Compiler & Linker Fundamentals for Arm Embedded Systems

Course 3 Cortex-M Software Development Fundamentals

Learning objectives

At the end of this module you will be able to:

- Choose a suitable compiler
- Discuss the purpose of registers as described by the Arm Application Procedure Call Standard (AAPCS)
- Select the compiler options suitable for your target project
- Mix C/assembly code
- Customize the application to target your system’s memory map
- Analyze diagnostics, warnings and errors
- Generate an image that can run on your target

Agenda

Compilers support for Arm Embedded Systems

Coding Considerations

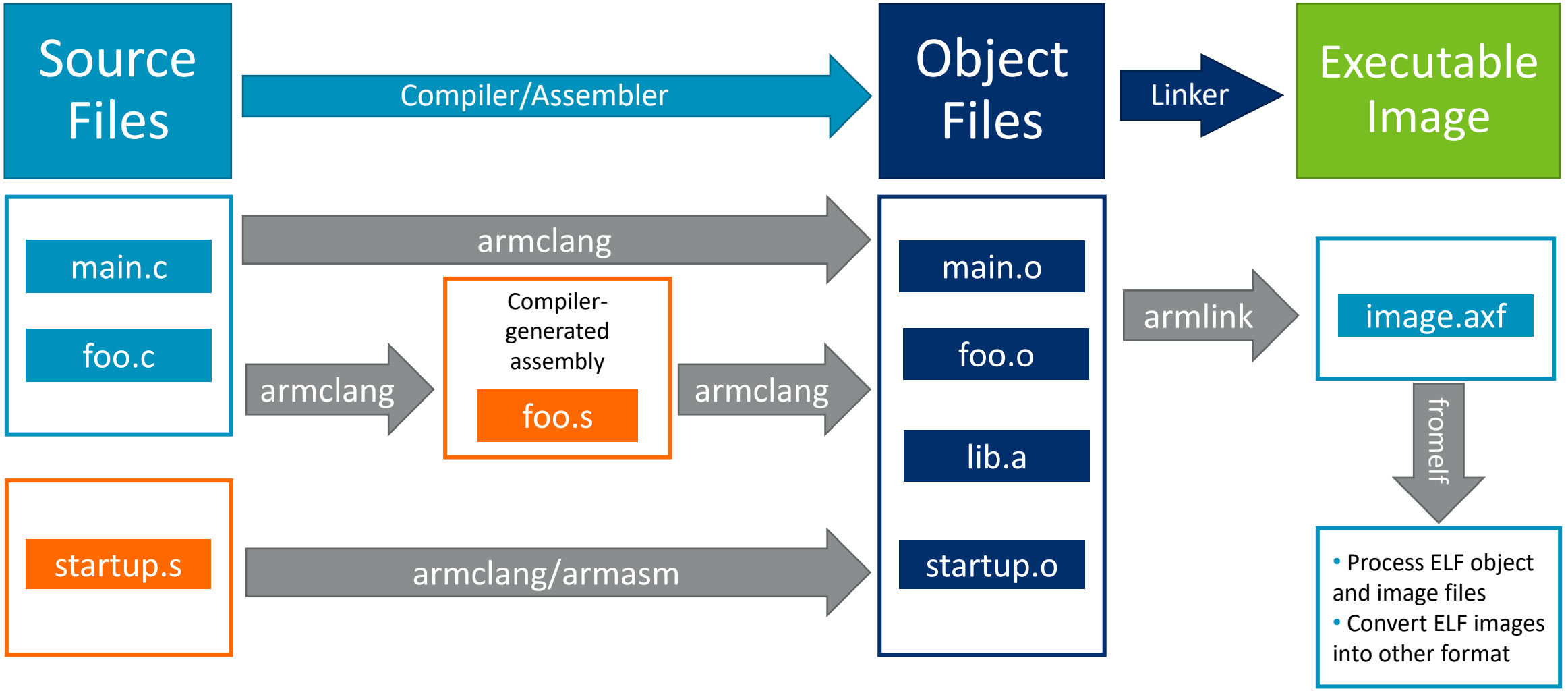
Compiler Usage

Mixing C/C++ with Assembly

Linker and Libraries

Arm Compiler Troubleshooting

General process to build a project



Compiler Support for the Arm Architecture

Different compilers have different support for the Arm architecture and Arm processors

- Arm Compiler 6
  - All A-profile, R-profile and M-profile variants
  - Focused on bare-metal software development
- The legacy Arm Compiler 5 supports Arm architectures from Armv4 to Armv7 inclusive

Open source compilers, such as GCC and third-party tool vendor compilers, also offer support the Arm Architecture

- https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm

Arm compilers should comply to the ABI for the Arm Architecture

Compiler support for Functional Safety

Some projects have functional safety (FuSa) requirements

Using an Arm compiler with long-term maintenance (LTM) simplifies and accelerates the development of safety-qualified systems

Arm periodically selects stable versions of Arm Compiler, as a basis for long-term support

- Arm Compiler 6.6.x
- Certain updates of Arm Compiler 5.06 and Arm Compiler 5.04

Arm Compiler versions with LTM are supported by an Arm Compiler Qualification Kit

- Safety manual
- Development process document
- Test and defect reports
- TÜV SÜD certificate with assessment reports
- Long-term technical support contract option

Introduction to Arm Compiler

Arm Compiler 6 (armclang) is based on LLVM Clang

- Higher level of compliance with OpenSource world
- More standardization and less fragmentation leading to ease of source code portability

Where can I get Arm Compiler?

- Arm DS/Keil-MDK integrates the latest version of Arm Compiler 6 (and Arm Compiler 5)
- Arm Compiler versions can be downloaded from https://developer.arm.com/tools-and-software/embedded/arm-compiler/downloads/version-6
- Arm Compiler requires a license

How do I use Arm Compiler?

- Compile the project in an IDE
  - Setting the compile options in the project properties
- Compile the project on the command-line
  - May need to set the environment variables beforehand

Which Arm Compiler version is recommended?

- For new projects / Armv8 architecture projects, Arm Compiler 6 is recommended
- For safety-critical code, Arm Compiler 6 for functional safety is recommended (stand alone license)

Toolchain differences between AC5 and AC6

Arm Compiler 6 uses the compiler tool armclang instead of armcc in Arm Compiler 5

- The command-line options for armclang are different to the command-line options for armcc

Arm Compiler 6 supports GNU assembly syntax

- Arm Compiler 6 includes an armclang integrated assembler to assemble assembly language source files written in GNU assembly syntax
- Arm Compiler 6 still includes armasm to support the legacy syntax
- Arm Compiler 5 supports armasm assembly syntax only

When migrating from AC5 to AC6, take care of the following

- Differences in the command-line options when invoking the compiler
- Differences in the adherence to language standards
- Differences in compiler specific keywords, attributes, and pragmas
- Differences in optimization and diagnostic behavior of the compiler

Further help is available in the Arm Compiler Migration & Compatibility Guide

- https://developer.arm.com/tools-and-software/embedded/arm-compiler/documentation

Arm Compiler support level definitions

Product features

- Full supported product features
- Beta product features
  - Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments
- Alpha product features
  - Alpha product features are not implementation complete, and are subject to change in future releases

Community features

- Arm makes no claims about community features, except when explicitly stated in the documentation
- Functionality might change significantly between feature releases
- Arm makes no guarantees that community features will remain functional across update releases

Unsupported features

- Specific features and use-cases are known not to function correctly, or are not intended for use with Arm Compiler 6

Language support

Source language modes in Arm Compiler 6.15

C		C++	
c90	gnu90	c++98	gnu++98
c99	gnu99	c++03	gnu++03
c11*	gnu11*	c++11	gnu++11
		c++14	gnu++14
		c++17*	gnu++17*

* Community features

Use `-std=<name>` to select one of the above language features

Default C++ standard is `gnu++14`

Default C standard is `gnu11`

For strict conformity to the standard use the `-Wpedantic` compiler option

Variable types supported for M-profile

The compiler supports these basic types:

`int / long`	32-bit (word) integer
`short`	16-bit (half-word) integer
`char`	8-bit byte, unsigned by default
`long long`	64-bit integer
`float`	32-bit single-precision IEEE floating point
`double`	64-bit double-precision IEEE floating point
`bool`	8-bit Boolean (C++ only)
`wchar_t`	32-bit “wide character” type (C++ only)
Pointers	32-bit integer addresses

Take care when porting legacy code from other vendors’ architectures

C++ support

C++ exception handling is on by default

- To disable exceptions, use `-fno-exceptions`

Support for libc++ with `-fno-exceptions` is limited

- Pre-compiled binaries of libc++ may throw exceptions, resulting in program termination
- Only the parts of libc++ that have been implemented in header files can be used with `-fno-exceptions`

The C++ initialization sequence is defined by the ABI

- Constructors handled by `__cpp_initialize_aeabi_`
- As part of this, destructors are registered using `__aeabi_atexit`

The `new` and `delete` operators call `malloc()` and `free()` functions

- Retarget `malloc()/free()` to provide your own heap support, not the `new/delete` operators

The C++ library is not thread-safe

- The `<thread>` library implementations within libc++ are not supported

Agenda

Compilers support for Arm Embedded Systems

Coding Considerations

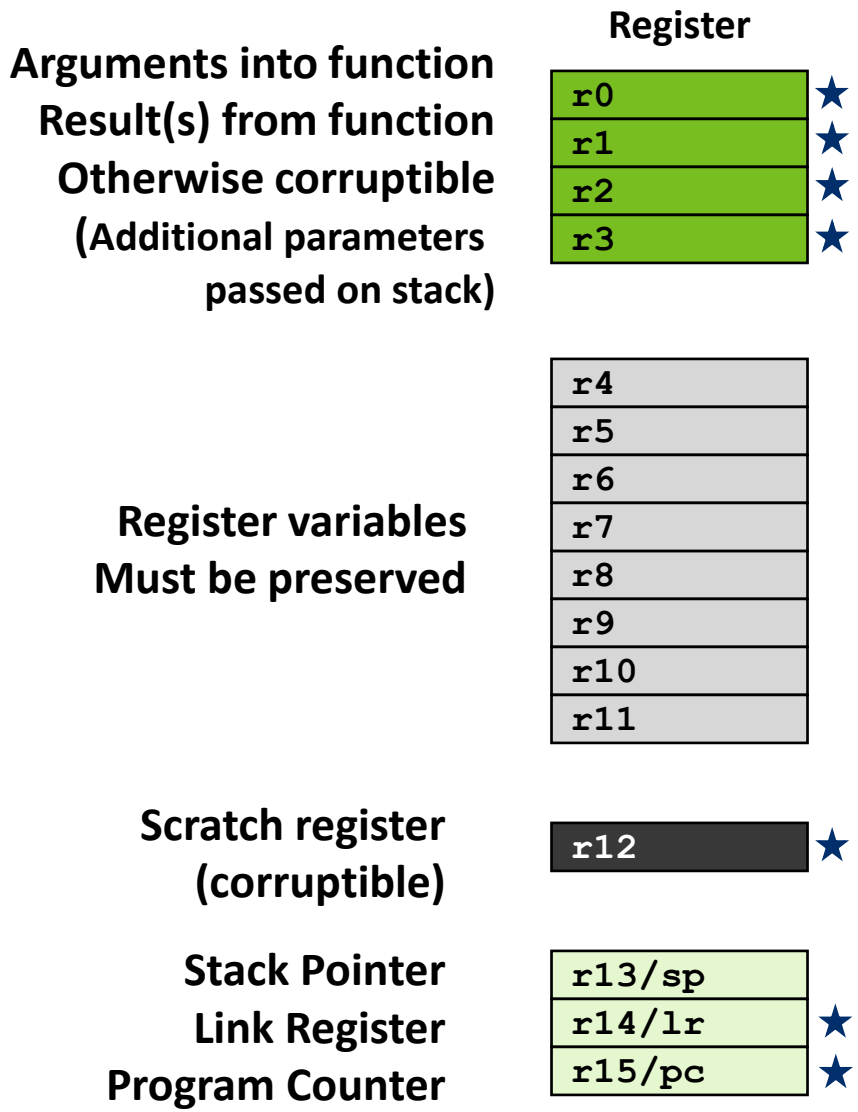
Compiler Usage

Mixing C/C++ with Assembly

Linker and Libraries

Arm Compiler Troubleshooting

Procedure Call Standard for the Arm Architecture (1)



The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see AAPCS)

xPSR flags may be corrupted by function call

Assembler code which links with compiled code must follow the AAPCS at external interfaces

The AAPCS is part of the ABI for the Arm Architecture

AAPCS requires that SP be 8-byte (2 word) aligned at externally visible boundaries

r14 can be used as a temporary register once value stacked

In M-profile systems, registers marked with a star ★ are automatically pushed on to the stack when an exception occurs

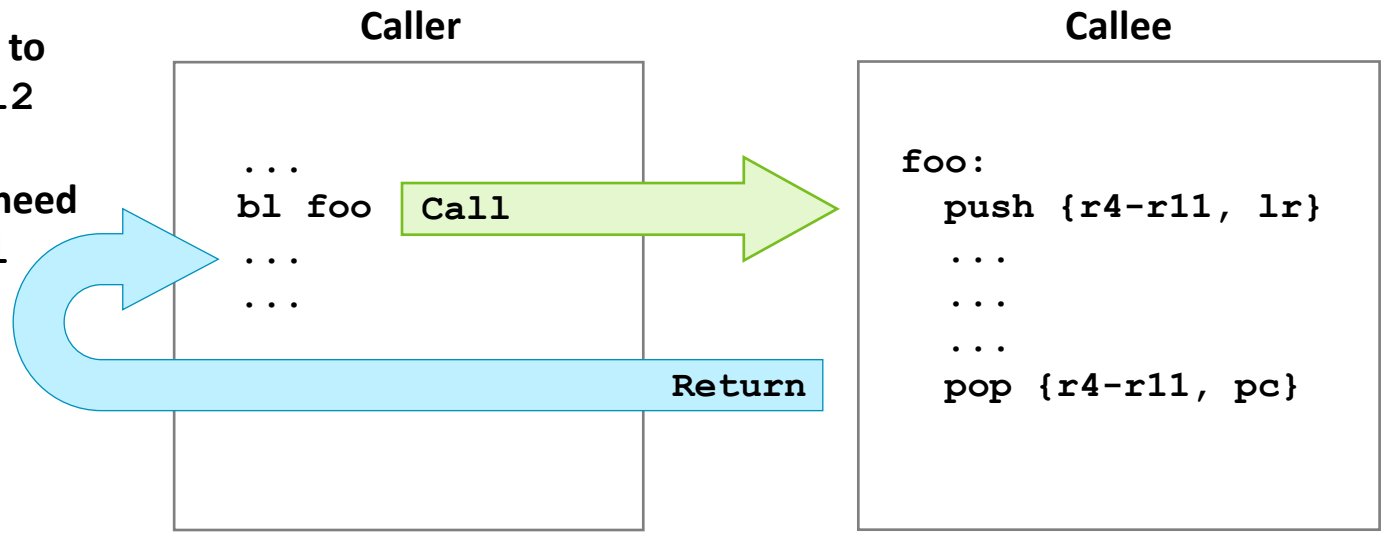
The xPSR ★ (processor state) is also pushed to the stack

Procedure Call Standard for the Arm Architecture (2)

Parameters passed in `r0 - r3`

Caller may need to save `r0 - r3, r12`

Caller does not need to save `r4 - r11`



Callee must preserve `r4 - r11, lr` if used by callee

Callee may corrupt `r0 - r3, r12`

Value returned in `r0` for `int` / `short` / `char` ;
in `r0` and `r1` for `long long`

AAPCS – Procedure Call Standard for the Arm Architecture

Parameters passing (1)

The first four word-sized parameters passed to a function will be transferred in registers r0-r3 (fast & efficient). If more arguments are needed, then the 5th, 6th and subsequent words will be passed on the stack, involves extra instructions and memory accesses

- Sub-word sized arguments will still use a whole register
- 64-bit arguments to functions will be passed in multiple registers or memory
  - must be passed in an even + consecutive odd register (i.e. r0+r1 or r2+r3)
  - 64-bit types must be 8-byte aligned in memory
- See AAPCS for more details

Therefore always try to limit arguments to 4 words or fewer

- If unavoidable, place most commonly used parameters in first 4 positions
- Or if arguments are in a structure then pass a pointer to the structure instead

C++ uses the first argument to pass the *this* pointer to non-static member functions, so at most 3 other words can be passed in registers

Parameter passing (2)

Examples compiled with

`-march=Armv8-m.base -O2`

4 parameters, no stack usage

```
int func1(int a, int b, int c, int d) {
    return a+b+c+d;
}
```

```
int caller1(void) {
    return func1(1, 2, 3, 4);
}
```

```
func1:
        .fnstart
        adds    r0, r1, r0
        adds    r0, r0, r2
        adds    r0, r0, r3
        bx      lr
```

```
caller1:
        .fnstart
        movs    r0, #1
        movs    r1, #2
        movs    r2, #3
        movs    r3, #4
        b       func1
```

6 parameters, stack used for 5th and 6th parameters

```
int func2(int a, int b, int c, int d, int e, int f) {
    return a+b+c+d+e+f;
}
```

```
int caller2(void) {
    return func2(1, 2, 3, 4, 5, 6);
}
```

```
func2:
        .fnstart
        adds    r0, r1, r0
        adds    r0, r0, r2
        adds    r0, r0, r3
        ldr     r1, [sp]
        adds    r0, r0, r1
        ldr     r1, [sp, #4]
        adds    r0, r0, r1
        bx      lr
```

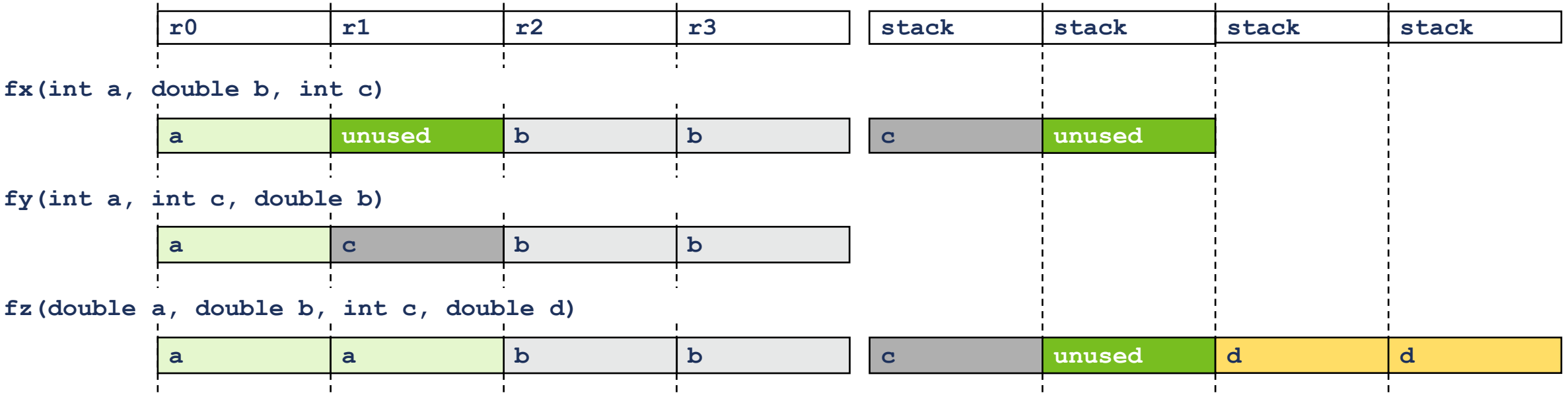
```
caller2:
        .fnstart
        push    {r7, lr}
        sub     sp, #8
        movs    r0, #6
        movs    r1, #5
        str     r1, [sp, #0]
        str     r0, [sp, #4]
        movs    r0, #1
        movs    r1, #2
        movs    r2, #3
        movs    r3, #4
        bl      func2
        add     sp, sp, #8
        pop     {r7, pc}
```

Parameter passing (3)

The AAPCS has rules about 64-bit types

- 64-bit types must be 8-byte aligned in memory
- 64-bit arguments to functions must be passed in an even + consecutive odd register (i.e. r0+r1 or r2+r3) or on the stack at an 8-byte aligned location

Registers or stack will be ‘wasted’ if arguments are listed in a sub-optimal order



Remember the hidden *this* argument in r0 for non-static C++ member functions

Agenda

Compilers support for Arm Embedded Systems

Coding Considerations

Compiler Usage

Mixing C/C++ with Assembly

Linker and Libraries

Arm Compiler Troubleshooting

Selecting compiler options

Toolchains like Arm Compiler 6 provide a wide range of options for different purposes:

Description	Example(s)
Targeting processors and architectures	-mcpu=cortex-m55+mve.fp+fp.dp
Floating-point support	-mfpu=*name* -mfloat-abi
Source language selection	-std=c99
Optimizations	-Oz
Code generation	-mno-unaligned-access, -fropi
Security/safety	-mcmse, -fstack-protector
Diagnostics	-pedantic

Example: build an application for an Armv8-M Mainline based processor with FPv5-D16 architecture, use hardware floating-point linkage, disable unaligned data accesses, and optimize for code size

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main -mfloat-abi=hard
-mfpu=fpv5-sp-d16 –mno-unaligned-access –Os -c main.c –o main.o
```

Optimization levels

Option	Description
`-O0` (default)	Minimum optimization with best debug view
`-O1`	Restricted optimization with good debug view
`-O2`	High optimization with limited debug view
`-O3`	Maximum optimization with very limited debug view
`-Ofast`	• Enables all the optimizations from level 3 • Enables optimizations performed with `-ffp-mode=fast`
`-Omax`	• Maximum optimization, enables all the optimizations from level fast • Automatically enables `-flto` • Caution: not fully standards-compliant for all code cases
`-Os`	Performs optimizations to reduce code size, balancing code size against code speed
`-Oz`	Performs optimizations to minimize image size
`-Omin`	• Enable all the optimizations from level `-Oz` • A basic set of link-time optimizations aimed at removing unused code and data • Unused virtual function elimination. This is a particular benefit to C++ users.

Use `-g`, `-gdwarf-4` for generating source level debug info (DWARF2/3 also available)

Optimization algorithms

Various optimizations are performed at different levels, including:

- **Redundant code elimination –** remove code that's not referenced
  - There are ways to prevent the compiler doing this
- **Tail-call optimization** - avoid unnecessary returns in function hierarchies
- **Instruction scheduling** - take advantage of dual-issue capability
- **Idiom recognition** - recognize what operation the programmer intended and generate an optimal solution
- **Inlining of functions** - improve performance at the potential expense of a larger image
  - Whether inlining is enabled is influenced by different factors
- **Loop transformation** - transform and restructure loops automatically
- **Link Time Optimization** - with single step optimization performs better static code path analysis
  - Enabled with `-flto`

Community features, `-mllvm` and `-debug-pass=Arguments`, can be used to obtain a list of all optimization methods performed by armclang for a specific optimization level

- ```
  armclang --target=arm-arm-none-eabi -mcpu=cortex-r52 -c -O1
  -v -mllvm -debug-pass=Arguments test.c
  ```
  ```

Selecting a target

Each Arm architecture variant has its own set of features

- Implementations of an architecture version may vary between cores

For Arm Compiler 6, you must specify a **`--target=option`**, where option is one of

- `arm-arm-none-eabi` generates A32/T32 instructions for AArch32 state
- `aarch64-arm-none-eabi` generates A64 instructions for AArch64 state

You can target a specific architecture version or CPU for the target project

- Specifying an architecture: **`-march=name[+[no]feature+…]`**
  - Allows the linker to select appropriate libraries for the target that take full advantage of the architectural features
  - Target Armv8-M Mainline:
    `armclang --target=arm-arm-none-eabi -march=armv8-m.main`
- Specifying a CPU: **`-mcpu=name[+[no]feature+…]`**
  - Used to enable/disable optional implementation options
  - Can enable further optimizations specific to the design of the implementation, e.g. pipeline-specific optimizations
  - Impacts portability of generated code
  - Target Cortex-M55 without the DSP Extension:
    `armclang --target=arm-arm-none-eabi -mcpu=Cortex-M55+nodsp`

Agenda

Compilers support for Arm Embedded Systems

Coding Considerations

Compiler Usage

Mixing C/C++ with Assembly

Linker and Libraries

Arm Compiler Troubleshooting

Mixing C and assembly

C/C++ and assembly can easily be mixed to:

- Access processor features which are not available from C
- Generate highly optimized code

Easy to make function calls between C, C++ and Assembly

- Make sure to conform to the procedure calling standard…

  …and import and export the relevant symbols

Different ways of Mixing C and assembly

- Calling functions defined in assembly files
- Inline Assembler
- Embedded Assembler
- Intrinsics, libraries and extensions

Calling assembly from C/C++

Define the routine in assembly and export its name

Call directly from C just like any other function

- Provide a function prototype in C
- Disable C++ name mangling with `extern "C"` if using the C++ compiler

Link as normal

```
extern void mystrcopy(char *d, const char *s);

void foo(void)
{
    const char *src = "Source";
    char dest[10];
    mystrcopy(dest, src);
}
```

Function Call

```
    .section  StringCopy, "x"
    .balign   4

    .global   mystrcopy

    .type     mystrcopy, "function"
mystrcopy:
    .func
    ldrb       r2, [r1], #1
    strb       r2, [r0], #1
    cmp        r2, #0
    bne        mystrcopy
    bx         lr
```

Inline assembly

Uses GNU inline assembler syntax

To use C variables in inline assembler,
specify them as input and output variables.
Maybe optimized by compiler.

To prevent the compiler
from corrupting registers used in assembly code,
add them to the clobbered list

Designed to access features that C cannot,
rather than for hand-coding

```
int asm_readbytes(int *a) {
    int v;
    __asm(
        "  ldrb %[out], [%[in]]\n"
        "  ldrb r1, [%[in], #1]\n"
        "  ldrb r2, [%[in], #2]\n"
        "  ldrb r3, [%[in], #3]\n"
        "  orr %[out], r1, lsl #8\n"
        "  orr %[out], r2, lsl #16\n"
        "  orr %[out], r3, lsl #24\n"
        : [out] "=&r" (v)    //output variables
        : [in] "r" (*a)      //input variabes
        : "r1", "r2", "r3"   //clobbered list
    );
    return v;
}
```

Embedded assembler

- To mark a function as an embedded assembler function, use `__attribute__((naked))`
- The compiler does not generate prologue and epilogue sequences for embedded assembler functions.
- Arguments will be passed in registers, loosely integrated into the compiler, not checked by the compiler.
- Embedded assembler functions cannot be inlined, either implicitly or explicitly.

```
__attribute__((naked)) int foo(int a, int b,
                               int c, int d,
                               int x)
{
    __asm(
        "LDR r12,[sp, #0]\n"
        "MLA r0,r12,r0,r1\n"
        "MLA r0,r0,r12,r2\n"
        "MLA r0,r0,r12,r3\n"
        "BX lr"
    );
}
```

Intrinsics, libraries and extensions

C/C++ standards do not define core-specific functionality

- The Arm Compiler intrinsics provide extra features to realize these operations

CMSIS: Cortex Microcontroller Software Interface Standard

- Allows Arm architecture-based microcontroller vendors to abstract out implementation specific operations by means of a standardized API
- Example: function to set interrupt priority mask, `void __set_PRIMASK(uint32_t priMask)`

ACLE: Arm C Language Extensions

- Adds the ability to perform Arm architecture specific features and functionality by means of special keywords and intrinsics
- The special register intrinsics, `XPSR`, `MSP`, `PSP`, `PRIMASK`, `CONTROL`

CMSE: Cortex Microcontroller Security Extensions

- Adds compiler support for the Armv8-M Security extension. Like CMSIS, this API is common across different toolchains
- New instructions: `SG`, `BXNS`, `BLXNS`, `TT`

Agenda

Compilers support for Arm Embedded Systems

Coding Considerations

Compiler Usage

Mixing C/C++ with Assembly

Linker and Libraries

Arm Compiler Troubleshooting

How does the linker know what to do?

The linker uses several inputs to decide what to do

- Command line
  - List of object files and user library files
  - Output file name
  - Other options, for example diagnostic information
- Description of the memory map
  - Command line options for simple images, scatter file for complex images
- Object files
  - Symbol table – contains information on what variables/functions are in the object file (definitions) and required (references) by the object file
  - Relocation information – informs the linker where it needs to fill in address information

For a link step to succeed it must match a single symbol definition to every reference

```
armlink object1.o lib1.a --info=inline -scatter=memory.scat -o image.axf
```

Object file | Library file | Diagnostic Info | Scatter file | Output file

Linking for a specific target

`--cpu` linker switch

- Enables generation of objects/images for a specific Arm core or architecture
- Use --cpu=list to list the architecture and processor names that are supported by this option

Without `--cpu` switch, the linker selects the ‘max’ architecture among all objects

- Allows the linker to select the suitable system libraries when building image, by referring to the objects
- e.g. Armv7-M object + Armv8-M object results in an Armv8-M image

With `--cpu` switch, linking fails if any of the object files relies on features that are incompatible with the selected core/architecture

- A guard to the third-party objects/libraries
- e.g. linking Armv7-A objects with `--cpu=8-M.Main` will result in linker error

Linker Optimizations

`--cpu` enables generation of objects/images for a specific ARM core or architecture

- Without the `--cpu` switch, the linker selects the 'max' architecture among all objects

The linker will remove unused sections to reduce image size. To preserve sections:

- Use `--keep` linker option
- Reference a symbol in that section (e.g. branch to a function)
- Mark a section with `ENTRY`, `+FIRST`/`+LAST`

To reduce image size the linker can compress RW data

- The linker analyses the RW data in an image to see if it can benefit from RW data compression

The linker can inline small functions in place of the branch instruction

- Enabled by **`--inline`**
- Linker can inline either one 32-bit or two 16-bit instructions

Link Time Optimization

Link Time Optimization (LTO) is form of inter-procedural optimization performed at the time of linking a program

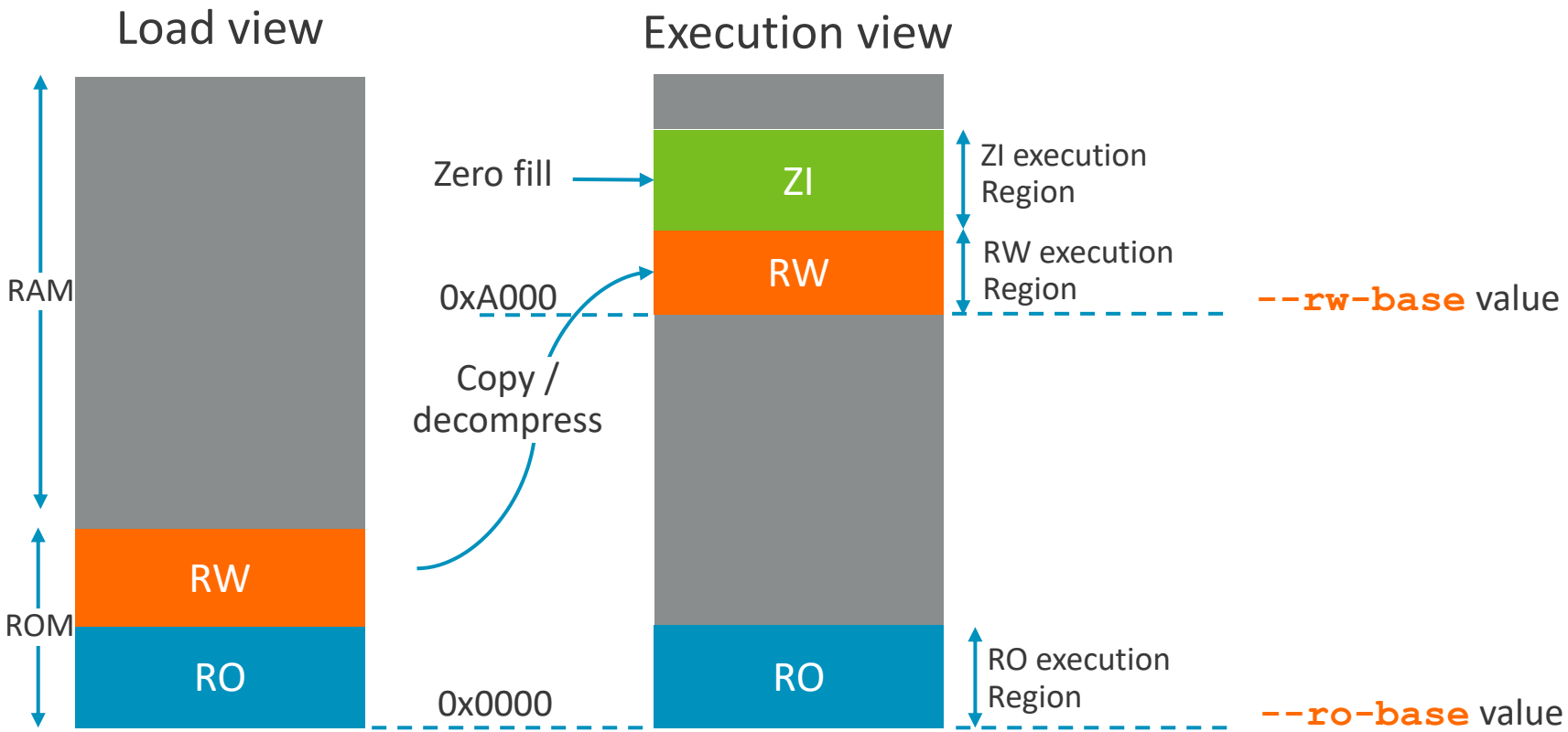
- More complete and powerful than Whole Program Optimization and multi-file compilation
- Tighter integration between compiler and linker””

Enable LTO with compiler option **`-flto`** and linker option **`--lto`**

- `armclang --target=arm-arm-none-eabi -flto -c foo.c -o foo.bc -march=armv8-m.main`
- `armclang --target=arm-arm-none-eabi -flto -c lto.c -o lto.bc -march=armv8-m.main`
- `armlink --lto foo.bc lto.bc -o lto.axf --entry=lto --cpu=8-M.Main`

Targeting your system's memory map

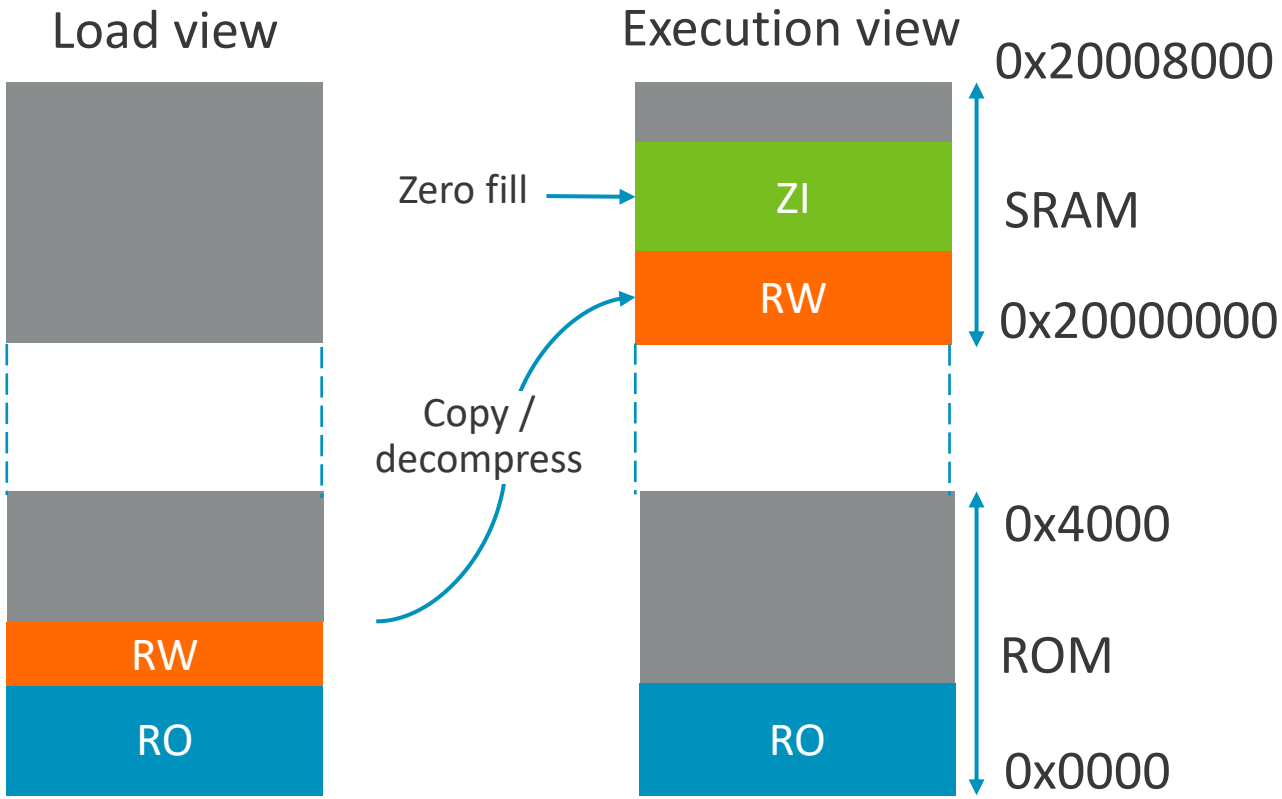
A simple image consists of a number of input sections of type RO, RW, XO, and ZI,
The linker collates the input sections to form the RO, RW, XO, and ZI output sections




```
armlink --cpu=8-M.Main --ro_base=0x0 --rw_base=0xA000
```

Scatter-loading mechanism and scatter-loading files

Scatter-loading file can control the grouping and placement of image components, like linker script in GCC describe the mappings of input files and output file, and to control the memory layout of the output file.




```
LOAD_ROM 0x0000 0x4000
{
   EXEC_ROM 0x0000 0x4000
   {
      *  (+RO)
   }
   SRAM 0x20000000 0x8000
   {
      *  (+RW, +ZI)
   }
}
```

Use a scatter-loading file for specific control

Place code and data at a specific address using a scatter file

- `__attribute__((section("foo")))` specifies that the global variable `gSquared` is to be placed in a section called `foo`

```
#include <stdio.h>
extern int sqr(int n1);
// Place in section foo
int gSquared __attribute__((section("foo")));
int main(void)
{
  gSquared=sqr(3);
  printf("Value squared is: %d\n", gSquared);
  return 0;
}
```

- Source file `function.c` containing the following code

```
int sqr(int n1)
{
  return n1*n1;
}
```

- Scatter file specifies that the section `foo` is to be placed in the `ER3` execution region

```
LR1 0x0000 0x20000
{
  ER1 0x0 0x2000
  {
    ; other read-only code+data
    *(+RO)
  }
  ER2 0x8000 0x2000
  {
    main.o
  }
  ER3 0x10000 0x2000
  {
    function.o
    *(foo) ; Place gSquared in ER3
  }
  ; RW and ZI data to be
  ; placed at 0x200000
  RAM 0x200000 (0x1FF00-0x2000)
  {
    *(+RW, +ZI)
  }
  ARM_LIB_STACK 0x800000 EMPTY -0x10000
  {
  }
  ARM_LIB_HEAP +0 EMPTY 0x10000
  {
  }
}
```

Arm Supplied Libraries

The ARM Compiler toolchain contains a number of libraries

- C libraries (ISO C90 compliant) / Microlib (optimized for small code size)
- Libc++ libraries to enable C++11 support
  - Currently Multithreaded support in Arm C++ libraries [ALPHA]
- C++ Runtime libraries
- Maths libraries
- Floating Point Support libraries
  - With hardware and software implementations of Floating-point functions.

User Libraries

Object and library files are treated differently

- All object files specified on the command line will be loaded by the linker
- Objects will only be loaded from libraries if they resolve a reference

Use libraries if you only expect a small amount of the contents to be linked into the image

The Arm Compiler toolchain contains a utility to create and maintain libraries called “armar”

Option	Usage	Example
--create	Create new libraries to hold object files	armar --create library.a object1.o object2.o
-x	Extract object files from existing libraries	armar -x library.a object2.o
-r	Replace existing object files	armar -r -a obj1.o mylib.a obj2.o
--sizes	View the sizes of the contents (code, data) of objects in the library	armar --sizes library.a
--zs	View the library’s symbol table	armar --zs library.a

Library search mechanism

There are multiple variants of each library for different architecture versions and build options including Thumb-only libraries

- The linker will select the most appropriate version of the libraries based on information embedded in the objects that are linked

The linker searches for Arm libraries in the following order:

- At the location specified with the command-line option **`--libpath`**
- In `../lib`, relative to the location of the armlink executable

The linker always searches user libraries before the Arm libraries.

- **`--no_scanlib`** do not search for the default Arm libraries
- **`--library=`*`name`*** uses only those libraries that are specified in the input file list to resolve references

Linker diagnostics

The linker can also generate detailed information on the contents of an image and the operations the linker has performed

`--info <option>` can be used to output useful information

- inline - Lists all functions inlined by the linker
- inputs - Lists the input symbols, objects and libraries
- libraries - Lists the libraries automatically selected by the linker
- sizes - Lists the code and data sizes for each object
- stack - Lists the stack usage of all local symbols
- totals - Lists the totals of the code and data sizes for input objects
- unused - Lists all unused sections that are eliminated
- veneers - Lists the linker-generated veneers

`--map` produces detailed information on the layout of the image

`--verbose` displays detailed information on the references and definitions in the object files and which object file have been loaded from libraries

Agenda

Compilers support for Arm Embedded Systems

Coding Considerations

Compiler Usage

Mixing C/C++ with Assembly

Linker and Libraries

Arm Compiler Troubleshooting

Arm Compiler Troubleshooting

Errors and warnings

- armclang provides extremely user-friendly diagnostics: https://clang.llvm.org/diagnostics.html
- Errors and warnings for other tool components, such as armlink, are documented in the Arm Compiler Errors and Warnings Reference Guide: https://developer.arm.com/documentation/100074/latest

Help from Arm Community

- https://community.arm.com/developer/tools-software/tools/f/arm-compilers-forum

Further help can be obtained from your support provider

- Provide a reproducible test case that includes:
  - Preprocessed source file: armclang <options> -E sourcefile.c > pp_sourcefile.c
  - Command-line options

Appendix

Selecting a target – GNU Toolchain

Choose GNU Toolchain for Arm processors

- GNU toolchain for embedded processors
  - Bare-metal development
  - Support for 32-bit Arm processors

- GNU toolchain for the A-profile processors
  - Application and Linux kernel development
  - Support for Arm Cortex-A family

You can target a specific architecture version or CPU for the target project

- Specifying the target ARM architecture: **`-march=name[+[no]extension+…]`**
  - Use **`-march`** for better compatibility
  - **`-mtune`** can be used in conjunction with **`-march`** to perform architecture-independent optimizations

- Specifying the target ARM processor : **`-mcpu=name[+[no]extension+…]`**
  - Use **`-mcpu`** for best performance

Language support – GNU Toolchain

Source language modes in 10.2.1

C		C++	
c89/c90/iso9899:1990	gnu89/gnu90	c++98/c++03	gnu++98/gnu++03
c99/iso9899:1999	gnu99	c++11	gnu++11
c11/iso9899:2011	gnu11	c++14	gnu++14
c17/c18/iso9899:2017	gnu17/gnu18	c++17/c++18	gnu++17/gnu++18
c2x*	gnu2x*	c++20/gnu++2a*	gnu++20/gnu++2a*

* Experimental and incomplete support

- Use `-std=<name>` to select one of the above language feature
- Use `gcc -v --help 2> /dev/null | grep -iv deprecated | grep "std="` to list all the supported languages

Optimization levels – GNU Toolchain

Option	Description
`-O0` (default)	Minimum optimization with best debug view
`-O1`	Restricted optimization with good debug view
`-O2`	High optimization with limited debug view
`-O3`	Maximum optimization with very limited debug view
`-Os`	Performs optimizations to reduce image size
`-Ofast`	Performs optimizations to reduce execution time

Use `-Q --help=optimizers` to find out the exact set of optimizations that are enabled at each level

Linker Script – GNU Toolchain

- Linker script is written in the linker command language
- Every link is controlled by a linker script
- Linker script describes how the sections in the input file should be mapped into the output file
- Also used to control the memory layout pf the output
- User can supply their own linker script by using the ‘-T’ command line option:
  - `ld –o prog –T gcc.ld main.o`

Linker Script vs Scatter File

The main difference between the two linkers is the use of `linker scripts` in GNU ld and `scatter files` in armlink.

Scatter File

```
LOAD_ROM 0x0000 0x4000
{
  EXEC_ROM 0x0000 0x4000
  {
        * (+RO)
  }
  SRAM 0x20000000 0x8000
  {
        * (+RW, +ZI)
  }
  ARM_LIB_HEAP +0 ALIGN 64 EMPTY 0xA0000 {}
  ARM_LIB_STACK +0 ALIGN 64 EMPTY 0x10000 {}
}
```

Linker Script

```
MEMORY
{
  rom (rx) : ORIGIN = 0x0, LENGTH=0x4000
  ram (wx): ORIGIN = 0x20000000, LENGTH = 0x8000
}
SECTIONS
{
. .text : { *(.text) }  > row
  .data : { *(.data) }  > raw
  .bss : { *(.bss) }  >  raw
  .heap (NOLOAD): {. = ALIGN(64); .= . + 0xA0000;}
  .stack (NOLOAD) : {.= . + 4 * 0x4000;}
}
```

Selecting a target – Arm Compiler 5

In Arm Compiler 5, it's essential to use the `–cpu` option so the correct libraries are selected

- Specify an architecture
  - To target the application to run on an Armv7-M implementation:
    ```
    armcc --cpu=7-M main.c
    ```
- Specify a CPU
  - To target the application to run on a Cortex-R4 processor:
    ```
    armcc --cpu=Cortex-R4 main.c
    ```

Optimization levels – Arm Compiler 5

Option	Description
`-O0`	Minimum optimization with best debug view
`-O1`	Restricted optimization with good debug view
`-O2` (default)	High optimization with limited debug view
`-O3`	Maximum optimization with very limited debug view
`-Ospace`	Performs optimizations to reduce image size
`-Otime`	Performs optimizations to reduce execution time

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!