Arm® Fortran Compiler

Version 20.3

Developer and Reference Guide

101380_2030_01_en

Arm® Fortran Compiler

Developer and Reference Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
1830-00	20 June 2018	Non-Confidential	Document release for Arm Fortran Compiler version 18.3
1840-00	17 July 2018	Non-Confidential	Update for Arm Fortran Compiler version 18.4
1900-00	02 November 2018	Non-Confidential	Update for Arm Fortran Compiler version 19.0
1910-00	08 March 2019	Non-Confidential	Update for Arm Fortran Compiler version 19.1
1920-00	07 June 2019	Non-Confidential	Update for Arm Fortran Compiler version 19.2
1930-00	30 August 2019	Non-Confidential	Update for Arm Fortran Compiler version 19.3
2000-00	29 November 2019	Non-Confidential	Update for Arm Fortran Compiler version 20.0
2010-00	23 April 2020	Non-Confidential	Update for Arm Fortran Compiler version 20.1
2010-01	23 April 2020	Non-Confidential	Documentation update 1 for Arm Fortran Compiler version 20.1
2020-00	25 June 2020	Non-Confidential	Update for Arm Fortran Compiler version 20.2
2030-00	04 September 2020	Non-Confidential	Update for Arm Fortran Compiler version 20.3
2030-01	16 October 2020	Non-Confidential	Documentation update 1 for Arm Fortran Compiler version 20.3

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

developer.arm.com

Contents

Arm® Fortran Compiler Developer and Reference Guide

Chapter 5 Fortran language reference

Chapter 6 Standards support

Chapter 7 Troubleshoot

List of Tables
Arm® Fortran Compiler Developer and Reference Guide

Preface

This preface introduces the *Arm® Fortran Compiler Developer and Reference Guide*.

It contains the following:

- *About this book* on page 10.

About this book

Provides information to help you use the Arm® Fortran Compiler component of Arm Compiler for Linux. Arm Fortran Compiler is an auto-vectorizing, Linux user-space Fortran compiler, tailored for Server and High Performance Computing (HPC) workloads. Arm Fortran Compiler supports popular Fortran and OpenMP standards and is tuned for Armv8-A based processors.

Using this book

This book is organized into the following chapters:

Chapter 1 Get started
This chapter introduces Arm Fortran Compiler (part of Arm Compiler for Linux and Arm Allinea Studio), and describes how to get started with the compiler, and where to find further support.

Chapter 2 Compile and link
This chapter describes the basic functionality of Arm Fortran Compiler, and describes how to compile your Fortran source with `armflang`.

Chapter 3 Optimize
This chapter describes the optimization-specific features supported in Arm Fortran Compiler.

Chapter 4 Compiler options
This chapter describes the options supported by `armflang`.

Chapter 5 Fortran language reference
This chapter can be used as a reference for the Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, and Fortran 2018 language features that are supported by Arm Fortran Compiler.

Chapter 6 Standards support
This chapter describes the support status of Arm Fortran Compiler with the Fortran language and OpenMP standards.

Chapter 7 Troubleshoot
This chapter describes how to diagnose problems when compiling applications using Arm Fortran Compiler.

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

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.

Feedback on content

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

- The title *Arm Fortran Compiler Developer and Reference Guide*.
- The number 101380_2030_01_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

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

Chapter 1
Get started

This chapter introduces Arm Fortran Compiler (part of Arm Compiler for Linux and Arm Allinea Studio), and describes how to get started with the compiler, and where to find further support.

It contains the following sections:

1.1 Arm® Fortran Compiler

Arm Fortran Compiler is a Linux user space Fortran compiler for server and High Performance Computing (HPC) Arm-based platforms. Arm Fortran Compiler is built on the open-source Clang front-end and the LLVM 9-based optimization and code generation back-end.

Arm Fortran Compiler supports modern Fortran (see *Fortran 2003* on page 6-147 and *Fortran 2008* on page 6-150), *OpenMP 4.0* on page 6-153, and *OpenMP 4.5* on page 6-154 standards, has a built-in autovectorizer, and is tuned for the 64-bit Armv8-A architecture. Arm Fortran Compiler also supports compiling for Scalable Vector Extension- (SVE-) and SVE2-enabled.

Arm Fortran Compiler is packaged with Arm C/C++ Compiler and Arm Performance Libraries in a single package called Arm Compiler for Linux. To use Arm Compiler for Linux, you must have a valid license for Arm Allinea Studio. Arm Allinea Studio is an end-to-end commercial suite of tools for developing Linux applications to run on Armv8-A-based targets. For more information about Arm Allinea Studio and how to license the tools, see the *Arm Allinea Studio web page*.

Resources

To learn more about Arm Fortran Compiler (part of Arm Compiler for Linux) and other Arm server and HPC tools, refer to the following information:

- **Arm Allinea Studio**:
  - *Arm Allinea Studio*
  - *Arm Fortran Compiler web page*
  - *Installation instructions*
  - *Release history*
  - *Supported platforms*
- **Porting guidance**:
  - *Porting and tuning resources*
  - *Arm GitLab Packages wiki*
  - *Arm HPC Ecosystem*
- **SVE and SVE2 information**:
  - *Scalable Vector Extension (SVE, and SVE2) information*
  - For an overview of SVE and why it is useful for HPC, see *Explore the Scalable Vector Extension (SVE)*.
  - For a list of SVE and SVE2 instructions, see the *Arm A64 Instruction Set Architecture*.
  - *White Paper: A sneak peek into SVE and VLA programming*. An overview of SVE with information on the new registers, the new instructions, and the Vector Length Agnostic (VLA) programming technique, with some examples.
  - *White Paper: Arm Scalable Vector Extension and application to Machine Learning*. In this white paper, code examples are presented that show how to vectorize some of the core computational kernels that are part of machine learning system. These examples are written with the Vector Length Agnostic (VLA) approach introduced by the Scalable Vector Extension (SVE).
  - *DWARF for the ARM 64-bit Architecture (AArch64) with SVE support*. This document describes the use of the DWARF debug table format in the Application Binary Interface (ABI) for the Arm 64-bit architecture.
  - *Procedure Call Standard for the ARM 64-bit Architecture (AArch64) with SVE support*. This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the Arm 64-bit architecture.
  - *Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A*. This supplement describes the Scalable Vector Extension to the Armv8-A architecture profile.
- **Support and sales**:
  - If you encounter a problem when developing your application and compiling with the Arm Fortran Compiler, see the *Troubleshoot* on page 7-155
  - *Contact Arm Support*
  - *Get software*

Note

An HTML version of this guide is available in the `<install_location>/<package_name>/share` directory of your product installation.

1.2 Get started with Arm® Fortran Compiler

Describes how to download and install Arm Compiler for Linux, and how to use Arm Fortran Compiler to compile Fortran source into an executable binary.

Prerequisites

Download and install Arm Compiler for Linux. You can download Arm Compiler for Linux from the *Arm Allinea Studio Downloads* page. Learn how to install and configure Arm Compiler for Linux, using the *Arm Compiler for Linux installation instructions* on the Arm Developer website.

Procedure

1. Load the environment module for Arm Compiler for Linux:
   a. As part of the installation, Arm recommends that your system administrator makes the Arm Compiler for Linux environment modules available to all users of the tool suite.

      To see which environment modules are available on your system, run:

      ```
      module avail
      ```

      Note

      If you cannot see the Arm Compiler for Linux environment module, but you know the installation location, use `module use` to update your `MODULEPATH` environment variable to include that location:

      ```
      module use <path/to/installation>/modulefiles/
      ```

      replacing `<path/to/installation>` with the path to your installation of Arm Compiler for Linux. The default installation location is `/opt/arm/`.

      `module use` sets your `MODULEPATH` environment variable to include the installation directory:

   b. To load the module for Arm Compiler for Linux, run:

      ```
      module load <architecture>/<linux_variant>/<linux_version>/arm-linux-compiler/
      <version>
      ```

      For example:

      ```
      module load Generic-AArch64/SUSE/12/arm-linux-compiler/20.3
      ```

   c. Check your environment. Examine the `PATH` variable. `PATH` must contain the appropriate `bin` directory from `<path/to/installation>`:

      ```
      echo $PATH
      /opt/arm/arm-linux-compiler-20.2_Generic-AArch64_SUSE-12_aarch64-linux/bin:...
      ```

   Note

   To automatically load the Arm Compiler for Linux every time you log into your Linux terminal, add the `module load` command for your system and product version to your `.profile` file.

2. To generate an executable binary, compile your application with Arm Fortran Compiler.

   Specify the input source filename, `<source>.<fortran-extension>`, and use `-o` to specify the output binary file, `<binary>`:

   ```
   armflang -o <binary> <source>.<fortran-extension>
   ```

Arm Fortran Compiler builds your binary `<binary>`.

To run your binary, use:

```
./<binary>
```

Example 1-1 Example: Compile and run a "Hello World" application

---

This example describes how to write, compile, and run a simple "Hello World" Fortran application.

1. Load the environment module for your system:

   ```
   module load <architecture>/<linux_variant>/<linux_version>/arm-linux-compiler/<version>
   ```

2. Create a "Hello World" application and save it in an `.f90` file, for example: `hello.f90`:

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

3. To generate an executable binary, compile your "Hello World" application with Arm Fortran Compiler.

   Specify the input file, `hello.f90`, and the binary name (using `-o`), `hello`:

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

4. Run the generated binary `hello`:

   ```
   ./hello
   ```

Next Steps

For more information about compiling and linking as separate steps, and how optimization levels effect auto-vectorization, see *Compile and link* on page 2-18.

1.3 Get support

To see a list of all the supported compiler options in your terminal, use:

```
armflang --help
```

or

```
man armflang
```

A description of each supported command-line option is available in *Compiler options* on page 4-49.

If you encounter a problem when developing your application and compiling with the Arm Compiler for Linux, see the *Troubleshoot* on page 7-155 topic.

If you encounter a problem when using Arm Compiler for Linux, contact the Arm Support team:

Contact Arm Support

Chapter 2
Compile and link

This chapter describes the basic functionality of Arm Fortran Compiler, and describes how to compile your Fortran source with `armflang`.

It contains the following sections:

2.1 Using the compiler

Describes how to generate executable binaries, compile and link object files, and enable optimization options, with Arm Fortran Compiler.

Compile and link

To generate an executable binary, compile your source file (for example, `source.f90`) with the `armflang` command:

```
armflang -o source.f90
```

A binary with the filename `source` is output.

Optionally, use the `-o` option to set the binary filename (for example, `binary`):

```
armflang -o binary source.f90
```

You can specify multiple source files on a single line. Each source file is compiled individually and then linked into a single executable binary. For example, to compile the source files `source1.f90` and `source2.f90`, use:

```
armflang -o binary source1.f90 source2.f90
```

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

```
armflang -c source1.f90
armflang -c source2.f90
armflang -o binary source1.o source2.o
```

Increase the optimization level

To control the optimization level, specify the `-O<level>` option on your compile line, and replace `<level>` with one of `0`, `1`, `2`, `3`, or `fast`. `-O0` option is the lowest, and the default, optimization level. `-Ofast` is the highest optimization level. Arm Fortran Compiler performs auto-vectorization at levels `-O2`, `O3`, and `-Ofast`.

For example, to compile `source.f90` into a binary called `binary`, and use the `-O3` optimization level, use:

```
armflang -O3 -o binary source.f90
```

Compile and optimize using CPU auto-detection

If you tell Arm Fortran Compiler what target CPU your application will run on, the compiler can make target-specific optimization decisions. Target-specific optimization decisions help ensure your application runs as efficiently as possible. To tell the compiler to make target-specific compilation decisions, use the `-mcpu=<target>` option and replace `<target>` with your target processor (from a supported list of targets). To see what processors are supported by the `-mcpu` option, see *4.44 -mcpu=* on page 4-98.

In addition, the `-mcpu` option also supports a `native` argument. `-mcpu=native` enables Arm Fortran Compiler to auto-detect the architecture and processor type of the CPU that you are running the compiler on.

For example, to auto-detect the target CPU and optimize the application for this target, use:

```
armflang -O3 -mcpu=native -o binary source.f90
```

The `-mcpu` option supports a range of Armv8-A-based Systems-on-Chip (SoC), including ThunderX2, Neoverse N1, and A64FX. When `-mcpu` is not specified, by default, `-mcpu=generic` is set, which generates portable output suitable for any Armv8-A-based target.

Note

- The optimizations that are performed from setting the `-mcpu` option (also known as hardware, or CPU, tuning) are independent of the optimizations that are performed from setting the `-O<level>` option.
- If you run the compiler on one target, but will run the application you are compiling on a different target, do not use `-mcpu=native`. Instead, use `-mcpu=<target>` where `<target>` is the target processor that you will run the application on.

Common compiler options

This section describes some common options to use with Arm Fortran Compiler.

Note

For more information about all the supported compiler options, run `man armflang`, `armflang --help`, or see *Compiler options* on page 4-49.

`-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 Executable and Linkable Format (ELF) object file (`<file>.o`). 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`, `-march=armv8-a+sve`, or `-march=armv8-a+sve2`.

Note

If you know what your target CPU is, Arm recommends using the `-mcpu` option instead of `-march`. For a complete list of supported targets, see *4.43 -march=* on page 4-97.

`-mcpu=native`

Enables the compiler to automatically detect the CPU you are running the compiler on, and optimize accordingly. The compiler selects a suitable target profile for that CPU. If you use `-mcpu`, you do not need to use the `-march` option.

`-mcpu` supports the following Armv8-A-based Systems-on-Chip (SoCs): ThunderX2, Neoverse N1, and A64FX.

Note

When `-mcpu` is not specified, it defaults to `-mcpu=generic` which generates portable output suitable for any Armv8-A-based target.

For more information, see *4.44 -mcpu=* on page 4-98.

`-O<level>`

Specifies the level of optimization to use when compiling source files. The supported options are: `-O0`, `-O1`, `-O2`, `-O3`, and `-Ofast`. The default is `-O0`. Auto-vectorization is enabled at `-O2`, `-O3`, and `-Ofast`

Warning

`-Ofast` performs aggressive optimizations that might violate strict compliance with language standards.

For more information, see *4.47 -O* on page 4-101.

`--config /path/to/<config-file>`

Passes the location of a configuration file to the compile command. Use a configuration file to specify a set of compile options to be run at compile time. The configuration file can be passed at compile time, or an environment variable can be set for it to be used for every invocation of the compiler. For more information about creating and using a configuration file, see *Configure Arm Compiler for Linux*.

`--help`

Describes the most common options that are supported by Arm Fortran Compiler. To see more detailed descriptions of all the options, use `man armflang`.

`--version`

Displays version information.

For a detailed description of all the supported compiler options, see *Compiler options* on page 4-49.

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

```
man armflang
```

Alternatively, if you use a bash terminal and have the ‘bash-completion’ package installed, you can use ‘command line completion’ (also known as ‘tab completion’). To complete the command or option that you are typing in your terminal, press the **Tab** button on your keyboard. If there are multiple options available to complete the command or option with, the terminal presents these as a list. If an option is specified in full, and you press **Tab**, Arm Compiler for Linux returns the supported arguments to that option.

For more information about how command line completion is enabled for bash terminal users of Arm Compiler for Linux, see the *Arm Allinea Studio installation instructions*.

Related tasks

2.2 Compile Fortran code for Arm SVE and SVE2-enabled processors on page 2-22

Related references

Chapter 4 Compiler options on page 4-49

2.2 Compile Fortran code for Arm SVE and SVE2-enabled processors

This topic describes how to use Arm Fortran Compiler to compile your code for Scalable Vector Extension- (SVE-) and SVE2-enabled target processors.

SVE and SVE2 support enables you to:

- Assemble source code containing SVE and SVE2 instructions.
- Disassemble ELF object files containing SVE and SVE2 instructions.
- Compile Fortran code for SVE and SVE2-enabled targets, with an advanced auto-vectorizer that is capable of taking advantage of the SVE and SVE2 features.

This tutorial shows you how to compile code to take advantage of SVE (or SVE2) functionality. The executable that is generated during the tutorial can only be run on SVE-enabled (or SVE2-enabled) hardware, or with Arm Instruction Emulator.

Prerequisites

- Your target must be SVE- or SVE2-enabled hardware, or you must download, install, and load the correct environment module for Arm Instruction Emulator. For more information about installing and setting up your environment for Arm Instruction Emulator, see *Install Arm Instruction Emulator*.
- Install Arm Compiler for Linux. For information about installing Arm Compiler for Linux, see *Install Arm Compiler for Linux*.
- Load the module for Arm Compiler for Linux. Run:

```
module load <architecture>/<linux_variant>/<linux_version>/arm-linux-compiler/<version>
```

Procedure

1. Compile your SVE or SVE2 source and specify an SVE-enabled (or SVE2-enabled) architecture:
   - To compile without linking to Arm Performance Libraries, set `-march` to the architecture and feature set you want to target:

     For SVE:

     ```
     armflang -O<level> -march=armv8-a+sve -o <binary> <source.f90>
     ```

     For SVE2:

     ```
     armflang -O<level> -march=armv8-a+sve2 -o <binary> <source.f90>
     ```
   - To compile and link to the SVE version of Arm Performance Libraries, set `-march` to the architecture and feature set you want to target and add the `-armpl=sve` option to your command line:

     For SVE:

     ```
     armflang -O<level> -march=armv8-a+sve -armpl=sve -o <binary> <source.f90>
     ```

     For SVE2:

     ```
     armflang -O<level> -march=armv8-a+sve2 -armpl=sve -o <binary> <source.f90>
     ```

     For more information about the supported options for `-armpl`, see the `-armpl` description in *Arm Fortran Compiler Options by Function* on page 4-52.

Note

For all of the preceding command lines, to get the best vectorizations for SVE, set `-O<level>` to be `-O2` or higher.

There are several SVE2 Cryptographic Extensions available: `sve2-aes`, `sve2-bitperm`, `sve2-sha3`, and `sve2-sm4`. Each extension is enabled using the `march` compiler option. For a full list of supported `-march` options, see *Arm Fortran Compiler Options by Function* on page 4-52.

Note

`sve2` also enables `sve`.

2. Run the executable:
   - On SVE(2)-enabled hardware:

     ```
     ./<binary>
     ```
   - Using Arm Instruction Emulator to emulate the SVE(2) instructions:

     ```
     armie -msve-vector-bits=<value> ./<binary>
     ```

     Replace `<value>` with the vector length to use (which must be a multiple of 128 bits up to 2048 bits).

     Note

     For more information about using Arm Instruction Emulator, see the *Arm Instruction Emulator documentation*.

2.3 Generate annotated assembly code

Arm Fortran Compiler can produce annotated assembly code. Generating annotated assembly code is a good first step to see how the compiler vectorizes loops.

Note

To use SVE functionality, you need to use a different set of compiler options. For more information, refer to *Compile Fortran code for Arm SVE and SVE2-enabled processors* on page 2-22.

Prerequisites

- Install Arm Compiler for Linux. For information about installing Arm Compiler for Linux, see *Install Arm Compiler for Linux*.
- Load the module for Arm Compiler for Linux, run:

```
module load <architecture>/<linux_variant>/<linux_version>/arm-linux-compiler/<version>
```

Procedure

1. Compile your source and specify an assembly code output:

```
armflang -S <source>.f90
```

The option `-S` is used to output assembly code.

The compiler outputs a `<source>.s` file.

2. Inspect the `<source>.s` file to see the annotated assembly code that was created.

Related tasks

2.2 Compile Fortran code for Arm SVE and SVE2-enabled processors on page 2-22

Chapter 3
Optimize

This chapter describes the optimization-specific features supported in Arm Fortran Compiler.

It contains the following sections:

3.1 Directives

Directives are used to provide additional information to the compiler, and to control the compilation of specific code blocks, for example, loops. This chapter describes what directives are supported in Arm Fortran Compiler.

To specify a compiler directive in your source file, use:

- For free-form Fortran, use `!dir$` to indicate a directive, or `!$omp` to indicate an OpenMP directive.
- For fixed-form Fortran, either `!dir$` or `cdir$` can be used to indicate a directive, and either `!$omp` or `c$omp` can be used to indicate an OpenMP directive.

  Warning

  Directives using `cdir$` or `c$omp` must start from the first column.

Note

To enable OpenMP directives, you must also include the `-fopenmp` compiler option in the compile command line.

For more information about which OpenMP directives are supported, see *Standards support* on page 6-146. For more information about the `-fopenmp` option, see *4.22 -fopenmp* on page 4-76.

This section contains the following subsections:

- *3.1.1 ivdep* on page 3-26.
- *3.1.2 vector always* on page 3-27.
- *3.1.3 novector* on page 3-28.
- *3.1.4 omp simd* on page 3-29.
- *3.1.5 unroll* on page 3-30.
- *3.1.6 nounroll* on page 3-30.

3.1.1 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

Source:

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

Parameters

None

Example: Using ivdep

Example usage of the `ivdep` directive.

```
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
```

Note

The example uses the free-form syntax. For fixed-form formats, replace `!dir$` with `cdir$`.

Command-line invocation

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

The `-Rpass` and `-Rpass-missed` options enable optimization remarks about vectorized loops to be reported. To learn more about optimization remarks, see *Optimization remarks* on page 3-43.

Outputs

1. With the pragma, the loop in the sum subroutine produces the following remark:

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

2. Without the pragma, the loop in the sum subroutine produces the following remark:

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

3.1.2 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.

Syntax

Command-line option:

None

Source:

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

Parameters

None

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
```

Note

The example uses the free-form syntax. For fixed-form formats, replace `!dir$` with `cdir$`.

Command-line invocation

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

The `-Rpass` and `-Rpass-missed` options enable optimization remarks about vectorized loops to be reported. To learn more about optimization remarks, see *Optimization remarks* on page 3-43.

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]
  ```

Related references

3.4 Optimization remarks on page 3-43

3.1.3 novector

Apply this directive to disable vectorization of a loop.

Note

Use this directive when vectorization would cause a performance regression, instead of a performance improvement.

Syntax

Command-line option:

None

Source:

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

Parameters

None

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

Note

The example uses the free-form syntax. For fixed-form formats, replace `!dir$` with `cdir$`.

Command-line invocation

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

The `-Rpass` and `-Rpass-missed` options enable optimization remarks about vectorized loops to be reported. To learn more about optimization remarks, see *Optimization remarks* on page 3-43.

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]
```

Related references

3.4 Optimization remarks on page 3-43

3.1.4 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`

Source:

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

Parameters

None

Note

Clauses for `omp simd` are not supported.

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

Command-line invocation

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

The `-Rpass` and `-Rpass-missed` options enable optimization remarks about vectorized loops to be reported. To learn more about optimization remarks, see *Optimization remarks* on page 3-43.

Outputs

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

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

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

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

Related references

Chapter 6 Standards support on page 6-146

3.1.5 unroll

Instructs the compiler optimizer to unroll a `DO` loop when optimization is enabled with an optimization level of `-02`, `-O3`, or `-Ofast`.

Syntax

Command-line option:

None

Source:

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

Parameters

None

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

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

Note

The example uses the free-form syntax. For fixed-form formats, replace `!dir$` with `cdir$`.

Related references

3.1.6 nounroll on page 3-30

3.4 Optimization remarks on page 3-43

4.1 Arm Fortran Compiler Options by Function on page 4-52

3.1.6 nounroll

Prevents the compiler optimizer from unrolling a `DO` loop when optimization is enabled with an optimization level of `-02`, `-O3`, or `-Ofast`.

Syntax

Command-line option:

None

Source:

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

Parameters

None

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
```

Note

The example uses the free-form syntax. For fixed-form formats, replace `!dir$` with `cdir$`.

Related references

3.1.5 unroll on page 3-30

3.4 Optimization remarks on page 3-43

4.1 Arm Fortran Compiler Options by Function on page 4-52

Related references

3.1.1 ivdep on page 3-26

3.1.2 vector always on page 3-27

3.1.3 novector on page 3-28

3.1.4 omp simd on page 3-29

3.1.5 unroll on page 3-30

3.1.6 nounroll on page 3-30

3.2 Link Time Optimization (LTO)

This section describes what Link Time Optimization (LTO) is, when LTO is useful, and how to compile with LTO. The section also provides reference information about the `llvm-ar` and `llvm-ranlib` LLVM utilities that are required to compile static libraries with LTO.

This section contains the following subsections:

3.2.1 What is Link Time Optimization (LTO)

Link Time Optimization is a form of interprocedural optimization that is performed at the time of linking application code. Without LTO, Arm Compiler for Linux compiles and optimizes each source file independently of one another, then links them to form the executable. With LTO, Arm Compiler for Linux can process, consume, and use inter-module dependency information from across all the source files to enable further optimizations at link time. LTO is particularly useful when source files that have already been compiled separately.

The following describes the workflow that Arm Compiler for Linux takes with and without LTO enabled, in more detail:

- Without LTO:
  1. Source files are translated into separate ELF object files (`.o`) and passed to the linker.
  2. The linker processes the separate ELF object files, together with library code, to create the ELF executable.
- With LTO:
  1. Source files are translated into a bitcode object files (`.o`), and passed to the linker. LLVM Bitcode is an intermediate form of code that is understood by the optimizer.
  2. To extract the module dependency information, the linker processes the bitcode and object files together and passes them to the LLVM optimizer utility, libLTO.
  3. The LLVM optimizer utility, libLTO, uses the module dependency information to filter out unused modules, and create a single highly optimized ELF object file. Additional optimizations are possible by knowing the module dependency information. The new ELF object file is returned to the linker.
  4. The linker links the new ELF object file with the remaining ELF object files and library code, to generate an ELF executable.

Limitations

LTO in Arm Compiler for Linux has some limitations:

- To compile static libraries, you must create a library archive file that libLTO can use at link time. `armllvm-ar`, as well as some open-source utility tools can create this archive file. For more information about `armllvm-ar`, see *armllvm-ar and reference* on page 3-36.
- Partial linking is not supported with LTO because partial linking only works with ELF objects, rather than bitcode files.
- If your library code calls a function that was defined in the source code, but is removed by libLTO, you might get linking errors.
- Bitcode objects are not guaranteed to be compatible across Arm Compiler for Linux versions. When linking with LTO, ensure that all your bitcode files are built using the same version of the compiler.
- You can not analyze LTO-optimized code using Arm Optimization Reports. Arm Optimization Reports analyzes object files that are generated by Arm Compiler for Linux before they are passed to the linker. Therefore, you can not use Arm Optimization Reports to investigate the vectorization decisions that LTO enables the linker to make.

3.2.2 Compile with Link Time Optimization (LTO)

This topic describes how to compile your Fortran source code with Link Time Optimization (LTO), using Arm Fortran Compiler.

Prerequisites

- Download and install Arm Compiler for Linux. You can download Arm Compiler for Linux from the *Arm Allinea Studio Downloads* page. Learn how to install and configure Arm Compiler for Linux, using the *Arm Compiler for Linux installation instructions* on the Arm Developer website.
- Load the environment module for Arm Compiler for Linux for your system.
- To compile your code with static libraries, you must create an archive of your libraries using an archive utility tool. Arm Compiler for Linux version 20.3+ includes variants of the LLVM archive utility tools `llvm-ar` (`armllvm-ar`) and `llvm-ranlib` (`armllvmran-lib`).

  If you use a Makefile to create the library archive and compile your application, open your Makefile and update any references of `llvm-ar` to `armllvm-ar`, and `llvm-ranlib` to `armllvm-ranlib`.

  Note

  If you use `ar` to create your archives, you must also use the LLVM Gold Plugin to enable `ar` to use LLVM bitcode object files. For more information, see the *LLVM gold plugin documentation*.

  For more information about `armllvm-ar`, see *armllvm-ar and reference* on page 3-36. For more information about `armllvm-ranlib`, see *armllvm-ranlib reference* on page 3-36.

Procedure

1. To generate an executable binary with LTO enabled, compile and link your code with `armflang`, and pass the `-flto` option:
   - For dynamic library compilation, use:

     ```
     armflang -O<level> -flto -o <binary> <sources>
     ```
   - For static library compilation:
     1. Compile, but do not link, your code with LTO:

        ```
        armflang -O<level> -flto -c <sources>
        ```

        The result is one or more `.o` files, one per source file that was passed to `armflang`.
     2. Create the archive file for your static library object files:

        ```
        armllvm-ar [config-options] [operation{modifiers)}] <archive> [<files>]
        armllvm-ranlib <archive>
        ```

        For example:

        ```
        armllvm-ar rc example-archive.a source1.o source2.o
        armllvm-ranlib example-archive.a
        ```

        `armllvm-ar` builds a single archive file from one or more `.o` files. `r` is an operation that instructs `armllvm-ar` to replace existing archive files or, if they are new files, add the files to the end of the archive. `c` is a modifier to `r` that disables the warning which informs you that an archive has been created.

        `armllvm-ranlib` builds an index for the `<archive>` file.

        For a more detailed description of `armllvm-ar`, see *armllvm-ar and reference* on page 3-36. For a more detailed description of `armllvm-ar`, see *armllvm-ranlib reference* on page 3-36.
     3. Link your remaining object files together with your archive file:

        ```
        armflang -O<level> -flto -o <binary> <sources>.o <archive>
        ```

Note

The `<archive>` file is used in place of the object files that where combined into the `<archive>` file by `armllvm-ar`.

2. (Optional) Use a tool like objdump to analyze the binary and view how the compiler optimized your code:

```
objdump -d <binary>
```

Arm Fortran Compiler builds your LTO-optimized binary `<binary>`.

To run your binary, use:

```
./<binary>
```

Example 3-1 Example: Compare code compiled with and without LTO

The following example application code is composed of two source files. `main.f90` contains the `main` function which calls and a second function, `foo`, contained in `foo.f90`. Compiling and analyzing example code without LTO enabled, then with LTO enabled, allows us to see the effect that LTO has on the application compilation.

Note

This example does not use static libraries.

1. Create the example source code files:
   a. Write and save the following code as a `main.f90` source file:

```
PROGRAM main
   IMPLICIT NONE
   REAL, EXTERNAL :: foo
   INTEGER :: i, numelts, numargs
   CHARACTER(len=256) :: filename, elts, progname
   REAL, DIMENSION(:), ALLOCATABLE :: data

   numargs = command_argument_count()

   IF (numargs .NE. 2) THEN
      CALL get_command_argument(0, progname)
      WRITE(*,*) "Incorrect arguments."
      WRITE(*,*) "  Usage: " // &
                 progname(1:len_trim(progname)) // &
                 " <filename> <size>'"
      STOP
   END IF

   CALL get_command_argument(1, filename)
   CALL get_command_argument(2, elts)
   READ(elts, *) numelts

   ALLOCATE(data(numelts))

   OPEN(42, FILE=filename, STATUS='old', &
      ACCESS='stream', FORM='unformatted')

   READ(42) data

   DO i = 1, numelts
      data(i) = foo(data(i))
   END DO

   REWIND(42)
   WRITE(42) data
   CLOSE(42)

   DEALLOCATE(data)
```

```
END PROGRAM
```

   b. Write and save the following code as a `foo.f90` source file:

```
REAL FUNCTION foo(val)
  foo = val*2.0
END FUNCTION
```

2. Use `armflang` to compile the code both without and with LTO enabled:
   a. To compile without LTO, into a binary called `binary-no-lto`, use:

```
armflang -O3 -o binary-no-lto main.f90 foo.f90
```

   b. To compile with LTO, into a binary called `binary-lto`, use:

```
armflang -O3 -flto -o binary-lto main.f90 foo.f90
```

3. To analyze the files to see the effect that LTO has on the generated code, use `objdump` to investigate the `main` function in the binary:

```
objdump -d binary-no-lto
```

In the following pseudo code:

- `{addr*}` represents an address. `{addr_main}`, `{addr_foo}`, and `{addr_loop_start}` are addresses that are given specific pseudo address names for the purpose of this example.
- {enc} represents the encoding.

For `binary-no-lto`, you can see separate functions `main` and `foo` in the following pseudo code:

```
{addr_main} <MAIN_>:
   ...
   {addr_loop_start}:    {enc}      add     x0, x8, x20
   {addr*}:              {enc}      bl      {addr_foo} <foo_>
   {addr*}:              {enc}      ldr     x8, [sp, #680]
   {addr*}:              {enc}      subs    x19, x19, #0x1
   {addr*}:              {enc}      str     s0, [x8, x20]
   {addr*}:              {enc}      add     x20, x20, #0x4
   {addr*}:              {enc}      b.ne    {addr_loop_start}
   ...
...

{addr_foo} <foo_>:
   {addr*}:       {enc}      ldr     s0, [x0]
   {addr*}:       {enc}      fadd    s0, s0, s0
   {addr*}:       {enc}      ret
```

`main` has a scalar loop with a branch to `foo` in it:

```
{addr*}:       {enc}        bl      {addr_foo} <foo_>
```

Whereas in `binary-lto`, you see one `main` function:

```
objdump -d binary-lto
```

Which gives:

```
{addr_main} <MAIN_>:
   ...
   {addr_loop_start}:     {enc}      ldr     q0, [x13], #16
   {addr*}:               {enc}      subs    x12, x12, #0x4
   {addr*}:               {enc}      fadd    v0.4s, v0.4s, v0.4s
   {addr*}:               {enc}      str     q0, [x14]
   {addr*}:               {enc}      mov     x14, x13
   {addr*}:               {enc}      b.ne    {addr_loop_start}
   ...
```

In `main` in `binary-lto`, the simple `foo` function has been inlined and transformed into a vectorized loop: `fadd v0.4s, v0.4s, v0.4s`.

Related references

3.2.3 armllvm-ar and reference on page 3-36

3.2.4 armllvm-ranlib reference on page 3-36

3.2.3 armllvm-ar and reference

This topic describes `armllvm-ar`. `armllvm-ar` is a utility tool provided in the Arm Compiler for Linux package, and is a variant of the LLVM `llvm-ar` utility tool.

`armllvm-ar` is an archiving tool that is similar to the Unix utility `ar`. However, unlike `ar`, `armllvm-ar` is able to understand the LLVM bitcode files that LLVM-based compilers produce when Link Time Optimization (LTO) is enabled.

`armllvm-ar` can archive several `.o` object (or bitcode object) files into a single archive library. As `armllvm-ar` archives the files, the tool creates a symbol table of the files. At link time, you can pass the archive to the compiler to link it into your application. When an archive is used by the compiler at link time, the symbol table enables linking to be performed faster than it would take the linker to link each file separately.

Note

For information about how `llvm-ar` differs from `ar`, see the *llvm-ar LLVM command documentation*.

Syntax

`armllvm-ar` can be run on the command line or through a Machine Readable Instruction (MRI) script. The following syntax is the command line syntax

```
armllvm-ar [config-options] [operation{modifiers)}] <archive> [<files>]
```

Note

`armllvm-ar` inherits the same syntax as `llvm-ar`.

Options for `armllvm-ar` are separated into *Configuration options*, *Operations*, and *Modifiers*:

- Configuration options are options that either configure how `llvm-ar` runs (for example how to set the default archive format), or are options to display help or version information.
- Operations are actions that are performed on an archive. You can only pass one operation to `armllvm-ar`.
- Modifiers control how the operation completes the action. You can specify multiple modifiers to an operation, however, each operation supports different modifiers.

Options, Operations, and Modifiers

`armllvm-ar` supports the same options, operations, and modifiers that are supported by LLVM's `llvm-ar` tool. To see the options, operations, and modifiers that are supported by both utility tools, see the *LLVM llvm-ar reference documentation*.

Outputs

A successful run of `armllvm-ar` returns `0` and creates an archive called <archive>, which normally has a `.a` suffix. A nonzero return value indicates an error.

Related references

3.2.4 armllvm-ranlib reference on page 3-36

3.2.4 armllvm-ranlib reference

This topic describes `armllvm-ranlib`. `armllvm-ranlib` is a utility tool provided in the Arm Compiler for Linux package, and is a variant of the LLVM `llvm-ranlib` utility tool.

Like, `llvm-ranlib` is a synonym to the LLVM archiver tool `llvm-ar -s`, `armllvm-ranlib` is a synonym for running `armllvm-ar -s`.

Note

For a full description of `llvm-ranlib` see the *llvm-ranlib LLVM command documentation*.

Related concepts

3.2.1 What is Link Time Optimization (LTO) on page 3-32

Related tasks

3.2.2 Compile with Link Time Optimization (LTO) on page 3-33

Related references

3.2.3 armllvm-ar and reference on page 3-36

3.2.4 armllvm-ranlib reference on page 3-36

3.3 Arm Optimization Report

Arm Optimization Report builds on the llvm-opt-report tool available in open-source LLVM. Arm Optimization Report shows you the optimization decisions that the compiler is making, in-line with your source code, enabling you to better understand the unrolling, vectorization, and interleaving behavior.

Unrolling

Unrolling is when a scalar loop is transformed to perform multiple iterations at once, but still as scalar instructions.

The unroll factor is the number of iterations of the original loop that are performed at once. Sometimes, loops with known small iteration counts are completely unrolled, such that no loop structure remains. In completely unrolled cases, the unroll factor is the total scalar iteration count.

Vectorization

Vectorization is when multiple iterations of a scalar loop are replaced by a single iteration of vector instructions.

The vectorization factor is the number of lanes in the vector unit, and corresponds to the number of scalar iterations that are performed by each vector instruction

Note

The true vectorization factor is unknown at compile-time for SVE, because SVE supports scalable vectors.

When SVE is enabled, Arm Optimization Report reports a vectorization factor that corresponds to a 128-bit SVE implementation.

If you are working with an SVE implementation with a larger vector width (for example, 256 bits or 512 bits), the number of scalar iterations that are performed by each vector instruction increases proportionally.

`SVE scaling factor = <true SVE vector width> / 128`

Loops vectorized using scalable vectors are annotated with `VS<F,I>`. For more information, see *arm-opt-report reference* on page 3-40.

Interleaving

Interleaving is a combination of vectorization followed by unrolling; multiple streams of vector instructions are performed in each iteration of the loop.

The combination of vectorization and unrolling information tells you how many iterations of the original scalar loop are performed in each iteration of the generated code.

```
Number of scalar iterations = <unroll factor> x <vectorization factor> x <interleave count>
x <SVE scaling factor>
```

Note

The number of scalar iterations is not an exact figure. For SVE code, the compiler can use the predication capabilities of SVE. For example, a 10-iteration scalar operation on 64-bit values takes 3 iterations on a 256-bit SVE-enabled target.

Reference

The annotations Arm Optimization Report uses to annotate the source code, and the options that can be passed to `arm-opt-report` are described in the **Arm Optimization Report reference**.

This section contains the following subsections:

- *3.3.1 How to use Arm Optimization Report* on page 3-39.
- *3.3.2 arm-opt-report reference* on page 3-40.

3.3.1 How to use Arm Optimization Report

This topic describes how to use Arm Optimization Report.

Prerequisites

Download and install Arm Compiler for Linux. For more information, see *Download Arm Compiler for Linux* and *Installation*.

Procedure

1. To generate a machine-readable `.opt.yaml` report, at compile time add `-fsave-optimization-record` to your command line.

   A `<filename>.opt.yaml` report is generated by Arm Compiler, where `<filename>` is the name of the binary.
2. To inspect the `<filename>.opt.yaml` report, as augmented source code, use `arm-opt-report`:

   ```
   arm-opt-report <filename>.opt.yaml
   ```

   Annotated source code appears in the terminal.

Example 3-2 Example

---

1. Create an example file called `example.f90` containing the following code:

   ```
   subroutine foo
     implicit none
     call bar()
   end subroutine foo

   subroutine test
     implicit none
     integer :: i
     integer, dimension(1600) :: res, p, d

     do i = 1, 1600
       res(i) = merge(res(i), res(i) + d(i), p(i) == 0)
     end do

     do i = 1, 16
       res(i) = merge(res(i), res(i) + d(i), p(i) == 0)
     end do

     call foo()
     call foo()
     call bar()
     call foo()
   end subroutine test
   ```
2. Compile the file, for example to a shared object called `example.o`:

   ```
   armflang -O3 -fsave-optimization-record -c -o example.o example.f90
   ```

   This generates a file, `example.opt.yaml`, in the same directory as the built object.

   For compilations that create multiple object files, there is a report for each build object.

   Note

   This example compiles to a shared object, however, you could also compile to a static object or to a binary.
3. View the `example.opt.yaml` file using `arm-opt-report`:

   ```
   arm-opt-report example.opt.yaml
   ```

Annotated source code is displayed in the terminal:

```
< example.f90
 1          | subroutine foo
 2          |   implicit none
 3          |   call bar()
 4          | end subroutine foo
 5          |
 6          | subroutine test
 7          |   implicit none
 8          |   integer :: i
 9          |   integer, dimension(1600) :: res, p, d
10          |
11          |
12          |   do i = 1, 1600
13          |     res(i) = merge(res(i), res(i) + d(i), p(i) == 0)
14     V4,2 |   end do
15          |
16          |
17          |   do i = 1, 16
18          |     res(i) = merge(res(i), res(i) + d(i), p(i) == 0)
19  U16     |   end do
20          |
21 I        |   call foo()
22 I        |   call foo()
23          |   call bar()
24 I        |   call foo()
25          | end subroutine test
```

The example Arm Optimization Report output is interpreted as follows:

- The `do` loop on line 12:
  - Is vectorized
  - Has a vectorization factor of four (there are four 32-bit integer lanes)
  - Has an interleave factor of two (the loop was unrolled twice)
- The `for` loop on line 19 is unrolled 16 times. This means it is completely unrolled, with no remaining loops.
- All three instances of `call foo()` are inlined

Related references

3.3.2 arm-opt-report reference on page 3-40

Related information

Arm Compiler for Linux and Arm Allinea Studio

Take a trial

Help and tutorials

3.3.2 arm-opt-report reference

This reference topic describes the options that are available for `arm-opt-report`. The topic also describes the annotations that `arm-opt-report` can use to annotate source code.

`arm-opt-report` uses a YAML optimization record, as produced by the `-fsave-optimization-record` option of LLVM, to output annotated source code that shows the various optimization decisions taken by the compiler.

Note

`-fsave-optimization-record` is not set by default by Arm Compiler for Linux.

Possible annotations are:

Annotation	Description
`I`	A function was inlined.
`U<N>`	A loop was unrolled `<N>` times.

(continued)

Annotation	Description
`V<F, I>`	A loop has been vectorized. Each vector iteration performed has the equivalent of `F*I` scalar iterations. Vectorization Factor, `F`, is the number of scalar elements that are processed in parallel. Interleave count, `I`, is the number of times the vector loop was unrolled.
`VS<F,I>`	A loop has been vectorized using scalable vectors. Each vector iteration performed has the equivalent of `N*F*I` scalar iterations, where `N` is the number of vector granules, which can vary according to the machine the application is run on. **Note** LLVM assumes a granule size of 128 bits when targeting SVE. `F` (Vectorization Factor) and `I` (Interleave count) are as described for `V<F,I>`.

Syntax

```
arm-opt-report [options] <input>
```

Options

Generic Options:

`--help`

Displays the available options (use `--help-hidden` for more).

`--help-list`

Displays a list of available options (`--help-list-hidden` for more).

`--version`

Displays the version information for `arm-opt-report`.

llvm-opt-report options:

`--hide-detrimental-vectorization-info`

Hides remarks about vectorization being forced despite the cost-model indicating that it is not beneficial.

`--hide-inline-hints`

Hides suggestions to inline function calls which are preventing vectorization.

`--hide-lib-call-remark`

Hides remarks about the calls to library functions that are preventing vectorization.

`--hide-vectorization-cost-info`

Hides remarks about the cost of loops that are not beneficial for vectorization.

`--no-demangle`

Does not demangle function names.

`-o=<string>`

Specifies an output file to write the report to.

`-r=<string>`

Specifies the root for relative input paths.

`-s`

Omits vectorization factors and associated information.

`--strip-comments`

Removes comments for brevity

`--strip-comments=<arg>`

Removes comments for brevity. Arguments are:

- `none`: Do not strip comments.
- `c`: Strip C-style comments.
- `c++`: Strip C++-style comments.
- `fortran`: Strip Fortran-style comments.

Outputs

Annotated source code.

Related tasks

3.3.1 How to use Arm Optimization Report on page 3-39

Related tasks

3.3.1 How to use Arm Optimization Report on page 3-39

Related references

3.3.2 arm-opt-report reference on page 3-40

3.4 Optimization remarks

Optimization remarks provide you with information about the choices that are made by the compiler. You can use them to see which code has been inlined or they can help you understand why a loop has not been vectorized.

By default, Arm Compiler for Linux prints optimization remark information to `stderr`. If this is your terminal output, you might want to redirect the terminal output to a separate file to store and search the remark information more easily.

To enable optimization remarks, pass one or more of the following `Rpass` options to `armflang` at compile time:

- `-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 a remark expression that you want see. The supported remark types are:

- `loop-vectorize`: Remarks about vectorized loops.
- `inline`: Remarks about inlining.
- `loop-unroll`: Remarks about unrolled loops.

`<regex>` can be one or more of the preceding remark types. If you filter for multiple types, separate each type with a pipe (`|`) character.

Alternatively, you can choose to print all optimization remark information by specifying `.*` for `<regex>`.

Note

Use `.*` with caution; depending on the size of code, and the level of optimization, the compiler can print a lot of information.

When you provide `-Rpass`, `armflang` generates debug line tables equivalent to passing `-gline-tables-only`, unless you instruct it not to by another debug controlling option. This default behavior ensures that source location information is available to print the remarks.

The general syntax to compile with optimization remarks enabled (`-Rpass[-<option>]`), optionally include debug options, and redirect the information to an output file (`<remarks-file.txt>`), is:

```
armflang -O<level> -Rpass[-<option>]=<remarks> <source> [<debug-option>] 2> <remarks-
file.txt>
```

Note

`2> <remarks-file.txt>` assumes a Bourne-shell syntax. You will need to replace this with the appropriate syntax to redirect output in your shell type.

This section contains the following subsection:
- *3.4.1 Enable optimization remarks* on page 3-43.

3.4.1 Enable optimization remarks

Describes how to enable optimization remarks and redirect the information they provide to an output file.

Prerequisites

Download and install Arm Compiler for Linux. You can download Arm Compiler for Linux from the *Arm Allinea Studio Downloads* page. Learn how to install and configure Arm Compiler for Linux, using the *Arm Compiler for Linux installation instructions* on the Arm Developer website.

Procedure

1. Compile your code with optimization remarks. To enable optimization remarks, pass one or more of `-Rpass=<regex>`, `-Rpass-missed=<regex>`, or `Rpass-analysis=<regex>` on your compile line.

   For example, to report all the remarks about vectorized loops (`-Rpass=loop-vectorize`), when compiling an input file called `source.f90`, use:

   ```
   armflang -O3 -Rpass=loop-vectorize source.f90 -gline-tables-only
   ```

   Result:

   ```
   example.f90:21: vectorized loop (vectorization width: 2,
   interleaved count: 1)
     [-Rpass=loop-vectorize]
       do i=1
   ```

2. Or, to print the optimization remark information to a separate file, instead of `stderr`, run:

   ```
   armflang -O<level> -Rpass[-<option>]=<remarks> <source> [<debug-option>] 2> <remarks-file>
   ```

   Replacing `2>` with the appropriate redirection syntax for the shell type

   For example, to redirect the output to a file called `vecreport.txt`, use:

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

A `<remarks-file.txt>` file is created which contains the optimization remarks.

Related information

Arm Fortran Compiler

Related tasks

3.4.1 Enable optimization remarks on page 3-43

3.5 Profile Guided Optimization (PGO)

Learn about Profile Guided Optimization (PGO) and how to use `llvm-profdata`. `llvm-profdata` is LLVM's utility tool for profiling data and displaying profile counter and function information. `llvm-profdata` is included in Arm Compiler for Linux.

Profile Guided Optimization (PGO) is a technique where you use profiling information to improve application run-time performance. To use PGO, you must generate profile information from an application, then recompile the application code while passing profile information to the compiler. The compiler can interpret and use the profile information to make informed optimization decisions. For example, when the compiler knows the frequency of a function call in an applications code, it can help the compiler make inlining decisions.

To enable the compiler to make the best optimization decisions for your applications code, you must pass profiling data that is representative of the applications typical workload. To generate profiling information that is representative of a typical workload, compile your application with your typical compiler options and run the application as you typically would.

The profile information can be generated from either:

- A sampling profiler
- An instrumented version of the code.

LLVM's documentation describes both methods. In this section, we only describe how to:

- Generate profile information from an instrumented version of the application code.
- Use `llvm-profdata` to combine and convert profile information from instrumented code into a format that the compiler can read as an input.

This section contains the following subsections:

- *3.5.1 How to compile with Profile Guided Optimization (PGO)* on page 3-45.
- *3.5.2 llvm-profdata reference* on page 3-47.

3.5.1 How to compile with Profile Guided Optimization (PGO)

Learn how to use Profile Guided Optimization (PGO) with Arm Fortran Compiler.

Note

The following procedure describes how to generate, and use, profile data using Arm Compiler for Linux. Profile data files generated by GCC compilers cannot be used by Arm Compiler for Linux.

Prerequisites

- Download and install Arm Compiler for Linux.
- Load the Arm Compiler for Linux environment module for your system.
- Add the `llvm-bin` directory to your `PATH`. For example:

  ```
  PATH=$PATH:<install-dir>/../llvm-bin
  ```

  Where `<install-dir>` is the Arm Compiler for Linux install location.

  Note

  To obtain `<install-dir>` for your system, load the Arm Compiler for Linux environment module and run `which armflang`. The returned path is your `<install-dir>`.

Procedure

1. Build an instrumented version of your application code. Compile your application with the `-fprofile-instr-generate` option:

   ```
   armflang -O<level> [options] -fprofile-instr-generate=<profdata_file>.profraw <source> -o
   <binary>
   ```

   Note

   - For good optimization, use `-O2` optimization level or higher.
   - To ensure that the instrumented executable represents the real executable, compile your application code with the same compiler options.
   - By default, if you do not specify a `<profdata_file>.profraw`, when you compile the application with `-fprofile-instr-generate`, the profile data is written to `default.profraw`.

     To change this behavior, either specify `<profdata_file>.profraw` on the compile line, or set the `LLVM_PROFILE_FILE="<profdata_file>.profraw"` when you run your application (see next step). Both `-fprofile-instr-generate` and `LLVM_PROFILE_FILE` can use the following modifiers to uniquely set profile data filename:
     — `%p` to state the process ID
     — `%h` to state the hostname
     — `%m` to state the unique profile name.

     For example, `LLVM_PROFILE_FILE="example-{%p|%h|%m}.profraw"`.

     If both `-fprofile-instr-generate` and `LLVM_PROFILE_FILE` are set, `LLVM_PROFILE_FILE` takes priority.

2. Run your application code with a typical workload. Either:
   - Run it with default behavior:

     ```
     ./<binary>
     ```

     The profile data is written to the profile data file specified in the previous step, or if no file was specified, to `default.profraw`.
   - Run the application and specify a new filename for the `.profraw` file using the `LLVM_PROFILE_FILE` environment variable:

     ```
     LLVM_PROFILE_FILE=<profdata_file>.profraw ./<binary>
     ```

     The profile data is written to `<profdata_file>.profraw`.
3. Combine and convert your `.profraw` files into a single processed `.profdata` file using the `llvm-profdata` tool `merge` command:
   - If you have a single `.profraw` file, use:

     ```
     llvm-profdata merge -output=<profdata_file>.profdata <file>.profraw
     ```

     Note

     Where you only have one `.profraw` file, no files are combined, however, you must still run the `merge` command to convert the file format.

   - If you have multiple `.profraw` files, you can combine and convert them into a single profile data file, `.profdata`, by either:
     — Passing each `.profraw` file in separately:

     ```
     llvm-profdata merge -output=<outfile>.profdata <filename1> [<filename2> ...]
     ```

     — Passing in all the `.profraw` files in a directory:

     ```
     llvm-profdata merge -output=<outfile>.profdata *.profraw
     ```

4. Recompile your application code and pass the profile data file, `<outfile>.profdata`, to `armflang` using the `-fprofile-instr-use=<outfile>.profdata` option:

   ```
   armflang -O<level> -fprofile-instr-use=<outfile>.profdata <source> -o <binary>
   ```

   This step can be repeated without having to regenerate a new profile data file. However, as compilation decisions change and change the output application code, `armflang` might get to a point where the profile data can no longer be used. At this point, `armflang` will output a warning.

Example 3-3 Example: Compiling code with PGO

This example uses 'foo.f90' as the source code file and 'foo-binary'.

1. Build an instrumented version of the `foo-binary` application code:

   ```
   armflang -O2 -fprofile-instr-generate foo.f90 -o foo-binary
   ```

2. Run `foo-binary` with a typical workload twice, creating separate `.profraw` files using their process ID to distinguish them:

   ```
   LLVM_PROFILE_FILE="foorun-%p.profraw"
   ./foo-binary
   ./foo-binary
   ```

3. Combine and convert the `.profraw` files into a single processed `.profdata` file:

   ```
   llvm-profdata merge -output=foorun.profdata foorun-*.profraw
   ```

4. Recompile the `foo-binary` application code passing the `foorun.profdata` profile data file to `armflang`:

   ```
   armflang -O2 -fprofile-instr-use=foorun.profdata foo.f90 -o foo-binary
   ```

Related concepts

3.5.2 llvm-profdata reference on page 3-47

Related information

LLVM's documentation

LLVM Command Guide

3.5.2 llvm-profdata reference

This topic describes the commands and lists the options for the `llvm-profdata` tool, for instrumentation-built profile data.

Note

Full documentation for the `llvm-profdata` is available online in the *LLVM Command Guide*.

In Arm Compiler for Linux, the `llvm-profdata` tool is located in `<install_dir>/arm-linux-compiler-*/llvm-bin`. To enable the `llvm-profdata` tool, add the `llvm-bin` directory to your `PATH`.

`llvm-profdata` accepts three commands: `merge`, `show`, and `overlap`. The following table describes each.

Table 3-1 Describes the commands for llvm-profdata

Command	Syntax	Description	Common options
`merge`	`llvm-profdata merge -instr [options] [filename1] {[filename2] ...}`	`merge` combines multiple, instrumentation-built, profile data files into a single, indexed, profile data file.	• `-weighted-files=<weight>,<filename>` • `-input-files=<path>` • `-sparse=true|false` • `-num-threads=<value>` • `-prof-sym-list=<path>` • `-compress-all-sections=true|false`
`show`	`llvm-profdata show -instr [options] [filename]`	`show` displays profile counter and (optional) function information for a profile data file.	• `-all-functions` • `-counts` • `-function=<string>` • `-text` • `-topn=<value>` • `-memop-sizes` • `-list-below-cutoff` • `-showcs`
`overlap`	`llvm-profdata overlap [options] [base profile] [test profile]`	`overlap` displays the overlap of profile counter information for two profile data files or, optionally, for any functions that match a given string (`<string>`).	• `-function=<string>` • `-value-cutoff=<value>` • `-cs`

Global options that all of the commands accept include:

- `-help`
- `-output=<filename>`

Related information

LLVM Command Guide

Related concepts

3.5.2 llvm-profdata reference on page 3-47

Related tasks

3.5.1 How to compile with Profile Guided Optimization (PGO) on page 3-45

Chapter 4
Compiler options

This chapter describes the options supported by `armflang`.

`armflang` provides many command-line options, including most Flang command-line options in addition to a number of Arm-specific options. Flang is a Fortran language front-end integrated with LLVM which, similar to Clang, supports community-supported options. Many common options, together with the Arm-specific options, are described in this chapter. The same options are also described in the tool through the `--help` option (run `armflang --help`), and in the `man` pages (run `man armflang`).

Additional information about community feature command-line options is available on the *Flang community GitHub web site*.

To see a list of arguments that Arm Fortran Compiler supports for a specific option, bash terminal users can also use command line completion (also known as tab completion). For example, to list the supported arguments for `-ffp-contract=` with `armflang` type the following command line into your terminal (but do not run it):

```
armflang -ffp-contract=
```

Press the **Tab** button on your keyboard. The arguments supported by `-ffp-contract=` return:

```
fast  off  on
```

Note

For more information about enabling this for other terminal types, see the *Arm Allinea Studio installation instructions*.

It contains the following sections:

- *4.58 -version* on page 4-112.
- *4.59 -Wl,* on page 4-113.
- *4.60 -Xlinker* on page 4-114.

4.1 Arm Fortran Compiler Options by Function

This provides a summary of the `armflang` command-line options that Arm Fortran Compiler supports.

Actions

Options that control what action to perform on the input.

Option	Description
4.8 -E on page 4-62	Stop after pre-processing. Output the pre-processed source.
4.52 -S on page 4-106	Stop after compiling the source and emit assembler files.
4.4 -c on page 4-58	Stop after compiling or assembling sources and do not link. This outputs object files.
4.22 -fopenmp on page 4-76	Enable ('-fopenmp') or disable ('-fno-openmp' [default]) OpenMP and link in the OpenMP library, libomp.
4.29 -fsyntax-only on page 4-83	Show syntax errors but do not perform any compilation.

File options

Options that specify input or output files.

Option	Description
4.38 -I on page 4-92	Add directory to include search path and Fortran module search path.
4.5 -config on page 4-59	Passes the location of a configuration file to the compile command.
4.40 -isystem on page 4-94	Add a directory to the include search path, before system header file directories.
4.48 -o on page 4-102	Write output to <file>.

Basic driver options

Options that affect basic functionality of the armclang or armflang driver.

Option	Description
4.2 -### on page 4-55	Print (but do not run) the commands to run for this compilation.
4.34 -gcc-toolchain= on page 4-88	Use the gcc toolchain at the given directory.
4.36 -help on page 4-90	Display available options.
4.37 -help-hidden on page 4-91	Display hidden options. Only use these options if advised to do so by your Arm representative.
4.49 -print-search-dirs on page 4-103	Print the paths that are used for finding libraries and programs.
4.57 -v on page 4-111	Show commands to run and use verbose output.
4.58 -version on page 4-112	Show the version number and some other basic information about the compiler.

Optimization options

Options that control what optimizations should be performed.

Option	Description
4.47 -O on page 4-101	Specifies the level of optimization to use when compiling source files.
4.3 -armpl= on page 4-56	Enable Arm Performance Libraries (ArmPL).
4.9 -fassociative-math on page 4-63	Allow ('-fassociative-math') or do not allow ('-fno-associative-math' [default]) the re-association of operands in a series of floating-point operations.

(continued)

Option	Description
4.12 -ffast-math on page 4-66	Enable ('-ffast-math') or disable ('-fno-fast-math' [default, except with '-Ofast']) aggressive, lossy floating-point optimizations.
4.15 -ffp-contract= on page 4-69	Controls when the compiler is permitted to form fused floating-point operations (for example, Fused Multiply-Add (FMA) operations).
4.17 -finline-functions on page 4-71	Inline ('-finline-functions') or do not inline ('-fno-inline-functions') suitable functions.
4.18 -flto on page 4-72	Enable ('-flto') or disable ('-fno-lto' [default]) Link Time Optimizations (LTO).
4.25 -fsave-optimization-record on page 4-79	Enable ('-fsave-optimization-record') or disable ('-fno-save-optimization-record' [default]) the generation of a YAML optimization record file.
4.26 -fsigned-zeros on page 4-80	Allow ('-fno-signed-zeros') or do not allow ('-fsigned-zeros' [default, except with '-Ofast']) optimizations that ignore the sign of floating point zeros.
4.27 -fsimdmath on page 4-81	Enable ('-fsimdmath') or disable ('-fno-simdmath' [default]) the vectorized libm library to support the vectorization of loops containing calls to basic library functions, such as those declared in math.h
4.30 -ftrapping-math on page 4-84	Tell the compiler to assume ('-ftrapping-math'), or not to assume ('-fno-trapping-math'), that floating point operations can trap. For example, divide by zero.
4.31 -fvectorize on page 4-85	Enable/disable loop vectorization (enabled by default).
4.43 -march= on page 4-97	Specifies the base architecture and extensions available on the target.
4.44 -mcpu= on page 4-98	Select which CPU architecture to optimize for.

Fortran Options

Options that affect the way Fortran workloads are compiled.

Option	Description
4.10 -fbackslash on page 4-64	Treat backslash as C-style escape character ('-fbackslash' [default]) or as a normal character ('-fno-backslash').
4.11 -fconvert= on page 4-65	Convert between big and little endian data format. Default is '-fconvert=native'.
4.13 -ffixed-form on page 4-67	Force fixed-form format Fortran. This is default for .f and .F files, and is the inverse of -ffree-form.
4.14 -ffixed-line-length- on page 4-68	Set line length (0 | 72 | 132 | none) in fixed-form format Fortran. Default is 72. 0 and none are equivalent and set the line length to a very large value (>132).
4.16 -ffree-form on page 4-70	Force free-form format for Fortran. This is default for .f90 and .F90 files, and is the inverse of -ffixed-form.
4.19 -fnative-atomics on page 4-73	Enable ('-fnative-atomics' [default]) or disable ('-fno-native-atomics') the use of native atomic instructions for OpenMP atomics.
4.21 -fno-fortran-main on page 4-75	Do not link in Fortran main.
4.23 -frealloc-lhs on page 4-77	Select semantics for assignments to allocatables.
4.24 -frecursive on page 4-78	Allocate all local arrays on the stack, allowing thread-safe recursion (enabled by default with -fopenmp).
4.28 -fstack-arrays on page 4-82	Place all automatic arrays on stack memory (enabled by default with -Ofast).
4.39 -i8 on page 4-93	Treat INTEGER and LOGICAL as INTEGER*8 and LOGICAL*8.

(continued)

Option	Description
4.45 -no-flang-libs on page 4-99	Do not link against Flang libraries.
4.51 -r8 on page 4-105	Treat REAL as REAL*8.

Development options

Options that facilitate code development.

Option	Description
4.32 -g on page 4-86	Generate source-level debug information.
4.33 -g0 on page 4-87	Disable generation of source-level debug information (default).
4.35 -gline-tables-only on page 4-89	Emit debug line number tables only.

Warning options

Options that control the behavior of warnings.

Option	Description
4.50 -Qunused-arguments on page 4-104	Do not emit a warning for unused driver arguments.
4.20 -fno-crash-diagnostics on page 4-74	Disable the auto-generation of preprocessed source files and a script for reproduction during a clang crash.

Preprocessor options

Options controlling the behavior of the preprocessor.

Option	Description
4.7 -D on page 4-61	Define <macro> to <value> (or 1 if <value> omitted).
4.56 -U on page 4-110	Undefine macro <macro>.
4.6 -cpp on page 4-60	Preprocess Fortran files.
4.46 -nocpp on page 4-100	Do not preprocess Fortran files.

Linker options

Options that are passed on to the linker or affect linking.

Option	Description
4.41 -L on page 4-95	Add a directory to the list of paths that the linker searches for user libraries.
4.59 -Wl, on page 4-113	Pass the comma separated arguments in <arg> to the linker.
4.60 -Xlinker on page 4-114	Pass <arg> to the linker.
4.42 -l on page 4-96	Search for the library named <library> when linking.
4.53 -shared on page 4-107	Create a shared object that can be linked against.
4.54 -static on page 4-108	Link against static libraries.
4.55 -static-arm-libs on page 4-109	Link against static Arm libraries.

4.2 -###

Print (but do not run) the commands to run for this compilation.

Syntax

```
armflang -###
```

4.3 -armpl=

Enable Arm Performance Libraries (ArmPL).

Instructs the compiler to load the optimum version of Arm Performance Libraries for your target architecture and implementation. This option also enables optimized versions of the C mathematical functions declared in the math.h library, tuned scalar and vector implementations of Fortran math intrinsics. This option implies `-fsimdmath`.

ArmPL provides libraries suitable for a range of supported CPUs. If you intend to use -armpl, you must also specify the required architecture using the -mcpu flag.

The -armpl option also enables:
- Optimized versions of the C mathematical functions declared in math.h.
- Optimized versions of Fortran math intrinsics.
- Auto-vectorization of C mathematical functions (disable this with -fno-simdmath).
- Auto-vectorization of Fortran math intrinsics (disable this with -fno-simdmath).

Default

By default, `-armpl` is not set (in other words, `OFF`)

Default argument behavior

If `-armpl` is set with no arguments, the default behavior of the option is `armpl=lp64,sequential`.

However, the default behavior of the arguments is also determined by the specification (or not) of the `-i8` (when using armflang) and `-fopenmp` options:

- If the `-i8` option is not specified, `lp64` is enabled by default. If `-i8` is specified, `ilp64` is enabled by default.
- If the `-fopenmp` option is not specified, `sequential` is enabled by default. If `-fopenmp` is specified, `parallel` is enabled by default.

In other words:
- Specifying `-armpl` sets `-armpl=lp64,sequential`.
- Specifying `-armpl` and `-i8` sets `-armpl=ilp64,sequential`.
- Specifying `-armpl` and `-fopenmp` sets `-armpl=lp64,parallel`.
- Specifying `-armpl`, `-i8`, and `-fopenmp` sets `-armpl=ilp64,parallel`.

Syntax

```
armflang -armpl=<arg1>,<arg2>...
```

Arguments

`lp64`

Use 32-bit integers. (default)

`ilp64`

Use 64-bit integers. Inverse of lp64. (default if using -i8 with armflang).

`sequential`

Use the single-threaded implementation of Arm Performance Libraries. (default)

`parallel`

Use the OpenMP multi-threaded implementation of Arm Performance Libraries. Inverse of sequential. (default if using `-fopenmp`)

`sve`

Use the ‘Generic’ SVE library from Arm Performance Libraries.

Note:

- To enable SVE compilation and library usage on SVE-enabled targets, use `-armpl -mcpu=native`.
- To enable SVE(2) compilation and library usage on a target without native support for these features, use `-armpl=sve -march=armv8-a+<Feature>`, where `<Feature>` is one of `sve`, `sve2`, `sve2-bitperm`, `sve2-aes`, `sve2-sha3`, or `sve2-sm4`.

4.4 -c

Stop after compiling or assembling sources and do not link. This outputs object files.

Syntax

```
armflang -c
```

4.5 -config

Passes the location of a configuration file to the compile command.

Use a configuration file to specify a set of compile options to be run at compile time. The configuration file can be passed at compile time, or an environment variable can be set for it to be used for every invocation of the compiler. For more information about creating and using a configuration file, see *https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-allinea-studio/installation/configure*.

Syntax

```
armflang --config <arg>
```

4.6 -cpp

Preprocess Fortran files.

Syntax

```
armflang -cpp
```

4.7 -D

Define <macro> to <value> (or 1 if <value> omitted).

Syntax

```
armflang -D<macro>=<value>
```

4.8 -E

Stop after pre-processing. Output the pre-processed source.

Syntax

```
armflang -E
```

4.9 -fassociative-math

Allow ('-fassociative-math') or do not allow ('-fno-associative-math' [default]) the re-association of operands in a series of floating-point operations.

For example, (a * b) + (a * c) => a * (b + c). Note: Using `-fassociative-math` violates the ISO C and C++ language standard.

Default

Default is `-fno-associative-math`.

Syntax

```
armflang -fassociative-math, -fno-associative-math
```

4.10 -fbackslash

Treat backslash as C-style escape character (‘-fbackslash’ [default]) or as a normal character (‘-fno-backslash’).

Default

Default is the C-style, `-fbackslash`.

Syntax

```
armflang -fbackslash, -fno-backslash
```

4.11 -fconvert=

Convert between big and little endian data format. Default is '-fconvert=native'.

Default

Default is `-fconvert=native`.

Syntax

```
armflang -fconvert={native \| swap \| big-endian \| little-endian}
```

4.12 -ffast-math

Enable (‘-ffast-math’) or disable (‘-fno-fast-math’ [default, except with ‘-Ofast’]) aggressive, lossy floating-point optimizations.

Using `-ffast-math` is equivalent to specifying the following options individually:

- `-fassociative-math`
- `-ffinite-math-only`
- `-ffp-contract=fast`
- `-fno-math-errno`
- `-fno-signed-zeros`
- `-fno-trapping-math`
- `-freciprocal-math`

Default

Default is `-fno-fast-math`, except where `-Ofast` is used. Using `-Ofast` enables `-ffast-math`.

Syntax

```
armflang -ffast-math
```

4.13 -ffixed-form

Force fixed-form format Fortran. This is default for .f and .F files, and is the inverse of -ffree-form.

Syntax

```
armflang -ffixed-form
```

4.14 -ffixed-line-length-

Set line length (0 | 72 | 132 | none) in fixed-form format Fortran. Default is 72. 0 and none are equivalent and set the line length to a very large value (>132).

Default

Default is `-ffixed-line-length-72`.

Syntax

```
armflang -ffixed-line-length-{0 \| 72 \| 132 \| none}
```

4.15 -ffp-contract=

Controls when the compiler is permitted to form fused floating-point operations (for example, Fused Multiply-Add (FMA) operations).

To generate better optimized code, allow the compiler to form fused floating-point operations.

On the compile line, `-ffp-contract` permits three arguments to control fused floating-point contract behavior: `OFF`, `ON`, and `FAST`. However, at the source level, you can also use the `STDC FP_CONTRACT={OFF|ON}` pragma to control the fused floating-point operation behavior for C/C++ code:

- When `-ffp-contract` is set to `{off|on}`, `STDC FP_CONTRACT={OFF|ON}` is still honoured where it is specified, and can switch the behavior.
- When `-ffp-contract` is set to `fast`, the behavior is always set to `FAST` and the `STDC FP_CONTRACT` pragma is ignored.

To generate better optimized code, allow the compiler to form fused floating-point operations.

Note

The fused floating-point instructions typically operate to a higher degree of accuracy than individual multiply and add instructions.

Default

For Fortran code, the default is `-ffp-contract=fast`. For C/C++ code, the default is `-ffp-contract=off`.

Syntax

```
armflang -ffp-contract={fast\|on\|off}
```

Arguments

`fast`

Use fused floating-point operations whenever possible, even if the operations are not permitted by the language standard. Note: Some fused floating-point contractions are not permitted by the C/C++ standard because they can lead to deviations from the expected results.

`on`

Use fused floating-point only when the language allows it. For example, for C/C++ code, floating-point contractions are allowed in a single C/C++ statement, however, for Fortran code, floating-point contractions are always enabled.

`off`

Do not use fused floating-point operations.

4.16 -ffree-form

Force free-form format for Fortran. This is default for .f90 and .F90 files, and is the inverse of -ffixed-form.

Syntax

```
armflang -ffree-form
```

4.17 -finline-functions

Inline ('-finline-functions') or do not inline ('-fno-inline-functions') suitable functions.

Note: For all `-finline-*` and `-fno-inline-*` options, the compiler ignores all but the last option that is passed to the compiler command.

Default

For `armclang|armclang++`, the default at `-O0` and `-O1` is `-fno-inline-functions`, and the default at `-O2` and higher is `-finline-functions`. For `armflang`, the default at all optimization levels is `-finline-functions`.

Syntax

```
armflang -finline-functions, -fno-inline-functions
```

4.18 -flto

Enable (‘-flto’) or disable (‘-fno-lto’ [default]) Link Time Optimizations (LTO).

You must pass the option to both the link and compile commands. When LTO is enabled, compiler object files contain an intermediate representation of the original code. When linking the objects together into a binary at link time, the compiler performs optimizations. It can allow the compiler to inline functions from different files, for example.

Default

Default is `-fno-lto`.

Syntax

```
armflang -flto, -fno-lto
```

4.19 -fnative-atomics

Enable ('-fnative-atomics' [default]) or disable ('-fno-native-atomics') the use of native atomic instructions for OpenMP atomics.

By default, armflang generates native atomic instructions for OpenMP atomic operations, falling back to libatomic when no suitable native instruction is available. Use `-fno-native-atomics` to disable this feature and have armflang generate code that use barriers to guarantee atomicity. This will normally result in a slower program.

Default

Default is `-fnative-atomics`.

Syntax

```
armflang -fnative-atomics, -fno-native-atomics
```

4.20 -fno-crash-diagnostics

Disable the auto-generation of preprocessed source files and a script for reproduction during a clang crash.

Default

By default, `-fno-crash-diagnostics` is disabled. The default behavior of the compiler enables crash diagnostics.

Syntax

```
armflang -fno-crash-diagnostics
```

4.21 -fno-fortran-main

Do not link in Fortran main.

Syntax

```
armflang -fno-fortran-main
```

4.22 -fopenmp

Enable (‘-fopenmp’) or disable (‘-fno-openmp’ [default]) OpenMP and link in the OpenMP library, libomp.

Default

Default is `-fno-openmp`.

Syntax

```
armflang -fopenmp, -fno-openmp
```

4.23 -frealloc-lhs

Select semantics for assignments to allocatables.

Fortran 2003 allows dynamic reallocation, which will error in Fortran 90/95. Use `-fno-realloc-lhs` to restore the F95 behavior. Default is F2003 semantics (`-frealloc-lhs`).

Default

Default is F2003 semantics (`-frealloc-lhs`).

Syntax

```
armflang -frealloc-lhs, -fno-realloc-lhs
```

4.24 -frecursive

Allocate all local arrays on the stack, allowing thread-safe recursion (enabled by default with -fopenmp).

In the absence of this flag, some large local arrays may be allocated in static memory. This reduces stack usage, but is not thread-safe.

Default

`-frecursive` is enabled by default with `-fopenmp`.

Syntax

```
armflang -frecursive
```

4.25 -fsave-optimization-record

Enable (‘-fsave-optimization-record’) or disable (‘-fno-save-optimization-record’ [default]) the generation of a YAML optimization record file.

Optimization records are files named <output name>.opt.yaml, which can be parsed by arm-opt-report to show what optimization decisions the compiler is making, in-line with your source code. For more information, see the ‘Optimize’ chapter in the compiler developer and reference guide.

Default

Default is `fno-save-optimization-record`.

Syntax

```
armflang -fsave-optimization-record, -fno-save-optimization-record
```

4.26 -fsigned-zeros

Allow (‘-fno-signed-zeros’) or do not allow (‘-fsigned-zeros’ [default, except with ‘-Ofast’]) optimizations that ignore the sign of floating point zeros.

Default

Default is `-fsigned-zeros`, except where `-Ofast` is used. Using `-Ofast` enables `-fno-signed-zeros`.

Syntax

```
armflang -fsigned-zeros, -fno-signed-zeros
```

4.27 -fsimdmath

Enable (‘-fsimdmath’) or disable (‘-fno-simdmath’ [default]) the vectorized libm library to support the vectorization of loops containing calls to basic library functions, such as those declared in math.h

Default

For `armclang|armclang++`, the default is `-fno-simdmath`. For `armflang`, the default is `-fsimdmath`.

Syntax

```
armflang -fsimdmath, -fno-simdmath
```

4.28 -fstack-arrays

Place all automatic arrays on stack memory (enabled by default with -Ofast).

Use this option if your Fortran code frequently performs small allocations and deallocations of memory. `-fstack-arrays` improves application performance by using memory on the stack instead of allocating it through malloc, or similar. For programs using very large arrays on particular operating systems, consider extending stack memory runtime limits.

Syntax

```
armflang -fstack-arrays, -fno-stack-arrays
```

4.29 -fsyntax-only

Show syntax errors but do not perform any compilation.

Syntax

```
armflang -fsyntax-only
```

4.30 -ftrapping-math

Tell the compiler to assume ('-ftrapping-math'), or not to assume ('-fno-trapping-math'), that floating point operations can trap. For example, divide by zero.

Possible traps include:

- Division by zero
- Underflow
- Overflow
- Inexact result
- Invalid operation.

Default

Default is `-ftrapping-math`, except where `-Ofast` is used. Using `-Ofast` enables `-fno-trapping-math`.

Syntax

```
armflang -ftrapping-math, -fno-trapping-math
```

4.31 -fvectorize

Enable/disable loop vectorization (enabled by default).

Syntax

```
armflang -fvectorize, -fno-vectorize
```

4.32 -g

Generate source-level debug information.

Syntax

```
armflang -g
```

4.33 -g0

Disable generation of source-level debug information (default).

Syntax

```
armflang -g0
```

4.34 -gcc-toolchain=

Use the gcc toolchain at the given directory.

Syntax

```
armflang --gcc-toolchain=<arg>
```

4.35 -gline-tables-only

Emit debug line number tables only.

Syntax

```
armflang -gline-tables-only
```

4.36 -help

Display available options.

Syntax

```
armflang -help, --help
```

4.37 -help-hidden

Display hidden options. Only use these options if advised to do so by your Arm representative.

Syntax

```
armflang --help-hidden
```

4.38 -I

Add directory to include search path and Fortran module search path.

Directories specified with the `-I` option apply to both the quote form of the include directive and the system header form. For example, #include “file” (quote form), and #include <file> (system header form). Directories specified with `-I` are searched before system include directories and, in `armclang|armclang++` only, after directories specified with `-iquote` (for the quoted form). If any directory is specified with both `-I` and `-isystem` then the directory is searched for as if it were only specified with `-isystem`.

For `armflang`, search for module-files in the directories that are specified with the `-I` option. Directories that are specified with `-I` are searched after the current working directory and before standard system module locations.

Syntax

```
armflang -I<dir>
```

4.39 -i8

Treat INTEGER and LOGICAL as INTEGER*8 and LOGICAL*8.

Syntax

```
armflang -i8
```

4.40 -isystem

Add a directory to the include search path, before system header file directories.

Directories specified with the `-isystem` option apply to both the quote form of the include directive and the system header form. For example, #include “file” (quote form), and #include <file> (system header form). Directories specified with the `-isystem` option are searched after directories specified with `-I` and before system header file directories. Directories specified with `-isystem` are treated as system directories. If any directory is specified with both `-I` and `-isystem` then the directory is searched for as if it were only specified with `-isystem`.

Syntax

```
armflang -isystem<directory>
```

4.41 -L

Add a directory to the list of paths that the linker searches for user libraries.

Syntax

```
armflang -L<dir>
```

4.42 -l

Search for the library named <library> when linking.

Syntax

```
armflang -l<library>
```

4.43 -march=

Specifies the base architecture and extensions available on the target.

Usage: `-march=<arg>` where `<arg>` is constructed as *name[+[no]feature+...]*:

name

`armv8-a` : Armv8 application architecture profile.

`armv8.1-a` : Armv8.1 application architecture profile.

`armv8.2-a` : Armv8.2 application architecture profile.

feature

Is the name of an optional architectural feature that can be explicitly enabled with +feature and disabled with +nofeature.

For AArch64, the following features can be specified:

- `crc` - Enable CRC extension. On by default for `-march=armv8.1-a` or higher.
- `crypto` - Enable Cryptographic extension.
- `fullfp16` - Enable FP16 extension.
- `lse` - Enable Large System Extension instructions. On by default for `-march=armv8.1-a` or higher.
- `sve` - Scalable Vector Extension (SVE). This feature also enables `fullfp16`. See *Scalable Vector Extension* for more information.
- `sve2`- Scalable Vector Extension version two (SVE2). This feature also enables `sve`. See *Arm A64 Instruction Set Architecture* for SVE and SVE2 instructions.
- `sve2-aes` - SVE2 Cryptographic extension. This feature also enables `sve2`.
- `sve2-bitperm` - SVE2 Cryptographic Extension. This feature also enables `sve2`.
- `sve2-sha3` - SVE2 Cryptographic Extension. This feature also enables `sve2`.
- `sve2-sm4` - SVE2 Cryptographic Extension. This feature also enables `sve2`.

Note

When enabling either the `sve2` or `sve` features, to link to the SVE-enabled version of Arm Performance Libraries, you must also include the `-armpl=sve` option. For more information about the supported options for `-armpl`, see the `-armpl` description.

Syntax

```
armflang -march=<arg>
```

4.44 -mcpu=

Select which CPU architecture to optimize for.

Syntax

```
armflang -mcpu=<arg>
```

Arguments

`native`

Auto-detect the CPU architecture from the build computer.

`thunderx2t99`

Optimize for Marvell ThunderX2 based computers.

`neoverse-n1`

Optimize for Neoverse N1 based computers.

`a64fx`

Optimize for Fujitsu A64FX based computers.

`generic`

Generate portable output suitable for any Armv8-A based computer.

4.45 -no-flang-libs

Do not link against Flang libraries.

Syntax

```
armflang -no-flang-libs
```

4.46 -nocpp

Do not preprocess Fortran files.

Syntax

```
armflang -nocpp
```

4.47 -O

Specifies the level of optimization to use when compiling source files.

Default

The default is `-O0`. However, for the best balance between ease of debugging, code size, and performance, it is important to choose an optimization level that is appropriate for your goals.

Syntax

```
armflang -O<level>
```

Arguments

`0`

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.

`1`

Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.

`2`

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.

`3`

Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.

`fast`

Enables all the optimizations from level 3 including those performed with the -ffp-mode=fast armclang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards. -Ofast implies -ffast-math.

4.48 -o

Write output to <file>.

Syntax

```
armflang -o<file>
```

4.49 -print-search-dirs

Print the paths that are used for finding libraries and programs.

Syntax

```
armflang -print-search-dirs, --print-search-dirs
```

4.50 -Qunused-arguments

Do not emit a warning for unused driver arguments.

Syntax

```
armflang -Qunused-arguments
```

4.51 -r8

Treat REAL as REAL*8.

Syntax

```
armflang -r8
```

4.52 -S

Stop after compiling the source and emit assembler files.

Syntax

```
armflang -S
```

4.53 -shared

Create a shared object that can be linked against.

Syntax

```
armflang -shared, --shared
```

4.54 -static

Link against static libraries.

This option prevents runtime dependencies on shared libraries. This is likely to result in larger binaries.

Syntax

```
armflang -static, --static
```

4.55 -static-arm-libs

Link against static Arm libraries.

This option prevents runtime dependencies on libraries shipped with Arm Compiler for Linux (such as libamath, libastring and Arm Performance Libraries). This is likely to result in larger binaries.

Syntax

```
armflang -static-arm-libs
```

4.56 -U

Undefine macro <macro>.

Syntax

```
armflang -U<macro>
```

4.57 -v

Show commands to run and use verbose output.

Syntax

```
armflang -v
```

4.58 -version

Show the version number and some other basic information about the compiler.

Syntax

```
armflang --version, --vsn
```

4.59 -Wl,

Pass the comma separated arguments in <arg> to the linker.

Syntax

```
armflang -Wl,<arg>,<arg2>...
```

4.60 -Xlinker

Pass <arg> to the linker.

Syntax

```
armflang -Xlinker <arg>
```

Chapter 5
Fortran language reference

This chapter can be used as a reference for the Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, and Fortran 2018 language features that are supported by Arm Fortran Compiler.

The support level for the latest Fortran standards (2003 and 2008) are described in *Standards support* on page 6-146.

For information about the Fortran standards, see the *JTC1/SC22/WG5 Fortran standards website*.

It contains the following sections:

5.1 Data types and file extensions

Describes the data types and file extensions that are supported by the Arm Fortran Compiler.

This section contains the following subsections:

- *5.1.1 Data types* on page 5-116.
- *5.1.2 Supported file extensions* on page 5-117.
- *5.1.3 Logical variables and constants* on page 5-118.
- *5.1.4 C/Fortran inter-language calling* on page 5-118.
- *5.1.5 Character* on page 5-119.
- *5.1.6 Complex* on page 5-119.
- *5.1.7 Fortran implementation notes* on page 5-119.

5.1.1 Data types

Arm Fortran Compiler provides the following intrinsic data types:

Table 5-1 Intrinsic data types

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

Table 5-1 Intrinsic data types (continued)

Data Type	Specified as	Size (bytes)
LOGICAL	LOGICAL	4
	LOGICAL*1	1
	LOGICAL([KIND=]1)	1
	LOGICAL*2	2
	LOGICAL([KIND=]2)	2
	LOGICAL*4	4
	LOGICAL([KIND=]4)	4
	LOGICAL*8	8
	LOGICAL([KIND=]8)	8
CHARACTER	CHARACTER	1
	CHARACTER([KIND=]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. The kind has the value of a real data type, with the decimal precision of at least 6 digits and an exponent range of at least 17 digits (single precision):

```
INTEGER, PARAMETER :: my_real_kind = SELECTED_REAL_KIND(6, 17)
...
REAL(my_real_kind) :: x
...
```

Or, alternatively, use the `ISO_FORTRAN_ENV` intrinsic module, which then makes the `REAL32` and `REAL64` kind parameters available to use. For example:

```
USE ISO_FORTRAN_ENV
INTEGER, PARAMETER :: my_real_kind = REAL32
...
REAL(my_real_kind) :: x
...
```

5.1.2 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 5-2 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.
file.fpp file.F	Fixed-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.

5.1.3 Logical variables and constants

This topic describes `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.

5.1.4 C/Fortran inter-language calling

This section provides some useful troubleshooting information when handling argument passing and return values for Fortran functions or subroutines that are 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.

5.1.5 Character

This topic describes how C/C++ functions call Fortran functions that return a `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.

5.1.6 Complex

This topic describes how to call Fortran functions that return a `COMPLEX` data type, from C or C++.

Fortran functions that return a `COMPLEX` data type cannot be directly called from C or C++. Instead, a workaround is possible by passing a C or 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);
```

5.1.7 Fortran implementation notes

Details information that is specific to the implementation of Fortran in Arm Fortran Compiler.

Implementation information:

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

  Note

  This behavior varies from compiler to compiler and is not defined in Fortran standards. The best practice is not to assume that arrays are filled with zeros when they are created.

Related concepts

5.1.4 C/Fortran inter-language calling on page 5-118

5.1.6 Complex on page 5-119

Related references

5.1.1 Data types on page 5-116

5.1.2 Supported file extensions on page 5-117

5.1.3 Logical variables and constants on page 5-118

5.1.5 Character on page 5-119

5.1.7 Fortran implementation notes on page 5-119

5.2 Intrinsics

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

This section contains the following subsections:

5.2.1 Fortran intrinsics 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 in 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.

5.2.2 Bit manipulation functions and subroutines

Functions and subroutines for manipulating bits.

Table 5-3 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

Table 5-3 Bit manipulation functions and subroutines (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`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
`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

5.2.3 Elemental character and logical functions

Elemental character logical conversion functions.

Table 5-4 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

Table 5-4 Elemental character and logical functions (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`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
`INDEX`	Return starting position of substring in 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

Table 5-4 Elemental character and logical functions (continued)

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

5.2.4 Vector/Matrix functions

Functions for vector or matrix multiplication.

Table 5-5 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.

5.2.5 Array reduction functions

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

Table 5-6 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

Table 5-6 Array reduction functions (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`MAXLOC`	Determine the position of the array element with the maximum 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
`MAXVAL`	Determine the maximum 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
`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

Table 5-6 Array reduction functions (continued)

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

5.2.6 String construction functions

Functions for constructing strings.

Table 5-7 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

5.2.7 Array construction manipulation functions

Functions for constructing and manipulating arrays.

Table 5-8 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
`OESHIFT`	Perform end-off shift on an array.	2 3 3 4	ARRAY, INTEGER ARRAY, INTEGER, Any ARRAY, INTEGER, INTEGER ARRAY, INTEGER, Any, INTEGER	ARRAY ARRAY ARRAY ARRAY, ARRAY
`MERGE`	Merge two arguments using 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 3	ARRAY, LOGICAL ARRAY, LOGICAL, VECTOR	ARRAY ARRAY

Table 5-8 Array construction and manipulation functions (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`RESHAPE`	Change the shape of an array.	2 3 3 4	ARRAY, INTEGER ARRAY, INTEGER, ARRAY ARRAY, INTEGER, INTEGER ARRAY, INTEGER, ARRAY, INTEGER	ARRAY ARRAY ARRAY 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.

5.2.8 General inquiry functions

Functions for general determining.

Table 5-9 General inquiry functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`ASSOCIATED`	Determine association status.	1 2	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

5.2.9 Numeric inquiry functions

Functions for determining numeric information.

Table 5-10 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

Table 5-10 Numeric inquiry functions (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`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
`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

5.2.10 Array inquiry functions

Functions for determining information about an array.

Table 5-11 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

5.2.11 Transfer functions

Functions for transferring types.

Table 5-12 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

5.2.12 Arithmetic functions

Functions for manipulating arithmetic.

Table 5-13 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
`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

Table 5-13 Arithmetic functions (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`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
`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)

Table 5-13 Arithmetic functions (continued)

Intrinsic	**Description**	**Num. of Arguments**	**Argument Type**	**Result**
`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 (nonzero)	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
`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 or COMPLEX	REAL
`SIND`	Return the sine (in degrees) of the specified value.	1	REAL or 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

Table 5-13 Arithmetic functions (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`SQRT`	Return the square root of the argument.	1	REAL or COMPLEX	REAL or 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

5.2.13 Miscellaneous functions

Functions for mixcellaneous use.

Table 5-14 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

5.2.14 Subroutines

Supported subroutines.

Table 5-15 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.	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)

5.2.15 Fortran 2003 functions

Fortran 2003-supported functions.

Table 5-16 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 that are passed on the command line when the containing program was invoked. If no command arguments are 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

5.2.16 Fortran 2008 functions

Fortran 2008-supported functions.

Table 5-17 Fortran 2008 functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result

`ACOSH`
`ASINH`
`ATANH`
	Inverse hyperbolic trigonometric functions	1	REAL	REAL

`BESSEL_J0`
`BESSEL_J1`
`BESSEL_JN`
`BESSEL_Y0`
`BESSEL_Y1`
`BESSEL_YN`
	Bessel function of:
	(J0) the first kind of order 0.
	(J1) the first kind of order 1.
	(JN) the first kind.
	(Y0) the second kind of order 0.
	(Y1) the second kind of order 1.
	(YN) the second kind.
		1
		1
		2 or 3
		1
		1
		2 or 3
			REAL
			REAL
			{INTEGER, REAL, or INTEGER}, INTEGER, REAL
			REAL
			REAL
			{INTEGER, REAL, or INTEGER}, INTEGER, REAL
				REAL
				REAL
				REAL
				REAL
				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(ARRAY, VALUE, DIM, MASK, KIND, BACK)`
	Or
	`FINDLOC(ARRAY, VALUE, MASK , KIND, BACK)`
		3 to 6	`ARRAY VALUE, DIM[, MASK, KIND, BACK]`
			Or
			`ARRAY, VALUE[, MASK, KIND, BACK]`
				INTEGER
				ARRAY

`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

Table 5-17 Fortran 2008 functions (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`NORM2`	Euclidean vector norm. **Syntax:** `NORM2(X[, DIM])` Where: * X shall be a REAL ARRAY. * DIM is an INTEGER SCALAR with a value in the range of 1 to n (where n is the rank of X). **Note** The current implementation experiences overflow for arguments containing elements whose square is at the boundary value for double-precision floating-point numbers. There is no such overflow for single-precision arguments.	1[, or 2]	REAL ARRAY[, INTEGER SCALAR]	The result is the same type as X. If DIM is not present, the result is SCALAR. If DIM is present, the result has rank n−1 and shape [d1,d2,…,dDIM−1,DIM +1,…,dn], where n is the rank of X, and [d1,d2,…,dn] is the shape of X.
`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

5.2.17 Unsupported functions

Unsupported Fortran 2008 functions:

Table 5-18 Unsupported functions

Intrinsic	Description	Num. of Arguments	Argument Type	Result

`ACOSH`
`ASINH`
`ATANH`
	Inverse hyperbolic trigonometric functions.	1	COMPLEX	COMPLEX

`BGE`	Bitwise greater than or equal to.	2	INTEGER, INTEGER	LOGICAL
`BGT`	Bitwise greater than.	2	INTEGER, INTEGER	LOGICAL
`BLE`	Bitwise less than or equal to.	2	INTEGER, INTEGER	LOGICAL
`BLT`	Bitwise less than.	2	INTEGER, INTEGER	LOGICAL
`DSHIFTL`	Combined left shift.	3	INTEGER or BOZ constant, INTEGER or BOZ constant, INTEGER	INTEGER
`DSHIFTR`	Combined right shift.	3	INTEGER or BOZ constant, INTEGER or BOZ constant, INTEGER	INTEGER”
`IALL`	Bitwise `AND` of array elements.	1	ARRAY	ARRAY
`IANY`	Bitwise `OR` of array elements.	1	ARRAY	ARRAY

`IPARITY`	Bitwise `XOR` of array elements.
	Syntax:
	`IALL(ARRAY[, DIM[, MASK]])`
	`IANY(ARRAY[, DIM[, MASK]])`
	`IPARITY(ARRAY[, DIM[, MASK]])`
		1	ARRAY	ARRAY

`IMAGE_INDEX`	Co-subscript to image index conversion.	2	COARRAY, INTEGER	INTEGER
`NUM_IMAGES`	Number of images.	0, 1, or 2	None, INTEGER, or INTEGER, LOGICAL	INTEGER
`THIS_IMAGE`	Co-subscript index of this image.	0, 1, or 2	None, INTEGER, INTEGER or COARRAY, INTEGER	INTEGER
`LCOBOUND`	Lower co-dimension of bounds of an array.	1	COARRAY	INTEGER

`UCOBOUND`	Upper co-dimension of bounds of an array.
	Syntax:
	`LCOBOUND(COARRAY[, DIM[, KIND]])`
	`UCOBOUND(COARRAY[, DIM[, KIND]])`
		1	COARRAY	INTEGER

`MASKL`	Left justified mask.	1[, or 2]	INTEGER[, INTEGER]	INTEGER

`MASKR`	Right justified mask.
	Syntax:
	`MASKL(I[, KIND])`
	`MASKR(I[, KIND])`
		1[, or 2]	INTEGER[, INTEGER]	INTEGER

Table 5-18 Unsupported functions (continued)

Intrinsic	Description	Num. of Arguments	Argument Type	Result
`MERGE_BITS`	Merge of bits under mask.	3	INTEGER, INTEGER, INTEGER	INTEGER
`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

5.2.18 Unsupported subroutines

Unsupported Fortran 2008 subroutines:

Table 5-19 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

Related references

5.2.1 Fortran intrinsics overview on page 5-121
5.2.2 Bit manipulation functions and subroutines on page 5-121
5.2.3 Elemental character and logical functions on page 5-122
5.2.4 Vector/Matrix functions on page 5-124
5.2.5 Array reduction functions on page 5-124
5.2.6 String construction functions on page 5-126
5.2.7 Array construction manipulation functions on page 5-126
5.2.8 General inquiry functions on page 5-127
5.2.9 Numeric inquiry functions on page 5-127
5.2.10 Array inquiry functions on page 5-128
5.2.11 Transfer functions on page 5-128

5.2.12 Arithmetic functions on page 5-129
5.2.13 Miscellaneous functions on page 5-132
5.2.14 Subroutines on page 5-132
5.2.15 Fortran 2003 functions on page 5-133
5.2.16 Fortran 2008 functions on page 5-133
5.2.17 Unsupported functions on page 5-135
5.2.18 Unsupported subroutines on page 5-137

5.3 Statements

Describes the Fortran statements that are supported in|fortrancompiler|.

The Fortran statements that are supported in the Arm Fortran Compiler, are:

Table 5-20 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 might 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.

Table 5-20 Supported Fortran statements (continued)

Statement	Language standard	Brief description
BLOCK	F08	Indicates where a BLOCK construct starts. The BLOCK construct defines an executable block of statements or constructs that can contain declarations. This allows you to declare variables closer to where they are used in your code. **Note** • To retain the status and value of a local variable of a BLOCK construct after the block ends, use the SAVE attribute. • SAVE-ed statements external to a block do not affect the local variables used internally in a block. • Control can not be transferred into a block from outside the block, except when the return is from a procedure call. Transfers in or out of the block are permitted. **Syntax**

```
<optional-name> BLOCK
   <optional-specification-part> ! One or more specification statements
   <statement-block>             ! Zero or more statements or constructs
END BLOCK <optional-name>
```

The following specification statements are not permitted:

- COMMON
- EQUIVALENCE
- IMPLICIT
- INTENT
- NAMELIST
- OPTIONAL
- SUBROUTINE
- VALUE

Statement	Language standard	Brief description
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.
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 might 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. Can be either static or dynamic form. **Note:** This statement has been marked as obsolescent. Obsolescent statements are now redundant and might be removed from future standards. This statement remains supported in the Arm Fortran Compiler.

Table 5-20 Supported Fortran statements (continued)

Statement	Language standard	Brief description
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 might 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 might 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.
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 might 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.

Table 5-20 Supported Fortran statements (continued)

Statement	Language standard	Brief description
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 might be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
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 might be removed from future standards. This statement remains supported in the Arm Fortran Compiler.
ERROR STOP	F2008	Stops the program execution and prevents any further execution of the program. ERROR STOP is similar to STOP, but ERROR STOP indicates that the program terminated in an error condition. **Note:** Also see STOP.
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 might 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.

Table 5-20 Supported Fortran statements (continued)

Statement	Language standard	Brief description
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 might 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, which must be declared in the code of the program unit containing the GOTO statement, and also must be unique in 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 might 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.
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 the intended use of a dummy argument, but can 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 in 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 might 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 can be omitted or that are optional.

Table 5-20 Supported Fortran statements (continued)

Statement	Language standard	Brief description
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.
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, to disallow access to type-bound procedures outside the defining module, a PRIVATE statement can appear after a CONTAINS statement, in a derived type.
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 can 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 might 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.

Table 5-20 Supported Fortran statements (continued)

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. **Note:** Also see ERROR STOP.
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 can 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 that they were introduced in, or the standard that they were last significantly changed.

Related information

WG5 Fortran Standards

Chapter 6
Standards support

This chapter describes the support status of Arm Fortran Compiler with the Fortran language and OpenMP standards.

It contains the following sections:

6.1 Fortran 2003

Details the support status with the Fortran 2003 standard.

Table 6-1 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 that are bound by name to a type	Yes
The PASS attribute	Yes
Procedures that are 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

Table 6-1 Fortran 2003 support (continued)

Fortran 2003 Feature	Support Status
Pointer INTENT	Yes
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

Table 6-1 Fortran 2003 support (continued)

Fortran 2003 Feature	Support Status
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

Note

For more information about the features that are listed in the table above, see *N1648 – ISO/IEC JTC1/SC22/WG5: The new features of Fortran 2003*.

6.2 Fortran 2008

Details the support status with the Fortran 2008 standard.

Table 6-2 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 that is 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	Yes Supports declaring multiple type-bound procedures in a single procedure statement.
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.

Table 6-2 Fortran 2008 support (continued)

Fortran 2008 feature	Support status
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	Yes
Recursive I/O	Yes
Execution control	
The BLOCK construct	Yes
Exit statement	No
Stop code	Yes
ERROR STOP	Yes
Intrinsic procedures for bit processsing	
Bit sequence comparison	No
Combined shifting	No
Counting bits	Yes
Masking bits	No
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. Also, atan2 cannot be accessed through atan.
Bessel functions	Yes
Error and gamma functions	Yes
Euclidean vector norms	Yes The current implementation experiences overflow for arguments containing elements whose square is at the boundary value for double-precision floating-point numbers. There is no such overflow for single-precision arguments.

Table 6-2 Fortran 2008 support (continued)

Fortran 2008 feature	Support status
Parity	No
Execute command line	No
Optional back argument added to maxloc and minloc	Yes
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

Note

For more information about the features that are listed in the table above, see *N1891 – ISO/IEC JTC1/SC22/WG5: The new features of Fortran 2008*.

6.3 OpenMP 4.0

Details the support status with the OpenMP 4.0 standard.

Table 6-3 OpenMP 4.0 support

OpenMP 4.0 Feature	Support
C/C++ Array Sections	N/A
Thread affinity policies	Yes
“simd” construct	Partial **Note:** No clauses are supported. `!$omp simd` can be used to force 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

6.4 OpenMP 4.5

Details the support status with the OpenMP 4.5 standard.

Table 6-4 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 7
Troubleshoot

This chapter describes how to diagnose problems when compiling applications using Arm Fortran Compiler.

It contains the following sections:

7.1 Application segfaults at -Ofast optimization level

A program runs correctly when the binary is built using the `-O3` optimization level, but encounters a runtime crash or segfault with `-Ofast` optimization level.

Condition

The runtime segfault only occurs when `-Ofast` is used to compile the code. The segfault disappears when you add the `-fno-stack-arrays` option to the compile line. .

The -fstack-arrays option is enabled by default at -Ofast

When the `-fstack-arrays` option is enabled, either on its own or enabled with `-Ofast` by default, the compiler allocates arrays for all sizes using the local stack for local and temporary arrays. This helps to improve performance, because it avoids slower heap operations with `malloc()` and `free()`. However, applications that use large arrays might reach the Linux stack-size limit at runtime and produce program segfaults. On typical Linux systems, a default stack-size limit is set, such as 8192 kilobytes. You can adjust this default stack-size limit to a suitable value.

Solution

Use `-Ofast -fno-stack-arrays` instead. The combination of `-Ofast -fno-stack-arrays` disables automatic arrays on the local stack, and keeps all other `-Ofast` optimizations. Alternatively, to set the stack so that it is larger than the default size, call `ulimit -s unlimited` before running the program.

If you continue to experience problems, *Contact Arm Support*.

7.2 Compiling with the -fpic option fails when using GCC compilers

Describes the difference between the `-fpic` and `-fPIC` options when compiling for Arm with GCC and Arm Compiler for Linux.

Condition

Failure can occur at the linking stage when building Position-Independent Code (PIC) on AArch64 using the lower-case `-fpic` compiler option with GCC compilers (gfortran, gcc, g++), in preference to using the upper-case `-fPIC` option.

Note

- This issue does not occur when using the `-fpic` option with Arm Compiler for Linux (`armflang`/`armclang`/`armclang++`), and it also does not occur on x86_64 because -fpic operates the same as `-fPIC`.
- PIC is code which is suitable for shared libraries.

Cause

Using the `-fpic` compiler option with GCC compilers on AArch64 causes the compiler to generate one less instruction per address computation in the code, and can provide code size and performance benefits. However, it also sets a limit of 32k for the Global Offset Table (GOT), and the build can fail at the executable linking stage because the GOT overflows.

Note

When building PIC with Arm Compiler for Linux on AArch64, or building PIC on x86_64, `-fpic` does not set a limit for the GOT, and this issue does not occur.

Solution

Consider using the `-fPIC` compiler option with GCC compilers on AArch64, because it ensures that the size of the GOT for a dynamically linked executable will be large enough to allow the entries to be resolved by the dynamic loader.

7.3 Error messages when installing Arm® Compiler for Linux

If you experience a problem when installing Arm Compiler for Linux, consider the following points.

- To perform a system-wide install, ensure that you have the correct permissions. If you do not have the correct permissions, the following errors are returned:
  - Systems using RPM Package Manager (RPM):

    ```
    error: can't create transaction lock on /var/lib/rpm/.rpm.lock (Permission denied)
    ```
  - Debian systems using dpkg:

    ```
    dpkg: error: requested operation requires superuser privilege
    ```
- If you install using the `--install-to <directory>` option, ensure that the system you are installing on has the required rpm or dpkg binaries installed. If it does not, the following errors are returned:
  - Systems using RPM Package Manager (RPM):

    ```
    Cannot find 'rpm' on your PATH. Unable to extract .rpm files.
    ```
  - Debian systems using dpkg:

    ```
    Cannot find 'dpkg' on your PATH. Unable to extract .deb files.
    ```

7.4 Error moving Arm® Compiler for Linux modulefiles

Describes a workaround to use if you move Arm Compiler for Linux environment modulefiles.

Condition

Affected: Arm Compiler for Linux 20.3 and prior releases.

Moving Arm® Compiler for Linux modulefiles causes them to stop working

Moving Arm Compiler for Linux modulefiles after they are installed causes Arm Compiler for Linux to stop working.

By default, Arm Compiler for Linux modulefiles are configured to find the Arm Compiler for Linux binaries at a location that is relative to the modulefiles. Moving or copying the modulefiles to a new location means that the installed binaries are no longer at the same relative location to the new modulefile location. When trying to locate binaries, the broken relative links between the new modulefile location and the location of the installed binaries causes the new modulefiles to fail.

Workaround

The workaround is to update the configuration of the modulefiles at the new location so that the new modulefiles point to the location of the installed Arm Compiler for Linux binaries.

1. Copy the `modulefiles` directory to a new location. For example:

   ```
   cp -r /opt/arm/modulefiles/ <path/to/new-location>/modulefiles
   ```

   Here, `<path/to/new-location>` represents the path to the new location that you have copied the `modulefiles` directory to.
2. For each modulefile in the new location, update the `package_prefix` line in that modulefile. The `package_prefix` line must point to the location of the original Arm Compiler for Linux installation so that the installed binaries can be correctly located.

   The path to the original Arm Compiler for Linux installation is represented by `<path/to/install-dir>`

   Change:

   ```
   set package_prefix     $root_prefix/<modulefile>
   ```

   to:

   ```
   set package_prefix     <path/to/install-dir>/<modulefile>
   ```

   Note

   `$root_prefix` is a variable that is defined and used in each Arm Compiler for Linux modulefile. `$root_prefix` is set to be the path to the `modulefiles` directory to which the modulefiles for Arm Compiler for Linux were originally installed.

   For example, for a default installation, `$root_prefix` is set to `/opt/arm/modulefiles`. If, at installation, you specified a custom install location, `$root_prefix` is set as the path to the `modulefiles` directory at the custom location your specified during installation.

   To change the `package_prefix` line in one step for all the modulefiles in a directory, you can use a tool like sed, for example:

   ```
   cd <path/to/new-location>
   find . -type f -print0 | xargs -0 sed -ri 's/set package_prefix          $root_prefix/set
   package_prefix          <path\/to\/install-dir>/g'
   ```

   replacing `<path/to/install-dir>` with the path to your Arm Compiler for Linux installation directory (`/opt/arm/modulefiles` for a default installation).

Related information

Arm Allinea Studio installation instructions