Arm RAN Acceleration Library Reference Guide

Version 20.10

Document Number 102249_2010_00

Chapter 1

Arm RAN Acceleration Library (ArmRAL) Reference Manual

1.1 About this book

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

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

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

1.2 Feedback

1.2.1 Feedback on this product

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

The product name.

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

1.2.2 Feedback on content

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

- The title Arm RAN Acceleration Library Reference Manual.
- The number 102249_2010_00_en.
- If applicable, the relevant page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1.3 Non-Confidential Proprietary Notice

1.4 Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

1.5 Product Status

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

1.6 Web Address

http://www.arm.com

1.7 Release Information

1.7.1 Document History

Issue	Date	Confidentiality	Change
2010-00	02 Oct 2020	Non-Confidential	New document for Arm RAN Acceleration Library v20.10

Chapter 2

Get started with Arm RAN Acceleration Library

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

2.1 Prerequisites

The library runs on most AArch64 cores, however some parts of the library have more strict requirements:

- To use the Cyclyic Redundancy Check (CRC) functions, you must run on a core that supports the A↩ Arch64 PMULL extension. If your machine supports the PMULL extension, pmull is listed under the "Features" list given in the `/proc/cpuinfo` file.

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

- A recent version of a C/C++ compiler, such as GCC. The library has been tested with GCC 7.1.0, 8.2.0, 9.3.0, and 10.2.0.
- A recent version of CMake (version 3.0.0, or higher).
- To build a local HTML version of the documentation, you must also have Doxygen installed.

2.2 Build Arm RAN Acceleration Library

After you download the library source, you must build the library before you can use it in your application development.

To build the library, use the following commands:

```
mkdir <build>
cd <build>
cmake [options] <path>
make
```

Substitute <build> with a directory name to build the library in, <path> with the path to the root directory of the library source, and [options] with the CMake options to build the library using.

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 or not to build the correctness tests and benchmarking code for the library, this 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 with -DBUILD_TESTING=On.

  Default is Off.

- -DARMRAL_ENABLE_WERROR=On|Off

  Enable -Werror when building the library and tests, which 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 additional 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.

2.3 Install Arm RAN Acceleration Library

If you have not already built the library, or if you want to rebuild the library to specify a custom install location, navigate to the unpacked product directory and run CMake:

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


Substitute:

- <build> with a directory name to build the library in.
- [options] with the CMake options to build the library using.
- (Optional) <install-dir> with a directory name to install the library in.
- <path> with the path to the root directory of the library source.

To install the library, run:

```
make install
```

For a default installation (without specifying -DCMAKE_INSTALL_PREFIX), the library is installed to /usr/local/lib/, and the headers to /usr/local/include/. For a custom installation make install installs the library to <install-dir>/lib/ and the headers to <install-dir>/include/.

An install creates an install_manifest.txt file in the library build directory. install_manifest.txt lists the locations that the library and header files are installed to.

2.4 Uninstall Arm RAN Acceleration Library

To uninstall the library, navigate to the library build directory and run:

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

2.5 Run the library tests

To run the included library tests, you must build the library with -DBUILD_TESTING=On. Once built, to run the tests, use:

```
make check
```

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

2.6 Run the benchmarks

All the functions in the library contain benchmarking code that contains preset problem sizes. To run the benchmark tests, you must have built your library with the -DBUILD_TESTING=On CMake option. Once built, to run the benchmarks, use:

```
make bench
```

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

2.7 Documentation

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

```
make docs
```

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

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Module Documentation

6.1 Vector functions

Modules

- Vector Multiply
- Vector Dot Product

6.1.1 Detailed Description

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

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

6.2 Matrix functions

Modules

- Complex Matrix Multiplication
- Complex Matrix Inversion

6.2.1 Detailed Description

Functions are provided for working with matrices, including:

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

6.3 Lower PHY support functions

Modules

- Sequence Generator
- Correlation Coefficient
- FIR filter
- Fast Fourier Transforms (FFT)

6.3.1 Detailed Description

The Lower PHY functions include support for:

- A Gold sequence generator
- A correlation coefficient of a pair of 16-bit integer arrays (in Q15 format).
- FIR filters. Supports both 16-bit integer and 32-bit floating-point datatypes. Support is provided for decimation factors of both one and two.
- Fast Fourier Transforms (FFTs). Supports both 16-bit integer and 32-bit floating-point datatypes.

6.4 Upper PHY support functions

Modules

- Modulation
- CRC24
- Polar Encoding

6.4.1 Detailed Description

The Lower PHY functions include support for:

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

6.5 Du-Ru IF support functions

Modules

- Mu-Law Companding
- Block Floating Point

6.5.1 Detailed Description

The Du-Ru IF functions include support for:

- Mu-law companding (both compression and expansion).
- Block floating-point compression and decompression, in both 8-bit and 12-bit formats.

6.6 Vector Multiply

Functions

- armral_status armral_cmplx_vecmul_i16 (int32_t n, const armral_cmplx_int16_t *a, const armral_cmplx_int16_t *b, armral_cmplx_int16_t *c)
- armral_status armral_cmplx_vecmul_i16_2 (int32_t n, const int16_t *a_re, const int16_t *a_im, const int16_t *b_re, const int16_t *b_im, int16_t *c_re, int16_t *c_im)
- armral_status armral_cmplx_vecmul_f32 (int32_t n, const armral_cmplx_f32_t *a, const armral_cmplx_f32_t *b, armral_cmplx_f32_t *c)
- armral_status armral_cmplx_vecmul_f32_2 (int32_t n, const float *a_re, const float *a_im, const float *b_re, const float *b_im, float *c_re, float *c_im)

6.6.1 Detailed Description

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 following algorithm is used:

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

6.6.2 Function Documentation

6.6.2.1 armral_cmplx_vecmul_f32()

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

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

```
C[n] = A[n] * B[n],    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:

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

Parameters

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

Returns

armral_status

6.6.2.2 armral_cmplx_vecmul_f32_2()

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

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

```
C[n] = A[n] * B[n],   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:

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

Parameters

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

Returns

armral_status

6.6.2.3 armral_cmplx_vecmul_i16()

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

This algorithm performs the element-wise complex multiplication between two complex input sequences, A and B, of the same length, (N).
The implementation uses saturating arithmetic, intermediate operations are computed 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],   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:

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

Parameters

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

Returns

armral_status

6.6.2.4 armral_cmplx_vecmul_i16_2()

```
armral_status armral_cmplx_vecmul_i16_2 (
            int32_t n,
```



```
            const int16_t * a_re,
            const int16_t * a_im,
            const int16_t * b_re,
            const int16_t * b_im,
            int16_t * c_re,
            int16_t * c_im )
```

This algorithm performs the element-wise complex multiplication between two complex [I and Q separated] input sequences, `A` and `B`, of the same length (`N`).
The implementation uses saturating arithmetic, intermediate operations are computed on 32-bit variables in Q31 format. The final result is right-shifted and narrowed to 16 bits, to convert it back into Q15 format.

```
C[n] = A[n] * B[n],   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:

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

Parameters

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

Returns

armral_status

6.7 Vector Dot Product

Functions

- armral_status armral_cmplx_vecdot_f32 (int32_t n, const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_b, armral_cmplx_f32_t *p_src_c)
- armral_status armral_cmplx_vecdot_f32_2 (int32_t n, const float *p_src_a_re, const float *p_src_a_im, const float *p_src_b_re, const float *p_src_b_im, float *p_src_c_re, float *p_src_c_im)
- armral_status armral_cmplx_vecdot_i16 (int32_t n, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *p_src_c)
- armral_status armral_cmplx_vecdot_i16_2 (int32_t n, const int16_t *p_src_a_re, const int16_t *p_↩ src_a_im, const int16_t *p_src_b_re, const int16_t *p_src_b_im, int16_t *p_src_c_re, int16_t *p_src_↩ c_im)
- armral_status armral_cmplx_vecdot_i16_32bit (int32_t n, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *p_src_c)
- armral_status armral_cmplx_vecdot_i16_2_32bit (int32_t n, const int16_t *p_src_a_re, const int16_t *p_src_a_im, const int16_t *p_src_b_re, const int16_t *p_src_b_im, int16_t *p_src_c_re, int16_t *p_↩ src_c_im)

6.7.1 Detailed Description

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 interleaved real and imaginary components:

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

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

The following algorithm is used:

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

6.7.2 Function Documentation

6.7.2.1 armral_cmplx_vecdot_f32()

```
armral_status armral_cmplx_vecdot_f32 (
            int32_t n,
            const armral_cmplx_f32_t * p_src_a,
            const armral_cmplx_f32_t * p_src_b,
            armral_cmplx_f32_t * p_src_c )
```

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

Parameters

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

Returns

armral_status

6.7.2.2 armral_cmplx_vecdot_f32_2()

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

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

Parameters

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

Returns

armral_status

6.7.2.3 armral_cmplx_vecdot_i16()

```
armral_status armral_cmplx_vecdot_i16 (
            int32_t n,
            const armral_cmplx_int16_t * p_src_a,
            const armral_cmplx_int16_t * p_src_b,
            armral_cmplx_int16_t * p_src_c )
```


This algorithm computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed. Array elements are assumed to be complex int16 in Q15 format and interleaved.
To avoid overflow issues input values are internally extended to 32-bit variables and all intermediate calculations results are saved in 64-bit internal variables. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Parameters

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

Returns

armral_status

6.7.2.4 armral_cmplx_vecdot_i16_2()

```
armral_status armral_cmplx_vecdot_i16_2 (
            int32_t n,
            const int16_t * p_src_a_re,
            const int16_t * p_src_a_im,
            const int16_t * p_src_b_re,
            const int16_t * p_src_b_im,
            int16_t * p_src_c_re,
            int16_t * p_src_c_im )
```

This algorithm computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed. Array elements are assumed to be int16 in Q15 format and separate arrays are used for real parts and imaginary parts of the input data.
To avoid overflow issues input values are internally extended to 32-bit variables and all intermediate calculations results are saved in 64-bit internal variables. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Parameters

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

Returns

armral_status

6.7.2.5 armral_cmplx_vecdot_i16_2_32bit()

```
armral_status armral_cmplx_vecdot_i16_2_32bit (
            int32_t n,
            const int16_t * p_src_a_re,
            const int16_t * p_src_a_im,
            const int16_t * p_src_b_re,
            const int16_t * p_src_b_im,
            int16_t * p_src_c_re,
            int16_t * p_src_c_im )
```

This algorithm computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed.
Array elements are assumed to be int16 in Q15 format and separate arrays are used for both the real parts and imaginary parts of the input data.
All intermediate calculation results are saved in 32-bit internal variables, saturating the value to prevent overflow. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Parameters

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

Returns

armral_status

6.7.2.6 armral_cmplx_vecdot_i16_32bit()

```
armral_status armral_cmplx_vecdot_i16_32bit (
            int32_t n,
            const armral_cmplx_int16_t * p_src_a,
            const armral_cmplx_int16_t * p_src_b,
            armral_cmplx_int16_t * p_src_c )
```

This algorithm computes the dot product between a pair of arrays of complex values. The arrays are multiplied element-by-element and then summed. Array elements are assumed to be complex int16 in Q15 format and interleaved.
All intermediate calculations results are saved in 32-bit internal variables, saturating the value to prevent overflow. To get the final result in Q15 and to avoid overflow, the accumulator narrows to 16 bits with saturation.

Parameters

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

Returns

armral_status

6.8 Complex Matrix Multiplication

Functions

- armral_status armral_cmplx_mat_mult_i16 (uint16_t m, uint16_t n, uint16_t k, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *p_dst)
- armral_status armral_cmplx_mat_mult_i16_32bit (uint16_t m, uint16_t n, uint16_t k, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *p_dst)
- armral_status armral_cmplx_mat_mult_f32 (uint16_t m, uint16_t n, uint16_t k, const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_b, armral_cmplx_f32_t *p_dst)
- armral_status armral_cmplx_mat_mult_2x2_f32 (const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_b, armral_cmplx_f32_t *p_dst)
- armral_status armral_cmplx_mat_mult_2x2_f32_iq (const float32_t *src_a_re, const float32_t *src_a_im, const float32_t *src_b_re, const float32_t *src_b_im, float32_t *dst_re, float32_t *dst_im)
- armral_status armral_cmplx_mat_mult_4x4_f32 (const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_b, armral_cmplx_f32_t *p_dst)
- armral_status armral_cmplx_mat_mult_4x4_f32_iq (const float32_t *src_a_re, const float32_t *src_a_im, const float32_t *src_b_re, const float32_t *src_b_im, float32_t *dst_re, float32_t *dst_im)
- armral_status armral_solve_2x2_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, uint32_t p_xstride, armral_fixed_point_index num_fract_bits_x)
- armral_status armral_solve_2x4_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, uint32_t p_xstride, armral_fixed_point_index num_fract_bits_x)
- armral_status armral_solve_4x4_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, uint32_t p_xstride, armral_fixed_point_index num_fract_bits_x)
- armral_status armral_solve_1x4_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, armral_fixed_point_index num_fract_bits_x)
- armral_status armral_solve_1x2_f32 (uint32_t num_sub_carrier, uint32_t type, 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)

6.8.1 Detailed Description

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.

6.8.2 Function Documentation

6.8.2.1 armral_cmplx_mat_mult_2x2_f32()

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

This algorithm performs an optimized product of two square 2x2 matrices. This algorithm assumes that matrix A (first matrix) is transposed before entering the `armral_cmplx_mat_mult_2x2_f32` function and is then processed as a column major matrix.
Matrix B (second matrix) is considered to be row major. In LTE/5G, you can use the `cmplx_mat_mult_2x2_f32` function in the equalization step in the formula:

```
x_hat = G * y
```

Equalization matrix `G` corresponds to the first input matrix (matrix A) of the function.
It is assumed that matrix `G` is transposed during its computation so that it is presented column-major on input.
The second input matrix (matrix B) is formed by four 4x1 vectors (`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 four distinct `x_hat` estimates.

Parameters

in	p_src_a	points to the first input complex matrix structure
in	p_src_b	points to the second input complex matrix structure
out	p_dst	points to the output complex matrix structure

Returns

armral_status

6.8.2.2 armral_cmplx_mat_mult_2x2_f32_iq()

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

This algorithm performs an optimized product of two square 2x2 matrices whose complex elements have already been separated into real component and imaginary component arrays.
This algorithm assumes that matrix A (first matrix) has been transposed before entering the `armral_cmplx_mat_mult_2x2_f32_iq` function and is then processed as a column major matrix.
Matrix B (second matrix) is considered to be row major. In LTE/5G, you can use the `armral_cmplx_mat_mult_2x2_f32_iq` function in the equalization step in the formula:

```
x_hat = G * y
```


Equalization matrix G corresponds to the first input matrix (matrix A) of the function. It is assumed that matrix G is transposed during its computation so that it is presented column-major on input.
The second input matrix (matrix B) is formed by two 2x1 vectors (y vectors in the preceding formula) so that each row of B represents a 2x1 vector output from each antenna port, and each call to armral_cmplx_mat_mult_2x2_f32_iq computes two distinct x_hat estimates.

Parameters

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

Returns

armral_status

6.8.2.3 armral_cmplx_mat_mult_4x4_f32()

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

This algorithm performs an optimized product of two square 4x4 matrices. This algorithm assumes that matrix A (first matrix) has been transposed before entering the armral_cmplx_mat_mult_4x4_f32 function and is then processed as a column-wise matrix.
Matrix B (second matrix) is considered to be row major. In LTE/5G, you can use the cmplx_mat_mult_4x4_f32 function in the equalization step in the formula:

```
x_hat = G * y
```

Equalization matrix G corresponds to the first input matrix (matrix A) of the function.
It is assumed that matrix G is transposed during its computation so that it is presented column-major on input.
The second input matrix (matrix B) is formed by four 4x1 vectors (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_hat estimates.

Parameters

in	p_src_a	points to the first input complex matrix structure
in	p_src_b	points to the second input complex matrix structure
out	p_dst	points to the output complex matrix structure

Returns

armral_status

6.8.2.4 armral_cmplx_mat_mult_4x4_f32_iq()

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

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

```
x_hat = G * y
```

Equalization matrix `G` corresponds to the first input matrix (matrix A) of the function. It is assumed that matrix `G` is transposed during its computation so that it is presented column-major on input.
The second input matrix (matrix B) is formed by four 4x1 vectors (`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_hat` estimates.

Parameters

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

Returns

armral_status

6.8.2.5 armral_cmplx_mat_mult_f32()

```
armral_status armral_cmplx_mat_mult_f32 (
            uint16_t m,
            uint16_t n,
            uint16_t k,
            const armral_cmplx_f32_t * p_src_a,
            const armral_cmplx_f32_t * p_src_b,
            armral_cmplx_f32_t * p_dst )
```


This algorithm performs the multiplication `A x B` for square matrices of float values, and assumes that matrices are saved in memory row major.

Parameters

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

Returns

armral_status

6.8.2.6 armral_cmplx_mat_mult_i16()

```
armral_status armral_cmplx_mat_mult_i16 (
            uint16_t m,
            uint16_t n,
            uint16_t k,
            const armral_cmplx_int16_t * p_src_a,
            const armral_cmplx_int16_t * p_src_b,
            armral_cmplx_int16_t * p_dst )
```

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

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

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

Parameters

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

Returns

armral_status

6.8.2.7 armral_cmplx_mat_mult_i16_32bit()

```
armral_status armral_cmplx_mat_mult_i16_32bit (
            uint16_t m,
            uint16_t n,
            uint16_t k,
            const armral_cmplx_int16_t * p_src_a,
            const armral_cmplx_int16_t * p_src_b,
            armral_cmplx_int16_t * p_dst )
```

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

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

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

Parameters

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

Returns

armral_status

6.8.2.8 armral_solve_1x2_f32()

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

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

```
x_hat = G * y
```


where y is a vector for the received signal, size corresponds to the number of antennae and x_hat 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 four.
When the type parameter is 1, the same G matrix is used for the equalization of four consecutive antenna inputs.
When the type parameter is 2, the same G matrix is used for the equalization of six consecutive antenna inputs.

Parameters

in	num_sub_carrier	number of sub-carrier to equalize
in	type	type 1 or type 2
in	p_y	points to the input received signal
in	p_ystride	stride between two Rx antennae
in	p_y_num_fract_bits	number of fractional bits in y conversion
in	p_g_real	points to the real part of coefficient matrix G
in	p_g_imag	points to the imag part coefficient matrix G
in	p_gstride	stride between elements of G
out	p_x	points to the output received signal
in	num_fract_bits_x	number of fractional bits in x

Returns

armral_status

6.8.2.9 armral_solve_1x4_f32()

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

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

```
x_hat = G * y
```


where y is a vector for the received signal, size corresponds to the number of antennae and x_hat 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 four.
When the type parameter is 1, the same G matrix is used for the equalization of four consecutive antenna inputs.
When the type parameter is 2, the same G matrix is used for the equalization of six consecutive antenna inputs.

Parameters

in	num_sub_carrier	number of sub-carrier to equalize
in	type	type 1 or type 2
in	p_y	points to the input received signal
in	p_ystride	stride between two Rx antennae
in	p_y_num_fract_bits	number of fractional bits in y conversion
in	p_g_real	points to the real part of coefficient matrix G
in	p_g_imag	points to the imag part coefficient matrix G
in	p_gstride	stride between elements of G
out	p_x	points to the output received signal
in	num_fract_bits_x	number of fractional bits in x

Returns

armral_status

6.8.2.10 armral_solve_2x2_f32()

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

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

```
x_hat = G * y
```


where y is a vector for the received signal, size corresponds to the number of antennae and x_hat 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 four.
When the type parameter is 1, the same G matrix is used for the equalization of four consecutive antenna inputs.
When the type parameter is 2, the same G matrix is used for the equalization of six consecutive antenna inputs.

Parameters

in	num_sub_carrier	number of sub-carriers to equalize
in	type	type 1 or type 2
in	p_y	points to the input received signal
in	p_ystride	stride between two Rx antennae
in	p_y_num_fract_bits	number of fractional bits in y
in	p_g_real	points to the real part of coefficient matrix G
in	p_g_imag	points to the imag part coefficient matrix G
in	p_gstride	stride between elements of G
out	p_x	points to the output received signal
in	p_xstride	stride between two layers
in	num_fract_bits_x	number of fractional bits in x

Returns

armral_status

6.8.2.11 armral_solve_2x4_f32()

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

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

```
x_hat = G * y
```


where y is a vector for the received signal, size corresponds to the number of antennae and x_hat 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 four.
When the type parameter is 1, the same G matrix is used for the equalization of four consecutive antenna inputs.
When the type parameter is 2, the same G matrix is used for the equalization of six consecutive antenna inputs.

Parameters

in	num_sub_carrier	number of sub-carrier to equalize
in	type	type 1 or type 2
in	p_y	points to the input received signal
in	p_ystride	stride between two Rx antennae
in	p_y_num_fract_bits	number of fractional bits in y
in	p_g_real	points to the real part of coefficient matrix G
in	p_g_imag	points to the imag part coefficient matrix G
in	p_gstride	stride between elements of G
out	p_x	points to the output received signal
in	p_xstride	stride between two layers
in	num_frac_bits_x	number of fractional bits in x

Returns

armral_status

6.8.2.12 armral_solve_4x4_f32()

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

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

```
x_hat = G * y
```


where `y` is a vector for the received signal, size corresponds to the number of antennae and `x_hat` 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 four.
When the type parameter is type 1, the same `G` matrix is used for the equalization of four consecutive antenna inputs.
When the type parameter is 2, the same `G` matrix is used for the equalization of six consecutive antenna inputs.

Parameters

`in`	num_sub_carrier	number of sub-carrier to equalize
`in`	type	type 1 or type 2
`in`	p_y	points to the input received signal
`in`	p_ystride	stride between two Rx antennae
`in`	p_y_num_fract_bits	number of fractional bits in `y`
`in`	p_g_real	points to the real part of coefficient matrix G
`in`	p_g_imag	points to the imag part coefficient matrix G
`in`	p_gstride	stride between elements of G
`out`	p_x	points to the output received signal
`in`	p_xstride	stride between two layers
`in`	num_frac_bits_x	number of fractional bits in `x`

Returns

armral_status

6.9 Complex Matrix Inversion

Functions

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

6.9.1 Detailed Description

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

6.9.2 Function Documentation

6.9.2.1 armral_cmplx_hermitian_mat_inverse_f32()

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

This algorithm computes the inverse of a complex hermitian squared matrix of size NxN.
The supported dimensions are 2x2, 4x4, 8x8, and 16x16.
The input matrix, which is filled in row major order, is made of complex float32_t elements in an interleaved form:

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

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

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

Parameters

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

Returns

armral_status

6.10 Sequence Generator

Functions

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

6.10.1 Detailed Description

Fills a pointer with a gold sequence of the specified length, generated from the specified seed.

6.10.2 Function Documentation

6.10.2.1 armral_seq_generator()

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

This algorithm generates a pseudo-random sequence (Gold Sequence) that is used in 4G and 5G networks to scramble data of a specific channel or to generate a specific sequence (for example for Downlink Reference Signal generation).
The sequence generator is the the same generator that is described in 3GPP 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, i=1,2...30
cinit = Sum(x2(i)*2^n, n=0..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.

Parameters

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

Returns

armral_status

6.11 Modulation

Functions

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

6.11.1 Detailed Description

Performs modulation and demodulation of digital signals. Modulation takes a bitstream and outputs a series of Q2.13 fixed-point complex symbols, with demodulation doing the reverse. 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).

6.11.2 Function Documentation

6.11.2.1 armral_demodulation()

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

This algorithm implements the soft-demodulation (or soft bit demapping) for QPSK, 16QAM, 64QAM, and 256QAM constellations.
The input sequence is assumed to be made of complex symbols `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 confidence of the demapping decision, component by component, instead of taking a hard decision and giving the bit value itself.

The LLRs calculations are made approximately with thresholds method, to have similar performance of the raw calculation, but with a lower complexity.
All the constellations mapping follow the 3GPP TS 38.211 V15.2.0, Chapter 5.1 Modulation mapper.

Parameters

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

Returns

armral_status

6.11.2.2 armral_modulation()

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

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

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

Parameters

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

Returns

armral_status

6.12 Correlation Coefficient

Functions

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

6.12.1 Detailed Description

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

6.12.2 Function Documentation

6.12.2.1 armral_corr_coeff_i16()

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

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

Pearson's correlation coefficient is calculated using:

```
           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)))
```

Parameters

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

Returns

armral_status

6.13 FIR filter

Functions

- armral_status armral_fir_filter_cf32 (uint32_t size, uint32_t taps, const armral_cmplx_f32_t *input, const armral_cmplx_f32_t *coeffs, armral_cmplx_f32_t *output)
- armral_status armral_fir_filter_cf32_decimate_2 (uint32_t size, uint32_t taps, const armral_cmplx_f32_t *input, const armral_cmplx_f32_t *coeffs, armral_cmplx_f32_t *output)
- armral_status armral_fir_filter_cs16 (uint32_t size, uint32_t taps, const armral_cmplx_int16_t *input, const armral_cmplx_int16_t *coeffs, armral_cmplx_int16_t *output)
- armral_status armral_fir_filter_cs16_decimate_2 (uint32_t size, uint32_t taps, const armral_cmplx_int16_t *input, const armral_cmplx_int16_t *coeffs, armral_cmplx_int16_t *output)

6.13.1 Detailed Description

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:

$$y[n] = b_0 x[n] + b_1 x[n-1] + ... + b_N x[n-N] = \sum_{i=0}^{N} b_i x[n-i]$$

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

6.13.2 Function Documentation

6.13.2.1 armral_fir_filter_cf32()

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

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

The size of the input data must be a multiple of four, and both the input array and the coefficients array are padded with zeros.

Parameters

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

Returns

armral_status

6.13.2.2 armral_fir_filter_cf32_decimate_2()

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

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

The size of the input data must be a multiple of four, and both the input array and the coefficients array are padded with zeros.

Parameters

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

Returns

armral_status

6.13.2.3 armral_fir_filter_cs16()

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

Computes a complex signed 16-bit integer FIR filter.

Parameters

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

Returns

armral_status

6.13.2.4 armral_fir_filter_cs16_decimate_2()

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

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

Parameters

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

Returns

armral_status

6.14 Mu-Law Companding

Functions

- armral_status armral_mu_law_compression (const int16_t *p_rb_in, int8_t *p_out, int8_t *comp_shift)
- armral_status armral_mu_law_expansion (const int8_t *p_comp_rb_in, int8_t comp_shift, int16_t *p_expanded_out)

6.14.1 Detailed Description

The Mu-law companding algorithm enables the compression of UP data over the fronthaul interface.

6.14.2 Function Documentation

6.14.2.1 armral_mu_law_compression()

```
armral_status armral_mu_law_compression (
            const int16_t * p_rb_in,
            int8_t * p_out,
            int8_t * comp_shift )
```

The Mu-law compression method combines a bit shift operation for dynamic range with a nonlinear piece wise approximation of the original logarithmic Mu-law. The algorithm uses Mu=8 for implementation efficiency. The Mu-law compression works on 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 12 complex output samples, each 8 bits wide, and a shift value.

Parameters

in	p_rb_in	points to the input Resource Block, 12 complex resource elements, therefore 24 interleaved values.
out	p_out	points to the output compressed RB, 12 complex resource elements with fixed word length of 8-bit, including sign, exponent, and mantissa.
out	comp_shift	the shift applied to the entire PRB.

Returns

armral_status

6.14.2.2 armral_mu_law_expansion()

```
armral_status armral_mu_law_expansion (
            const int8_t * p_comp_rb_in,
```



```
int8_t comp_shift,
int16_t * p_expanded_out )
```

The Mu-law expansion method is a logical reverse function of the compression method. The MU-law expansion works on a fixed block size of one Resource Block (RB). Each block consists of 12 8-bit complex resource elements. Each block taken as input is expanded into 12 complex output samples, each 16 bits wide, and a shift value.

Parameters

in	p_comp_rb_in	points to the input compressed Resource Block (8 bits), 12 complex resource elements, therefore 24 interleaved values.
in	comp_shift	the shift applied to the entire PRB in compression.
out	p_expanded_out	points to the output expanded RB, 12 complex resource elements with fixed word length of 16 bits.

Returns

armral_status

6.15 Block Floating Point

Functions

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

  Block floating-point compression to 8-bit.
- armral_status armral_block_float_compr_9bit (uint32_t n_prb, const armral_cmplx_int16_t *src, armral_compressed_data_9bit *dst)

  Block floating point compression to 9-bit.
- armral_status armral_block_float_compr_12bit (uint32_t n_prb, const armral_cmplx_int16_t *src, armral_compressed_data_12bit *dst)

  Block floating point compression to 12-bit.
- armral_status armral_block_float_decompr_8bit (uint32_t n_prb, const armral_compressed_data_8bit *src, armral_cmplx_int16_t *dst)

  Block floating-point decompression from 8 bit.
- armral_status armral_block_float_decompr_9bit (uint32_t n_prb, const armral_compressed_data_9bit *src, armral_cmplx_int16_t *dst)

  Block floating point decompression from 9 bit.
- armral_status armral_block_float_decompr_12bit (uint32_t n_prb, const armral_compressed_data_12bit *src, armral_cmplx_int16_t *dst)

  Block floating point decompression from 12 bit.

6.15.1 Detailed Description

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

6.15.2 Function Documentation

6.15.2.1 armral_block_float_compr_12bit()

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

The algorithm operates on a fixed block size of one Resource Block (RB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 12-bit samples and one unsigned exponent.

Parameters

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

Returns

armral_status

6.15.2.2 armral_block_float_compr_8bit()

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

The algorithm operates on a fixed block size of one Resource Block (RB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 9-bit samples and one unsigned exponent.

Parameters

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

Returns

armral_status

6.15.2.3 armral_block_float_compr_9bit()

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

The algorithm operates on a fixed block size of one Resource Block (RB). Each block consists of 12 16-bit complex resource elements. Each block taken as input is compressed into 24 9-bit samples and one unsigned exponent.

Parameters

in	n_prb	number of input resource blocks
in	src	points to the input complex samples sequence
out	dst	point to the output 12 bit block

Returns

armral_status

6.15.2.4 armral_block_float_decompr_12bit()

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

The algorithm operates on a fixed block size of one Resource Block (RB). Each block consists of 12 12-bit complex resource elements and an unsigned exponent. Each block taken as input is expanded into 12 16-bit complex samples.

Parameters

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

Returns

armral_status

6.15.2.5 armral_block_float_decompr_8bit()

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

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

Parameters

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

Returns

armral_status

6.15.2.6 armral_block_float_decompr_9bit()

```
armral_status armral_block_float_decompr_9bit (
            uint32_t n_prb,
```



```
    const armral_compressed_data_9bit * src,
    armral_cmplx_int16_t * dst )
```

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

Parameters

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

Returns

armral_status

6.16 CRC24

Functions

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

6.16.1 Detailed Description

Computes a 24-bit 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 + x^0
CRC24B polynomial = x^24 + x^23 + x^6 + x^5 + x^1 + x^0
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 + x^0
```

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

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

6.16.2 Function Documentation

6.16.2.1 armral_crc24_a_be()

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

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

Parameters

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

Returns

armral_status

6.16.2.2 armral_crc24_a_le()

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

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

Parameters

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

Returns

armral_status

6.16.2.3 armral_crc24_b_be()

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

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

Parameters

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

Returns

armral_status

6.16.2.4 armral_crc24_b_le()

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

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

Parameters

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

Returns

armral_status

6.16.2.5 armral_crc24_c_be()

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

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

Parameters

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

Returns

armral_status

6.16.2.6 armral_crc24_c_le()

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


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

Parameters

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

Returns

armral_status

6.17 Polar Encoding

Functions

- armral_status armral_polar_encoder (uint16_t n, const uint32_t *p_u_seq_in, uint32_t *p_d_seq_out)
- armral_status armral_polar_decoder (uint16_t n, uint16_t k, const int8_t *p_llr_in, uint32_t *p_u_↩ seq_out)

6.17.1 Detailed Description

Polar codes are used to encode the Uplink Control Information (UCI) over the PUCCH and PUSCH, and the Downlink Control Information (DCI) over the PDCCH, in downlink. 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 3GPP TS 38.212 V16.0.0 (2019-12).

6.17.2 Function Documentation

6.17.2.1 armral_polar_decoder()

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

Decodes `k` real information bits from a Polar-encoded message of length `n`, given as input as a sequence of 8-bit log-likelihood ratios.

The decoder is implemented using the Successive Cancellation (SC) method.

Parameters

in	n	polar code length in bits, must be a power of 2
in	k	number of real information bits (message + CRC) within the codeword of length N, K must be < N
in	p_llr_in	points to the input sequence of LLR bytes.
out	p_u_seq_out	points to the output decoded sequence `[u]` of bits `[u(0), u(1), ..., u(N-1)]`.

Returns

armral_status

6.17.2.2 armral_polar_encoder()

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

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

Parameters

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

Returns

armral_status

6.18 Fast Fourier Transforms (FFT)

Typedefs

- typedef struct armral_fft_plan_t armral_fft_plan_t

Enumerations

- enum armral_fft_direction_t { ARMRAL_FFT_FORWARDS = -1, ARMRAL_FFT_BACKWARDS = 1 }

Functions

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

  Creates a plan to solve a complex fp32 FFT.
- armral_status armral_fft_execute_cf32 (const armral_fft_plan_t *p, const armral_cmplx_f32_t *x, armral_cmplx_f32_t *y)

  Performs a single FFT using the specified plan and arrays.
- armral_status armral_fft_destroy_plan_cf32 (armral_fft_plan_t **p)

  Destroys an FFT plan.
- armral_status armral_fft_create_plan_cs16 (armral_fft_plan_t **p, int n, armral_fft_direction_t dir)

  Creates a plan to solve a complex int16 (Q0.15 format) FFT.
- armral_status armral_fft_execute_cs16 (const armral_fft_plan_t *p, const armral_cmplx_int16_t *x, armral_cmplx_int16_t *y)

  Performs a single FFT using the specified plan and arrays.
- armral_status armral_fft_destroy_plan_cs16 (armral_fft_plan_t **p)

  Destroys an FFT plan.

6.18.1 Detailed Description

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);
```

6.18.2 Typedef Documentation

6.18.2.1 armral_fft_plan_t

```
typedef struct armral_fft_plan_t armral_fft_plan_t
```

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

6.18.3 Enumeration Type Documentation

6.18.3.1 armral_fft_direction_t

```
enum armral_fft_direction_t
```

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

Enumerator

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

6.18.4 Function Documentation

6.18.4.1 armral_fft_create_plan_cf32()

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

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

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

Parameters

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

Returns

armral_status

6.18.4.2 armral_fft_create_plan_cs16()

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

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

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

Parameters

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

Returns

armral_status

6.18.4.3 armral_fft_destroy_plan_cf32()

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

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

Parameters

`in,out`	p	A pointer to the FFT plan pointer. The pointer must be the value that is returned by an earlier call to `armral_fft_create_↩ plan_cf32`. On function exit, the value that is pointed to is set to `NULL`.

Returns

armral_status

6.18.4.4 armral_fft_destroy_plan_cs16()

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

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

Parameters

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

Returns

armral_status

6.18.4.5 armral_fft_execute_cf32()

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

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

Parameters

`in`	p	A pointer to the FFT plan. The pointer is the value that is filled in by an earlier call to `armral_fft_create_plan_cf32`.
`in`	x	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`.
`out`	y	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`.

Returns

armral_status

6.18.4.6 armral_fft_execute_cs16()

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

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

Parameters

in	p	A pointer to the FFT plan. The pointer is the value that is filled in by an earlier call to `armral_fft_create_plan_cs16`.
in	x	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`.
out	y	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`.

Returns

armral_status

Chapter 7

Data Structure Documentation

7.1 armral_cmplx_f32_t Struct Reference

`#include <armral.h>`

Data Fields

- float re
  32-bit real component
- float im
  32-bit imaginary component

7.1.1 Detailed Description

32-bit floating-point complex data type.

7.1.2 Field Documentation

7.1.2.1 im

`float armral_cmplx_f32_t::im`

7.1.2.2 re

`float armral_cmplx_f32_t::re`

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

- armral.h

7.2 armral_cmplx_int16_t Struct Reference

`#include <armral.h>`

Data Fields

- int16_t re
  16-bit real component
- int16_t im
  16-bit imaginary component

7.2.1 Detailed Description

16-bit signed int complex data type.

7.2.2 Field Documentation

7.2.2.1 im

`int16_t armral_cmplx_int16_t::im`

7.2.2.2 re

`int16_t armral_cmplx_int16_t::re`

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

- armral.h

7.3 armral_compressed_data_12bit Struct Reference

`#include <armral.h>`

Data Fields

- int8_t mantissa [36]
  Packed data, 12 bits per element.
- int8_t exp
  Block exponent, in the range 0-8 (inclusive)

7.3.1 Detailed Description

The structure for a 12-bit compressed block.
See armral_block_float_compr_12bit and armral_block_float_decompr_12bit.

7.3.2 Field Documentation

7.3.2.1 exp

```
int8_t armral_compressed_data_12bit::exp
```

7.3.2.2 mantissa

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

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

- armral.h

7.4 armral_compressed_data_8bit Struct Reference

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

Data Fields

- int8_t mantissa [24]
  Packed data, 8 bits per element.
- int8_t exp
  Block exponent, in the range 0-4 (inclusive)

7.4.1 Detailed Description

The structure for an 8-bit compressed block.
See armral_block_float_compr_8bit and armral_block_float_decompr_8bit.

7.4.2 Field Documentation

7.4.2.1 exp

```
int8_t armral_compressed_data_8bit::exp
```

7.4.2.2 mantissa

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

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

- armral.h

7.5 armral_compressed_data_9bit Struct Reference

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

Data Fields

- int8_t mantissa [27]
  Packed data, 9 bits per element.
- int8_t exp
  Block exponent, in the range 0-5 (inclusive)

7.5.1 Detailed Description

The structure for a 9-bit compressed block.
See armral_block_float_compr_9bit and armral_block_float_decompr_9bit.

7.5.2 Field Documentation

7.5.2.1 exp

```
int8_t armral_compressed_data_9bit::exp
```

7.5.2.2 mantissa

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

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

- armral.h

Chapter 8

File Documentation

8.1 armral.h File Reference

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

Data Structures

- struct armral_cmplx_int16_t
- struct armral_cmplx_f32_t
- struct armral_compressed_data_8bit
- struct armral_compressed_data_9bit
- struct armral_compressed_data_12bit

Macros

- #define ARMRAL_NUM_COMPLEX_SAMPLES 12

Typedefs

- typedef struct armral_fft_plan_t armral_fft_plan_t

Enumerations

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

Functions

- armral_status armral_cmplx_vecmul_i16 (int32_t n, const armral_cmplx_int16_t *a, const armral_cmplx_int16_t *b, armral_cmplx_int16_t *c)
- armral_status armral_cmplx_vecmul_i16_2 (int32_t n, const int16_t *a_re, const int16_t *a_im, const int16_t *b_re, const int16_t *b_im, int16_t *c_re, int16_t *c_im)
- armral_status armral_cmplx_vecmul_f32 (int32_t n, const armral_cmplx_f32_t *a, const armral_cmplx_f32_t *b, armral_cmplx_f32_t *c)
- armral_status armral_cmplx_vecmul_f32_2 (int32_t n, const float *a_re, const float *a_im, const float *b_re, const float *b_im, float *c_re, float *c_im)
- armral_status armral_cmplx_vecdot_f32 (int32_t n, const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_b, armral_cmplx_f32_t *p_src_c)
- armral_status armral_cmplx_vecdot_f32_2 (int32_t n, const float *p_src_a_re, const float *p_src_a_im, const float *p_src_b_re, const float *p_src_b_im, float *p_src_c_re, float *p_src_c_im)
- armral_status armral_cmplx_vecdot_i16 (int32_t n, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *p_src_c)
- armral_status armral_cmplx_vecdot_i16_2 (int32_t n, const int16_t *p_src_a_re, const int16_t *p_src_a_im, const int16_t *p_src_b_re, const int16_t *p_src_b_im, int16_t *p_src_c_re, int16_t *p_src_c_im)
- armral_status armral_cmplx_vecdot_i16_32bit (int32_t n, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *p_src_c)
- armral_status armral_cmplx_vecdot_i16_2_32bit (int32_t n, const int16_t *p_src_a_re, const int16_t *p_src_a_im, const int16_t *p_src_b_re, const int16_t *p_src_b_im, int16_t *p_src_c_re, int16_t *p_src_c_im)
- armral_status armral_cmplx_mat_mult_i16 (uint16_t m, uint16_t n, uint16_t k, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *p_dst)
- armral_status armral_cmplx_mat_mult_i16_32bit (uint16_t m, uint16_t n, uint16_t k, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *p_dst)
- armral_status armral_cmplx_mat_mult_f32 (uint16_t m, uint16_t n, uint16_t k, const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_b, armral_cmplx_f32_t *p_dst)
- armral_status armral_cmplx_mat_mult_2x2_f32 (const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_b, armral_cmplx_f32_t *p_dst)
- armral_status armral_cmplx_mat_mult_2x2_f32_iq (const float32_t *src_a_re, const float32_t *src_a_im, const float32_t *src_b_re, const float32_t *src_b_im, float32_t *dst_re, float32_t *dst_im)
- armral_status armral_cmplx_mat_mult_4x4_f32 (const armral_cmplx_f32_t *p_src_a, const armral_cmplx_f32_t *p_src_b, armral_cmplx_f32_t *p_dst)
- armral_status armral_cmplx_mat_mult_4x4_f32_iq (const float32_t *src_a_re, const float32_t *src_a_im, const float32_t *src_b_re, const float32_t *src_b_im, float32_t *dst_re, float32_t *dst_im)
- armral_status armral_solve_2x2_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, uint32_t p_xstride, armral_fixed_point_index num_fract_bits_x)
- armral_status armral_solve_2x4_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, uint32_t p_xstride, armral_fixed_point_index num_fract_bits_x)
- armral_status armral_solve_4x4_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, uint32_t p_xstride, armral_fixed_point_index num_fract_bits_x)
- armral_status armral_solve_1x4_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, armral_fixed_point_index num_fract_bits_x)

- armral_status armral_solve_1x2_f32 (uint32_t num_sub_carrier, uint32_t type, const armral_cmplx_int16_t *p_y, uint32_t p_ystride, const armral_fixed_point_index *p_y_num_fract_bits, const float32_t *p_g_real, const float32_t *p_g_imag, uint32_t p_gstride, armral_cmplx_int16_t *p_x, armral_fixed_point_index num_fract_bits_x)
- armral_status armral_cmplx_hermitian_mat_inverse_f32 (uint16_t size, uint16_t par, const armral_cmplx_f32_t *p_src, armral_cmplx_f32_t *p_dst)
- armral_status armral_seq_generator (uint16_t sequence_len, uint32_t seed, uint8_t *p_dst)
- armral_status armral_modulation (uint32_t nbits, armral_modulation_type mod_type, const int8_t *p_src, armral_cmplx_int16_t *p_dst)
- armral_status armral_demodulation (uint32_t n_symbols, int16_t amp, uint16_t noise_power, armral_modulation_type mod_type, const armral_cmplx_int16_t *p_src, int8_t *p_dst)
- armral_status armral_corr_coeff_i16 (int32_t n, const armral_cmplx_int16_t *p_src_a, const armral_cmplx_int16_t *p_src_b, armral_cmplx_int16_t *c)
- armral_status armral_fir_filter_cf32 (uint32_t size, uint32_t taps, const armral_cmplx_f32_t *input, const armral_cmplx_f32_t *coeffs, armral_cmplx_f32_t *output)
- armral_status armral_fir_filter_cf32_decimate_2 (uint32_t size, uint32_t taps, const armral_cmplx_f32_t *input, const armral_cmplx_f32_t *coeffs, armral_cmplx_f32_t *output)
- armral_status armral_fir_filter_cs16 (uint32_t size, uint32_t taps, const armral_cmplx_int16_t *input, const armral_cmplx_int16_t *coeffs, armral_cmplx_int16_t *output)
- armral_status armral_fir_filter_cs16_decimate_2 (uint32_t size, uint32_t taps, const armral_cmplx_int16_t *input, const armral_cmplx_int16_t *coeffs, armral_cmplx_int16_t *output)
- armral_status armral_mu_law_compression (const int16_t *p_rb_in, int8_t *p_out, int8_t *comp_shift)
- armral_status armral_mu_law_expansion (const int8_t *p_comp_rb_in, int8_t comp_shift, int16_t *p_expanded_out)
- armral_status armral_block_float_compr_8bit (uint32_t n_prb, const armral_cmplx_int16_t *src, armral_compressed_data_8bit *dst)

  Block floating-point compression to 8-bit.
- armral_status armral_block_float_compr_9bit (uint32_t n_prb, const armral_cmplx_int16_t *src, armral_compressed_data_9bit *dst)

  Block floating point compression to 9-bit.
- armral_status armral_block_float_compr_12bit (uint32_t n_prb, const armral_cmplx_int16_t *src, armral_compressed_data_12bit *dst)

  Block floating point compression to 12-bit.
- armral_status armral_block_float_decompr_8bit (uint32_t n_prb, const armral_compressed_data_8bit *src, armral_cmplx_int16_t *dst)

  Block floating-point decompression from 8 bit.
- armral_status armral_block_float_decompr_9bit (uint32_t n_prb, const armral_compressed_data_9bit *src, armral_cmplx_int16_t *dst)

  Block floating point decompression from 9 bit.
- armral_status armral_block_float_decompr_12bit (uint32_t n_prb, const armral_compressed_data_12bit *src, armral_cmplx_int16_t *dst)

  Block floating point decompression from 12 bit.
- armral_status armral_crc24_a_le (uint32_t size, const uint64_t *input, uint64_t *crc24)
- armral_status armral_crc24_a_be (uint32_t size, const uint64_t *input, uint64_t *crc24)
- armral_status armral_crc24_b_le (uint32_t size, const uint64_t *input, uint64_t *crc24)
- armral_status armral_crc24_b_be (uint32_t size, const uint64_t *input, uint64_t *crc24)
- armral_status armral_crc24_c_le (uint32_t size, const uint64_t *input, uint64_t *crc24)
- armral_status armral_crc24_c_be (uint32_t size, const uint64_t *input, uint64_t *crc24)
- armral_status armral_polar_encoder (uint16_t n, const uint32_t *p_u_seq_in, uint32_t *p_d_seq_out)
- armral_status armral_polar_decoder (uint16_t n, uint16_t k, const int8_t *p_llr_in, uint32_t *p_u_seq_out)
- armral_status armral_fft_create_plan_cf32 (armral_fft_plan_t **p, int n, armral_fft_direction_t dir)

  Creates a plan to solve a complex fp32 FFT.
- armral_status armral_fft_execute_cf32 (const armral_fft_plan_t *p, const armral_cmplx_f32_t *x, armral_cmplx_f32_t *y)

Performs a single FFT using the specified plan and arrays.

- armral_status armral_fft_destroy_plan_cf32 (armral_fft_plan_t **p)

  Destroys an FFT plan.
- armral_status armral_fft_create_plan_cs16 (armral_fft_plan_t **p, int n, armral_fft_direction_t dir)

  Creates a plan to solve a complex int16 (Q0.15 format) FFT.
- armral_status armral_fft_execute_cs16 (const armral_fft_plan_t *p, const armral_cmplx_int16_t *x, armral_cmplx_int16_t *y)

  Performs a single FFT using the specified plan and arrays.
- armral_status armral_fft_destroy_plan_cs16 (armral_fft_plan_t **p)

  Destroys an FFT plan.

8.1.1 Macro Definition Documentation

8.1.1.1 ARMRAL_NUM_COMPLEX_SAMPLES

```
#define ARMRAL_NUM_COMPLEX_SAMPLES 12
```

The number of complex samples in each compressed block.

8.1.2 Enumeration Type Documentation

8.1.2.1 armral_fixed_point_index

```
enum armral_fixed_point_index
```

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

Enumerator

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

Enumerator

ARMRAL_FIXED_POINT_INDEX_Q11_4	1 sign bit, 11 integer bits, 4 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q12_3	1 sign bit, 12 integer bits, 3 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q13_2	1 sign bit, 13 integer bits, 2 fractional bits
ARMRAL_FIXED_POINT_INDEX_Q14_1	1 sign bit, 14 integer bits, 1 fractional bit
ARMRAL_FIXED_POINT_INDEX_Q15_0	1 sign bit, 15 integer bits, 0 fractional bits

8.1.2.2 armral_modulation_type

```
enum armral_modulation_type
```

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

Enumerator

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

8.1.2.3 armral_status

```
enum armral_status
```

Error status returned by functions in the library.

Enumerator

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

