functions for working with vectors.

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

- Vector elementwise multiplication (vector multiply)

- Vector dot product

3.1.1 Vector Multiply

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

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

The vector multiplication algorithm is:

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

3.1.1.1 armral_cmplx_vecmul_i16

This 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], where 0 ≤ n < N-1
```

where:

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

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

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

where:

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

Syntax

Defined in `armral.h` on line `291`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `int32_t`.

The number of samples in each vector.

a

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

Points to the first input vector.

b

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

Points to the second input vector.

c

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

Points to the output vector.

3.1.1.2 armral_cmplx_vecmul_i16_2

This 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], where 0 ≤ n < N-1
```

where:

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

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

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

Syntax

Defined in `armral.h` on line `331`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

    A read-only parameter of type `int32_t`.

    The number of samples in each vector.

a_re

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

    Points to the real part of the first input vector.

a_im

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

Points to the imaginary part of the first input vector.

b_re

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

Points to the real part of the second input vector.

b_im

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

Points to the imaginary part of the second input vector.

c_re

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

Points to the real part of the output result.

c_im

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

Points to the imaginary part of the output result.

3.1.1.3 armral_cmplx_vecmul_f32

This 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], where 0 ≤ n < N-1
```

where:

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

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

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

where:

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

Syntax

Defined in `armral.h` on line `371`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `int32_t`.

The number of samples in each vector.

a

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

Points to the first input vector.

b

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

Points to the second input vector.

c

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

Points to the output vector.

3.1.1.4 armral_cmplx_vecmul_f32_2

This 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], where 0 ≤ n < N-1
```

where:

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

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

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

Syntax

Defined in `armral.h` on line `408`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

 A read-only parameter of type `int32_t`.

 The number of samples in each vector.

a_re

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

 Points to the real part of the first input vector.

a_im

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

 Points to the imaginary part of the first input vector.

b_re

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

 Points to the real part of the second input vector.

b_im

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

 Points to the imaginary part of the second input vector.

c_re

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

 Points to the real part of the output result.

c_im

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

 Points to the imaginary part of the output result.

3.1.2 Vector Dot Product

Computes the dot product of two complex vectors.

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

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

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

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

The dot product algorithm is:

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

3.1.2.1 armral_cmplx_vecdot_f32

This 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.

Syntax

Defined in `armral.h` on line `457`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `int32_t`.

The number of samples in each vector.

p_src_a

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

Points to the first complex input vector.

p_src_b

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

Points to the second complex input vector.

p_src_c

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

Points to the output complex vector.

3.1.2.2 armral_cmplx_vecdot_f32_2

This 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.

Syntax

Defined in `armral.h` on line `479`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `int32_t`.

The number of samples in each vector.

p_src_a_re

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

Points to the real part of the first input vector.

p_src_a_im

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

Points to the imaginary part of the first input vector.

p_src_b_re

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

Points to the real part of the second input vector.

p_src_b_im

> A read-only parameter of type `const float *`.
>
> Points to the imaginary part of the second input vector.

p_src_c_re

> A write-only parameter of type `float *`.
>
> Points to the real part of the output result.

p_src_c_im

> A write-only parameter of type `float *`.
>
> Points to the imaginary part of the output result.

3.1.2.3 armral_cmplx_vecdot_i16

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.

Syntax

Defined in `armral.h` on line `500`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p_src_a

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Points to the first input vector.

p_src_b

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Points to the second input vector.

n

> A read-only parameter of type `int32_t`.

The number of samples in each vector.

p_src_c

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

Points to the output complex result.

3.1.2.4 armral_cmplx_vecdot_i16_2

This 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.

Syntax

Defined in `armral.h` on line `524`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

p_src_a_re

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

Points to the real part of first input vector.

p_src_a_im

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

Points to the imag part of first input vector.

p_src_b_re

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

Points to the real part of second input vector.

p_src_b_im

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

Points to the imag part of second input vector.

n

A read-only parameter of type `int32_t`.

The number of samples in each vector.

p_src_c_re

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

Points to the real part of output complex result.

p_src_c_im

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

Points to the imag part of output complex result.

3.1.2.5 armral_cmplx_vecdot_i16_32bit

This 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.

Syntax

Defined in `armral.h` on line `546`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `int32_t`.

The number of samples in each vector.

p_src_a

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

Points to the first input vector.

p_src_b

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

Points to the second input vector.

p_src_c

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

Points to the output complex result.

3.1.2.6 armral_cmplx_vecdot_i16_2_32bit

This 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.

Syntax

Defined in `armral.h` on line `571`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `int32_t`.

The number of samples in each vector.

p_src_a_re

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

Points to the real part of the first input vector.

p_src_a_im

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

Points to the imaginary part of the first input vector.

p_src_b_re

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

Points to the real part of the second input vector.

p_src_b_im

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

Points to the imaginary part of the second input vector.

p_src_c_re

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

Points to the real part of the output result.

p_src_c_im

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

Points to the imaginary part of the output result.

3.2 Matrix functions

Functions for working with matrices.

Functions are provided for working with matrices, including:

- Matrix-vector multiplication for 16-bit integer datatypes.

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

- Matrix inversion. Supports the 32-bit floating-point datatype.

3.2.1 Complex Matrix-Vector Multiplication

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

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

3.2.1.1 armral_cmplx_mat_vec_mult_i16

This algorithm performs the multiplication `A x` for matrix `A` and vector `x`, and assumes that:

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

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

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

Syntax

Defined in `armral.h` on line `607`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows in matrix `A`.

n

A read-only parameter of type `uint16_t`.

The number of columns in matrix `A`.

p_src_a

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

Points to the input matrix.

p_src_x

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

Points to the input vector.

p_dst

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

Points to the output vector.

3.2.1.2 armral_cmplx_mat_vec_mult_batch_i16

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

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

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

The matrix elements are interleaved such that all elements for a particular location within the matrix are stored together. This means that, for instance, the first `num_mats` complex numbers

stored are the first element of each of the matrices in the batch. The offset to the next location in the same matrix is given by the `num_mats` batch size:

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

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

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

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

Syntax

Defined in `armral.h` on line `654`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_mats`

　　A read-only parameter of type `uint16_t`.

　　The number of input matrices.

`num_vecs_per_mat`

　　A read-only parameter of type `uint16_t`.

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

`m`

　　A read-only parameter of type `uint16_t`.

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

`n`

　　A read-only parameter of type `uint16_t`.

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

p_src_a

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Points to the input matrix.

p_src_x

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Points to the input vector.

p_dst

> A write-only parameter of type `armral_cmplx_int16_t *`.
>
> Points to the output vector.

3.2.1.3 armral_cmplx_mat_vec_mult_batch_i16_pa

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

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

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

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

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

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

representing an identical format to the input matrices.

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

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

Syntax

Defined in `armral.h` on line `710`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_mats

> A read-only parameter of type `uint16_t`.
>
> The number of input matrices.

num_vecs_per_mat

> A read-only parameter of type `uint16_t`.
>
> The number of input and output vectors for each input matrix. The total number of input vectors is `num_mats * num_vecs_per_mat`. There are the same number of output vectors.

m

> A read-only parameter of type `uint16_t`.
>
> The number of rows (M) in each matrix A.

n

> A read-only parameter of type `uint16_t`.
>
> The number of columns (N) in each matrix A.

p_srcs_a

> A read-only parameter of type `const armral_cmplx_int16_t **`.
>
> Points to the input matrix structure.

p_srcs_x

> A read-only parameter of type `const armral_cmplx_int16_t **`.
>
> Points to the input vector structure.

p_dsts

> A write-only parameter of type `armral_cmplx_int16_t **`.
>
> Points to the output vector structure.

3.2.1.4 armral_cmplx_mat_vec_mult_i16_32bit

This algorithm performs the multiplication `A x` for matrix `A` and vector `x`, and assumes that:

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

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

A 32-bit Q0.31 saturating accumulator is used internally. If you need a larger range, consider using armral_cmplx_mat_vec_mult_i16 instead. To get a Q15 result, the final result is narrowed to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `733`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

    A read-only parameter of type `uint16_t`.

    The number of rows in matrix `A`.

n

    A read-only parameter of type `uint16_t`.

    The number of columns in matrix `A`.

p_src_a

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

    Points to the input matrix.

p_src_x

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

    Points to the input vector.

p_dst

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

    Points to the output matrix.

3.2.1.5 armral_cmplx_mat_vec_mult_batch_i16_32bit

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

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

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

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

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

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

A 32-bit Q0.31 saturating accumulator is used internally. If you need a larger range, consider using armral_cmplx_mat_vec_mult_batch_i16 instead. To get a Q15 result, the final result is narrowed to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `774`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_mats

A read-only parameter of type `uint16_t`.

The number of input matrices.

num_vecs_per_mat

A read-only parameter of type `uint16_t`.

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

m

A read-only parameter of type `uint16_t`.

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

n

A read-only parameter of type `uint16_t`.

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

p_src_a

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

Points to the input matrix.

p_src_x

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

Points to the input vector.

p_dst

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

Points to the output vector.

3.2.1.6 armral_cmplx_mat_vec_mult_batch_i16_32bit_pa

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

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

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

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

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

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

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

representing an identical format to the input matrices.

A 32-bit Q0.31 saturating accumulator is used internally. If you need a larger range, consider using armral_cmplx_mat_vec_mult_batch_i16_pa instead. To get a Q15 result, the final result is narrowed to 16 bits with saturation.

Syntax

Defined in `armral.h` on line `825`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_mats

A read-only parameter of type `uint16_t`.

The number of input matrices.

num_vecs_per_mat

A read-only parameter of type `uint16_t`.

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

m

A read-only parameter of type `uint16_t`.

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

n

A read-only parameter of type `uint16_t`.

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

p_srcs_a

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

Points to the input matrix structure.

p_srcs_x

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

Points to the input vector structure.

p_dsts

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

Points to the output vector structure.

3.2.1.7 armral_cmplx_mat_vec_mult_f32

This algorithm performs the multiplication `A` `x` for matrix `A` and vector `x`, and assumes that:

- Matrix and vector elements are float values.

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

Syntax

Defined in `armral.h` on line `843`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows in matrix `A`.

n

A read-only parameter of type `uint16_t`.

The number of columns in matrix `A`.

p_src_a

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

Points to the first input matrix.

p_src_x

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

Points to the input vector.

p_dst

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

Points to the output matrix.

3.2.2 Complex Matrix-Matrix Multiplication

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

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

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

3.2.2.1 armral_cmplx_mat_mult_i16

This algorithm performs the multiplication `A` `B` for matrices, and assumes that:

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

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

Syntax

Defined in `armral.h` on line `883`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows in matrix `A`.

n

A read-only parameter of type `uint16_t`.

The number of columns in matrix `A`.

k

A read-only parameter of type `uint16_t`.

The number of columns in matrix `B`.

p_src_a

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Points to the first input matrix.

p_src_b

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Points to the second input matrix.

p_dst

> A write-only parameter of type `armral_cmplx_int16_t *`.
>
> Points to the output matrix.

3.2.2.2 armral_cmplx_mat_mult_i16_32bit

This algorithm performs the multiplication A B for matrices, and assumes that:

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

A 32-bit 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.

Syntax

Defined in `armral.h` on line `907`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

> A read-only parameter of type `uint16_t`.
>
> The number of rows in matrix A.

n

> A read-only parameter of type `uint16_t`.
>
> The number of columns in matrix A.

k

A read-only parameter of type `uint16_t`.

The number of columns in matrix B.

p_src_a

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

Points to the first input matrix.

p_src_b

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

Points to the second input matrix.

p_dst

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

Points to the output matrix.

3.2.2.3 armral_cmplx_mat_mult_f32

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

Syntax

Defined in `armral.h` on line `923`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

m

A read-only parameter of type `uint16_t`.

The number of rows in matrix A.

n

A read-only parameter of type `uint16_t`.

The number of columns in matrix A.

k

A read-only parameter of type `uint16_t`.

The number of columns in matrix `B`.

p_src_a

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

Points to the first input matrix.

p_src_b

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

Points to the second input matrix.

p_dst

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

Points to the output matrix.

3.2.2.4 armral_cmplx_mat_mult_2x2_f32

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 assumed 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:

```
xˆ = G y
```

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

The second input matrix (matrix `B`) is formed by two 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 `xˆ` estimates.

Syntax

Defined in `armral.h` on line `954`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

p_src_a

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

Points to the first input matrix.

p_src_b

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

Points to the second input matrix.

p_dst

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

Points to the output matrix.

3.2.2.5 armral_cmplx_mat_mult_2x2_f32_iq

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:

```
x^ = G y
```

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

The second input matrix (matrix B) is formed by two 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_iq` computes two distinct x^ estimates.

Syntax

Defined in `armral.h` on line `989`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

src_a_re

> A read-only parameter of type `const float32_t *`.
>
> Points to the real part of the first input matrix.

src_a_im

> A read-only parameter of type `const float32_t *`.
>
> Points to the imag part of the first input matrix.

src_b_re

> A read-only parameter of type `const float32_t *`.
>
> Points to the real part of the second input matrix.

src_b_im

> A read-only parameter of type `const float32_t *`.
>
> Points to the imag part of the second input matrix.

dst_re

> A write-only parameter of type `float32_t *`.
>
> Points to the real part of the output matrix.

dst_im

> A write-only parameter of type `float32_t *`.
>
> Points to the imag part of the output matrix.

3.2.2.6 armral_cmplx_mat_mult_4x4_f32

This algorithm performs an optimized product of two square 4x4 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:

```
x^ = G y
```

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

The algorithm assumes that matrix G is transposed during computation so that the matrix presents as column-major on input.

The second input matrix (matrix B) is formed by four 4x1 vectors (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 `cmplx_mat_mult_4x4_f32` computes four distinct x^ estimates.

Syntax

Defined in `armral.h` on line `1022`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

p_src_a

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

   Points to the first input matrix.

p_src_b

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

   Points to the second input matrix.

p_dst

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

   Points to the output matrix.

3.2.2.7 armral_cmplx_mat_mult_4x4_f32_iq

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:

```
x^ = G y
```

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

The second input matrix (matrix `B`) is formed by four 4x1 vectors (`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 `x^` estimates.

Syntax

Defined in `armral.h` on line `1056`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

src_a_re

> A read-only parameter of type `const float32_t *`.
>
> Points to the real part of the first input matrix.

src_a_im

> A read-only parameter of type `const float32_t *`.
>
> Points to the imag part of the first input matrix.

src_b_re

> A read-only parameter of type `const float32_t *`.
>
> Points to the real part of the second input matrix.

src_b_im

> A read-only parameter of type `const float32_t *`.
>
> Points to the imag part of the second input matrix.

dst_re

> A write-only parameter of type `float32_t *`.
>
> Points to the real part of the output matrix.

dst_im

> A write-only parameter of type `float32_t *`.
>
> Points to the imag part of the output matrix.

3.2.2.8 armral_solve_2x2_f32

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

```
xˆ = G y
```

where `y` is a vector for the received signal, size corresponds to the number of antennae and `xˆ` 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` (`G11`, `G12`, `G21`, `G22`) 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 is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1098`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_sub_carrier`

 A read-only parameter of type `uint32_t`.

 The number of subcarriers to equalize.

`num_sc_per_g`

 A read-only parameter of type `uint32_t`.

 The number of subcarriers per `G` matrix.

p_y

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

Points to the input received signal.

p_ystride

A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

p_y_num_fract_bits

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

The number of fractional bits in `y`.

p_g_real

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

The real part of coefficient matrix `G`.

p_g_imag

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

The imag part of coefficient matrix `G`.

p_gstride

A read-only parameter of type `uint32_t`.

The stride between elements of `G`.

p_x

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

Points to the output received signal.

p_xstride

A read-only parameter of type `uint32_t`.

The stride between two layers.

num_fract_bits_x

A read-only parameter of type `armral_fixed_point_index`.

The number of fractional bits in `x`.

3.2.2.9 armral_solve_2x4_f32

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

```
x^ = G y
```

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

`G` is the equalization complex matrix and is assumed to be a 2x4 matrix. I and Q components of `G` elements are assumed to be stored separated in memory.

Also, each coefficient of `G` (`G11`, `G12`, `G21`, `G22`) 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 is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1142`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_sub_carrier

A read-only parameter of type `uint32_t`.

The number of subcarrier to equalize.

num_sc_per_g

A read-only parameter of type `uint32_t`.

The number of subcarriers per `G` matrix.

p_y

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

Points to the input received signal.

p_ystride

A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

p_y_num_fract_bits

> A read-only parameter of type `const armral_fixed_point_index *`.
>
> The number of fractional bits in `y`.

p_g_real

> A read-only parameter of type `const float32_t *`.
>
> The real part of coefficient matrix `G`.

p_g_imag

> A read-only parameter of type `const float32_t *`.
>
> The imag part of coefficient matrix `G`.

p_gstride

> A read-only parameter of type `uint32_t`.
>
> The stride between elements of `G`.

p_x

> A write-only parameter of type `armral_cmplx_int16_t *`.
>
> Points to the output received signal.

p_xstride

> A read-only parameter of type `uint32_t`.
>
> The stride between two layers.

num_fract_bits_x

> A read-only parameter of type `armral_fixed_point_index`.
>
> The number of fractional bits in `x`.

3.2.2.10 armral_solve_4x4_f32

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

```
x^ = G y
```

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

`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` (`G11`, `G12`, `G21`, `G22`) 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 is also available for cases where the number of subcarriers per G matrix is equal to one.

Syntax

Defined in `armral.h` on line `1186`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_sub_carrier

A read-only parameter of type `uint32_t`.

The number of subcarrier to equalize.

num_sc_per_g

A read-only parameter of type `uint32_t`.

The number of subcarriers per G matrix.

p_y

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

Points to the input received signal.

p_ystride

A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

p_y_num_fract_bits

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

The number of fractional bits in y.

p_g_real

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

The real part of coefficient matrix G.

p_g_imag

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

> The imag part of coefficient matrix `G`.

p_gstride

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

> The stride between elements of `G`.

p_x

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

> Points to the output received signal.

p_xstride

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

> The stride between two layers.

num_fract_bits_x

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

> The number of fractional bits in `x`.

3.2.2.11 armral_solve_1x4_f32

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

```
xˆ = G y
```

where `y` is a vector for the received signal, size corresponds to the number of antennae and `xˆ` 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` (`G11`, `G12`, `G21`, `G22`) 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 is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1230`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_sub_carrier

A read-only parameter of type `uint32_t`.

The number of subcarrier to equalize.

num_sc_per_g

A read-only parameter of type `uint32_t`.

The number of subcarriers per `G` matrix.

p_y

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

Points to the input received signal.

p_ystride

A read-only parameter of type `uint32_t`.

The stride between two Rx antennae.

p_y_num_fract_bits

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

The number of fractional bits in `y` conversion.

p_g_real

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

The real part of coefficient matrix `G`.

p_g_imag

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

The imag part of coefficient matrix `G`.

p_gstride

A read-only parameter of type `uint32_t`.

The stride between elements of `G`.

p_x

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

Points to the output received signal.

num_fract_bits_x

A read-only parameter of type `armral_fixed_point_index`.

The number of fractional bits in `x`.

3.2.2.12 armral_solve_1x2_f32

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

```
x^ = G y
```

where `y` is a vector for the received signal, size corresponds to the number of antennae and `x^` is the estimate of the transmitted signal, size corresponds to the number of layers. `G` is the equalization complex matrix and is assumed to be a 2x4 matrix. I and Q components of `G` elements are assumed to be stored separated in memory.

Also, each coefficient of `G` (`G11`, `G12`, `G21`, `G22`) 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 is also available for cases where the number of subcarriers per `G` matrix is equal to one.

Syntax

Defined in `armral.h` on line `1273`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_sub_carrier

      A read-only parameter of type `uint32_t`.

      The number of subcarrier to equalize.

num_sc_per_g

      A read-only parameter of type `uint32_t`.

      The number of subcarriers per `G` matrix.

p_y

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

      Points to the input received signal.

p_ystride

      A read-only parameter of type `uint32_t`.

      The stride between two Rx antennae.

p_y_num_fract_bits

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

      The number of fractional bits in `y` conversion.

p_g_real

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

      The real part of coefficient matrix `G`.

p_g_imag

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

      The imag part of coefficient matrix `G`.

p_gstride

      A read-only parameter of type `uint32_t`.

      The stride between elements of `G`.

p_x

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

      Points to the output received signal.

num_fract_bits_x

      A read-only parameter of type `armral_fixed_point_index`.

      The number of fractional bits in `x`.

3.2.3 Complex Matrix Inversion

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

3.2.3.1 armral_cmplx_hermitian_mat_inverse_f32

This algorithm computes the inverse of a single complex Hermitian square matrix of size `N x N`.

The supported dimensions are `2 x 2`, `3 x 3`, `4 x 4`, `8 x 8`, and `16 x 16`.

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

Syntax

Defined in `armral.h` on line `1303`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

    A read-only parameter of type `uint32_t`.

    The size of the input matrix.

`p_src`

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

    Points to the input matrix structure.

`p_dst`

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

    Points to the output matrix structure.

3.2.3.2 armral_cmplx_mat_inverse_f32

This algorithm computes the inverse of a single complex square matrix of size `N x N`.

The supported dimensions are `2 x 2`, `3 x 3`, `4 x 4`, `8 x 8`, and `16 x 16`.

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

Syntax

Defined in `armral.h` on line `1317`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

`size`

A read-only parameter of type `uint32_t`.

The size of the input matrix.

`p_src`

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

Points to the input matrix structure.

`p_dst`

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

Points to the output matrix structure.

3.2.3.3 armral_cmplx_hermitian_mat_inverse_batch_f32

This algorithm computes the inverse of a batch of `M` complex Hermitian square matrices, each of size `N x N`.

The supported matrix dimensions are `2 x 2`, `3 x 3`, and `4 x 4`.

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

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

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

Syntax

Defined in `armral.h` on line `1344`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

`num_mats`

A read-only parameter of type `uint32_t`.

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

`size`

A read-only parameter of type `uint32_t`.

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

`p_src`

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

Points to the input matrix structure.

`p_dst`

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

Points to the output matrix structure.

3.2.3.4 armral_cmplx_hermitian_mat_inverse_batch_f32_pa

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

The supported matrix dimensions are `2 x 2`, `3 x 3`, and `4 x 4`.

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

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

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

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

Syntax

Defined in `armral.h` on line `1376`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

num_mats

A read-only parameter of type `uint32_t`.

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

size

A read-only parameter of type `uint32_t`.

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

p_srcs

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

Points to the input matrix structure.

p_dsts

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

Points to the output matrix structure.

3.2.4 SVD decomposition of single complex matrix

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

3.2.4.1 armral_svd_cf32

This algorithm performs the Singular Value Decomposition (SVD) of an `m` x `n` single complex matrix `A`, where `m` ≥ `n` and `A` is stored in column-major order. The SVD of `A` is a two-sided decomposition in the form `A` = `U` Σ `V^H`, with `U` an `m` x `m` single complex orthogonal matrix. Note that we only

store the first n columns of U because there are at most n non-zero singular values. V is an $n \times n$ single complex orthogonal matrix, and Σ is an $m \times n$ real matrix. Entries $\Sigma_{\{i, i\}}$, $i < n$ contain the singular values, and other entries in Σ are zero. We only store the singular values, not the full matrix Σ. The singular values $\Sigma_{\{i,i\}}$ are stored in vector s for $0 \le i < n$. The matrices U and V^H are implicitly used in the algorithm, unless parameter vect is specified to be true, in which case the left and right singular vectors (respectively) are stored in U and V^H in column-major order. This means that singular vectors are stored contiguously in U, and are non-contiguous in V^H. Note that it is V^H that is returned, not V.

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

Syntax

Defined in `armral.h` on line `2789`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

vect

A read-only parameter of type `bool`.

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

m

A read-only parameter of type `int`.

The number of rows in matrix A.

n

A read-only parameter of type `int`.

The number of columns in matrix A.

a

A parameter of type `armral_cmplx_f32_t *`.

On entry contains the $m \times n$ matrix on which to perform the SVD. On exit contains the Householder reflectors used to perform the bidiagonalization of A.

s

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

The vector of singular values.

u

> A write-only parameter of type `armral_cmplx_f32_t *`.
>
> The left singular vectors, if required. If `vect` is true, `u` is populated with the left singular vectors in the SVD. A memory of `m x n` is assumed to have been allocated before the call to this method.

vt

> A write-only parameter of type `armral_cmplx_f32_t *`.
>
> The right singular vectors, if required. If `vect` is true, `vt` is populated with the right singular vectors in the SVD. A memory of `n x n` is assumed to have been allocated before the call to this method.

3.3 Lower PHY support functions

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

The Lower PHY functions include support for:

- A Gold Sequence generator

- A correlation coefficient of a pair of 16-bit integer arrays (in Q15 format)

- FIR filters. Supports both 16-bit integer and 32-bit floating-point datatypes. Support is provided for decimation factors of both one and two.

- Fast Fourier Transforms (FFTs). Supports both 16-bit integer and 32-bit floating-point datatypes.

3.3.1 Sequence Generator

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

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

3.3.1.1 armral_seq_generator

This 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 the 3GPP Technical Specification (TS) 36.211, Chapter 7.2. The generator uses two polynomials, `x1` and `x2`, defined as:

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

to generate the output sequence:

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

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

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

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

Syntax

Defined in `armral.h` on line `1431`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

sequence_len

A read-only parameter of type `uint16_t`.

The length of the sequence in bits (cinit).

seed

A read-only parameter of type `uint32_t`.

The random sequence starting point.

p_dst

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

Points to the output bits.

3.3.2 Correlation Coefficient

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

3.3.2.1 armral_corr_coeff_i16

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

Pearson's correlation coefficient is calculated using:

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

Syntax

Defined in `armral.h` on line `1542`:

```
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);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `int32_t`.

The number of complex samples in each vector.

p_src_a

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

Points to the first input vector.

p_src_b

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

Points to the second input vector.

c

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

Points to the result.

3.3.3 FIR filter

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

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

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

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

3.3.3.1 armral_fir_filter_cf32

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

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

Syntax

Defined in `armral.h` on line `1586`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

> A read-only parameter of type `uint32_t`.
>
> The number of complex samples in input.

taps

> A read-only parameter of type `uint32_t`.
>
> The number of taps of the FIR filter.

input

> A read-only parameter of type `const armral_cmplx_f32_t *`.
>
> Points to the input samples buffer.

coeffs

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

>   Points to the coefficients array.

output

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

>   Points to the output array.

3.3.3.2 armral_fir_filter_cf32_decimate_2

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

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

Syntax

Defined in `armral.h` on line `1608`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

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

>   The number of complex samples in input.

taps

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

>   The number of taps of the FIR filter.

input

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

>   Points to the input samples buffer.

coeffs

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

>   Points to the coefficients array.

output

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

Points to the output array.

3.3.3.3 armral_fir_filter_cs16

Computes a complex signed 16-bit integer FIR filter.

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

Syntax

Defined in `armral.h` on line `1626`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of complex samples in input.

taps

A read-only parameter of type `uint32_t`.

The number of taps of the FIR filter.

input

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

Points to the input samples buffer.

coeffs

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

Points to the coefficients array.

output

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

Points to the output array.

3.3.3.4 armral_fir_filter_cs16_decimate_2

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

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

Syntax

Defined in `armral.h` on line `1648`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of complex samples in input.

taps

A read-only parameter of type `uint32_t`.

The number of taps of the FIR filter.

input

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

Points to the input samples buffer.

coeffs

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

Points to the coefficients array.

output

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

Points to the output array.

3.3.4 Fast Fourier Transforms (FFT)

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

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

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

3.3.4.1 armral_fft_create_plan_cf32

Creates a plan to solve a complex fp32 FFT.

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.

Syntax

Defined in `armral.h` on line `2517`:

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

Returns

An `armral_status` value that indicates success or failure

Parameters

p

A parameter of type `armral_fft_plan_t **`.

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

n

A read-only parameter of type `int`.

The problem size to be solved by this FFT plan.

dir

A write-only parameter of type `armral_fft_direction_t`.

The direction to be solved by this FFT plan.

3.3.4.2 armral_fft_execute_cf32

Performs a single FFT using the specified plan and arrays.

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

Syntax

Defined in `armral.h` on line `2537`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

> A read-only parameter of type `const armral_fft_plan_t *`.
>
> A pointer to the FFT plan. The pointer is the value that is filled in by an earlier call to armral_fft_create_plan_cf32.

x

> A read-only parameter of type `const armral_cmplx_f32_t *`.
>
> The input array for this FFT. The length must be the same as the value of `n` that was previously passed to armral_fft_create_plan_cf32.

y

> A write-only parameter of type `armral_cmplx_f32_t *`.
>
> The output array for this FFT. The length must be the same as the value of `n` that was previously passed to armral_fft_create_plan_cf32.

3.3.4.3 armral_fft_destroy_plan_cf32

Destroys an FFT plan.

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

Syntax

Defined in `armral.h` on line `2554`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

> A parameter of type `armral_fft_plan_t **`.
>
> A pointer to the FFT plan pointer. The pointer must be the value that is returned by an earlier call to armral_fft_create_plan_cf32. On function exit, the value that is pointed to is set to `NULL`.

3.3.4.4 armral_fft_create_plan_cs16

Creates a plan to solve a complex int16 (Q0.15 format) FFT.

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.

Syntax

Defined in `armral.h` on line `2575`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

> A parameter of type `armral_fft_plan_t **`.
>
> A pointer to the resulting plan pointer. On output `*p` is a valid pointer, to be passed to armral_fft_execute_cs16.

n

> A read-only parameter of type `int`.

The problem size to be solved by this FFT plan.

dir

A write-only parameter of type `armral_fft_direction_t`.

The direction to be solved by this FFT plan.

3.3.4.5 armral_fft_execute_cs16

Performs a single FFT using the specified plan and arrays.

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

Syntax

Defined in `armral.h` on line `2595`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

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

A pointer to the FFT plan. The pointer is the value that is filled in by an earlier call to armral_fft_create_plan_cs16.

x

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

The input array for this FFT. The length must be the same as the value of `n` that was previously passed to armral_fft_create_plan_cs16.

y

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

The output array for this FFT. The length must be the same as the value of `n` that was previously passed to armral_fft_create_plan_cs16.

3.3.4.6 armral_fft_destroy_plan_cs16

Destroys an FFT plan.

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

Syntax

Defined in `armral.h` on line `2612`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

p

> A parameter of type `armral_fft_plan_t **`.
>
> A pointer to the FFT plan pointer. The pointer must be the value that is returned by an earlier call to armral_fft_create_plan_cs16. On function exit, the value that is pointed to is set to `NULL`.

3.4 Upper PHY support functions

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

The Upper PHY functions include support for:

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

3.4.1 Modulation

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

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

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

3.4.1.1 armral_modulation

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

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

Syntax

Defined in `armral.h` on line `1469`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

nbits

A read-only parameter of type `uint32_t`.

The number of input modulated bits.

mod_type

A read-only parameter of type `armral_modulation_type`.

The type of modulation to perform.

p_src

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

Points to input bit flow.

p_dst

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

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

3.4.1.2 armral_demodulation

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

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

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

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

The LLRs calculations are made approximately with thresholds method, to have similar performance of the raw calculation, but with a lower complexity. The base of the logarithm used depends on the noise power and the specified `ulp` .

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

Syntax

Defined in `armral.h` on line `1506`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_symbols

  A read-only parameter of type `uint32_t`.

  The number of complex symbols in the input.

ulp

  A read-only parameter of type `uint16_t`.

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

mod_type

  A read-only parameter of type `armral_modulation_type`.

  The modulation type.

p_src

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

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

p_dst

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

      Points to the output byte seq.

3.4.2 CRC

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

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

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

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

3.4.2.1 armral_crc24_a_le

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

Syntax

Defined in `armral.h` on line `2156`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

      A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

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

Points to the input byte sequence.

crc24

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

The computed 24-bit CRC result.

3.4.2.2 armral_crc24_a_be

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

Syntax

Defined in `armral.h` on line `2168`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

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

Points to the input byte sequence.

crc24

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

The computed 24-bit CRC result.

3.4.2.3 armral_crc24_b_le

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

Syntax

Defined in `armral.h` on line `2180`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

> A read-only parameter of type `uint32_t`.
>
> The number of bytes of the given buffer.

input

> A read-only parameter of type `const uint64_t *`.
>
> Points to the input byte sequence.

crc24

> A write-only parameter of type `uint64_t *`.
>
> The computed 24-bit CRC result.

3.4.2.4 armral_crc24_b_be

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

Syntax

Defined in `armral.h` on line `2192`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

> A read-only parameter of type `uint32_t`.
>
> The number of bytes of the given buffer.

input

> A read-only parameter of type `const uint64_t *`.
>
> Points to the input byte sequence.

crc24

> A write-only parameter of type `uint64_t *`.
>
> The computed 24-bit CRC result.

3.4.2.5 armral_crc24_c_le

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

Syntax

Defined in `armral.h` on line `2204`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

> A read-only parameter of type `uint32_t`.
>
> The number of bytes of the given buffer.

input

> A read-only parameter of type `const uint64_t *`.
>
> Points to the input byte sequence.

crc24

> A write-only parameter of type `uint64_t *`.
>
> The computed 24-bit CRC result.

3.4.2.6 armral_crc24_c_be

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

Syntax

Defined in `armral.h` on line `2216`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

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

Points to the input byte sequence.

crc24

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

The computed 24-bit CRC result.

3.4.2.7 armral_crc16_le

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

Syntax

Defined in `armral.h` on line `2228`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

> A read-only parameter of type `uint32_t`.
>
> The number of bytes of the given buffer.

input

> A read-only parameter of type `const uint64_t *`.
>
> Points to the input byte sequence.

crc16

> A write-only parameter of type `uint64_t *`.
>
> The computed 16-bit CRC result.

3.4.2.8 armral_crc16_be

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

Syntax

Defined in `armral.h` on line `2240`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

> A read-only parameter of type `uint32_t`.
>
> The number of bytes of the given buffer.

input

> A read-only parameter of type `const uint64_t *`.
>
> Points to the input byte sequence.

crc16

> A write-only parameter of type `uint64_t *`.
>
> The computed CRC on 16 bit.

3.4.2.9 armral_crc11_le

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

Syntax

Defined in `armral.h` on line `2252`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

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

Points to the input byte sequence.

crc11

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

The computed 11-bit CRC result.

3.4.2.10 armral_crc11_be

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

Syntax

Defined in `armral.h` on line `2264`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

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

Points to the input byte sequence.

crc11

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

The computed CRC on 11 bit.

3.4.2.11 armral_crc6_le

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

Syntax

Defined in `armral.h` on line `2276`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

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

Points to the input byte sequence.

crc6

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

The computed 6-bit CRC result.

3.4.2.12 armral_crc6_be

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

Syntax

Defined in `armral.h` on line `2288`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

size

A read-only parameter of type `uint32_t`.

The number of bytes of the given buffer.

input

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

Points to the input byte sequence.

crc6

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

The computed CRC on 6 bit.

3.4.3 Polar encoding

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

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

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

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

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

3.4.3.1 armral_polar_frozen_mask

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

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

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

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

Syntax

Defined in `armral.h` on line `2359`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

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

e

A read-only parameter of type `uint32_t`.

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

k

A read-only parameter of type `uint32_t`.

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

n_pc

A read-only parameter of type `uint32_t`.

The number of parity bits in the encoded message.

n_pc_wm

A read-only parameter of type `uint32_t`.

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

frozen

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

The output `frozen` mask, length `n` bytes. Elements corresponding to `frozen` bits are set to all ones, everything else set to zero.

3.4.3.2 armral_polar_subchannel_interleave

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

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

Syntax

Defined in `armral.h` on line `2386`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

The polar code size `n`.

kplus

A read-only parameter of type `uint32_t`.

The number of information bits plus the number of parity bits: $K' = K + n\_pc$.

frozen

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

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

c

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

The input codeword, of length K bits.

u

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

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

3.4.3.3 armral_polar_subchannel_deinterleave

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

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

Syntax

Defined in `armral.h` on line `2413`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

k

A read-only parameter of type `uint32_t`.

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

frozen

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

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

u

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

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

c

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

The output codeword, of length `K` bits.

3.4.3.4 armral_polar_encoder

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

Syntax

Defined in `armral.h` on line `2429`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

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

p_u_seq_in

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

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

p_d_seq_out

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

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

3.4.3.5 armral_polar_decoder

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

If `l=1`, the decoder uses a Successive Cancellation (SC) method. If `l>1` , the decoder uses a Successive Cancellation List (SCL) method instead. `l` candidate codewords are maintained and returned, sorted by worsening path metric (in other words, the first returned value is the most likely to be correct). Not all list sizes are supported. Unsupported values of `n` or `l` will return `ARMRAL_ARGUMENT_ERROR`.

Syntax

Defined in `armral.h` on line `2455`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n

A read-only parameter of type `uint32_t`.

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

frozen

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

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

l

A read-only parameter of type `uint32_t`.

The list size to be used in decoding.

p_llr_in

A read-only parameter of type `const int8_t *`.

Points to the input sequence of LLR bytes.

p_u_seq_out

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

Points to `l` decoded sequences, ordered by decreasing path metric, each of length `n` bits.

3.4.4 Low-Density Parity Check (LDPC)

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

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

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

3.4.4.1 armral_ldpc_get_base_graph

Uses the identifier of a base graph to get the data structure that describes a base graph.

Syntax

Defined in `armral.h` on line `2695`:

```
const armral_ldpc_base_graph_t *
armral_ldpc_get_base_graph(armral_ldpc_graph_t bg);
```

Returns

A pointer to an LDPC base graph.

Parameters

bg

A read-only parameter of type `armral_ldpc_graph_t`.

Enum identifier of the base graph to get.

3.4.4.2 armral_ldpc_encode_block

Performs encoding using LDPC as layed out in the 3GPP Technical Specification (TS) 38.212.

Syntax

Defined in `armral.h` on line `2713`:

```
armral_status armral_ldpc_encode_block(const uint8_t *data_in,
                                       armral_ldpc_graph_t bg, uint32_t z,
```

```
                                                    uint8_t *data_out);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

data_in

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

The message bits to encode. It is assumed that the number of bits stored in `data_in` fits into a single code block.

bg

A read-only parameter of type `armral_ldpc_graph_t`.

Identifier for the base graph to use for encoding.

z

A read-only parameter of type `uint32_t`.

The lifting size.

data_out

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

The codeword to be transmitted. `data_out` has the first two columns for the base graphs punctured, and contains the parity bits for the message passed in.

3.4.4.3 armral_ldpc_decode_block

Syntax

Defined in `armral.h` on line `2727`:

```
armral_status armral_ldpc_decode_block(const int8_t *llrs,
                                       armral_ldpc_graph_t bg, uint32_t z,
                                       uint32_t num_its, uint8_t *data_out);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

llrs

A read-only parameter of type `const int8_t *`.

The log likelihood ratios for each of the bits received.

`bg`

A read-only parameter of type `armral_ldpc_graph_t`.

The type of base graph to use for the decoding.

`z`

A read-only parameter of type `uint32_t`.

The lifting size to use.

`num_its`

A read-only parameter of type `uint32_t`.

The maximum number of iterations of the LDPC decoder to run.

`data_out`

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

The decoded bits.

3.5 DU-RU IF support functions

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

The DU-RU IF functions include support for:

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

3.5.1 Mu-Law Compression

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

3.5.1.1 armral_mu_law_compr_8bit

The Mu-Law compression method combines a bit-shift operation for dynamic range with a nonlinear piece-wise approximation of the original logarithmic Mu-Law. The Mu-Law compression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 12 complex output samples, each 8 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1680`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

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

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_8bit *`.

Points to the output 8-bit data and exponent.

scale

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

Phase compensation term to use, or NULL.

3.5.1.2 armral_mu_law_compr_9bit

The Mu-Law compression method combines a bit-shift operation for dynamic range with a nonlinear piece-wise approximation of the original logarithmic Mu-Law. The Mu-Law compression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 12 complex output samples, each 9 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1702`:

```
armral_status armral_mu_law_compr_9bit(uint32_t n_prb,
                                       const armral_cmplx_int16_t *src,
                                       armral_compressed_data_9bit *dst,
```

```
                                    const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

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

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_9bit *`.

Points to the output 9-bit data and shift.

scale

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

Phase compensation term to use, or NULL.

3.5.1.3 armral_mu_law_compr_14bit

The Mu-Law compression method combines a bit-shift operation for dynamic range with a nonlinear piece-wise approximation of the original logarithmic Mu-Law. The Mu-Law compression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 12 complex output samples, each 14 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1723`:

```
armral_status armral_mu_law_compr_14bit(uint32_t n_prb,
                                    const armral_cmplx_int16_t *src,
                                    armral_compressed_data_14bit *dst,
                                    const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

> A read-only parameter of type `uint32_t`.
>
> The number of input resource blocks.

src

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Points to the input complex samples sequence.

dst

> A write-only parameter of type `armral_compressed_data_14bit *`.
>
> Points to the output 14-bit data and shift.

scale

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Phase compensation term to use, or NULL.

3.5.1.4 armral_mu_law_decompr_8bit

The Mu-Law decompression method is a logical reverse function of the compression method. The Mu-Law decompression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 8-bit complex resource elements. Each block taken as input is expanded into 12 complex output samples, each 16 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values.

Syntax

Defined in `armral.h` on line `1744`:

```
armral_status armral_mu_law_decompr_8bit(uint32_t n_prb,
                                         const armral_compressed_data_8bit *src,
                                         armral_cmplx_int16_t *dst,
                                         const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

> A read-only parameter of type `uint32_t`.
>
> The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_8bit *`.

Points to the input 8-bit data and shift.

dst

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

Points to the output complex samples sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.1.5 armral_mu_law_decompr_9bit

The Mu-Law decompression method is a logical reverse function of the compression method. The Mu-Law decompression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 9-bit complex resource elements. Each block taken as input is expanded into 12 complex output samples, each 16 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values.

Syntax

Defined in `armral.h` on line `1765`:

```
armral_status armral_mu_law_decompr_9bit(uint32_t n_prb,
                                         const armral_compressed_data_9bit *src,
                                         armral_cmplx_int16_t *dst,
                                         const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_9bit *`.

Points to the input 9-bit data and shift.

dst

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

Points to the output complex samples sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.1.6 armral_mu_law_decompr_14bit

The Mu-Law decompression method is a logical reverse function of the compression method. The Mu-Law decompression operates on `n_prb` Resource Blocks (RB) of fixed size. Each block consists of 12 14-bit complex resource elements. Each block taken as input is expanded into 12 complex output samples, each 16 bits wide, and the shift applied to the block.

A phase-compensation factor, stored in `*scale`, is used to scale values.

Syntax

Defined in `armral.h` on line `1786`:

```
armral_status armral_mu_law_decompr_14bit(
    uint32_t n_prb, const armral_compressed_data_14bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_14bit *`.

Points to the input 14-bit data and shift.

dst

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

Points to the output complex samples sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.2 Block Scaling Compression

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

3.5.2.1 armral_block_scaling_compr_8bit

The algorithm operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 8-bit post-scaled samples and a common unsigned scaling factor.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1818`:

```
armral_status
armral_block_scaling_compr_8bit(uint32_t n_prb, const armral_cmplx_int16_t *src,
                                armral_compressed_data_8bit *dst,
                                const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

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

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_8bit *`.

Points to the output 8-bit data and a scaling factor.

scale

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

Phase compensation term to use, or NULL.

3.5.2.2 armral_block_scaling_compr_9bit

The algorithm operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 9-bit post-scaled samples and a common unsigned scaling factor.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1839`:

```
armral_status
armral_block_scaling_compr_9bit(uint32_t n_prb, const armral_cmplx_int16_t *src,
                                armral_compressed_data_9bit *dst,
                                const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

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

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_9bit *`.

Points to the output 9-bit data and a scaling factor.

scale

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

Phase compensation term to use, or NULL.

3.5.2.3 armral_block_scaling_compr_14bit

The algorithm operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 14-bit post-scaled samples and a common unsigned scaling factor.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1859`:

```
armral_status armral_block_scaling_compr_14bit(
    uint32_t n_prb, const armral_cmplx_int16_t *src,
    armral_compressed_data_14bit *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

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

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_14bit *`.

Points to the output 14-bit data and a scaling factor.

scale

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

Phase compensation term to use, or NULL.

3.5.2.4 armral_block_scaling_decompr_8bit

The algorithm operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 8-bit complex post-scaled resource elements and an unsigned scaling factor. Each block taken as input is expanded into 12 16-bit complex samples.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1878`:

```
armral_status armral_block_scaling_decompr_8bit(
    uint32_t n_prb, const armral_compressed_data_8bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_8bit *`.

Points to the input 8-bit data and scaling factor.

dst

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

Points to the output complex samples sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.2.5 armral_block_scaling_decompr_9bit

The algorithm operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 9-bit complex post-scaled resource elements and an unsigned scaling factor. Each block taken as input is expanded into 12 16-bit complex samples.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1897`:

```
armral_status armral_block_scaling_decompr_9bit(
    uint32_t n_prb, const armral_compressed_data_9bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_9bit *`.

Points to the input 9-bit data and a scaling factor.

dst

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

Points to the output complex samples sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.2.6 armral_block_scaling_decompr_14bit

The algorithm operates on a fixed block size of one Physical Resource Block (PRB). Each block consists of 12 14-bit complex post-scaled resource elements and an unsigned scaling factor. Each block taken as input is expanded into 12 16-bit complex samples.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1917`:

```
armral_status armral_block_scaling_decompr_14bit(
    uint32_t n_prb, const armral_compressed_data_14bit *src,
    armral_cmplx_int16_t *dst, const armral_cmplx_int16_t *scale);
```

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_14bit *`.

Points to the input 14-bit data and a scaling factor.

dst

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

Points to the output complex samples sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.3 Block Floating Point

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

3.5.3.1 armral_block_float_compr_8bit

Block floating-point compression to 8-bit.

The 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.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1948`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

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

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_8bit *`.

Points to the output 8-bit data and exponent.

scale

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

Phase compensation term to use, or NULL.

3.5.3.2 armral_block_float_compr_9bit

Block floating point compression to 9-bit big-endian.

The 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.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `1973`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

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

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_9bit *`.

Points to the output 9-bit data and exponent.

scale

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

Phase compensation term to use, or NULL.

3.5.3.3 armral_block_float_compr_12bit

Block floating point compression to 12-bit big-endian.

The 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.

Syntax

Defined in `armral.h` on line `1995`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

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

Points to the input complex samples sequence.

dst

A write-only parameter of type `armral_compressed_data_12bit *`.

Points to the output 12-bit data and exponent.

scale

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

Phase compensation term to use, or NULL.

3.5.3.4 armral_block_float_compr_14bit

Block floating point compression to 14-bit big-endian.

The 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.

A phase-compensation factor, stored in `*scale`, is used to scale values before compression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2020`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

> A read-only parameter of type `uint32_t`.
>
> The number of input resource blocks.

src

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Points to the input complex samples sequence.

dst

> A write-only parameter of type `armral_compressed_data_14bit *`.
>
> Points to the output 14-bit data and exponent.

scale

> A read-only parameter of type `const armral_cmplx_int16_t *`.
>
> Phase compensation term to use, or NULL.

3.5.3.5 armral_block_float_decompr_8bit

Block floating-point decompression from 8 bit.

The 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.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2042`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_8bit *`.

Points to the input compressed block sequence.

dst

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

Points to the complex output sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.3.6 armral_block_float_decompr_9bit

Block floating point decompression from 9 bit big-endian.

The 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.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2066`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_9bit *`.

Points to the input compressed block sequence.

dst

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

Points to the complex output sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.3.7 armral_block_float_decompr_12bit

Block floating point decompression from 12 bit big-endian.

The 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.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2090`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_12bit *`.

Points to the input compressed block sequence.

dst

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

Points to the complex output sequence.

scale

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

Phase compensation term to use, or NULL.

3.5.3.8 armral_block_float_decompr_14bit

Block floating point decompression from 14 bit big-endian.

The 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.

A phase-compensation factor, stored in `*scale`, is used to scale values after decompression in the case that `scale` is non-NULL.

Syntax

Defined in `armral.h` on line `2114`:

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

Returns

An `armral_status` value that indicates success or failure.

Parameters

n_prb

A read-only parameter of type `uint32_t`.

The number of input resource blocks.

src

A read-only parameter of type `const armral_compressed_data_14bit *`.

Points to the input compressed block sequence.

dst

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

Points to the complex output sequence.

scale

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

Phase compensation term to use, or NULL.

4 Data Structures

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

4.1 armral_cmplx_f32_t

32-bit floating-point complex data type.

Syntax

Defined in `armral.h` on line 182:

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

4.2 armral_cmplx_int16_t

16-bit signed integer complex data type.

Syntax

Defined in `armral.h` on line 174:

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

4.3 armral_compressed_data_12bit

The structure for a 12-bit compressed block.

See armral_block_float_compr_12bit and armral_block_float_decompr_12bit.

Syntax

Defined in `armral.h` on line 220:

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

4.4 armral_compressed_data_14bit

The structure for a 14-bit compressed block.

See armral_block_float_compr_14bit and armral_block_float_decompr_14bit.

Syntax

Defined in `armral.h` on line 231:

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

4.5 armral_compressed_data_8bit

The structure for an 8-bit compressed block.

See armral_block_float_compr_8bit and armral_block_float_decompr_8bit.

Syntax

Defined in `armral.h` on line 198:

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

4.6 armral_compressed_data_9bit

The structure for a 9-bit compressed block.

See armral_block_float_compr_9bit and armral_block_float_decompr_9bit.

Syntax

Defined in `armral.h` on line 209:

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

4.7 armral_ldpc_base_graph_t

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

Syntax

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

```
typedef struct {
  ///
  /// The number of rows in the base graph.
  uint32_t nrows;
  /// The number of columns in the base graph which are associated with message
  /// bits. Punctured columns are included.
  uint32_t nmessage_bits;
  /// The number of block columns that are in the codeword. `ncodeword_bits` is
  /// the number of columns in the base graph minus the two punctured columns.
  uint32_t ncodeword_bits;
  /// The indices of the start of a row in the base graph, which you can use to
  /// index into the `col_inds` array to get the column indices of the non-zero
  /// entries in a row of the base graph.
  const uint32_t *row_start_inds;
  /// The indices of the non-zero columns in the base graph. Each of the entries
  /// in a row are stored contiguously. The start of a row is identified by
  /// indices stored in the `row_start_inds` array. For example, the start of
  /// row with index (zero-based) `2` is at index `row_start_inds[2]`.
  const uint32_t *col_inds;
  /// The shifts applied to the identity matrix to give the matrix at each
  /// non-zero column in the base graph. The shifts for all lifting sets are
  /// stored in this array. All shifts for one lifting set are stored before the
  /// next lifting set. This means that the shifts for lifting set with index
  /// (zero-based) `3`, and row with index `5` is at index
  /// `(row_start_inds[5] + 3) * 8`, where `8` is the number of lifting
  /// sets.
  const uint32_t *shifts;
} armral_ldpc_base_graph_t;
```

5 Macros

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

5.1 ARMRAL_NUM_COMPLEX_SAMPLES

The number of complex samples in each compressed block.

Syntax

Defined in `armral.h` on line `190`:

```
#define ARMRAL_NUM_COMPLEX_SAMPLES 12
```

6 Enumerations

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

6.1 armral_status

Error status returned by functions in the library.

Syntax

Defined in `armral.h` on line `100`:

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

6.2 armral_modulation_type

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

Syntax

Defined in `armral.h` on line `109`:

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

6.3 armral_fixed_point_index

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

Syntax

Defined in `armral.h` on line `120`:

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

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

6.4 armral_polar_frozen_bit_type

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

Syntax

Defined in `armral.h` on line `165`:

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

6.5 armral_fft_direction_t

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

Syntax

Defined in `armral.h` on line `2493`:

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

6.6 armral_ldpc_graph_t

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

Syntax

Defined in `armral.h` on line `2644`:

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

7 Type Aliases

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

7.1 armral_fft_plan_t

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

Syntax

Defined in `armral.h` on line `2486`:

```
typedef struct armral_fft_plan_t armral_fft_plan_t;
```