Arm Fortran Compiler Reference Guide

Version 19.1.0

Document Number 101380_1910_00

CONTENTS

Release information

Table 1: Document history

Issue	Date	Confidentiality	Change
1830_00	20 June 2018	Non-Confidential	18.3.0
1840_00	27 July 2018	Non-Confidential	18.4.0
1900_00	2 November 2018	Non-Confidential	19.0.0
1910_00	8 March 2019	Non-Confidential	19.1.0

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

Product Status

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

Web Address

http://www.arm.com

CHAPTER

ONE

OVERVIEW

Gives an overview of the Arm Fortran Compiler, the information within this book, and provides information on how to get further support.

1.1 Arm Fortran Compiler

Arm Fortran Compiler is an auto-vectorizing, Linux user-space Fortran compiler, tailored for High Performance Computing (HPC) and scientific workloads. It is built on the open-source Flang front-end and the LLVM-based optimization and code generation back-end. It supports popular Fortran and OpenMP standards and is tuned for 64-bit Armv8-A architecture.

Arm Fortran Compiler is available in combination with Arm C/C++ Compiler, Arm Performance Libraries, Arm Forge, and Arm Performance Reports as part of the Arm Allinea Studio. Arm Allinea Studio is the end-to-end commercial suite for building and porting HPC applications on Arm.

1.2 About this book

This document contains information on the adherance of the Arm Fortran Compiler with the various Fortran standards. It also describes the compatibility with various Fortran language features, statements and instrinsics. In addition, it describes the available compiler options, includes some *Getting started* content, and provides information and examples on using some of the compiler features.

This guide is not a tutorial, instead it is intended for application programmers who have a basic understanding of Fortran concepts and standards.

1.3 Getting help

You can find further help and resources on the Arm Developer website. If you need further assistance, Contact Arm Support.

CHAPTER

TWO

GET STARTED

Arm Fortran Compiler is an auto-vectorizing compiler for the 64-bit Arm®v8-A architecture. This getting started tutorial shows how to install, compile Fortran code, use different optimization levels and generate an executable.

The Arm Fortran Compiler tool chain for the 64-bit Arm®v8-A architecture enables you to compile Fortran code for Arm®v8-A compatible platforms, with an advanced auto-vectorizer capable of taking advantage of SIMD features.

2.1 Installation

Refer to Installing Arm Compiler for HPC for information on installing Arm Fortran Compiler.

2.2 Configuring environment

As part of the installation, your administrator should have made the Arm Compiler for HPC environment module available. To see which environment modules are available:

```
module avail
```

Note: You may need to configure the MODULEPATH environment variable to include the installation directory:

```
export MODULEPATH=$MODULEPATH:/opt/arm/modulefiles/
```

To configure your Linux environment to make Arm Fortran Compiler for HPC available:

```
module load <architecture>/<linux_variant>/<linux_version>/suites/
arm-compiler-for-hpc/<version>
```

For example:

```
module load Generic-AArch64/SUSE/12/suites/arm-compiler-for-hpc/19.1
```

You can check your environment by examining the PATH variable. It should contain the appropriate bin directory from /opt/arm, as installed in the previous section:

```
echo $PATH
/opt/arm/arm-compiler-for-hpc-19.1_Generic-AArch64_SUSE-
12_aarch64-linux/bin:...
```

Note: You might want to consider adding the module load command to your .profile to run it automatically every time you log in.

2.3 Compiling and running a simple "Hello World" program

This example illustrates how to compile and run a simple "Hello World" Fortran program.

1. Create a simple "hello world" program and save it in a file. In our case, we have saved it in a file named hello.f90

```
program hello
  print *, 'hello world'
  end
```

2. To generate an executable binary, compile your program with Arm Fortran Compiler for HPC.

```
armflang -o hello hello.f90
```

3. Now you can run the generated binary `hello` as shown below

```
./hello
```

In the following sections we discuss the available compiler options in more detail and, towards the end of this tutorial, discuss compiling Fortran code for SVE-enabled targets.

2.4 Generating executable binaries from Fortran code

To generate an executable binary, compile a program using:

```
armflang -o example1 example1.f90
```

You can also specify multiple source files on a single line. Each source file is compiled individually and then linked into a single executable binary:

```
armflang -o example1 example1a.f90 example1b.f90
```

2.5 Compiling and linking object files as separate steps

To compile each of your source files individually into an object file, specify the `-c` (compile-only) option, and then pass the resulting object files into another invocation of `armflang` to link them into an executable binary.

```
armflang -c -o file1a.o file1a.f90
armflang -c -o file1b.o file1b.f90
armflang -o file1 file1a.o file2a.o
```

2.6 Increasing the optimization level

To increase the optimization level, use the `-O<level>` option. The `-O0` option is the lowest optimization level, while `-O3` is the highest. Arm Fortran Compiler only performs auto-vectorization at `-O2` and higher, and uses `-O0` as the default setting. The optimization flag can be specified when generating a binary, such as:

```
armflang -O3 -o example1 example1.f90
```

The optimization flag can also be specified when generating an object file:

```
armflang -O3 -c -o example1a.o example1a.f90
armflang -O3 -c -o example1b.o example1b.f90
```

or when linking object files:

```
armflang -O3 -o example1 example1a.o example1b.o
```

2.7 Compiling and optimizing using CPU auto-detection

Arm Fortran Compiler supports the use of the `-mcpu=native` option, for example:

```
armflang -O3 -mcpu=native -o example1 example1.f90
```

This option enables the compiler to automatically detect the architecture and processor type of the CPU it is being run on, and optimize accordingly.

This option supports a range of Arm®v8-A based SoCs, including ThunderX2.

Note: The optimization performed according to the auto-detected architecture and processor is independent of the optimization level denoted by the `-O<level>` option.

2.8 Compiling Fortran code for SVE-enabled target architectures

The Arm Fortran Compiler toolchain for the 64-bit Armv8-A architecture supports the Scalable Vector Extensions (SVE), enabling you to:

- Assemble source code containing SVE instructions.
- Disassemble ELF object files containing SVE instructions.
- Compile C and C++ code for SVE-enabled targets, with an advanced auto-vectorizer capable of taking advantage of SVE features.

To optimize Fortran code for an SVE-enabled target, enable auto-vectorization by using optimization level `-O2` or `-O3`, and specify an SVE-enabled target architecture using the `-march=` option:

```
armflang -O3 -march=armv8-a+sve -o example1 example1.f90
```

In this example, the Armv8-A target architecture is specified.

You can also specify multiple source files on a single line. Each source file is compiled individually and then linked into a single executable binary:

```
armflang -O3 -march=armv8-a+sve -o example2 example2a.f90 example2b.f90
```

2.9 Common compiler options

`-S`

Outputs assembly code, rather than object code. Produces a text `.s` file containing annotated assembly code.

`-c`

Performs the compilation step, but does not perform the link step. Produces an ELF object `.o` file. To later link object files into an executable binary, run `armflang` again, passing in the object files.

`-o file`

Specifies the name of the output file.

`-march=name[+[no]feature]`

Targets an architecture profile, generating generic code that runs on any processor of that architecture. For example `-march=armv8-a+sve`.

`-mcpu=native`

Enables the compiler to automatically detect the CPU it is being run on and optimize accordingly. This supports a range of Armv8-A based SoCs, including ThunderX2.

`-Olevel`

Specifies the level of optimization to use when compiling source files. The default is `-O0`.

`--help`

Describes the most common options supported by Arm Fortran Compiler for HPC.

`--version`

Displays version information.

For a detailed descriptions of all the supported compiler options, see *Compiler options*.

To view the supported options on the command-line, use the `man` pages:

```
man armflang
```

2.10 Get support

Command line help is accessible through the `--help` option:

```
armflang --help
```

If you have problems and would like to contact our support team, Get in touch.

CHAPTER

THREE

COMPILER OPTIONS

This page lists the command-line options currently supported by `armflang` within Arm Fortran Compiler.

The supported options are also available within the `man` pages built into the tool. To view them, use:

```
man armflang
```

3.1 Actions

Control what action to perform on the input.

Table 1: Compiler actions

Option	Description
`-E`	Only run the preprocessor. **Usage** `armflang -E`
`-S`	Only run preprocess and compilation steps. **Usage** `armflang -S`
`-c`	Only run preprocess, compile, and assemble steps. **Usage** `armflang -c`
`-fopenmp`	Enable OpenMP and link in the OpenMP library, libomp. **Usage** `armflang -fopenmp`
`-fsyntax-only`	Show syntax errors but do not perform any compilation. **Usage** `armflang -fsyntax-only`

3.2 File options

Specify input or output files.

Table 2: Compiler file options

Option	Description
`-I<dir>`	Add directory to include search path. **Usage** `armflang -I<dir>`

Continued on next page

Table 2 – continued from previous page

Option	Description

`-include <file>`	Include file before parsing.
	Usage
	`armflang -include <file>`
	Or
	`armflang --include <file>"`

`-o <file>`	Write output to `<file>`. **Usage** `armflang -o <file>`

3.3 Basic driver options

Configure basic functionality of the `armflang` driver.

Table 3: Compiler basic driver options

Option	Description
`--gcc-toolchain=<arg>`	Use the gcc toolchain at the given directory. **Usage** `armflang --gcc-toolchain=<arg>`

`-help`
`--help`
	Display available options.
	Usage
	`armflang -help`
	`armflang --help`

`--help-hidden`	Display hidden options. Only use these options if advised to do so by your Arm representative. **Usage** `armflang --help-hidden`

`-v`	Show commands to run and use verbose output.
	Usage
	`armflang -v`
	`--version`

`--vsn"`	Show the version number and some other basic information about the compiler.
	Usage
	`armflang --version`
	`armflang --vsn"`

3.4 Optimization options

Control optimization behavior and performance.

Table 4: Compiler optimization options

Option	Description
`-O0`	Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image. This is the default optimization level. **Usage** `armflang -O0`

Continued on next page

Table 4 – continued from previous page

Option	Description
`-O1`	Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug. **Usage** `armflang -O1`
`-O2`	High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information. **Usage** `armflang -O2`
`-O3`	Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels. **Usage** `armflang -O3`
`-Ofast`	Enable all the optimizations from level 3, including those performed with the -ffp-mode=fast armflang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards. **Usage** `armflang -Ofast`
`-ffast-math`	Allow aggressive, lossy floating-point optimizations. **Usage** `armflang -ffast-math`
`-ffinite-math-only`	Enable optimizations that ignore the possibility of NaN and +/-Inf. **Usage** `armflang -ffinite-math-only`
`-ffp-contract={fast|on|off}`	Controls when the compiler is permitted to form fused floating-point operations (such as FMAs). fast: Always (default). on: Only in the presence of the FP_CONTRACT pragma. off: Never. **Usage** `armflang -ffp-contract={fast|on|off}`

`-finline`
`-fno-inline`
	Enable or disable inlining (enabled by default).
	Usage
	`armflang -finline`
	(enable)
	`armflang -fno-inline`
	(disable)

`-fstack-arrays`
`-fnostack-arrays`
	Place all automatic arrays on stack memory.
	For programs using very large arrays on particular operating systems, consider extending stack memory runtime limits. Enabled by default at optimization level `-Ofast`.
	Usage
	`armflang -fstack-arrays`
	(enable)
	`armflang -fnostack-arrays`
	(disable)

Continued on next page

Table 4 – continued from previous page

Option	Description
`-fstrict-aliasing`	Tells the compiler to adhere to the aliasing rules defined in the source language. In some circumstances, this flag allows the compiler to assume that pointers to different types do not alias. Enabled by default when using `-Ofast`. **Usage** `armflang -fstrict-aliasing`

`-funsafe-math-optimizations`
`-fno-unsafe-math-optimizations`
	This option enables reassociation and reciprocal math optimizations, and does not honor trapping nor signed zero.
	Usage
	`armflang -funsafe-math-optimizations`
	(enable)
	`armflang-fno-unsafe-math-optimizations`
	(disable)

`-fvectorize`
`-fno-vectorize`
	Enable/disable loop vectorization (enabled by default).
	Usage
	`armflang -fvectorize`
	(enable)
	`armflang -fno-vectorize`
	(disable)

`-mcpu=<arg>`	Select which CPU architecture to optimize for `-mcpu=native` causes the compiler to auto-detect the CPU architecture from the build computer. **Usage** `armflang -mcpu=<arg>`

3.5 Workload compilation options

Configure how Fortran workloads compile.

Table 5: Compiler workload compilation options

Option	Description

`-frealloc-lhs`
`-fno-realloc-lhs`
	`-frealloc-lhs` uses Fortran 2003 standard semantics for assignments to allocatables. An allocatable object on the left-hand side of an assignment is automatically allocated, or re-allocated, to match the dimensions of the right-hand side. This is the default behavior.
	`-fno-realloc-lhs` uses Fortran 95 standard semantics for assignments to allocatables. The left-hand side of an allocatable assignment is assumed to be allocated with the correct dimensions. Incorrect behavior can occur if the left-hand side is not allocated with the correct dimensions.
	Note: In Arm Fortran Compiler versions 19.0 and earlier, `-Mallocatable=03` was supported instead of `-frealloc-lhs`, and `-Mallocatable=95` was supported instead of `-fno-realloc-lhs`.
	Usage
	`armflang -frealloc-lhs`
	`armflang -fno-realloc-lhs`

Continued on next page

Table 5 – continued from previous page

Option	Description
`-cpp`	Preprocess Fortran files. **Usage** `armflang -cpp`

`-fbackslash`
`-fno-backslash`
	Treat backslash as C-style escape character (`-fbackslash`) or as a normal character (`-fno-backslash`).
	Usage
	`armflang -fbackslash`
	(enable)
	`armflang -fno-backslash`
	(disable)

`-fconvert={native|swap|big-endian little-endian}`	Convert between big and little endian data format. Default = native. **Usage** `armflang -fconvert={native|swap|big-endian little-endian}`
`-ffixed-form`	Force fixed-form format Fortran. This is default for .f and .F files, and is the inverse of `-ffree-form`. **Usage** `armflang -ffixed-form`
`-ffixed-line-length={0|72|132|none}`	Set line length in fixed-form format Fortran. Default = 72. 0 and none are equivalent and set the line length to a very large value (> 132). **Usage** `armflang -ffixed-line-length={72|132}`
`-ffree-form`	Force free-form format for Fortran. This is default for .f90 and .F90 files, and is the inverse of `-ffixed-form`. **Usage** `armflang -ffree-form`
`-fma`	Enable generation of FMA instructions. **Usage** `armflang -fma`
`-fno-fortran-main`	Do not link in Fortran main. **Usage** `armflang -fno-fortran-main`
`-frecursive`	Allocate all local arrays on the stack, allowing thread-safe recursion. In the absence of this flag, some large local arrays may be allocated in static memory. This reduces stack , but is not thread-safe. This flag is enabled by default when `-fopenmp` is given. **Usage** `armflang -frecursive`
`-i8`	Treat `INTEGER` and `LOGICAL` as `INTEGER*8` and `LOGICAL*8`. **Usage** `armflang -i8`
`-no-flang-libs`	Do not link against Flang libraries. **Usage** `armflang -no-flang-libs`
`-nocpp`	Don't preprocess Fortran files. **Usage** `armflang -nocpp`
`-nofma`	Disable generation of FMA instructions. **Usage** `armflang -nofma`

Continued on next page

Table 5 – continued from previous page

Option	Description
`-r8`	Treat `REAL` as `REAL*8`. **Usage** `armflang -r8`
`-static-flang-libs`	Link using static Flang libraries. **Usage** `armflang -static-flang-libs`

3.6 Development options

Support code development.

Table 6: Compiler development options

Option	Description

`-fcolor-diagnostics`
`-fno-color-diagnostics`
	Use colors in diagnostics.
	Usage
	`armflang -fcolor-diagnostics`
	Or
	`armflang -fno-color-diagnostics`

`-g`	Generate source-level debug information. **Usage** `armflang -g`

3.7 Warning options

Control the behavior of warnings.

Table 7: Compiler warning options

Option	Description
`-W<warning>` `-Wno-<warning>`	Enable or disable the specified warning. **Usage** `armflang -W<warning>`
`-Wall`	Enable all warnings. **Usage** `armflang -Wall`
`-w`	Suppress all warnings. **Usage** `armflang -w`

3.8 Pre-processor options

Control pre-processor behavior.

Table 8: Compiler pre-processing options

Option	Description
`-D <macro>=<value>`	Define `<macro>` to `<value>` (or 1 if `<value>` is omitted). **Usage** `armflang -D<macro>=<value>`

Continued on next page

Table 8 – continued from previous page

Option	Description
-U	Undefine macro <macro>. **Usage** armflang -U<macro>

3.9 Linker options

Control linking behavior and performance.

Table 9: Compiler linker options

Option	Description
-Wl,<arg>	Pass the comma separated arguments in <arg> to the linker. **Usage** armflang -Wl,<arg>, <arg2>...
-Xlinker <arg>	Pass <arg> to the linker. **Usage** armflang -Xlinker <arg>
-l<library>	Search for the library named <library> when linking. **Usage** armflang -l<library>
-larmflang	At link time, include this option to use the default Fortran libarmflang runtime library for both serial and parallel (OpenMP) Fortran workloads. **Note:** • This option is set by default when linking using armflang. • You need to explicitly include this option if you are linking with armclang instead of armflang at link time. • This option only applies to link time operations. **Usage** armclang -larmflang See notes in description.
-larmflang-nomp	At link time, use this option to avoid linking against the OpenMP Fortran runtime library. **Note:** • Enabled by default when compiling and linking using armflang with the -fno-openmp option. • You need to explicitly include this option if you are linking with armclang instead of armflang at link time. • Should not be used when your code has been compiled with the -lomp or -fopenmp options. • Use this option with care. When using this option, do not link to any OpenMP-utilizing Fortran runtime libraries in your code. • This option only applies to link time operations. **Usage** armclang -larmflang-nomp See notes in description.

Continued on next page

Table 9 – continued from previous page

Option	Description

`-shared`
`--shared`
	Causes library dependencies to be resolved at runtime by the loader.
	This is the inverse of -static. If both options are given, all but the last option will be ignored.
	Usage
	`armflang -shared`
	Or
	`armflang --shared`

`-static`
`--static`
	Causes library dependencies to be resolved at link time.
	This is the inverse of `-shared`. If both options are given, all but the last option is ignored.
	Usage
	`armflang -static`
	Or
	`armflang --static`

To link serial or parallel Fortran workloads using `armclang` instead of `armflang`, include the `-larmflang` option to link with the default Fortran runtime library for serial and parallel Fortran workloads. You also need to pass any options required to link using the required mathematical routines for your code.

To statically link, in addition to passing `-larmflang` and the mathematical routine options, you also need to pass:

- `-static`
- `-lomp`
- `-lrt`

To link serial or parallel Fortran workloads using `armclang` instead of `armflang`, without linking against the OpenMP runtime libraries, instead pass `-armflang-nomp` at link time. For example, pass:

- `-larmflang-nomp`
- Any mathematical routine options, for example: `-lm` or `-lamath`.

Again, to statically link, in addition to `-larmflang-nomp` and the mathematical routine options, you also need to pass:

- `-static`
- `-lrt`

Warning:

- Do not link against any OpenMP-utlizing Fortran runtime libraries when using this option.
- All lockings and thread local storage will be disabled.
- Arm does not recommend using the `-larmflang-nomp` option for typical workloads. Use this option with caution.

Note: The `-lompstub` option (for linking against libompstub) might still be needed if you have imported `omp_lib` in your Fortran code but not compiled with `-fopenmp`.

CHAPTER

FOUR

FORTRAN DATA TYPES AND FILE EXTENSIONS

This topic describes, the data types and file extensions supported by Arm Fortran Compiler.

4.1 Data types

Arm Fortran Compiler provides the following intrinsic data types:

Table 1: Intrinsic data types

** Data Type**	**Specified as**	**Size (bytes)**
INTEGER	INTEGER INTEGER*1 INTEGER([KIND=]1) INTEGER*2 INTEGER([KIND=]2) INTEGER*4 INTEGER([KIND=]4) INTEGER*8 INTEGER([KIND=]8)	4 1 1 2 2 4 4 8 8
REAL	REAL REAL*4 REAL([KIND=]4) REAL*8 REAL([KIND=]8)	4 4 4 8 8
DOUBLE PRECISION	DOUBLE PRECISION (same as REAL*8, no KIND parameter is permitted)	16
COMPLEX	COMPLEX COMPLEX*8 COMPLEX([KIND=]4) COMPLEX*16 COMPLEX([KIND=]8)	4 8 8 16 16
DOUBLE COMPLEX	DOUBLE COMPLEX (same as COMPLEX*8, no KIND parameter is permitted)	8

Continued on next page

Table 1 – continued from previous page

** Data Type**	Specified as	Size (bytes)
LOGICAL	LOGICAL LOGICAL*1 LOGICAL([KIND=]1) LOGICAL*2 LOGICAL([KIND=]2) LOGICAL*4 LOGICAL([KIND=]4) LOGICAL*8 LOGICAL([KIND=]8)	4 1 1 2 2 4 4 8 8
CHARACTER	CHARACTER CHARACTER([KIND=]1)	1 1
BYTE	BYTE (same as INTEGER([KIND=]1))	1

Note:

- The default entries are the first entries for each intrinsic data type.
- To determine the kind type parameter of a representation method, use the intrinsic function `KIND`.

For more portable programs, define a `PARAMETER` constant using the appropriate `SELECTED_INT_KIND` or `SELECTED_REAL_KIND` functions, as appropriate.

For example, this code defines a `PARAMETER` constant for an `INTEGER` kind that has 9 digits:

```
INTEGER, PARAMETER :: MY_INT_KIND = SELECTED_INT_KIND(9)
...
INTEGER(MY_INT_KIND) :: J
...
```

4.2 Supported file extensions

The extensions `f90`, `.f95`, `.f03`, and `.f08` are used for modern, free-form source code conforming to the Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 standards, respectively.

The extensions `.F90`, `.F95`, `.F03`, and `.F08` are used for modern, free-form source code that require preprocessing, and conform to the Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 standards, respectively.

The `.f` and `.for` extensions are typically used for older, fixed-form code such as FORTRAN77.

The file extensions that are compatible with Arm Fortran Compiler are:

Table 2: Supported file extensions.

File Extension	Interpretation
a.out	Executable output file.
file.a	Library of object files.
file.f file.for	Fixed-format Fortran source file.

Continued on next page

Table 2 – continued from previous page

File Extension	Interpretation
file.fpp file.F	Free-format Fortran source file that requires preprocessing.
file.f90 file.f95 file.f03 file.f08	Free-format Fortran source file.
file.F90 file.F95 file.F03 file.F08	Free-format Fortran source file that requires preprocessing.
file.o	Compiled object file.
file.s	Assembler source file.

4.3 Logical variables and constants

A `LOGICAL` constant is either `True` or `False`. The Fortran standard does not specify how variables of `LOGICAL` type are represented. However, it does require `LOGICAL` variables of default kind to have the same storage size as default `INTEGER` and `REAL` variables.

For Arm Fortran Compiler:

- `.TRUE.` corresponds to -1 and has a default storage size of 4-bytes.
- `.FALSE.` corresponds to 0 and has a default storage size of 4-bytes.

Note: Some compilers represent `.TRUE.` and `.FALSE.` as 1 and 0, respectively.

4.4 C/Fortran inter-language calling

This section provides some useful troubleshooting information when handling argument passing and return values for Fortran functions or subroutines called from C/C++ code.

In Fortran, arguments are passed by reference. Here, reference means the address of the argument is passed, rather than the argument itself. In C/C++, arguments are passed by value, except for strings and arrays, which are passed by reference.

C/C++ provides some flexibility when solving passing difference with Fortran. Usually, intelligent use of the & and * operators in argument passing enables you to call Fortran from C/C++, and in argument declarations when Fortran is calling C/C++.

Fortran functions which return `CHARACTER` or `COMPLEX` data types require special consideration when called from C/C++ code.

4.5 Character

Fortran functions that return a character require the *calling* C/C++ function to have two arguments to describe the result:

1. The first argument provides the address of the returned character.
2. The second argument provides the length of the returned character.

For example, the Fortran function:

```
CHARACTER*(*) FUNCTION CHF( C1, I)
     CHARACTER*(*) C1
     INTEGER I
END
```

when called in C/C++, has an extra declaration:

```
extern void chf_();
     char tmp[10];
     char c1[9];
     int i;
     chf_(tmp, 10, c1, &i, 9);
```

The argument, `tmp`, provides the address, and the length is defined with the second argument, `10`.

Note:

- Fortran functions declared with a character return length, for example `CHARACTER*4 FUNCTION CHF()`, still require the second parameter to be supplied to the calling C/C++ code.
- The value of the character function is not automatically NULL-terminated.

4.6 Complex

Fortran functions that return a `COMPLEX` data type cannot be directly called from C/C++. Instead, a workaround is possible by passing a C/C++ function a pointer to a memory area. This memory area can then be calling the `COMPLEX` function and storing the returned value.

For example, the Fortran function:

```
SUBROUTINE INTER_CF(C, I)
  COMPLEX C
  COMPLEX CF
  C = CF(I)
  RETURN
END

COMPLEX FUNCTION CF(I)
  . . .
END
```

when called in C/C++ is completed using a memory pointer:

```
extern void inter_cf_();
    typedef struct {float real, imag;} cplx;
    cplx c1;
    int i;
    inter_cf_(&c1, &i);
```

4.7 Arm Fortran Compiler Fortran implementation notes

Additional information specific to the Arm Fortran Compiler:

- **Arm Fortran Compiler does not initialize arrays or variables with** zeros.

Note: This behavior varies from compiler to compiler and is not defined within Fortran standards. It is best practice to not assume arrays are filled with zeros when created.

CHAPTER

FIVE

FORTRAN STATEMENTS

This topic describes the Fortran statements supported within Arm Fortran Compiler.

5.1 Statements

The Fortran statements supported within the Arm Fortran Compiler, are:

Table 1: Supported Fortran statements

Statement	Language standard	Brief description
ACCEPT	F77	Causes formatted input to be read on standard input.
ALLOCATABLE	F90	Specifies that an array with fixed rank, but deferred shape, is available for a future ALLOCATE statement.
ALLOCATE	F90	Allocates storage for each allocatable array, pointer object, or pointer-based variable that appears in the statements; declares storage for deferred-shape arrays. **Note:** Arm Fortran Compiler does not initialize arrays or variables with zeros. It is best practice to not assume that arrays are filled with zeros when created.
ASSIGN	F77	Assigns a statement label to a variable. **Note:** This statement is a deleted feature in the Fortran standard, but remains supported in the Arm Fortran Compiler.
ASSOCIATE	F2003	Associates a name either with a variable or with the value of an expression, while in a block.
ASYNCHRONOUS	F77	Warns the compiler that incorrect results may occur for optimizations involving movement of code across wait statements, or statements that cause wait operations.
BACKSPACE	F77	Positions the file that is connected to the specified unit, to before the preceding record.
BLOCK DATA	F77	Introduces several non-executable statements that initialize data values in COMMON tables.
BYTE	F77 ext	Establishes the data type of a variable by explicitly attaching the name of a variable to a 1-byte integer, overriding implied data typing.
CALL	F77	Transfers control to a subroutine.
CASE	F90	Begins a case-statement-block portion of a SELECT CASE statement.

Continued on next page

Table 1 – continued from previous page

Statement	Language standard	Brief description
CHARACTER	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a character data type, overriding the implied data typing. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
CLOSE	F77	Terminates the connection of the specified file to a unit.
COMMON	F77	Defines global blocks of storage that are either sequential or non-sequential. May be either static or dynamic form. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
COMPLEX	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a complex data type, overriding implied data typing.
CONTAINS	F90 F2003	Precedes a subprogram, a function or subroutine, and indicates the presence of the subroutine or function definition inside a main program, external subprogram, or module subprogram. In F2003, a CONTAINS statement can also appear in a derived type immediately before any type-bound procedure definitions.
CONTINUE	F77	Passes control to the next statement.
CYCLE	F90	Interrupts a DO construct execution and continues with the next iteration of the loop.
DATA	F77	Assigns initial values to variables before execution. **Note:** This statement amongst execution statements has been marked as obsolescent. This functionality is redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
DEALLOCATE	F90	Causes the memory that is allocated for each pointer-based variable or allocatable array that appears in the statement to be deallocated (freed). Also may be used to deallocate storage for deferred-shape arrays.
DECODE	F77 ext	Transfers data between variables or arrays in internal storage and translates that data from character form to internal form, according to format specifiers.
DIMENSION	F90	Defines the number of dimensions in an array and the number of elements in each dimension.

Continued on next page

Table 1 – continued from previous page

Statement	Language standard	Brief description
DO (Iterative)	F90	Introduces an iterative loop and specifies the loop control index and parameters. **Note:** Label form DO statements have been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
DO WHILE	F77	Introduces a logical DO loop and specifies the loop control expression.
DOUBLE COMPLEX	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to a double complex data type. This overrides the implied data typing.
DOUBLE PRECISION	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a double precision data type, overriding implied data typing.
ELSE	F77	Begins an ELSE block of an IF block, and encloses a series of statements that are conditionally executed.
ELSE IF	F77	Begins an ELSE IF block of an IF block series, and encloses statements that are conditionally executed.
ELSE WHERE	F90	The portion of the WHERE ELSE WHERE construct that permits conditional masked assignments to the elements of an array, or to a scalar, zero-dimensional array.
ENCODE	F77 ext	Transfers data between variables or arrays in internal storage and translates that data from internal to character form, according to format specifiers.
END	F77	Terminates a segment of a Fortran program.
END ASSOCIATE	F2003	Terminates an ASSOCIATE block.
END DO	F77	Terminates a DO or DO WHILE loop.
END FILE	F77	Writes an ENDFILE record to the files.
END IF	F77	Terminates an IF ELSE or ELSE IF block.
END MAP	F77 ext	Terminates a MAP declaration.
END SELECT	F90	Terminates a SELECT declaration.
END STRUCTURE	F77 ext	Terminates a STRUCTURE declaration.
END UNION	F77 ext	Terminates a UNION declaration.
END WHERE	F90	Terminates a WHERE ELSE WHERE construct.
ENTRY	F77	Allows a subroutine or function to have more than one entry point. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.

Continued on next page

Table 1 – continued from previous page

Statement	Language standard	Brief description
EQUIVALENCE	F77	Allows two or more named regions of data memory to share the same start address. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
EXIT	F90	Interrupts a DO construct execution and continues with the next statement after the loop.
EXTERNAL	F77	Identifies a symbolic name as an external or dummy procedure which can then be used as an argument.
FINAL	F2003	Specifies a final subroutine inside a derived type.
FORALL	F95	Provides, as a statement or construct, a parallel mechanism to assign values to the elements of an array. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
FORMAT	F77	Specifies format requirements for input or output.
FUNCTION	F77	Introduces a program unit; all the statements that follow apply to the function itself.
GENERIC	F2003	Specifies a generic type-bound procedure inside a derived type.
GOTO (Assigned)	F77	Transfers control so that the statement identified by the statement label is executed next. **Note:** This statement is a deleted feature in the Fortran standard, but remains supported in the Arm Fortran Compiler.
GOTO (Computed)	F77	Transfers control to one of a list of labels, according to the value of an expression. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
GOTO (Unconditional)	F77	Unconditionally transfers control to the statement with the label label, which must be declared within the code of the program unit containing the GOTO statement, and must be unique within that program unit.
IF (Arithmetic)	F77	Transfers control to one of three labeled statements, depending on the value of the arithmetic expression. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
IF (Block)	F77	Consists of a series of statements that are conditionally executed.

Continued on next page

Table 1 – continued from previous page

Statement	Language standard	Brief description
IF (Logical)	F77	Executes or does not execute a statement based on the value of a logical expression.
IMPLICIT	F77	Redefines the implied data type of symbolic names from their initial letter, overriding implied data types.
IMPORT	F2003	Gives access to the named entities of the containing scope.
INCLUDE	F77 ext	Directs the compiler to start reading from another file.
INQUIRE	F77	Inquires about the current properties of a particular file or the current connections of a particular unit.
INTEGER	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to an integer data type, overriding implied data types.
INTENT	F90	Specifies intended use of a dummy argument, but may not be used in a specification statement of a main program.
INTERFACE	F90	Makes an implicit procedure an explicit procedure where the dummy parameters and procedure type are known to the calling module; Also overloads a procedure name.
INTRINSIC	F77	Identifies a symbolic name as an intrinsic function and allows it to be used as an actual argument.
LOGICAL	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to a logical data type, overriding implied data types.
MAP	F77 ext	Designates each unique field or group of fields within a UNION statement.
MODULE	F90	Specifies the entry point for a Fortran 90, or Fortran 95, module program unit. A module defines a host environment of scope of the module, and may contain subprograms that are in the same scoping unit.
NAMELIST	F90	Allows the definition of NAMELIST groups for NAMELIST-directed I/O.
NULLIFY	F90	Disassociates a pointer from its target.
OPEN	F77	Connects an existing file to a unit, creates and connects a file to a unit, creates a file that is pre-connected, or changes certain specifiers of a connection between a file and a unit.
OPTIONAL	F90	Specifies dummy arguments that may be omitted or that are optional.
OPTIONS	F77 ext	Confirms or overrides certain compiler command-line options.
PARAMETER	F77	Gives a symbolic name to a constant.
PAUSE	F77	Stops program execution. **Note:** This statement is a deleted feature in the Fortran standard, but remains supported in the Arm Fortran Compiler.
POINTER	F90	Provides a means for declaring pointers.

Continued on next page

Table 1 – continued from previous page

Statement	Language standard	Brief description
PRINT	F77	Transfers data to the standard output device from the items that are specified in the output list and format specification.
PRIVATE	F90 F2003	Specifies that entities that are defined in a module are not accessible outside of the module. PRIVATE can also appear inside a derived type to disallow access to its data components outside the defining module. In F2003, a PRIVATE statement may appear after CONTAINS statement of the type, to disallow access to type-bound procedures outside the defining module.
PROCEDURE	F2003	Specifies a type-bound procedure, procedure pointer, module procedure, dummy procedure, intrinsic procedure, or an external procedure.
PROGRAM	F77	Specifies the entry point for a linked Fortran program.
PROTECTED	F2003	Protects a module variable against modification from outside the module in which it was declared.
PUBLIC	F90	Specifies that entities that are defined in a module are accessible outside of the module.
PURE	F95	Indicates that a function or subroutine has no side effects.
READ	F77	Transfers data from the standard input device to the items specified in the input and format specifications.
REAL	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a data type, overriding implied data types.
RECORD	F77 ext	A VAX Fortran extension, defines a user-defined aggregate data item.
RECURSIVE	F90	Indicates whether a function or subroutine may call itself recursively.
RETURN	F77	When used in a subroutine, causes a return to the statement following a CALL. When used in a function, returns to the relevant arithmetic expression. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and may be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
REWIND	F77	Positions the file at the start. The statement has no effect if the file is already positioned at the start, or if the file is connected but does not exist.
SAVE	F77	Retains the definition status of an entity after a RETURN or END statement in a subroutine or function that has been executed.
SELECT CASE	F90	Begins a CASE construct.

Continued on next page

Table 1 – continued from previous page

Statement	Language standard	Brief description
SELECT TYPE	F2003	Provides the capability to execute alternative code depending on the dynamic type of a polymorphic entity, and to gain access to dynamic parts. The alternative code is selected using the TYPE IS statement for a specific dynamic type, or the CLASS IS statement for a specific type (and all its type extensions). Use the optional class default statement to specify all other dynamic types that do not match a specified TYPE IS or CLASS IS statement. Like the CASE construct, the code consists of a several blocks and, at most, one is selected for execution.
SEQUENCE	F90	A derived type qualifier that specifies the ordering of the storage that is associated with the derived type. This statement specifies storage for use with COMMON and EQUIVALENCE statements.
STOP	F77	Stops program execution and precludes any further execution of the program.
STRUCTURE	F77 ext	A VAX extension to FORTRAN 77 that defines an aggregate data type.
SUBROUTINE	F77	Introduces a subprogram unit.
TARGET	F90	Specifies that a data type may be the object of a pointer variable (for example, pointed to by a pointer variable). Types that do not have the TARGET attribute cannot be the target of a pointer variable.
THEN	F77	Part of an IF block statement, surrounds a series of statements that are conditionally executed.
TYPE	F90 F2003	Begins a derived type data specification or declares variables of a specified user-defined type. Use the optional EXTENDS statement with TYPE to indicate a type extension in F2003.
UNION	F77 ext	A multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields.
USE	F90	Gives a program unit access to the public entities or to the named entities in the specified module.
VOLATILE	F77 ext	Inhibits all optimizations on the variables, arrays and common blocks that it identifies.
WAIT	F2003	Performs a wait operation for specified pending asynchronous data transfer operations.
WHERE	F90	Permits masked assignments to the elements of an array or to a scalar, zero-dimensional array.
WRITE	F77	Transfers data to the standard output device from the items that are specified in the output list and format specification.

*See WG5 Fortran Standards

Note: The denoted language standards indicate the standard they were introduced in, or the standard they were last significantly changed.

5.1.1 Related information

- WG5 Fortran Standards

CHAPTER

SIX

FORTRAN INTRINSICS

The Fortran language standards implemented in the Arm Fortran Compiler are Fortran 77, Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008. This topic details the supported and unsupported Fortran intrinsics within Arm Fortran Compiler.

6.1 Overview

An intrinsic is a function made available for a given language standard, for example, Fortran 95. Intrinsic functions accept arguments and return values. When an intrinsic function is called within the source code, the compiler replaces the function with a set of automatically-generated instructions. It is best practice to use these intrinsics to enable the compiler to optimize the code most efficiently.

Note: The intrinsics listed in the following tables are specific to Fortran 90/95, unless explicitly stated.

6.2 Bit manipulation functions and subroutines

Functions and subroutines for manipulating bits.

Table 1: Bit manipulation functions and subroutines

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`AND`	Perform a logical AND on corresponding bits of the arguments.	2	Any, except CHAR or COMPLEX	INTEGER or LOGICAL
`BIT_SIZE`	Return the number of bits (the precision) of the integer argument.	1	INTEGER	INTEGER
`BTEST`	Test the binary value of a bit in a specified position of an integer argument.	2	INTEGER, INTEGER	LOGICAL
`IAND`	Perform a bit-by-bit logical AND on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER
`IBCLR`	Clear one bit to zero.	2	INTEGER, INTEGER >=0	INTEGER
`IBITS`	Extract a sequence of bits.	3	INTEGER, INTEGER >=0, INTEGER >=0	INTEGER
`IBSET`	Set one bit to one.	2	INTEGER, INTEGER >=0	INTEGER
`IEOR`	Perform a bit-by-bit logical exclusive OR on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER

Continued on next page

Table 1 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`IOR`	Perform a bit-by-bit logical OR on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER
`ISHFT`	Perform a logical shift.	2	INTEGER, INTEGER	INTEGER
`ISHFTC`	Perform a circular shift of the rightmost bits.	2 or 3	INTEGER, INTEGER or INTEGER, INTEGER, INTEGER	INTEGER
`LSHIFT`	Perform a logical shift to the left.	2	INTEGER, INTEGER	INTEGER
`MVBITS`	Copy bit sequence.	5	INTEGER(IN), INTEGER(IN), INTEGER(IN), INTEGER(IN, OUT), INTEGER(IN)	N/A
`NOT`	Perform a bit-by-bit logical complement on the argument.	2	INTEGER	INTEGER
`OR`	Perform a logical OR on each bit of the arguments.	2	Any except CHAR or COMPLEX	INTEGER or LOGICAL
`POPCNT`	Return the number of one bits. (F2008)	1	INTEGER or bits	INTEGER
`POPPAR`	Return the bitwise parity. (F2008)	1	INTEGER or bits	INTEGER
`RSHIFT`	Perform a logical shift to the right.	2	INTEGER, INTEGER	INTEGER
`SHIFT`	Perform a logical shift.	2	Any except CHAR or COMPLEX, INTEGER	INTEGER or LOGICAL
`XOR`	Perform a logical exclusive OR on each bit of the arguments.	2	INTEGER, INTEGER	INTEGER
`ZEXT`	Zero-extend the argument.	1	INTEGER or LOGICAL	INTEGER

6.3 Elemental character and logical functions

Elemental character logical conversion functions.

Table 2: Elemental character and logical functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`ACHAR`	Return character in specified ASCII collating position.	1	INTEGER	CHARACTER
`ADJUSTL`	Left adjust string.	1	CHARACTER	CHARACTER
`ADJUSTR`	Right adjust string.	1	CHARACTER	CHARACTER
`CHAR`	Return character with specified ASCII value.	1	LOGICAL*1 INTEGER	CHARACTER CHARACTER
`IACHAR`	Return position of character in ASCII collating sequence.	1	CHARACTER	INTEGER
`ICHAR`	Return position of character in the character set's collating sequence.	1	CHARACTER	INTEGER

Continued on next page

Table 2 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
INDEX	Return starting position of substring within first string.	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER INTEGER
LEN	Return the length of string.	1	CHARACTER	INTEGER
LEN_TRIM	Return the length of the supplied string minus the number of trailing blanks.	1	CHARACTER	INTEGER
LGE	Test the supplied strings to determine if the first string is lexically greater than or equal to the second. **Note:** From F2008, character kind ASCII is also supported.	2	CHARACTER, CHARACTER	LOGICAL
LGT	Test the supplied strings to determine if the first string is lexically greater than the second. **Note:** From F2008, character kind ASCII is also supported.	2	CHARACTER, CHARACTER	LOGICAL
LLE	Test the supplied strings to determine if the first string is lexically less than or equal to the second. **Note:** From F2008, character kind ASCII is also supported.	2	CHARACTER, CHARACTER	LOGICAL
LLT	Test the supplied strings to determine if the first string is lexically less than the second. **Note:** From F2008, character kind ASCII is also supported.	2	CHARACTER, CHARACTER	LOGICAL
LOGICAL	Logical conversion.	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL

Continued on next page

Table 2 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
SCAN	Scan string for characters in set.	2	CHARACTER, CHARACTER	INTEGER
		3	CHARACTER, CHARACTER, LOGICAL	INTEGER
VERIFY	Determine if string contains all characters in set.	2	CHARACTER, CHARACTER	INTEGER
		3	CHARACTER, CHARACTER, LOGICAL	INTEGER

6.4 Vector/Matrix functions

Functions for vector or matric multiplication.

Table 3: Vector and matrix functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
DOT_PRODUCT	Perform dot product on two vectors.	2	INTEGER, REAL, COMPLEX, or LOGICAL	INTEGER, REAL, COMPLEX, or LOGICAL
MATMUL	Perform matrix multiply on two matrices.	2	INTEGER, REAL, COMPLEX, or LOGICAL	INTEGER, REAL, COMPLEX, or LOGICAL

Note: All matrix outputs are the same type as the argument supplied.

6.5 Array reduction functions

Functions for determining information from, or calculating using, the elements in an array.

Table 4: Array reduction functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
ALL	Determine if all array values are true.	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
ANY	Determine if any array value is true.	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
COUNT	Count true values in array.	1 2	LOGICAL LOGICAL, INTEGER	INTEGER INTEGER
MAXLOC	Determine the position of the array element with the maximum value.	1 2 2 3 1 2 2 3	INTEGER INTEGER, LOGICAL INTEGER, INTEGER INTEGER, INTEGER, LOGICAL REAL REAL, LOGICAL REAL, INTEGER REAL, INTEGER , LOGICAL	INTEGER INTEGER INTEGER INTEGER REAL REAL REAL REAL
MAXVAL	Determine the maximum value of the array elements.	1 2 2 3 1 2 2 3	INTEGER INTEGER, LOGICAL INTEGER, INTEGER INTEGER, INTEGER, LOGICAL REAL REAL, LOGICAL REAL, INTEGER REAL, INTEGER , LOGICAL	INTEGER INTEGER INTEGER INTEGER REAL REAL REAL REAL

Continued on next page

Table 4 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
MINLOC	Determine the position of the array element with the minimum value.	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	INTEGER
		2	INTEGER, INTEGER	INTEGER
		3	INTEGER, INTEGER, LOGICAL	INTEGER
		1	REAL	REAL
		2	REAL, LOGICAL	REAL
		2	REAL, INTEGER	REAL
		3	REAL, INTEGER , LOGICAL	REAL
MINVAL	Determine the minimum value of the array elements.	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	INTEGER
		2	INTEGER, INTEGER	INTEGER
		3	INTEGER, INTEGER, LOGICAL	INTEGER
		1	REAL	REAL
		2	REAL, LOGICAL	REAL
		2	REAL, INTEGER	REAL
		3	REAL, INTEGER , LOGICAL	REAL
PRODUCT	Calculate the product of the elements of an array.	1	NUMERIC	NUMERIC
		2	NUMERIC, LOGICAL	NUMERIC
		2	NUMERIC, INTEGER	NUMERIC
		3	NUMERIC, INTEGER, LOGICAL	NUMERIC

Continued on next page

Table 4 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
SUM	Calculate the sum of the elements of an array.	1 2 2 3	NUMERIC NUMERIC, LOGICAL NUMERIC, INTEGER NUMERIC, INTEGER, LOGICAL	NUMERIC NUMERIC NUMERIC NUMERIC

6.6 String construction functions

Functions for constructing strings.

Table 5: String construction functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
REPEAT	Concatenate copies of a string.	2	CHARACTER, INTEGER	CHARACTER
TRIM	Remove trailing blanks from a string.	1	CHARACTER	CHARACTER

6.7 Array construction manipulation functions

Functions for constructing and manipulating arrays.

Table 6: Array construction and manipulation functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
CSHIFT	Perform circular shift on an array.	2 3	ARRAY, INTEGER ARRAY, INTEGER, INTEGER	ARRAY ARRAY

Continued on next page

Table 6 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
OESHIFT	Perform end-off shift on an array.	2	ARRAY, INTEGER	ARRAY
		3	ARRAY, INTEGER, Any	ARRAY
		3	ARRAY, INTEGER, INTEGER	ARRAY
		4	ARRAY, INTEGER, Any, INTEGER	ARRAY, ARRAY
MERGE	Merge two arguments based on the logical mask.	3	Any, Any, LOGICAL The second argument must be of the same type as the first argument.	Any
PACK	Pack an array into a rank-one array.	2	ARRAY, LOGICAL	ARRAY
		3	ARRAY, LOGICAL, VECTOR	ARRAY
RESHIFT	Change the shape of an array.	2	ARRAY, INTEGER	ARRAY
		3	ARRAY, INTEGER, ARRAY	ARRAY
		3	ARRAY, INTEGER, INTEGER	ARRAY
		4	ARRAY, INTEGER, ARRAY, INTEGER	ARRAY
SPREAD	Replicate an array by adding a dimension.	3	Any, INTEGER, INTEGER	ARRAY
TRANSPOSE	Transpose an array of rank two.	1	ARRAY (m, n)	ARRAY (n, m)
UNPACK	Unpack a rank-one array into an array of multiple dimensions.	3	VECTOR, LOGICAL, ARRAY	ARRAY

Note: All ARRAY outputs are the same type as the argument supplied.

6.8 General inquiry functions

Functions for general determining.

Table 7: General inquiry functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
ASSOCIATED	Determine association status.	12	POINTER, POINTER, . . . , POINTER, TARGET	LOGICAL LOGICAL
KIND	Determine the kind of an argument.	1	Any intrinsic type	INTEGER
PRESENT	Determine presence of optional argument.	1	Any	LOGICAL

6.9 Numeric inquiry functions

Functions for determining numeric information.

Table 8: Numeric inquiry functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
DIGITS	Determine the number of significant digits.	1 1	INTEGER REAL	INTEGER
EPSILON	Smallest number that can be represented.	1	REAL	REAL
HUGE	Largest number that can be represented.	1 1	INTEGER REAL	INTEGER REAL
MAXEXPONENT	Value of the maximum exponent.	1	REAL	INTEGER
MINEXPONENT	Value of the minimum exponent.	1	REAL	INTEGER
PRECISION	Decimal precision.	1 1	REAL COMPLEX	INTEGER INTEGER
RADIX	Base of the model.	1 1	INTEGER REAL	INTEGER INTEGER
RANGE	Decimal exponent range.	1 1 1	INTEGER REAL COMPLEX	INTEGER INTEGER INTEGER

Continued on next page

Table 8 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
SELECTED_INT_KIND	Kind-type titlemeter in range.	1	INTEGER	INTEGER
SELECTED_REAL_KIND	Kind-type titlemeter in range. **Syntax:** SELECTED_REAL_KIND(P [,R]) where P is precision and R is the range.	1 2	INTEGER INTEGER, INTEGER	INTEGER INTEGER
TINY	Smallest positive number that can be represented.	1	REAL	REAL

6.10 Array inquiry functions

Functions for determining information about an array.

Table 9: Array inquiry functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
ALLOCATED	Determine if an array is allocated.	1	ARRAY	LOGICAL
LBOUND	Determine the lower bounds.	1 2	ARRAY ARRAY, INTEGER	INTEGER
SHAPE	Determine the shape.	1	Any	INTEGER
SIZE	Determine the number of elements.	1 2	ARRAY ARRAY, INTEGER	INTEGER
UBOUND	Determine the upper bounds.	1 2	ARRAY ARRAY, INTEGER	INTEGER

6.11 Transfer functions

Functions for transferring types.

Table 10: Transfer functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
TRANSFER	Change the type but maintain bit representation.	2 3	Any, Any Any, Any, INTEGER	Any*

*Must be of the same type as the second argument

6.12 Arithmetic functions

Functions for manipulating arithmetic.

Table 11: Arithmetic functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
ABS	Return absolute value of the supplied argument.	1	INTEGER, REAL, or COMPLEX	INTEGER, REAL, or COMPLEX
ACOS	Return the arccosine (in radians) of the specified value.	1	REAL	REAL
ACOSD	Return the arccosine (in degrees) of the specified value.	1	REAL	REAL
AIMAG	Return the value of the imaginary part of a complex number.	1	COMPLEX	REAL
AINT	Truncate the supplied value to a whole number.	2	REAL, INTEGER	REAL
AND	Perform a logical AND on corresponding bits of the arguments.	2	Any, except CHAR or COMPLEX	INTEGER or LOGICAL
ANINT	Return the nearest whole number to the supplied argument.	2	REAL, INTEGER	REAL
ASIN	Return the arcsine (in radians) of the specified value.	1	REAL	REAL
ASIND	Return the arcsine (in degrees) of the specified value.	1	REAL	REAL
ATAN	Return the arctangent (in radians) of the specified value.	1	REAL	REAL
ATAN2	Return the arctangent (in radians) of the specified pair of values.	2	REAL, REAL	REAL
ATAN2D	Return the arctangent (in degrees) of the specified pair of values.	1	REAL, REAL	REAL
ATAND	Return the arctangent (in degrees) of the specified value.	1	REAL	REAL
CEILING	Return the least integer greater than or equal to the supplied real argument.	2	REAL, KIND	INTEGER

Continued on next page

Table 11 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
CMPLX	Convert the supplied argument or arguments to complex type.	2 3	{INTEGER, REAL, or COMPLEX,}, {INTEGER, REAL, or COMPLEX} {INTEGER, REAL, or COMPLEX}, {INTEGER or REAL}, KIND	COMPLEX COMPLEX
COMPL	Perform a logical complement on the argument.	1	Any, except CHAR or COMPLEX	N/A
COS	Return the cosine (in radians) of the specified value.	1	REAL COMPLEX	REAL
COSD	Return the cosine (in degrees) of the specified value.	1	REAL COMPLEX	REAL
COSH	Return the hyperbolic cosine of the specified value.	1	REAL	REAL
DBLE	Convert to double precision real.	1	INTEGER, REAL, or COMPLEX	REAL
DCMPLX	Convert the argument or supplied arguments to double complex type.	1 2	INTEGER, REAL, or COMPLEX INTEGER, REAL	DOUBLE COMPLEX DOUBLE COMPLEX
DPROD	Double precision real product.	2	REAL, REAL	REAL (double precision)
EQV	Perform a logical exclusive NOR on the arguments.	2	Any, except CHAR or COMPLEX	INTEGER or LOGICAL
EXP	Exponential function.	1	REAL COMPLEX	REAL COMPLEX
EXPONENT	Return the exponent part of a real number.	1	REAL	INTEGER
FLOOR	Return the greatest integer less than or equal to the supplied real argument.	1 2	REAL REAL, KIND	REAL KIND
FRACTION	Return the fractional part of a real number.	1	REAL	INTEGER
IINT	Convert a value to a short integer type.	1	INTEGER, REAL, or COMPLEX	INTEGER
ININT	Return the nearest short integer to the real argument.	1	REAL	INTEGER

Continued on next page

Table 11 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
INT	Convert a value to integer type.	1 2	INTEGER, REAL, or COMPLEX {INTEGER, REAL, or COMPLEX}, KIND	INTEGER INTEGER
INT8	Convert a real value to a long integer type.	1	REAL	INTEGER
IZEXT	Zero-extend the argument.	1	LOGICAL or INTEGER	INTEGER
JINT	Convert a value to an integer type.	1	INTEGER, REAL, or COMPLEX	INTEGER
JNINT	Return the nearest integer to the real argument.	1	REAL	INTEGER
KNINT	Return the nearest integer to the real argument.	1	REAL	INTEGER (long)
LOG	Return the natural logarithm.	1	REAL or COMPLEX	REAL
LOG10	Return the common logarithm.	1	REAL	REAL
MAX	Return the maximum value of the supplied arguments.	2 or more	INTEGER or REAL (all of same kind)	Same as argument type
MIN	Return the minimum value of the supplied arguments.	2 or more	INTEGER or REAL (all of same kind)	Same as argument type
MOD	Find the remainder.	2 or more	{INTEGER or REAL}, {INTEGER or REAL} (all of same kind)	Same as argument type
MODULO	Return the modulo value of the arguments.	2 or more	{INTEGER or REAL}, {INTEGER or REAL} (all of same kind)	Same as argument type
NEAREST	Return the nearest different number that can be represented, by a machine, in a given direction.	2	REAL, REAL (non-zero)	REAL
NEQV	Perform a logical exclusive OR on the arguments.	2	Any, except CHAR or COMPLEX	INTEGER or LOGICAL
NINT	Convert a value to integer type.	1 2	REAL REAL, KIND	INTEGER
REAL	Convert the argument to real.	1 2	INTEGER, REAL, or COMPLEX {INTEGER, REAL, or COMPLEX}, KIND	REAL REAL
RRSPACING	Return the reciprocal of the relative spacing of model numbers near the argument value.	1	REAL	REAL

Continued on next page

Table 11 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
SET_ EXPONENT	Return the model number whose fractional part is the fractional part of the model representation of the first argument and whose exponent part is the second argument.	2	REAL, INTEGER	REAL
SIGN	Return the absolute value of A times the sign of B. Syntax: SIGN(A, B)	2	{INTEGER or REAL}, {INTEGER or REAL}	Same as argument
SIN	Return the sine (in radians) of the specified value.	1	REAL COMPLEX	REAL
SIND	Return the sine (in degrees) of the specified value.	1	REAL COMPLEX	REAL
SINH	Return the hyperbolic sine of the specified value.	1	REAL	REAL
SPACING	Return the relative spacing of model numbers near the argument value.	1	REAL	REAL
SQRT	Return the square root of the argument.	1	REAL COMPLEX	REAL COMPLEX
TAN	Return the tangent (in radians) of the specified value.	1	REAL	REAL
TAND	Return the tangent (in degrees) of the specified value.	1	REAL	REAL
TANH	Return the hyperbolic tangent of the specified value.	1	REAL	REAL

6.13 Miscellaneous functions

Functions for mixcellaneous use.

Table 12: Miscellaneous functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
LOC	Return the argument address.	1	NUMERIC	INTEGER
NULL	Assign a disassociated status.	0 1	 POINTER	POINTER POINTER

6.14 Subroutines

Supported subroutines.

Table 13: Subroutines

Intrinsic	Description	Num. of Arguments	Argument Type
`CPU_TIME`	Return processor time.	1	REAL (OUT)
`DATE_AND_TIME`	Return the date and time.	4 (all optional)	DATE (CHARACTER, OUT) TIME (CHARACTER, OUT) ZONE (CHARACTER, OUT) VALUES (INTEGER, OUT)
`RANDOM_NUMBER`	Generate pseudo-random numbers.	1	REAL (OUT)
`RANDOM_SEED`	Set or query pseudo-random number generator.	0 1 1 1	SIZE (INTEGER, OUT) PUT (INTEGER ARRAY, IN) GET (INTEGER ARRAY, OUT)
`SYSTEM_CLOCK`	Query the real time clock.	3 (optional)	COUNT (INTEGER, OUT) COUNT_RATE (REAL, OUT) COUNT_MAX (INTEGER, OUT)

6.15 Fortran 2003 functions

Fortran 2003-supported functions.

Table 14: Fortran 2003 functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
COMMAND _ARGUMENT _COUNT	Return a scalar of type default integer that is equal to the number of arguments passed on the command line when the containing program was invoked. If there were no command arguments passed, the result is 0.	0	None	INTEGER
EXTENDS_TYPE _OF	Determine whether the dynamic type of A is an extension type of the dynamic type of B. **Syntax:** `EXTENDS_TYPE _OF(A, B)`	2	Objects of extensible type	LOGICAL SCALAR
GET_COMMAND _ARGUMENT	Return the specified command line argument of the command that invoked the program.	1 to 4	INTEGER plus optionally: CHAR, INTEGER, INTEGER	A command argument
GET_COMMAND	Return the entire command line that was used to invoke the program.	0 to 3	CHAR, INTEGER, INTEGER	A command line
GET_ENVIRONM ENT_VARIABLE	Return the value of the specified environment variable.	1 to 5	CHAR, CHAR, INTEGER, INTEGER, LOGICAL	Stores the value of NAME in VALUE
IS_IOSTAT _END	Test whether a variable has the value of the I/O status: 'end of file'.	1	INTEGER	LOGICAL
IS_IOSTAT _EOR	Test whether a variable has the value of the I/O status: 'end of record'.	1	INTEGER	LOGICAL
LEADZ	Count the number of leading zero bits.	1	INTEGER or bits	INTEGER
MOVE_ALLOC	Move an allocation from one allocatable object to another.	2	Any type and rank	None
NEW_LINE	Return the newline character.	1	CHARACTER	CHARACTER
SAME_TYPE _AS	Determine whether the dynamic type of A is the same as the dynamic type of B. **Syntax:** `SAME_TYPE_AS (A, B)`	2	Objects of extensible type	LOGICAL SCALAR
SCALE	Return the value A * B where B is the base of the number system in use for A. **Syntax:** ‘‘ SCALE(A, B)‘‘	2	REAL, INTEGER	REAL

6.16 Fortran 2008 functions

Fortran 2008-supported functions.

Table 15: Fortran 2008 functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
ACOSH, ASINH, ``ATANH	Inverse hyperbolic trigonometric functions	1	REAL	REAL
	Bessel function of:			
BESSEL_J0	the first kind of order 0.	1	REAL	REAL
BESSEL_J1	the first kind of order 1.	1	REAL	REAL
BESSEL_JN	the first kind.	2 or 3	{INTEGER, REAL, or INTEGER}, INTEGER, REAL	REAL
BESSEL_Y0	the second kind of order 0.	1	REAL	REAL
BESSEL_Y1	the second kind of order 1.	1	REAL	REAL
BESSEL_YN	the second kind.	2 or 3	{INTEGER, REAL, or INTEGER}, INTEGER, REAL	REAL
C_SIZEOF	Calculates the number of bytes of storage the expression A 'occupies'. **Syntax:** `C_SIZEOF(A)`	1	Any	INTEGER
COMPILER _OPTIONS	Options passed to the compiler.	None	None	STRING
COMPILER _VERSION	Compiler version string.	None	None	CHARACTER
ERF	Error function.	1	REAL	REAL
ERFC	Complementary error function.	1	REAL	REAL
ERFC _SCALED	Exponentially- scaled complementary error function.	1	REAL	REAL
FINDLOC	Finds the location of a specified value in an array. **Syntax:** `FINDLOC(ARRA Y, VALUE, DIM, MASK, KIND, BACK)` Or `FINDLOC(ARRA Y, VALUE, MASK , KIND, BACK)`	3 to 6	ARRAY VALUE, DIM[, MASK, KIND, BACK], Or ARRAY, VALUE[, MASK, KIND, BACK]	INTEGER ARRAY

Continued on next page

Table 15 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
GAMMA	Computes Gamma of A. For positive, integer values of X.	1	REAL (not zero or negative)	REAL
LOG_GAMMA	Computes the natural logarithm of the absolute value of the Gamma function.	1	REAL (not zero or negative)	REAL
HYPOT	Euclidean distance function.	2	REAL, REAL	REAL
IS _CONTIGUOUS	Tests the contiguity of an array.	1	ARRAY	LOGICAL
LEADZ	Returns the number of leading zero bits of an integer.	1	INTEGER	INTEGER
POPCNT	Return the number of one bits.	1	INTEGER	INTEGER
POPPAR	Return the bitwise parity.	1	INTEGER	INTEGER
SELECTED_REAL_KIND	Kind type titlemeter in range. **Syntax:** SELECTED_REAL_KIND(P[, R, RADIX]) where P is precision and R is the range. **Note:** Radix argument added for F2008.	1 2 3	INTEGER INTEGER, INTEGER INTEGER, INTEGER, INTEGER	INTEGER INTEGER INTEGER
STORAGE_SIZE	Storage size of argument A, in bits. **Syntax:** STORAGE_SIZE(A[, KIND])	1[, 2]	SCALAR or ARRAY[, INTEGER]	INTEGER
TRAILZ	Number of trailing zero bits of an integer	1	INTEGER	INTEGER

6.17 Unsupported functions

Unsupported Fortran 2008 functions:

Table 16: Unsupported functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
ACOSH ASINH ATANH	Inverse hyperbolic trigonometric fucntions.	1	COMPLEX	COMPLEX

Continued on next page

Table 16 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
BGE BGT BLE BLT	Bitwise greater than or equal to. Bitwise greater than. Bitwise less than or equal to. Bitwise less than.	2 2 2 2	INTEGER, INTEGER INTEGER, INTEGER INTEGER, INTEGER INTEGER, INTEGER	LOGICAL LOGICAL LOGICAL LOGICAL
DSHIFTL DSHIFTR	Combined left shift. Combined right shift.	3 3	INTEGER or BOZ constant, INTEGER or BOZ constant, INTEGER INTEGER or BOZ constant, INTEGER or BOZ constant, INTEGER	INTEGER INTEGER
IALL IANY IPARITY	Bitwise AND of array elements. Bitwise OR of array elements. Bitwise XOR of array elements. **Syntax:** ‘‘ INTRINSIC(ARRAY[, DIM[, MASK]])‘‘	1 1 1	ARRAY ARRAY ARRAY	ARRAY ARRAY ARRAY
IMAGE_INDEX NUM_IMAGES THIS_IMAGE	Co-subscript to image index conversion. Number of images. Co-subscript index of this image.	2 0, 1, or 2 0, 1, or 2	COARRAY, INTEGER None, INTEGER, or INTEGER, LOGICAL None, INTEGER, INTEGER or COARRAY, INTEGER	INTEGER INTEGER INTEGER

Continued on next page

Table 16 – continued from previous page

Intrinsic	Description	Num. of Arguments	Argument Type	Result
LCOBOUND	Lower co-dimension of bounds of an array.	1	COARRAY	INTEGER
UCOBOUND	Upper co-dimension of bounds of an array. **Syntax:** ‘‘ INTRINSIC(COARRAY[, DIM[, KIND]])‘‘	1	COARRAY	INTEGER
MASKL	Left justified mask.	1[, or 2]	INTEGER[, INTEGER]	INTEGER
MASKR	Right justified mask. **Syntax:** `INTRINSIC(I[, KIND])`	1[, or 2]	INTEGER[, INTEGER]	INTEGER
MERGE_BITS	Merge of bits under mask.	3	INTEGER, INTEGER, INTEGER	INTEGER
NORM2	Euclidean vector norm. **Syntax:** `NORM2(ARRAY[, DIM])`	1[, or 2]	REAL ARRAY[, INTEGER]	ARRAY
PARITY	Reduction with exclusive OR. **Syntax:** `PARITY(MASK[, DIM])`	1[, or 2]	LOGICAL ARRAY[,INTEGER]	LOGICAL
SHIFTA	Right shift with fill.	2	INTEGER, INTEGER	INTEGER
SHIFTL	Left shift.	2	INTEGER, INTEGER	INTEGER
SHIFTR	Right shift.	2	INTEGER, INTEGER	INTEGER

6.18 Unsupported subroutines

Unsupported Fortran 2008 subroutines:

Table 17: Unsupported subroutines

Intrinsic	Description	Num. of Arguments	Argument Type
`ATOMIC_DEFINE`	Defines the variable ATOM with the value VALUE atomically. **Syntax:** `ATOMIC_DEFINE(ATOM, VALUE[, STAT])`	2[, or 3]	{INTEGER or LOGICAL}, {INTEGER or LOGICAL}[, INTEGER]
`ATOMIC_REF`	Atomically assigns the value of the variable ATOM to VALUE. **Syntax:** `ATOMIC_REF(ATOM, VALUE[, STAT ])`	2[, or 3]	{INTEGER or LOGICAL}, {INTEGER or LOGICAL}[, INTEGER]
`EXECUTE_COMMAND _LINE`	Execute a shell command. **Syntax:** `EXECUTE_COMMAND_ LINE(COMMAND[, WAIT, EXITSTAT, CMDSTAT, CMDMSG])`	1	STRING

CHAPTER

SEVEN

DIRECTIVES

Directives are used to provide additional information to the compiler, and to control the compilation of specific code blocks, for example, loops.

Specify compiler directives as markers in your source file.

Note: To enable OpenMP directives, you must also use the `-fopenmp` compiler option. For more information on supported OpenMP directives, see *Standards support*. For more information on the `-fopenmp` compiler options, see *Actions*.

Directives supported by Arm Fortran Compiler:

7.1 ivdep

Apply this general-purpose directive to a loop to force the vectorizer to ignore memory dependencies of iterative loops, and proceed with the vectorization.

Syntax

Command-line option:

None

Code:

```
!dir$ ivdep
  <loops>
```

Parameters

None

Example: Using ivdep

Example usage of the `ivdep` directive.

Code example:

```
subroutine sum(myarr1,myarr2,ub)
  integer, pointer :: myarr1(:)
  integer, pointer :: myarr2(:)
  integer :: ub

  !dir$ ivdep
  do i=1,ub
    myarr1(i) = myarr1(i)+myarr2(i)
  end do
end subroutine
```

Command-line invocation:

```
armflang -O3 <test>.f90  -S -Rpass-missed=loop-vectorize
-Rpass=loop-vectorize
```

Outputs:

1. With the pragma, the loop given below says the following:

```
remark vectorized loop (vectorization width: 2, interleaved
count: 1) [-Rpass=loop-vectorize]
```

2. Without the pragma, the loop given below says the following:

```
```

remark: loop not vectorized [-Rpass-missed=loop-vectorize]

7.2 vector always

Apply this directive to force vectorization of a loop. The directive tells the vectorizer to ignore any potential cost-based implications.

Note: The loop needs to be able to be vectorized.

7.2.1 Syntax

Command-line option:

None

Code:

```
!dir$ vector always
  <loops>
```

7.2.2 Parameters

None

7.2.3 Example: Using vector always

Example usage of the `vector always` directive.

Code example:

```
subroutine add(a,b,c,d,e,ub)
  implicit none
  integer :: i, ub
  integer, dimension(:) :: a, b, c, d, e

  !dir$ vector always
  do i=1, ub
    e(i) = a(c(i)) + b(d(i))
  end do
end subroutine add
```

Command-line invocation:

```
armflang -O3 <test>.f90  -S -Rpass-missed=loop-vectorize -Rpass=loop-vectorize
```

7.2.4 Outputs

- With the pragma, the output for the example is:

```
remark: vectorized loop (vectorization width: 4, interleaved
count: 1) [-Rpass=loop-vectorize]
```

- Without the pragma, the output for the example is:

```
remark: the cost-model indicates that vectorization is not beneficial [-Rpass-
↪missed=loop-vectorize]
```

7.2.5 Related information

- *Optimization remarks*

7.3 novector

Apply this directive to disable vectorization of a loop.

Note: Use this directive when vectorization would cause a regression instead of an improvement.

7.3.1 Syntax

Command-line option:

None

Code:

```
!dir$ novector
  <loops>
```

7.3.2 Parameters

None

7.3.3 Example: Using novector

Example usage of the `novector` directive.

Code example:

```
subroutine add(arr1,arr2,arr3,ub)
  integer :: arr1(ub), arr2(ub), arr3(ub)
  integer :: i

  !dir$ novector
```

(continues on next page)

(continued from previous page)

```
  do i=1,ub
    arr1(i) = arr1(i) + arr2(i)
  end do
end subroutine add
```

Command-line invocation:

```
armflang -O3 <test>.f90  -S -Rpass-missed=loop-vectorize -Rpass=loop-vectorize
```

7.3.4 Outputs

- With the pragma, the output for the example is:

```
remark: loop not vectorized [-Rpass-missed=loop-vectorize]
```

- Without the pragma, the output for the example is:

```
remark: vectorized loop (vectorization width: 4, interleaved count: 2)
[-Rpass=loop-vectorize]
```

7.3.5 Related information

- *Optimization remarks*

7.4 omp simd

Apply this OpenMP directive to a loop to indicate that the loop can be transformed into a SIMD loop.

Syntax

Command-line option:

`-fopenmp`

Code:

```
!$omp simd
  <do-loops>
```

Parameters

None

Example: Using omp simd

Example usage of the `omp simd` directive.

Code example:

```
subroutine sum(myarr1,myarr2,myarr3,myarr4,myarr5,ub)
  integer, pointer :: myarr1(:)
  integer, pointer :: myarr2(:)
  integer, pointer :: myarr3(:)
  integer, pointer :: myarr4(:)
  integer, pointer :: myarr5(:)
  integer :: ub

  !$omp simd
```

(continues on next page)

(continued from previous page)

```
  do i=1,ub
    myarr1(i) = myarr2(myarr4(i))+myarr3(myarr5(i))
  end do
end subroutine
```

Command-line invocation:

```
.. code-block:: Shell
```

armflang -O3 -fopenmp <test>.f90 -S -Rpass-missed=loop-vectorize -Rpass=loop-vectorize

Outputs:

1. With the pragma, the loop given below says the following:

   ```
   .. code-block:: Shell
   ```

   remark vectorized loop (vectorization width: 2, interleaved count: 1) [-Rpass=loop-vectorize]

2. Without the pragma, the loop given below says the following:

   ```
   .. code-block:: Shell
   ```

   remark: loop not vectorized [-Rpass-missed=loop-vectorize]

7.5 unroll

Instructs the compiler optimizer to unroll a DO loop when optimization is enabled with the compiler optimization flags `-O2` or higher.

7.5.1 Syntax

Command-line option:

None

Code:

```
!dir$ unroll
  <loops>
```

7.5.2 Parameters

None

7.5.3 Example: Using unroll

Example usage of the `unroll` directive.

Code example:

```
subroutine add(a,b,c,d)
 integer, parameter :: m = 1000
 integer :: a(m), b(m), c(m), d(m)
 integer :: i
```

(continues on next page)

(continued from previous page)

```
 !DIR$ UNROLL
   do i =1, m
     b(i) = a(i) + 1
     d(i) = c(i) + 1
   end do
end subroutine add
```

7.5.4 Related information

- *nounroll*
- *Optimization remarks*
- *Optimization options*

7.6 nounroll

Prevents the unrolling of DO loops when optimization is enabled with the compiler optimization flags `-O2` or higher.

7.6.1 Syntax

Command-line option:

None

Code:

```
!dir$ nounroll
  <loops>
```

7.6.2 Parameters

None

7.6.3 Example: Using nounroll

Example usage of the `nounroll` directive.

Code example:

```
subroutine add(a,b,c,d)
 integer, parameter :: m = 1000
 integer :: a(m), b(m), c(m), d(m)
 integer :: i

 !DIR$ NOUNROLL
   do i =1, m
     b(i) = a(i) + 1
     d(i) = c(i) + 1
   end do
end subroutine add
```

7.6.4 Related information

- *unroll*
- *Optimization remarks*
- *Optimization options*

CHAPTER

EIGHT

OPTIMIZATION REMARKS

This short tutorial describes how to enable and use optimization remarks with Arm Fortran Compiler.

8.1 Optimization remarks

This short tutorial describes how to enable optimization remarks and pipe the information they provide to an output file.

Optimization remarks provide you with information about the choices made by the compiler. They can be used to see which code has been inlined or can help you understand why a loop has not been vectorized.

By default, Arm Fortran Compiler prints compilation information to stderr. Optimization remarks prints this optimization information to the terminal, or you can choose to pipe them to an output file.

8.1.1 Procedure

1. To enable optimization remarks, choose from following `Rpass` options:

   Table 1: Optimization remarks options

Option	Description
`-Rpass=<regex>`	Information about what the compiler has optimized.
`-Rpass-analysis=<regex>`	Information about what the compiler has analyzed.
`-Rpass-missed=<regex>`	Information about what the compiler failed to optimize.

   For each option, replace `<regex>` with an expression for the type of remarks you wish to view.

   Recommended `<regexp>` queries are:

   - `-Rpass=\(loop-vectorize\|inline\|loop-unroll)`
   - `-Rpass-missed=\(loop-vectorize\|inline\|loop-unroll)`
   - `-Rpass-analysis=\(loop-vectorize\|inline\|loop-unroll)`

   where `loop-vectorize` will filter remarks regarding vectorized loops, `inline` for remarks regarding inlining, and `loop-unroll` for remarks about unrolled loops.

   Note: To search for all remarks, use the expression `.*`. However, use this expression with care because a lot of information can print depending on the size of your code and the level of optimization performed.

2. To generate the required debug information, you must combine the `-Rpass` option with any of the following `-g` flags:

Table 2: Optimization remarks flags

Flag	Description
`-g`	Emits debug information into the binary.
`-gline-tables-only`	Only emits line table debug information into the binary.

3. To compile with optimization remarks enabled, debug information specified, and pipe the information to an output file, pass the selected above options and debug information to `armflang`, and use `> <outputfile>`:

```
armflang -O<level> -Rpass=<option> <example.f90> <debug_information> 2>
↪<output_file.txt>
```

8.1.2 Example: Fortran example using armflang

This example shows you how to enable and pipe optimization remarks for an example program, `example.f90`.

1. Pass `-Rpass` with the regular expression `loop-vectorize` to `armflang`, use:

```
armflang -O3 -Rpass=loop-vectorize example.F90 -gline-tables-only
```

This results in the following example output in the terminal:

```
example.F90:21: vectorized loop (vectorization width: 2,
interleaved count: 1)
  [-Rpass=loop-vectorize]
    do i=1
```

2. Pipe loop vectorization optimization remarks to a file called `vecreport.txt`, use:

```
armflang -O3 -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize
-Rpass-missed=loop-vectorize example.F90 -gline-tables-only
2> vecreport.txt
```

CHAPTER

NINE

STANDARDS SUPPORT

The support status of Arm Fortran Compiler with the Fortran and OpenMP standards.

9.1 Fortran 2003

The following table details the support status with the Fortran 2003 standard.

Table 1: Fortran 2003 support

Fortran 2003 Feature	Support Status
ISO TR 15580 IEEE Arithmetic	Yes
ISO TR 15581 Allocatable Enhancements	
Dummy arrays	Yes
Function results	Yes
Structure components	Yes
Data enhancements and object orientation	
Parameterized derived types	Yes
Procedure pointers	Yes
Finalization	Yes
Procedures bound by name to a type	Yes
The PASS attribute	Yes
Procedures bound to a type as operators	Yes
Type extension	Yes
Overriding a type-bound procedure	Yes
Enumerations	Yes
ASSOCIATE construct	Yes
Polymorphic entities	Yes
SELECT TYPE construct	Yes
Deferred bindings and abstract types	Yes
Allocatable scalars	Yes
Allocatable character length	Yes
Miscellaneous enhancements	Yes
Structure constructor changes	Yes
Generic procedure interfaces with the same name as a type	Yes
The allocate statement	Yes
Source specifier	Yes
Errmsg specifier	Yes
Assignment to an allocatable array	Yes
Transferring an allocation	Yes
More control of access from a module	Yes
Renaming operators on the USE statement	Yes
Pointer assignment	Yes
Pointer INTENT	Yes

Continued on next page

Table 1 – continued from previous page

Fortran 2003 Feature	Support Status
The VOLATILE attribute	Yes One or more issues are observed with this feature.
The IMPORT statement	Yes
Intrinsic modules	Yes
Access to the computing environment	Yes
Support for international character sets	Partial Only `selected_char_kind` is supported.
Lengths of names and statements	
names = 63	Yes
statements = 256	Yes
Binary, octal and hex constants	Yes
Array constructor syntax	Yes
Specification and initialization expressions	Yes A few intrinsics which are not commonly used are not supported.
Complex constants	Yes
Changes to intrinsic functions	Yes
Controlling IEEE underflow	Yes
Another IEEE class value	Yes
I/O enhancements	Yes
Derived type I/O	Yes One or more issues are observed with this feature.
Asynchronous I/O	Yes One or more issues are observed with this feature.
FLUSH statement	Yes
IOMSG= specifier	Yes
Stream access input/output	Yes
ROUND= specifier	Yes Not supported for write.
DECIMAL= specifier	Yes
SIGN= specifier	Yes `processor_defined` does not work for open.
Kind type parameters of integer specifiers	Yes
Recursive input/output	Yes
Intrinsic function for newline character	Yes
Input and output of IEEE exceptional values	Yes Read does not work for NaN(s).
Comma after a P edit descriptor	Yes
Interoperability with	
Interoperability of intrinsic types	Yes
Interoperability with C pointers	Yes
Interoperability of derived types	Yes
Interoperability of variables	Yes
Interoperability of procedures	Yes
Interoperability of global data	Yes

9.2 Fortran 2008

The following table details the support status with the Fortran 2008 standard.

Table 2: Fortran 2008 support

Fortran 2008 feature	Support status
Submodules	Yes
Coarrays	No
Performance enhancements	
do concurrent	Partial The `do concurrent` syntax is accepted. The code generated is serial.
Contiguous attribute	Yes
Data Declaration	
Maximum rank + corank = 15	No
Long integers	Yes
Allocatable components of recursive type	No
Implied-shape array	No
Pointer initialization	No
Data statement restrictions lifted	No
Kind of a forall index	No
Type statement for intrinsic types	No
Declaring type-bound procedures	No
Value attribute is permitted for any nonallocatable nonpointer noncoarray	No
In a pure procedure the intent of an argument need not be specified if it has the value attribute	Yes
Accessing data objects	
Simply contiguous arrays rank remapping to rank>1 target	Yes
Omitting an ALLOCATABLE component in a structure constructor	No
Multiple allocations with SOURCE=	No
Copying the properties of an object in an ALLOCATE statement	Yes
MOLD= specifier for ALLOCATE	Yes
Copying bounds of source array in ALLOCATE	Yes
Polymorphic assignment	No
Accessing real and imaginary parts	Partial Not supported for complex arrays.
Pointer function reference is a variable	No
Elemental dummy argument restrictions lifted	Yes
Input/Output	
Finding a unit when opening a file	Yes
g0 edit descriptor	No
Unlimited format item	No
Recursive I/O	Yes
Execution control	
The BLOCK construct	No
Exit statement	No
Stop code	Yes
ERROR STOP	No
Intrinsic procedures for bit processsing	
Bit sequence comparison	No
Combined shifting	No
Counting bits	Yes
Masking bits	No

Continued on next page

Table 2 – continued from previous page

Fortran 2008 feature	Support status
Shifting bits	No
Merging bits	No
Bit transformational functions	No
Intrinsic procedures and modules	
Storage size	Yes
Optional argument RADIX added to SELECTED REAL	No
Extensions to trigonometric and hyperbolic intrinsics	Partial Complex types are not accepted for acosh, asinh and atanh. Additionally, atan2 cannot be accessed via atan.
Bessel functions	Yes
Error and gamma functions	Yes
Euclidean vector norms	No
Parity	No
Execute command line	No
Optional back argument added to maxloc and minloc	No
Find location in an array	Yes
String comparison	Yes
Constants	Yes
COMPILER_VERSION	Yes
COMPILER_OPTIONS	Yes
Function for C sizeof	Yes
Added optional argument for IEEE_SELECTED_REAL_KIND	No
Programs and procedures	
Save attribute for module and submodule data	Partial One or more issues are observed with this feature.
Empty contains section	Partial Not supported for procedures.
Form of end statement for internal and module procedures	Yes
Internal procedure as an actual argument	Yes
Null pointer or unallocated allocatable as absent dummy arg.	Partial Not supported for null pointer.
Non pointer actual for pointer dummy argument	Yes
Generic resolution by procedureness	No
Generic resolution by pointer vs. allocatable	Yes
Impure elemental procedures	Yes
Entry statement becomes obsolescent	Yes
Source form	
Semicolon at line start	Yes

9.3 OpenMP 4.0

The following table details the support status with the OpenMP 4.0 standard.

Table 3: OpenMP 4.0 support

OpenMP 4.0 Feature	Support
C/C++ Array Sections	N/A
Thread affinity policies	Yes

Continued on next page

Table 3 – continued from previous page

OpenMP 4.0 Feature	Support
"simd" construct	Partial **Note:** No clauses are supported. `!$omp simd` can be used to forge a loop to be vectorized.
"declare simd" construct	No
Device constructs	No
Task dependencies	No
"taskgroup" construct	Yes
User defined reductions	No
Atomic capture swap	Yes
Atomic seq_cst	No
Cancellation	Yes
OMP_DISPLAY_ENV	Yes

9.4 OpenMP 4.5

The following table details the support status with the OpenMP 4.5 standard.

Table 4: OpenMP 4.5 support

OpenMP 4.5 Feature	Support
doacross loop nests with ordered	No
"linear" clause on loop construct	No
"simdlen" clause on simd construct	No
Task priorities	No
"taskloop" construct	Yes
Extensions to device support	No
"if" clause for combined constructs	Yes
"hint" clause for critical construct	No
"source" and "sink" dependence types	No
C++ reference types in data sharing attribute clauses	N/A
Reductions on C/C++ array sections	N/A
"ref", "val", "uval" modifiers for linear clause	No
Thread affinity query functions	Yes
Hints for lock API	Yes

CHAPTER

TEN

FURTHER RESOURCES

This topic describes the Fortran statements supported within Arm Fortran Compiler.

10.1 Further resources

To learn more about Arm Fortran Compiler and other Arm tools, refer to the following information on the Arm Developer website:

- Arm Fortran Compiler
- Installation instructions
- Release history
- Supported platforms
- Porting and tuning
- Packages wiki
- Help and tutorials
- Arm Allinea Studio
- Get software
- Arm HPC tools
- Arm HPC Ecosystem
- Scalable Vector Extension (SVE)
- Contact Arm Support