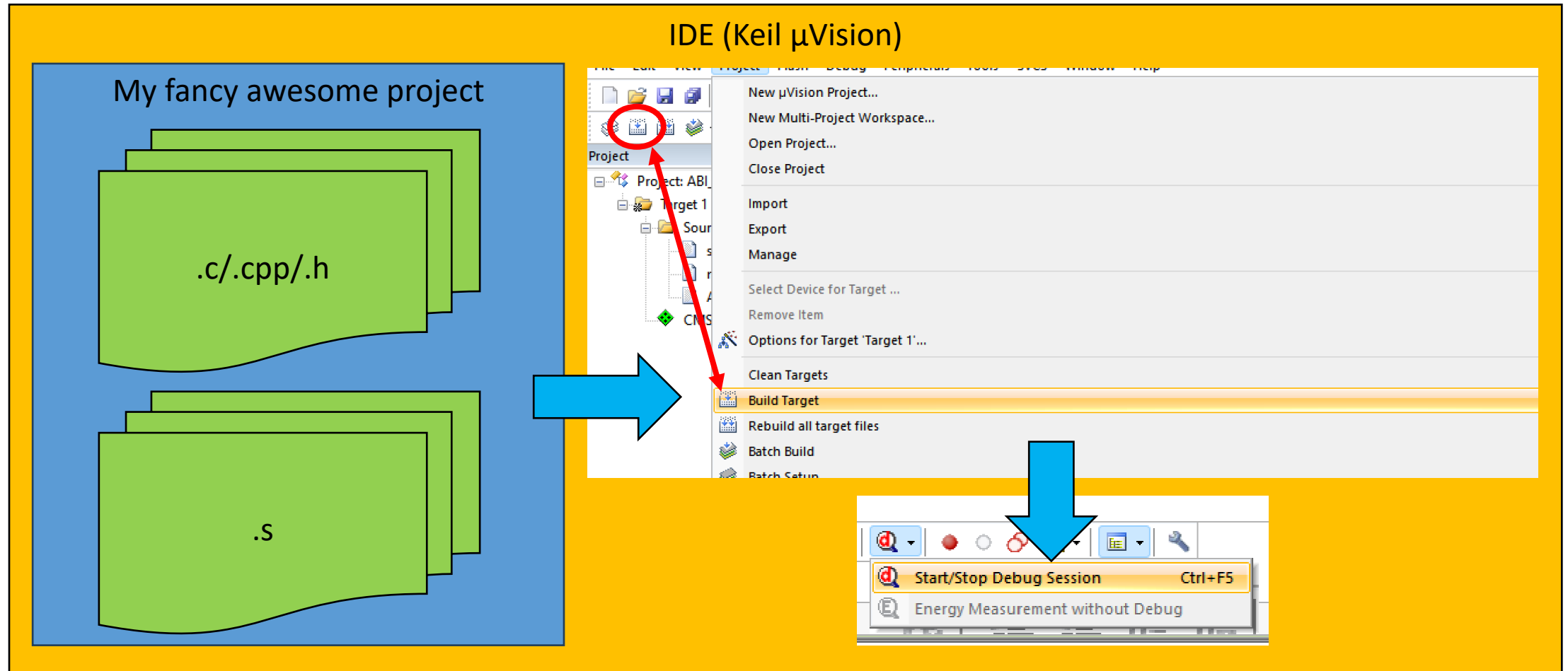


# From Source code to Executable The Arm Toolchain for Embedded Systems

Francesco Angione, Paolo Bernardi

# How is an executable produced from the source code?



# Remainder!!

- Debugging is a very, very, very long painful process.
- Tools, especially the compiler, are your best (and worst) friends!
- The more information you provide to the toolchain, the fewer chances to have different results than the one you have in mind!
- Knowledge of toolchains easily allows you to debug your code.



# Outline

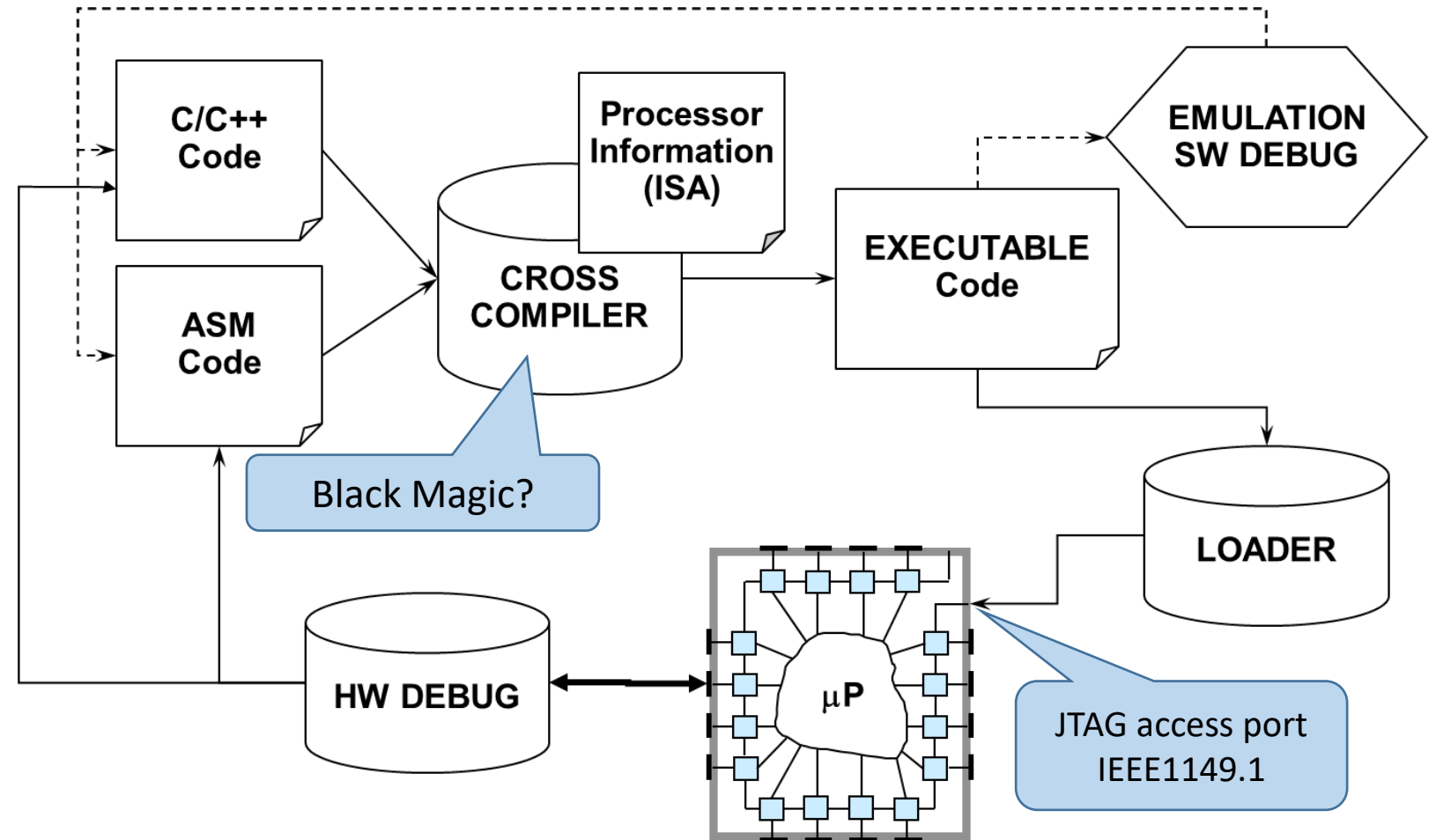
- What is a toolchain?
  - The Arm toolchain.
  - Investigating the compilation output files.
- How does a System-on-Chip start the program?
  - The Arm “Magic secret sauce”.

# Outline

- What is a toolchain?
  - The Arm toolchain.
  - Investigating the compilation output files.
- How does a System-on-Chip start the program?
  - The Arm “Magic secret sauce”.

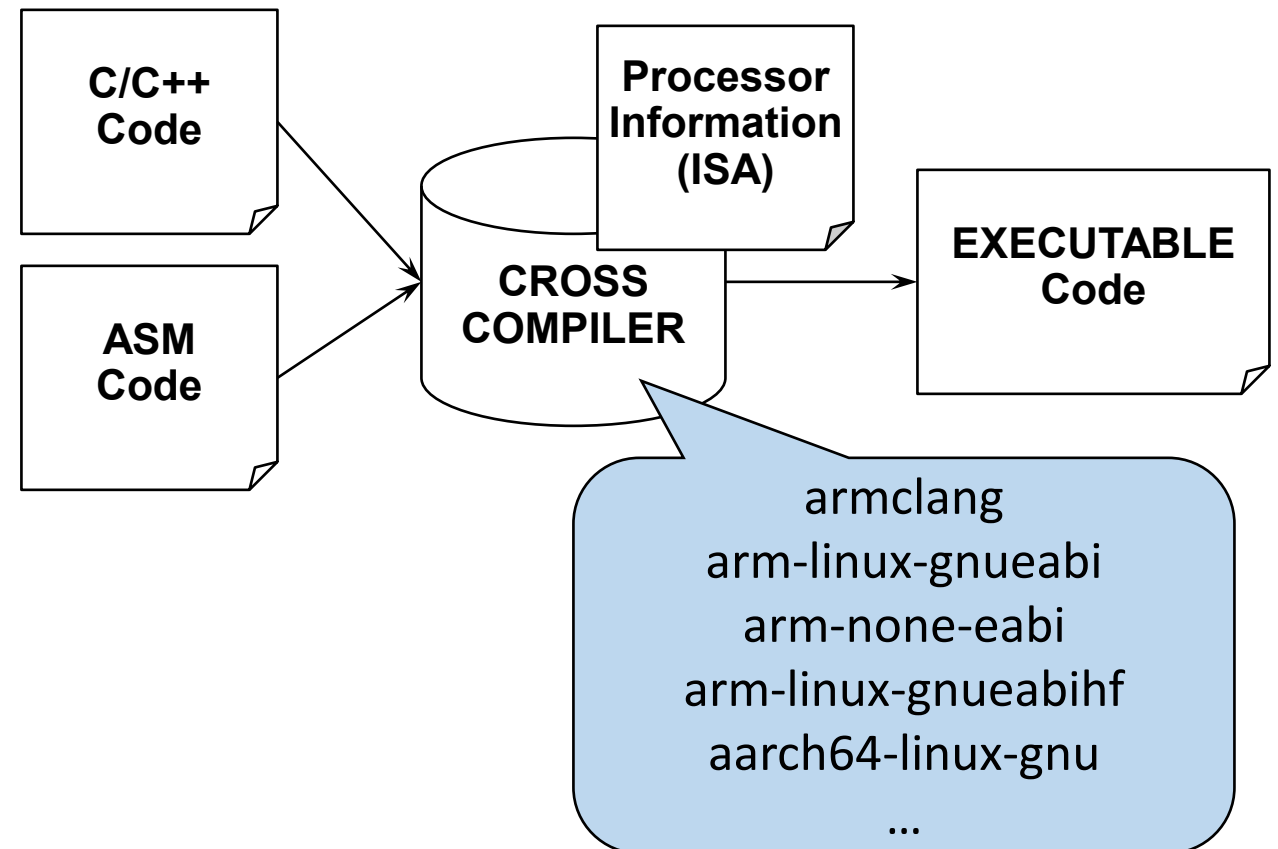
# What is a toolchain?

- A set of **programming tools**.
- Used for complex development tasks or to create software products.



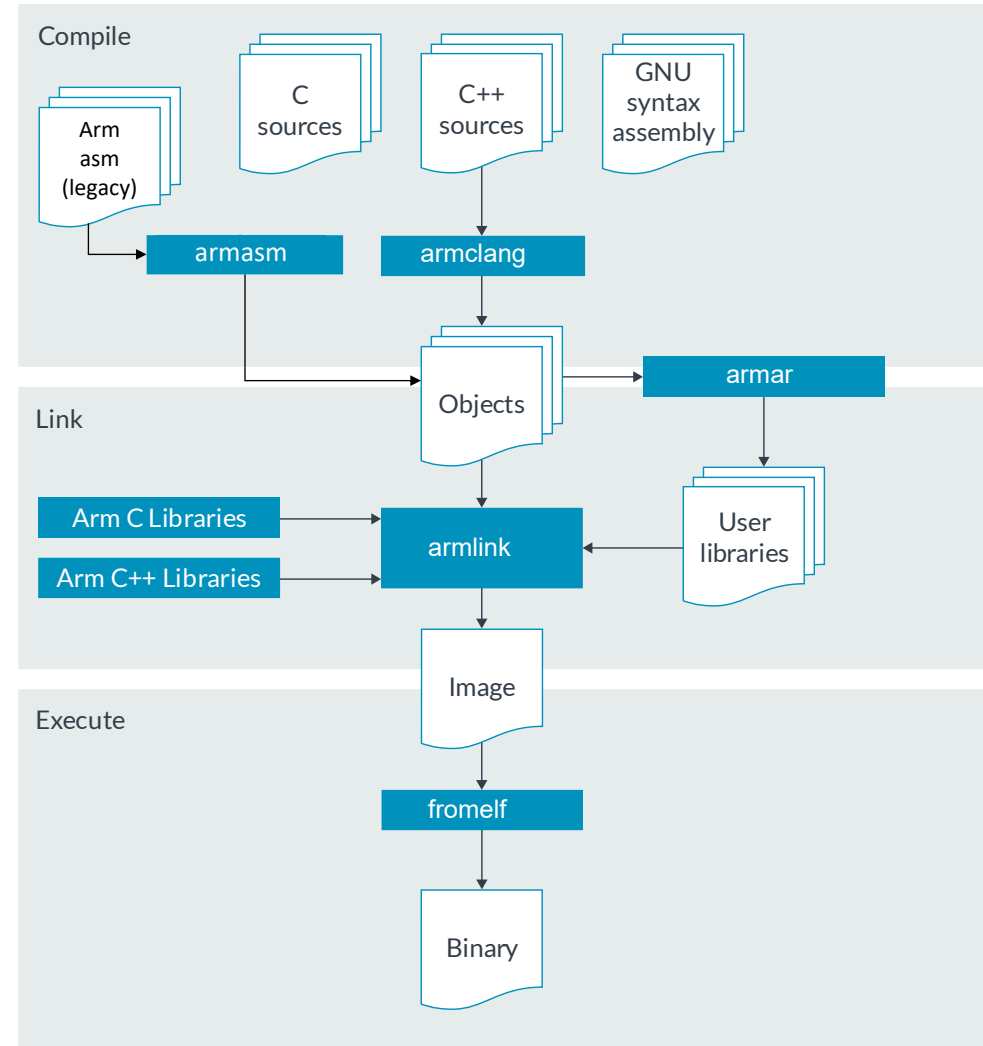
# What does the cross-compiler?

- It is a compiler capable of creating executable code **for a platform other than the one on which the compiler is running.**
- It includes a set of programming tools (toolchain).
- Preprocess the source code.
- Translate high level code in machine code.
- Introduce already developed library.



# The Arm Toolchain

- Based on an enhanced version of *clang* (frontend) and *llvm project*, with proprietary customizations.
- The toolchain is composed of 3 different phases:
  - Preprocessing and compilation phase (armasm for legacy support).
  - Link phase.
  - Execute phase.



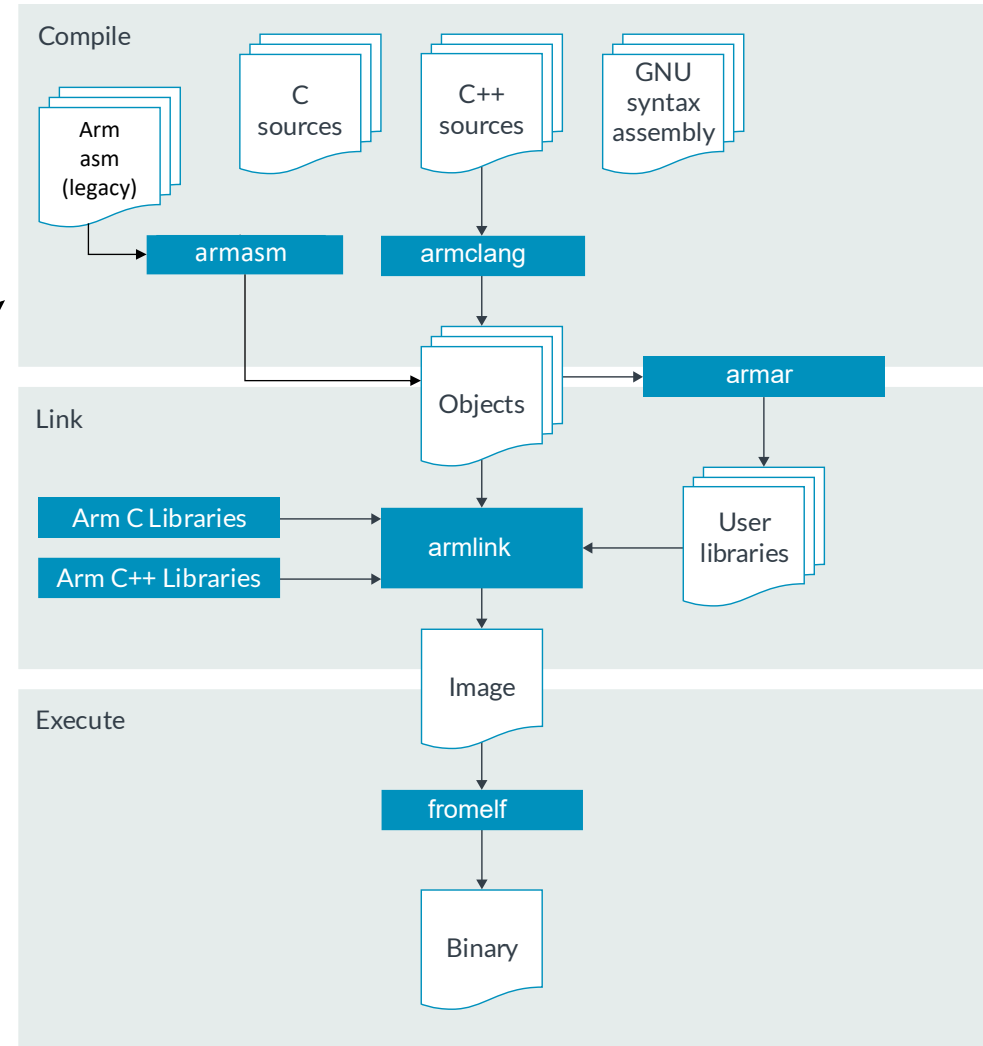
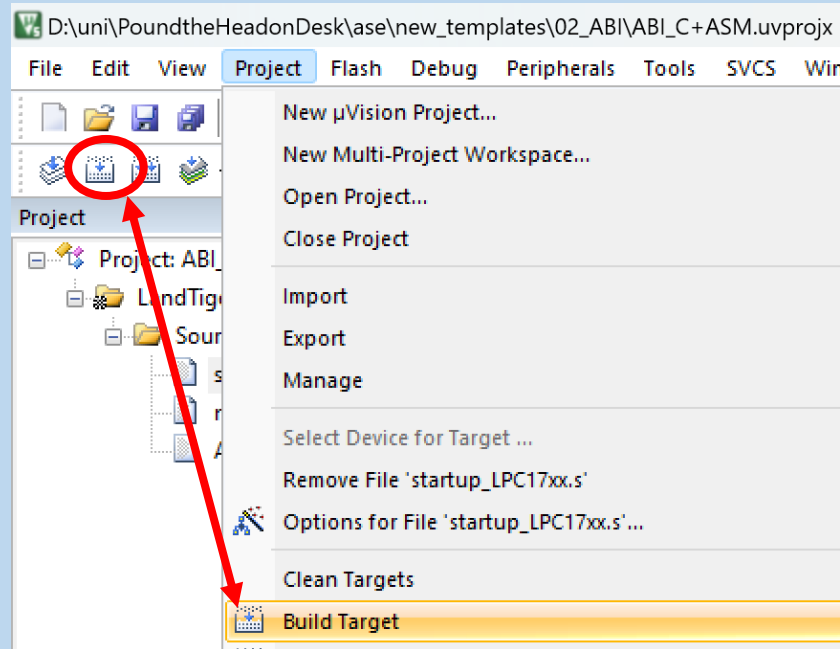


# The Arm Toolchain

- Based on an enhanced version of *clang*

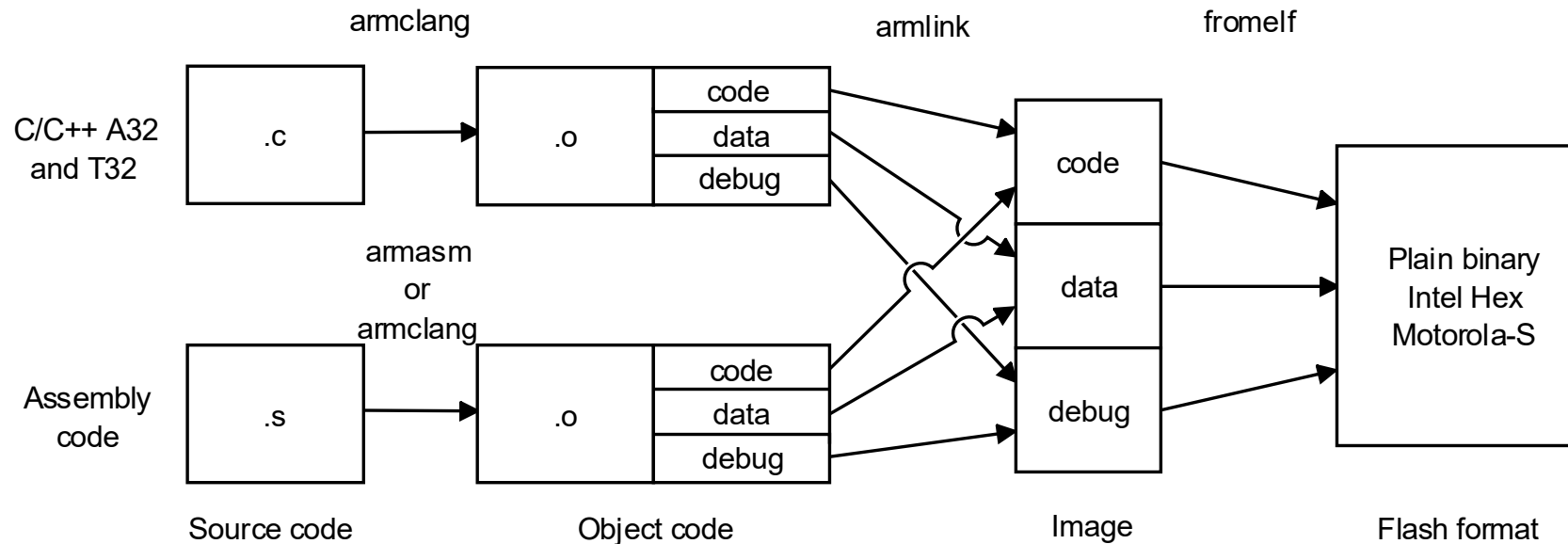
All the three phases included in:

- Toolchain

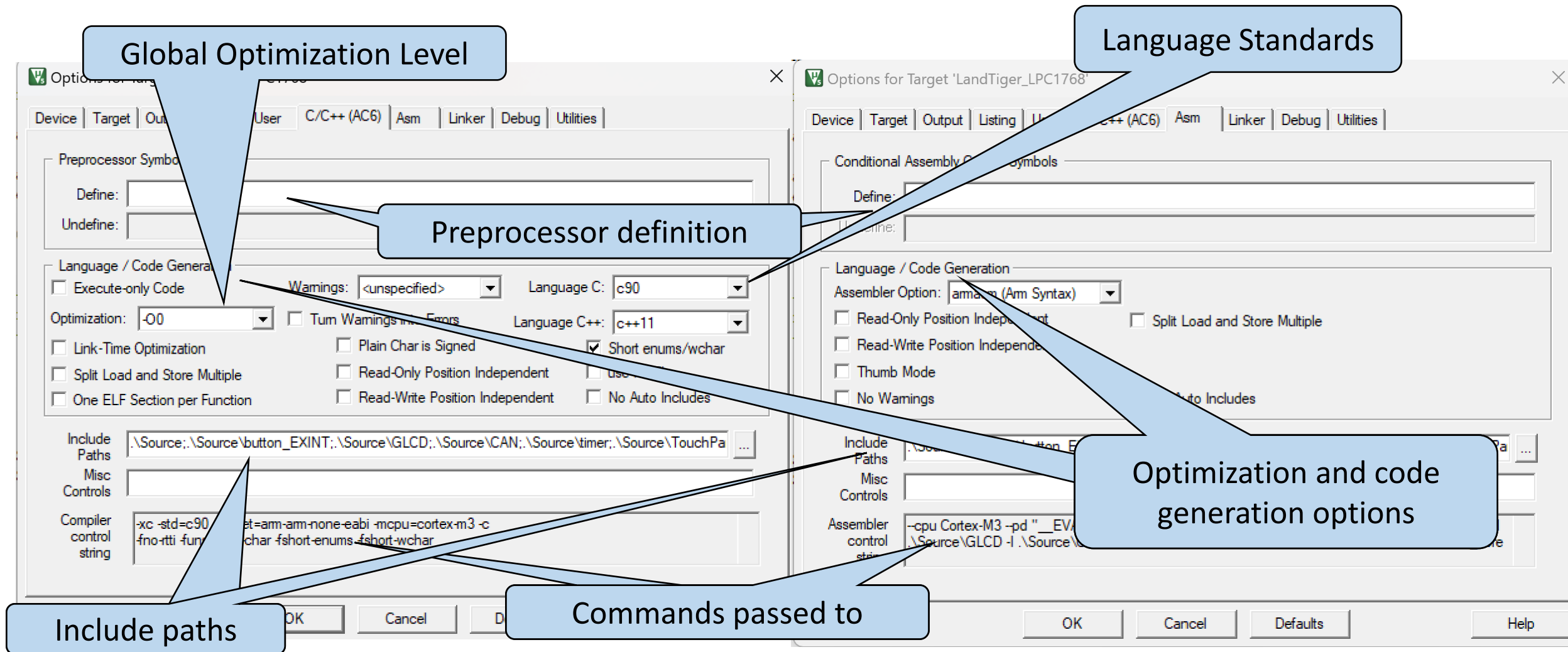


# The compile phase – armclang and armasm

- The compile phase create object files (.o).
- Use armclang to compile high level code such as c or c++.
- Use armasm to assemble existing assembly code written in armasm syntax.
- Use armclang to assemble assembly language code, or inline assembly, written in GNU syntax.



# The compile phase – armclang6 and armasm



# Example – On the fly variable declaration

- Index declaration as C++ like.
- See build log.

```
1
2
3
4 volatile int my_array[N];
5 int main(void) {
6
7     int i;
8
9     for (i=0; i<N; i++) {
10         my_array[i]=i*3;
11     }
12     while(1);
13 }
14
```

```
1
2
3
4 volatile int my_array[N];
5 int main(void) {
6
7     for ( int i=0; i<N; i++) {
8         my_array[i]=i*3;
9     }
10
11     while(1);
12 }
13
```

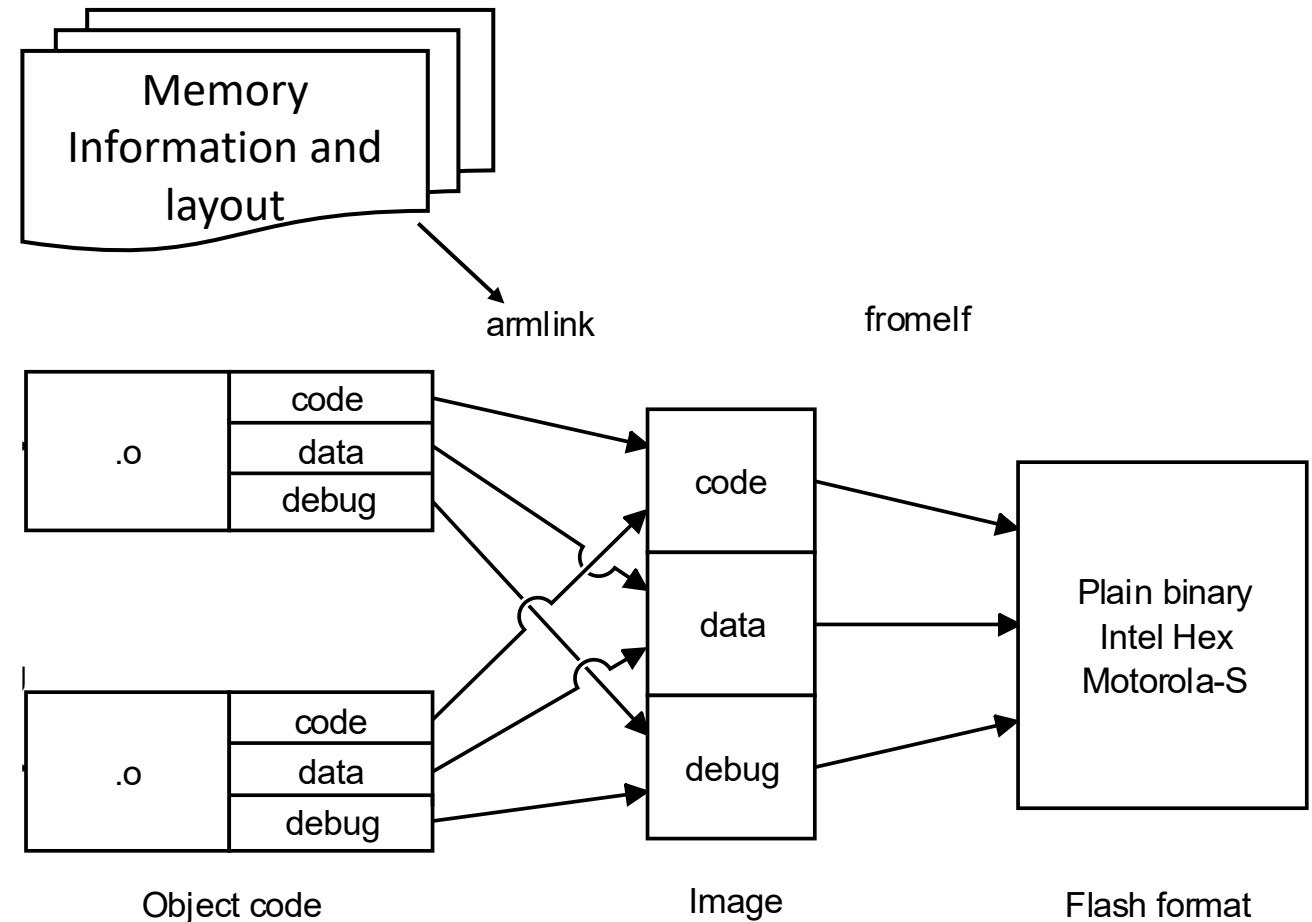
```
*** Using Compiler 'V6.22', folder: 'D:\programmi\keil\ARM\ARMCLANG\Bin'
Rebuild target 'LandTiger_LPC1768'
assembling ASM_funct.s...
assembling startup_LPC17xx.s...
creating list file for main.c...
creating preprocessor file for main.c...
Source/main.c(8): warning: GCC does not allow variable declarations in for loop initializers before C99 [-Wgcc-compat]
8 |         for ( int i=0; iSoftware Packages used:
```

# The compile phase – Optimization levels

- They strongly affect the performance and size of executables (and **machine code** in the executable).
- Different optimizations level:
  - Level 0 (O0): Turns off most optimizations. Generated code that directly corresponds to the source code.
  - Level 1 (O1): Restricted optimization. The best debug view for the trade-off between image size, performance, and debug.
  - Level 2 (O2): High optimization. The debug view might be less satisfactory.
  - Level 3 (O3): Very high optimization. A poor debug view.
  - Fast (Of), Max (Omax), Size (Os - Oz).

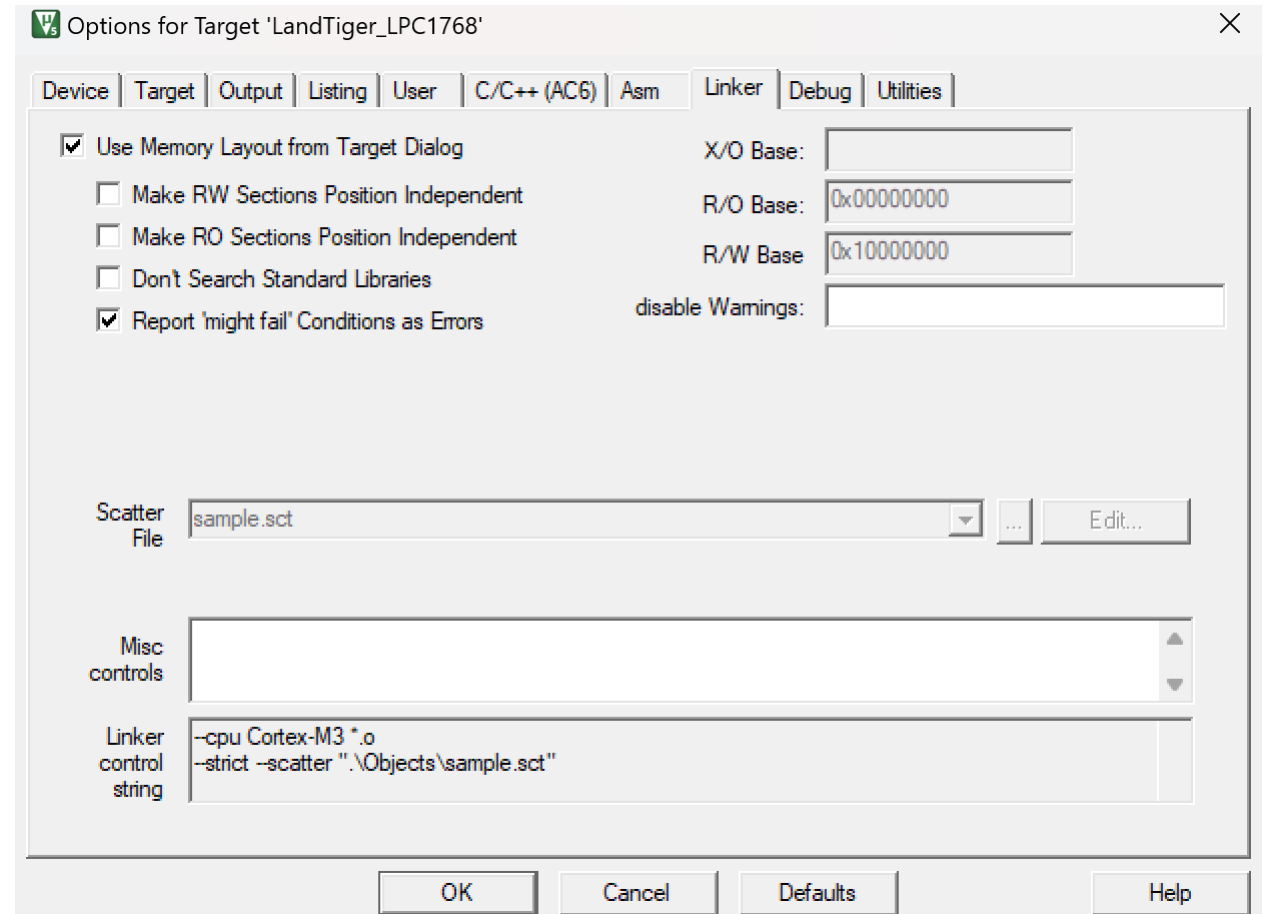
# The link phase – armlink

- Link all object files into a single executable file (or another object file) by merging similar sections.
- It needs memory information to organize the image memory layout.
- It resolves:
  - Functions and variables (their symbols/label is substituted with an address).
  - Linker symbol (**different from functions and variables**).
- It eliminates unused sections **regardless of the optimization level**:
  - Removes unreachable code and data from the final image.



# Memory Information and layout

- You can specify the entry point at the startup (i.e., the function called at the system boot).
- You can specify the memory information:
  - By command line using armlink tool.
  - By passing a scatter file.
- You can specify additional custom code and data sections.



# Memory Information and layout

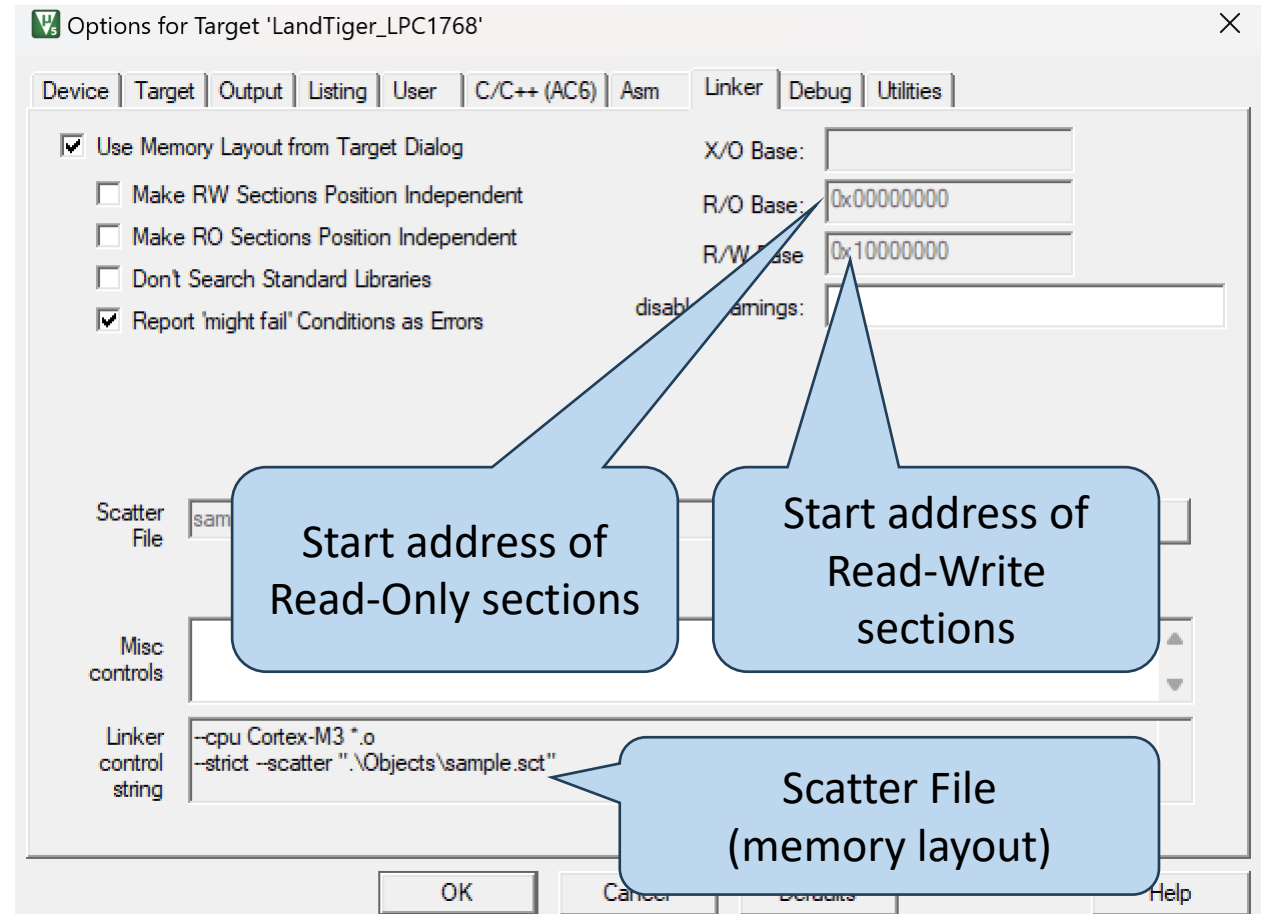
- You can specify the memory layout at the start of the system by calling

```
FLASH_LOAD 0x20000000
{
    RW 0x20000000 ; RW
    * (+RW-DATA)
}
```

Start address of  
Zero Init sections

- By passing a scatter file.
- You can specify additional custom code and data sections.

```
ER_ZI 0x405000
{
    * (+ZI)
}
```





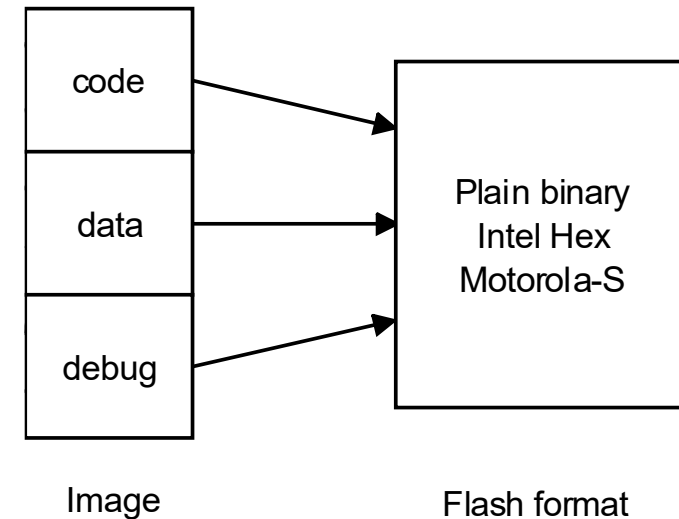
# The execute phase - fromelf

- Process object and image files.
- Convert ELF images into other formats for use by ROM tools or for direct loading into memory. The formats available are:
  - Plain binary.
  - Motorola 32-bit S-record.
  - Intel Hex-32.
  - Byte oriented hexadecimal.
- Display information about the input file, for example, disassembly output or symbol listings.

## Binary

00000b68	88	00	00	00	10	02	00	00	08	00	00	00	0c	00	00	00	.....	
00000b70	bc	00	00	00	18	02	00	00	08	00	00	00	0c	00	00	00	.....	
00000b80	88	00	00	00	24	02	00	00	02	00	00	00	d8	00	00	00	.....\$.....	
00000b90	03	00	04	a5	00	00	00	04	01	41	53	4d	5f	66	75	6e	63	.....ASM_func
00000ba0	74	2e	73	00	43	6f	6d	70	20	6f	6e	65	6e	74	3a	20	41	.....t.s.Component: A
00000bb0	52	4d	20	43	6f	6d	70	69	6c	65	72	20	35	2e	30	36	66	.....RM Compiler 5.06
00000bc0	20	75	70	64	61	74	65	20	36	20	28	62	75	69	6c	64	61	.....update 6 (build
00000bd0	20	37	35	30	29	20	54	6f	6f	6c	3a	20	61	72	6d	61	61	.....750) Tool: arma

# Motorola

[illegible]

# The execute phase - fromelf

- Process object and image files.
- Convert ELF images into other formats for use by other tools for direct loading in target hardware. Formats available:

- Plain binary.

- Motorola 32-bit S-record.

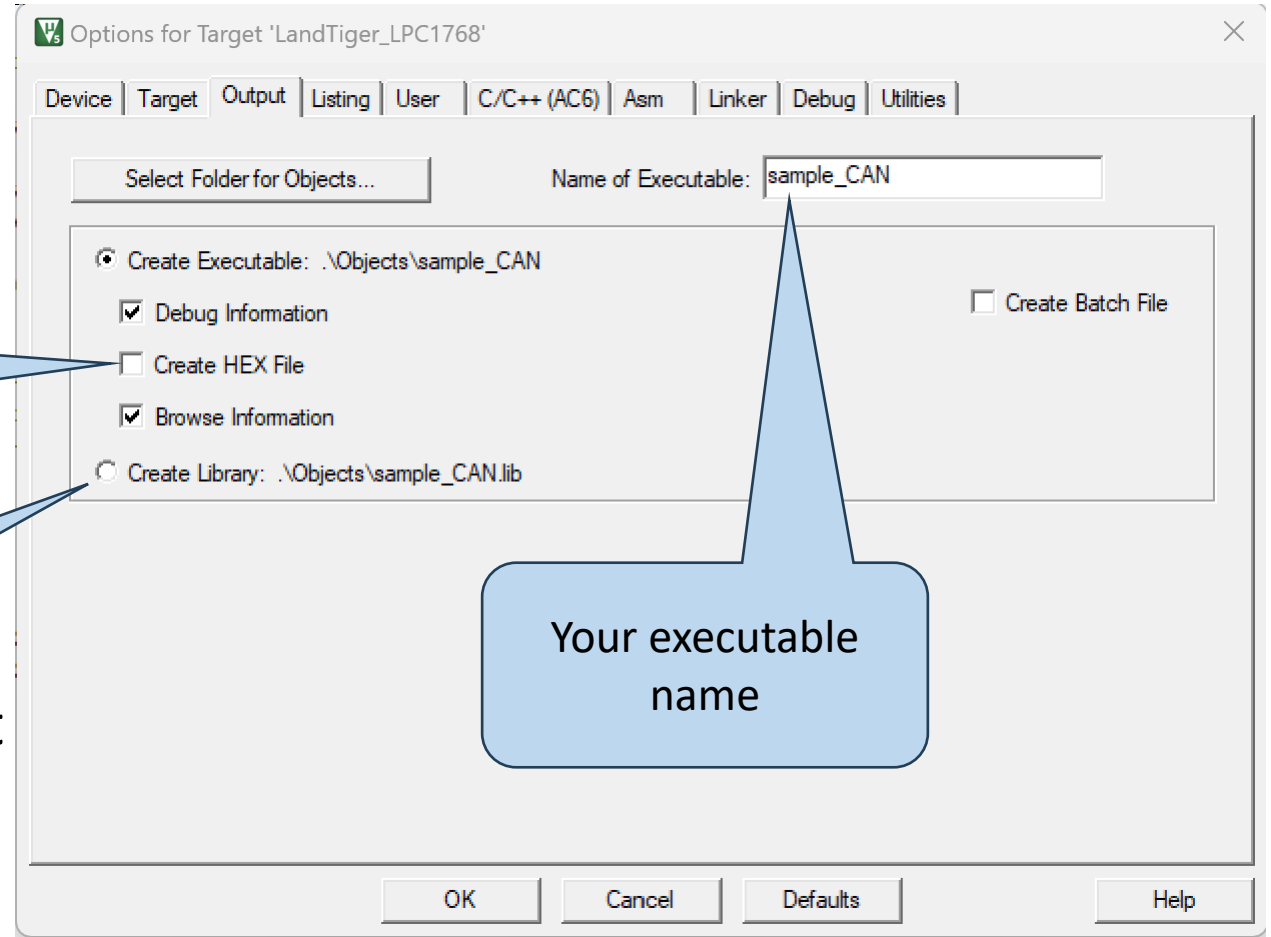
- Intel Hex-32.

- Byte oriented.

- Display information in output file, for example, object or symbol listing.

If you want a hex file

If you want a library to be included into another project

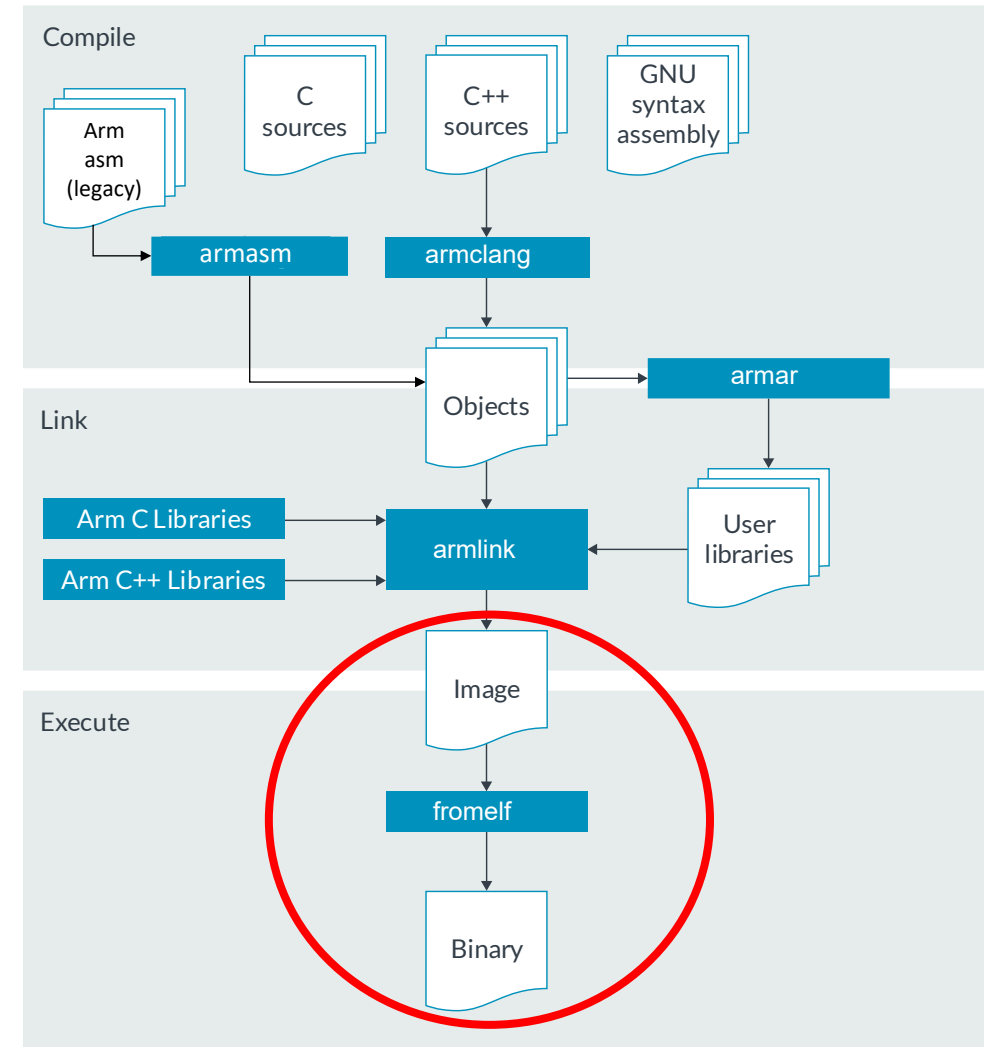


# Outline

- What is a toolchain?
  - The Arm toolchain.
  - Investigating the compilation output files.
- How does a System-on-Chip start the program?
  - The Arm “Magic secret sauce”.

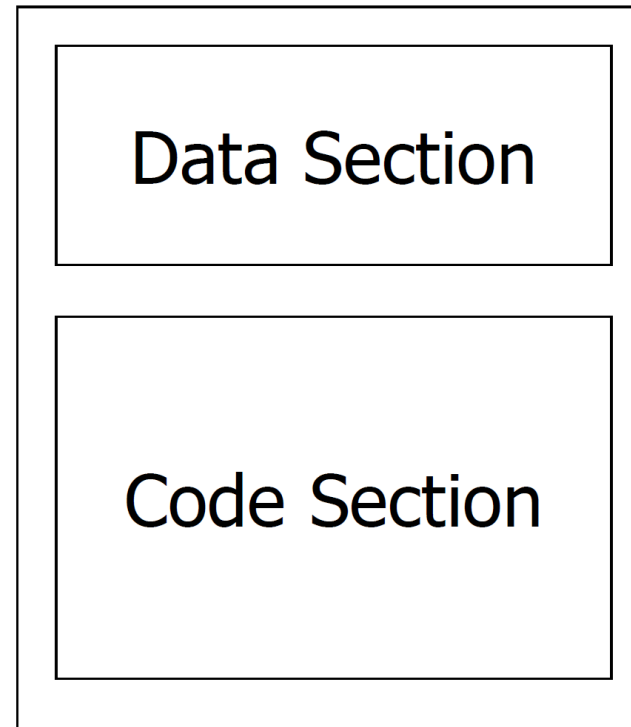
# Investigating the compilation output files.

- The Arm toolchain produces:
  - The executable file.
  - The listing, dependencies files.
  - The map file.
  - The build log and static call graph file.



# The executable

- The overall image (from the source code) is converted into an executable (.exe, .elf, .axf for Arm).
- Data and Code sections are in the executable.

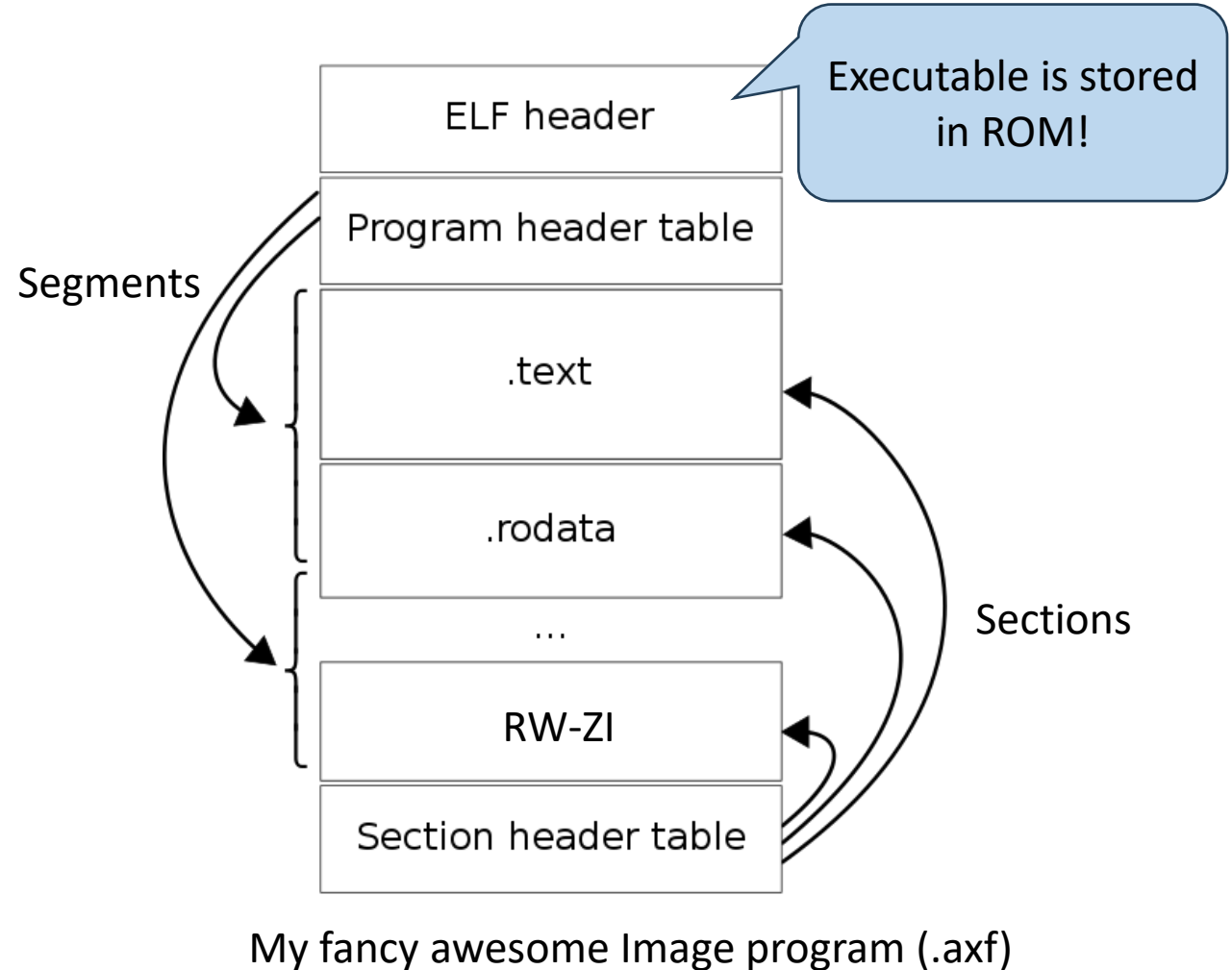


My fancy awesome Image program

- Data Section
  - Variables
  - Constants
- Code Section
  - Program
  - Routines
  - Subroutines

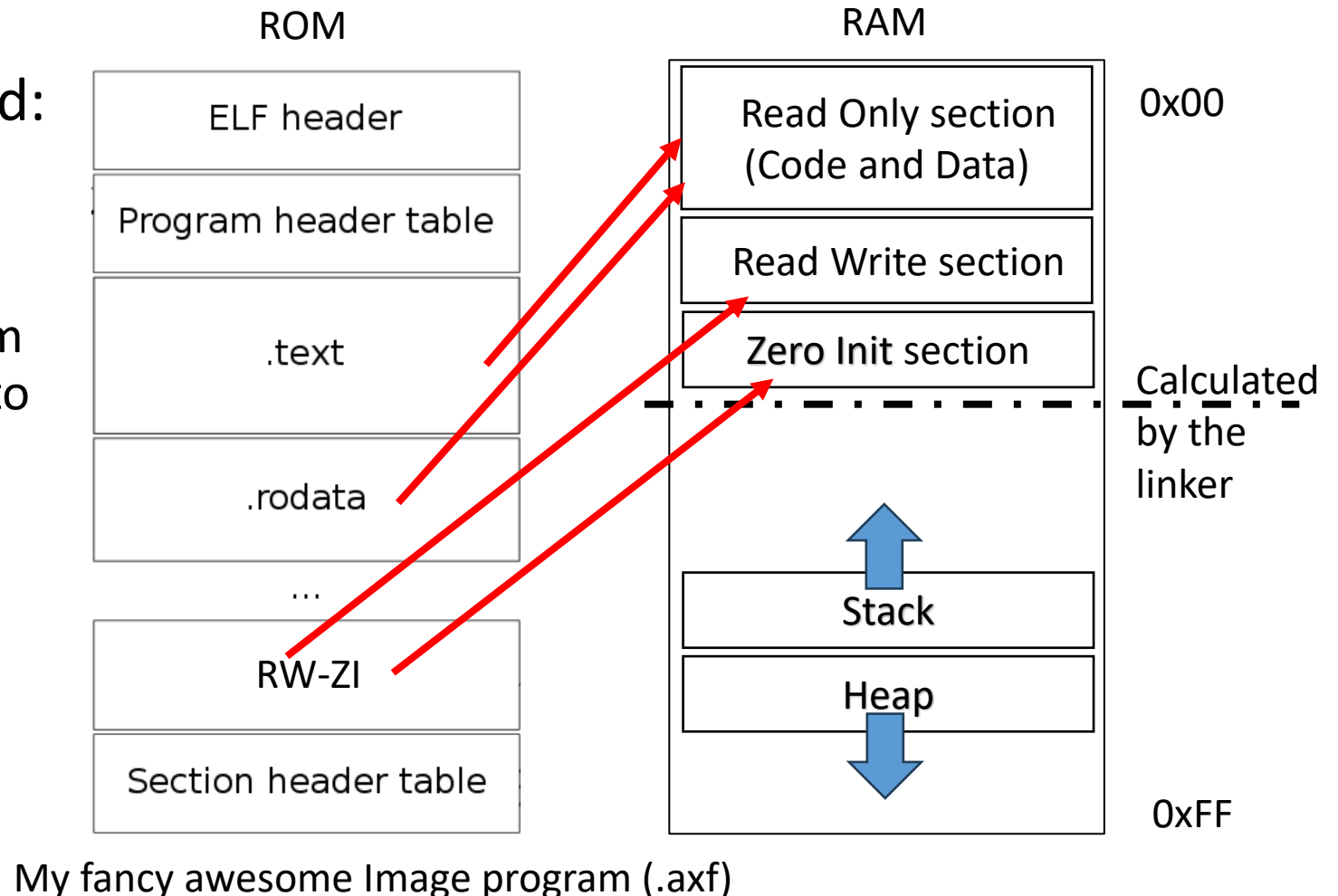
# The executable – Load view

- The overall image (from the source code) is converted into an executable (.exe, .elf, .axf for Arm).
- Data and Code sections are in the executable.
- Composed of:
  - Entry address.
  - Stack and heap information.
  - Sections, used by the linker.
  - Segments, used by the loader (at runtime).



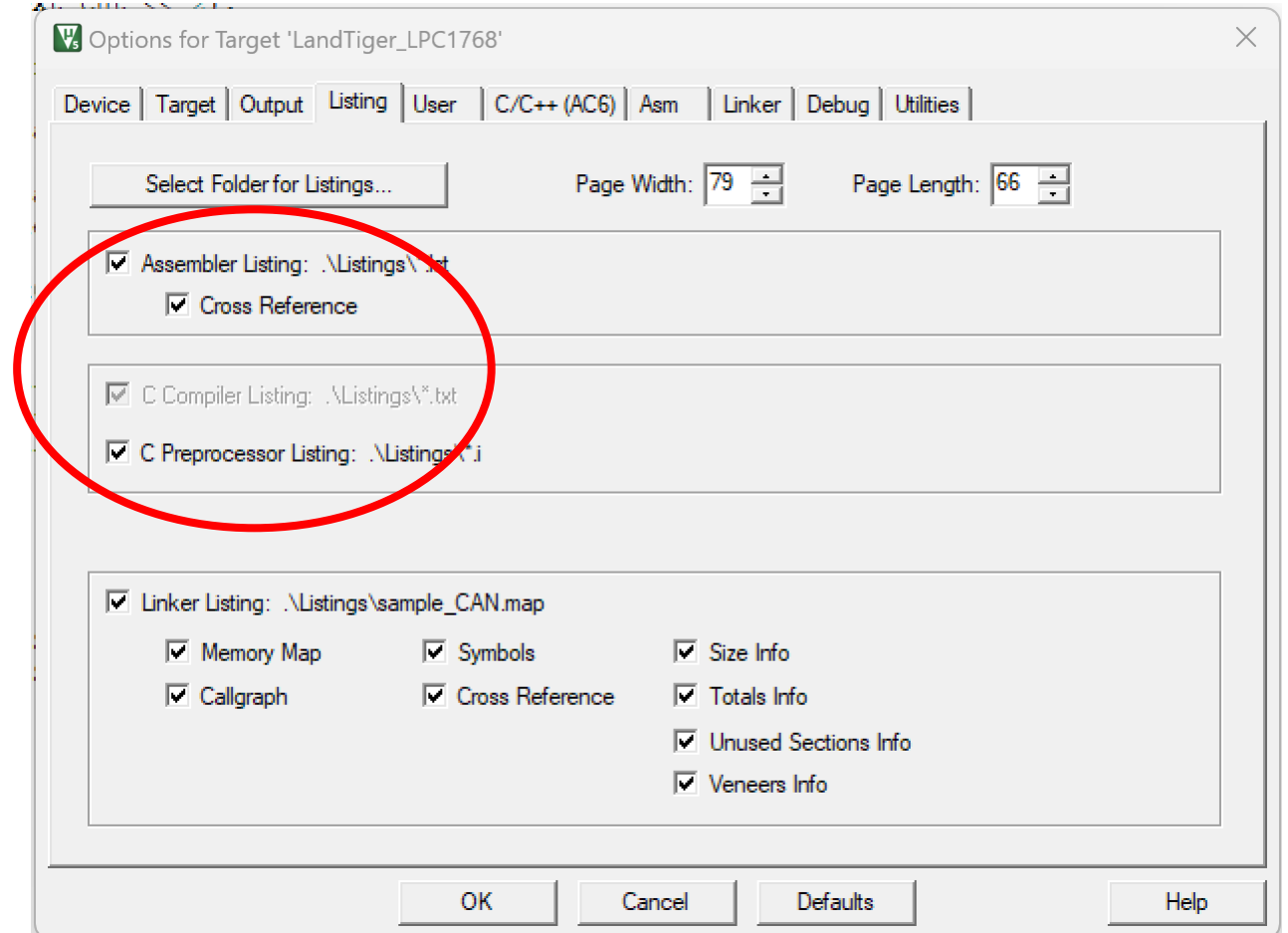
# The executable – Execution view

- Before the image is executed:
  - Move executable segments from ROM to their execution addresses in RAM.
  - RW data must be copied from its load address in the ROM to its execution address in the RAM.
- Runtime memory layout information is calculated offline:
  - Stack and heap execution address and size.



# The Listing and dependencies files

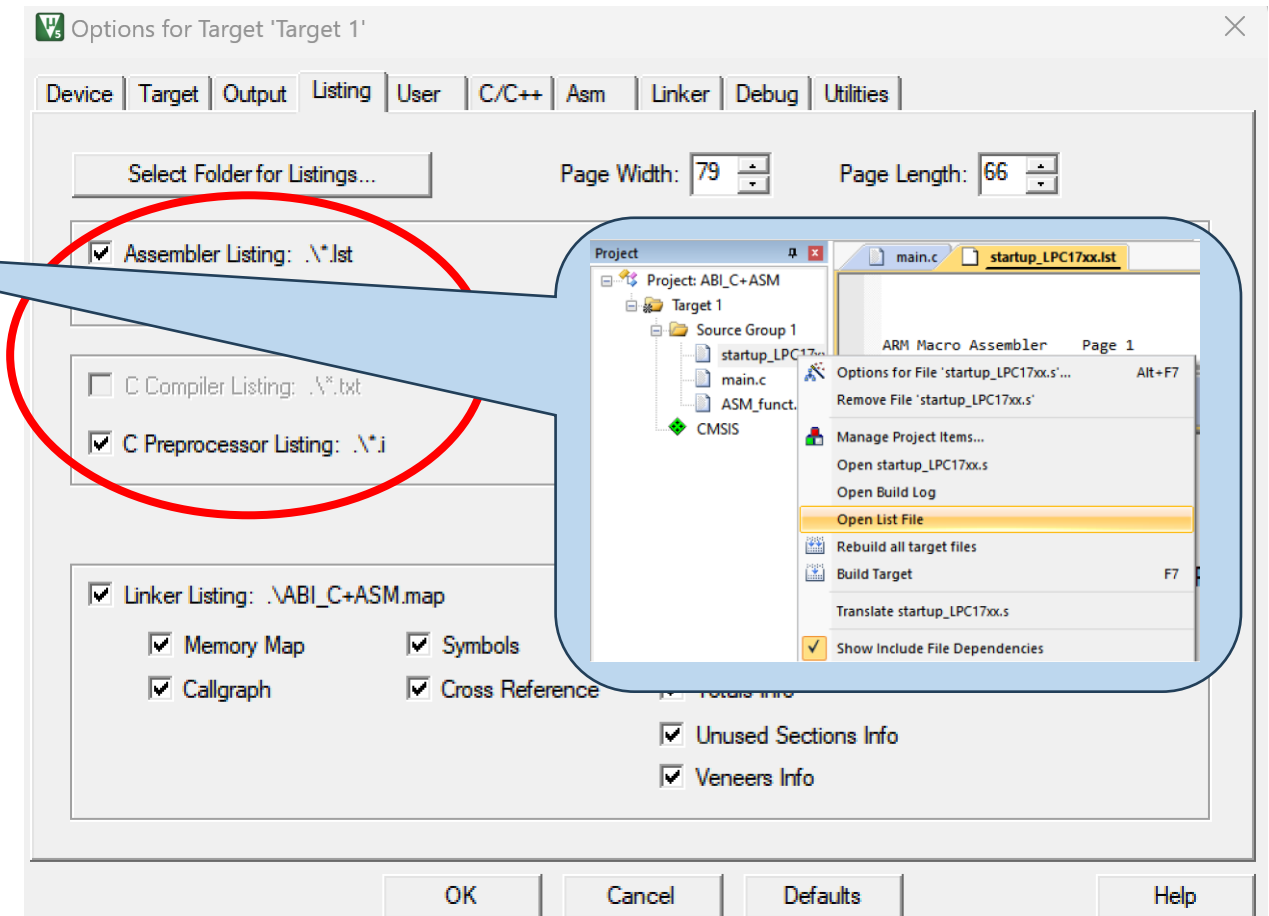
- Dependencies files are generated (.d) and used by the toolchain (information needed during the link phase!).
- The project dependencies are in the .dep file.
- Listing files are debugging files showing how the code is translated in machine code.





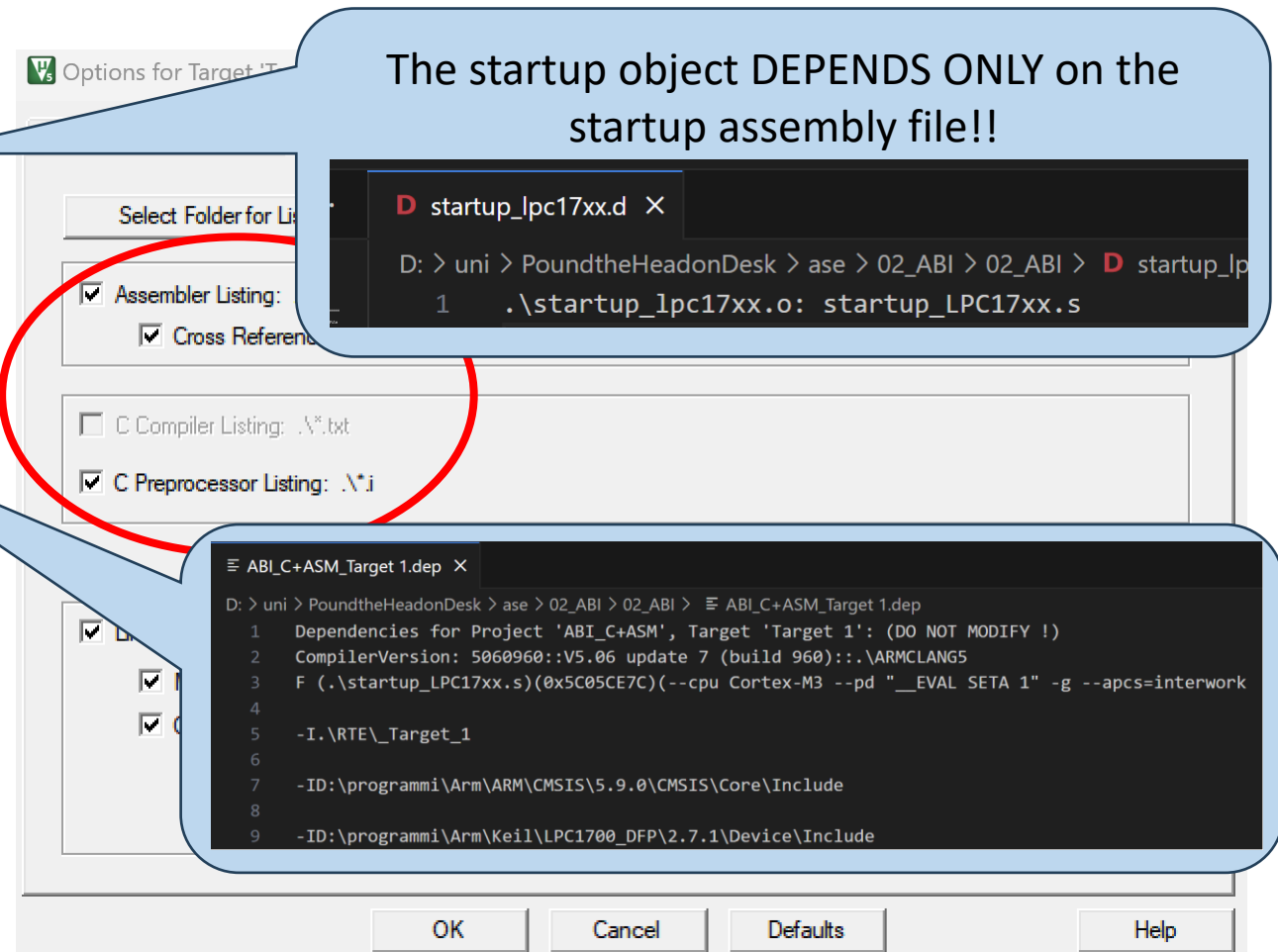
# The Listing and dependencies files

- Dependencies files are generated (.d) and used by the toolchain (information needed during the link phase!).
- The project dependencies are in the .dep file.
- Listing files are debugging files showing how the code is translated in machine code.



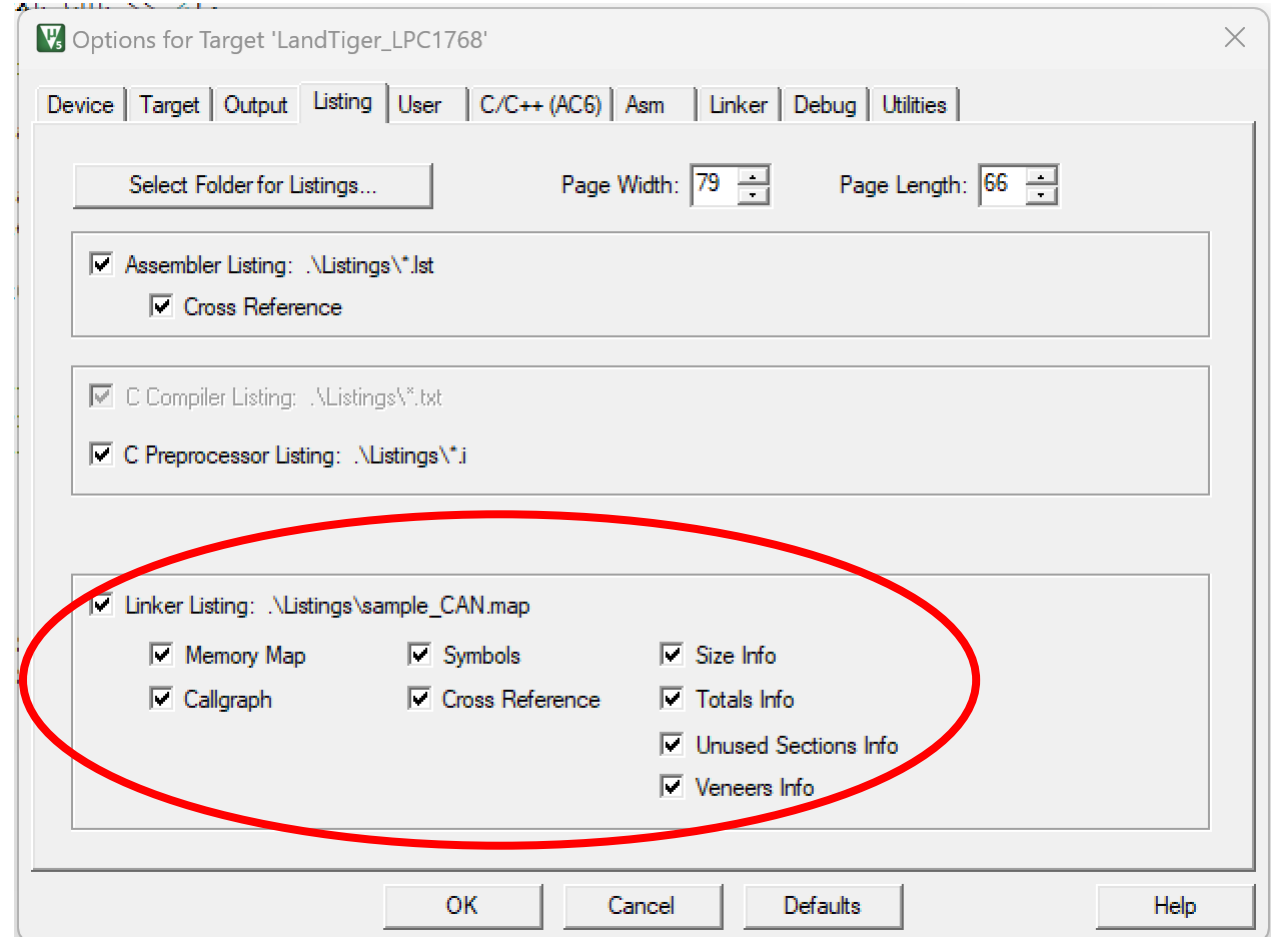
# The Listing and dependencies files

- Dependencies files are generated (.d) and used by the toolchain (information needed during the link phase!).
- The project dependencies are in the .dep file.
- Listing files are debugging files showing how the code is translated in machine code.



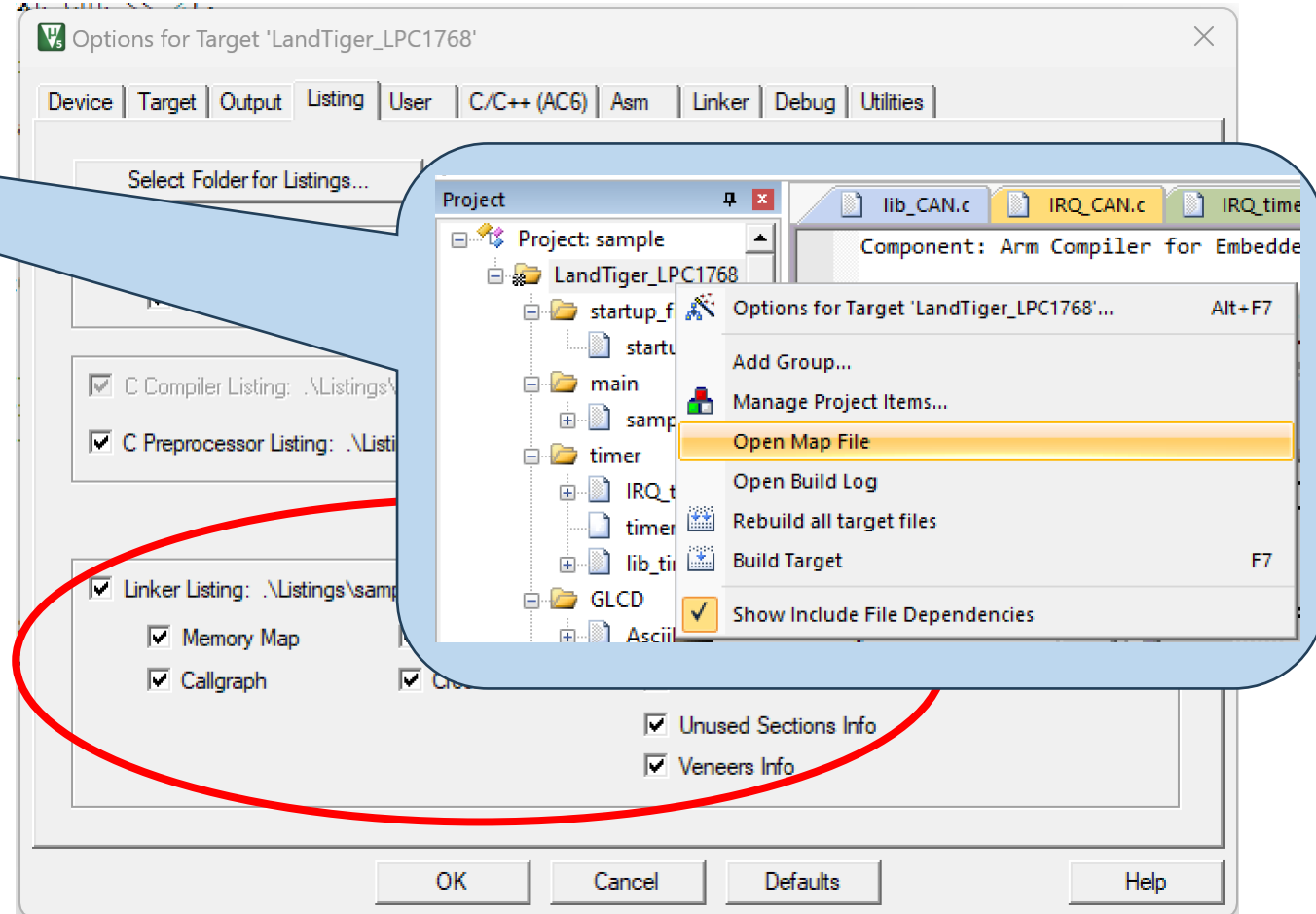
# The map

- It is the output “log” of the link phase.
- It includes the memory map, symbols table, cross references and sizes.
- It may be used by debugging tools.



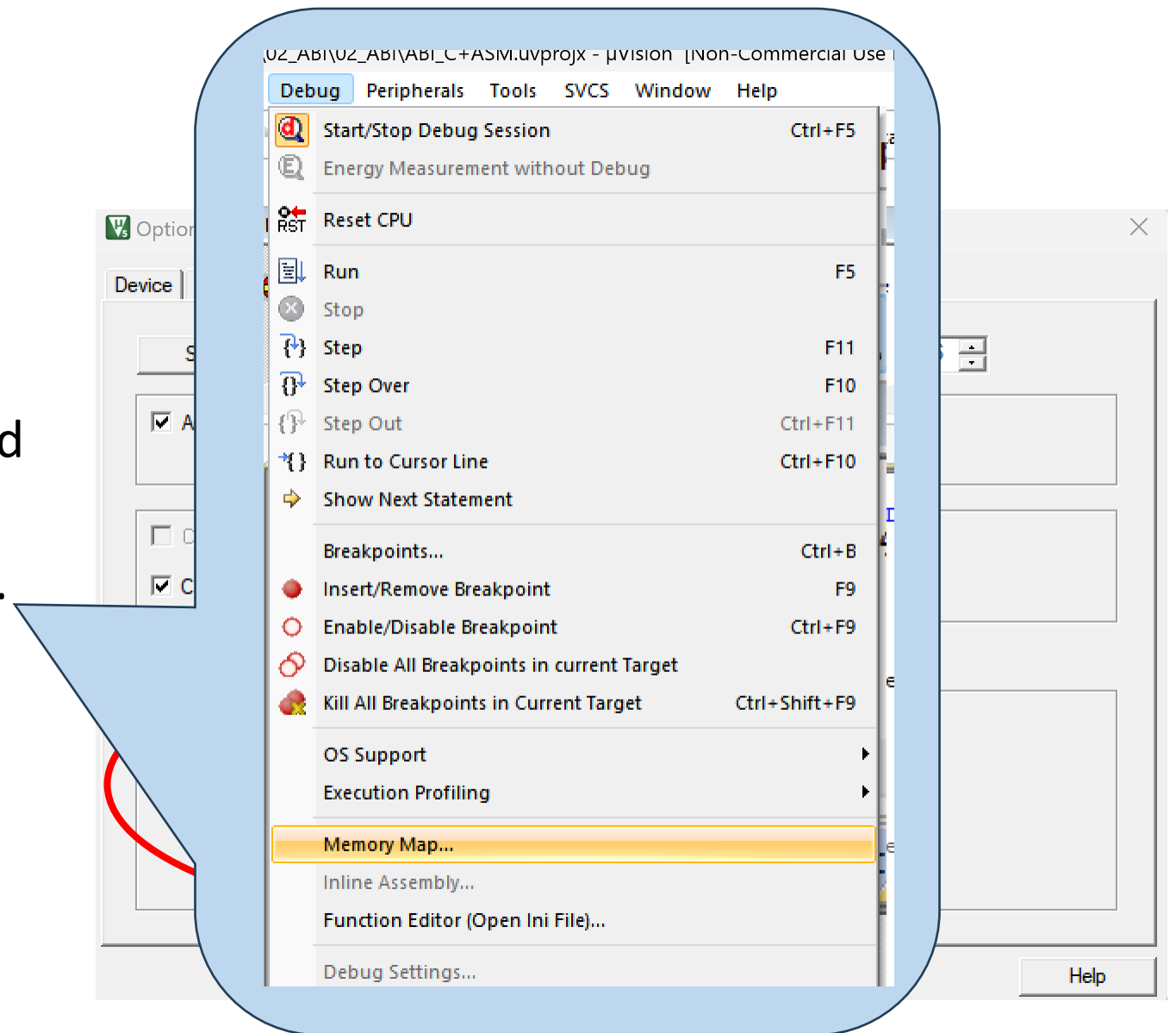
# The map

- It is the output “log” of the link phase.
- It includes the memory map, symbols table, cross references and sizes.
- It may be used by debugging tools.



# The map

- It is the output “log” of the link phase.
- It includes the memory map, symbols table, cross references and sizes.
- It may be used by debugging tools.



# Example – Where do they belong to?

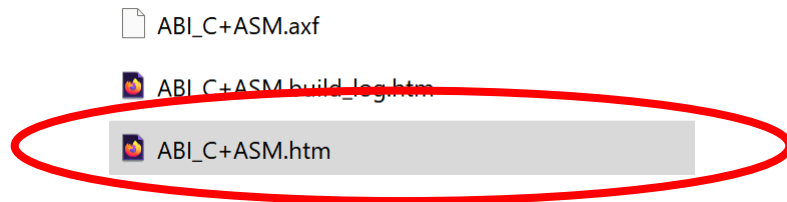
- Where the compiler is putting the variables?
- See the map file.

Image Symbol Table					
Local Symbols					
Symbol Name	Value	Offset	Type	Size	Object(Section)
Global Symbols					
Symbol Name	Value	Offset	Type	Size	Object(Section)

```
1
2
3  const int pippo[]= {21312,44321};
4  int this_is_zero;
5  short int this_is_not_zero=0xcafe;
6  volatile int my_array[N];
7
8
9
10 int main(void) {
11     int i=0;
12
13     volatile int value=pippo[0];
14
15     for ( i=0;i<N;i++){
16         my_array[i]=i*3;
17     }
18
19     while(1);
20 }
21
```

# The static call graph file

- It is another debugging output “log” of the link phase.
- It is a control-flow graph.
- It represents the calling relationships between functions in the executable.
- In **Objects** folder:



## Static Call Graph for image .\ABI\_C+ASM.axf

#<CALLGRAPH># ARM Linker, 5060960: Last Updated: Sun Dec 03 13:41:14 2023

Maximum Stack Usage = 16 bytes + Unknown(Functions without stacksize, Cycles, Untraceable Function Pointers)

Call chain for Maximum Stack Depth:

\_\_rt\_entry\_main ⇒ main

Functions with no stack information

- [\\_user\\_initial\\_stackheap](#)
- [ASM\\_func](#)

Mutually Recursive functions

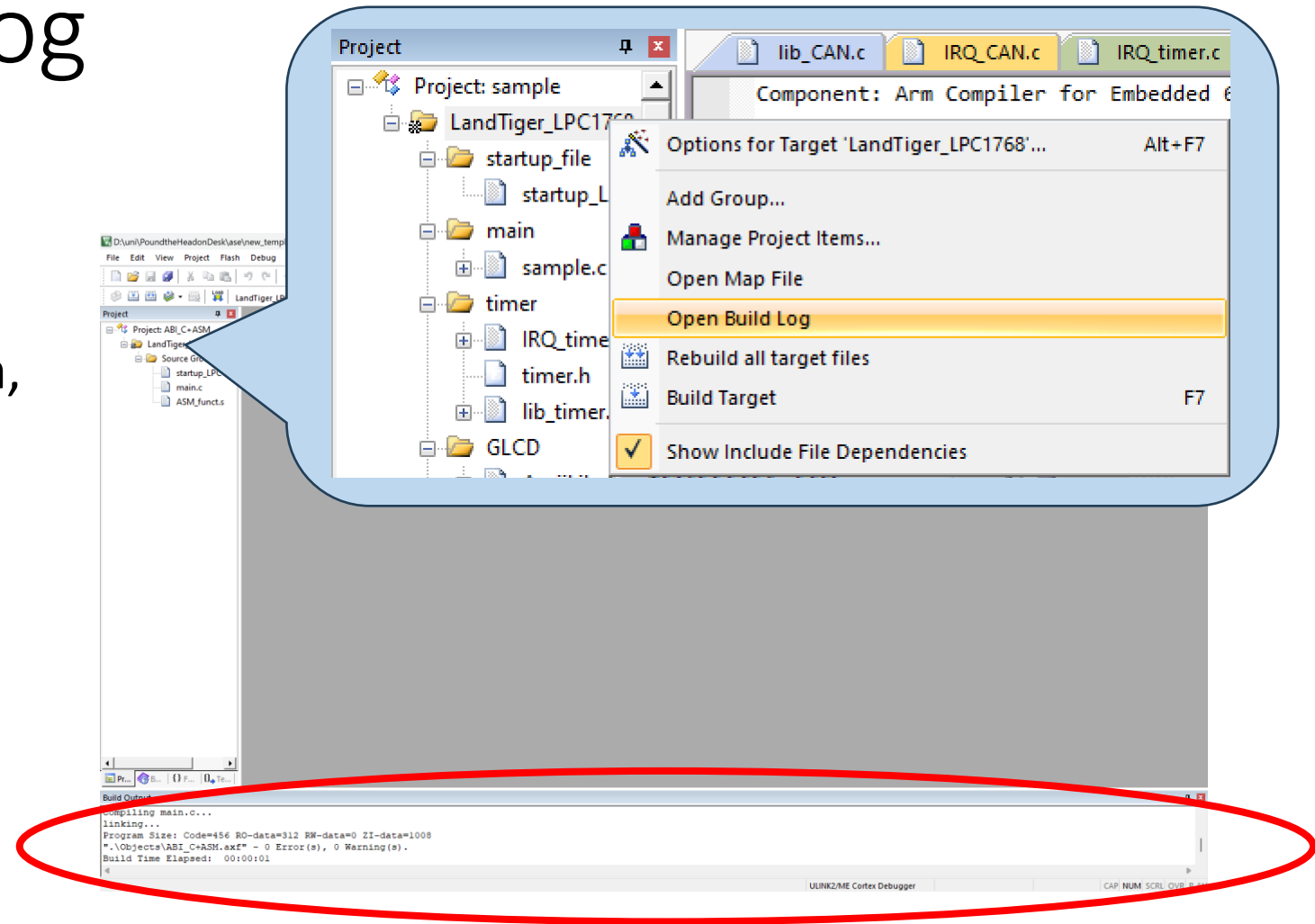
- [NMI\\_Handler](#) ⇒ [NMI\\_Handler](#)
- [HardFault\\_Handler](#) ⇒ [HardFault\\_Handler](#)
- [MemManage\\_Handler](#) ⇒ [MemManage\\_Handler](#)
- [BusFault\\_Handler](#) ⇒ [BusFault\\_Handler](#)
- [UsageFault\\_Handler](#) ⇒ [UsageFault\\_Handler](#)
- [SVC\\_Handler](#) ⇒ [SVC\\_Handler](#)
- [DebugMon\\_Handler](#) ⇒ [DebugMon\\_Handler](#)
- [PendSV\\_Handler](#) ⇒ [PendSV\\_Handler](#)
- [SysTick\\_Handler](#) ⇒ [SysTick\\_Handler](#)
- [ADC\\_IRQHandler](#) ⇒ [ADC\\_IRQHandler](#)

Function Pointers

- [ADC\\_IRQHandler](#) from startup\_lpc17xx.o(text) referenced from startup\_lpc17xx.o(RESET)
- [ASM\\_func](#) from asm\_func.o(asm\_functions) referenced from asm\_func.o(asm\_functions)
- [BOD\\_IRQHandler](#) from startup\_lpc17xx.o(text) referenced from startup\_lpc17xx.o(RESET)

# The build output log

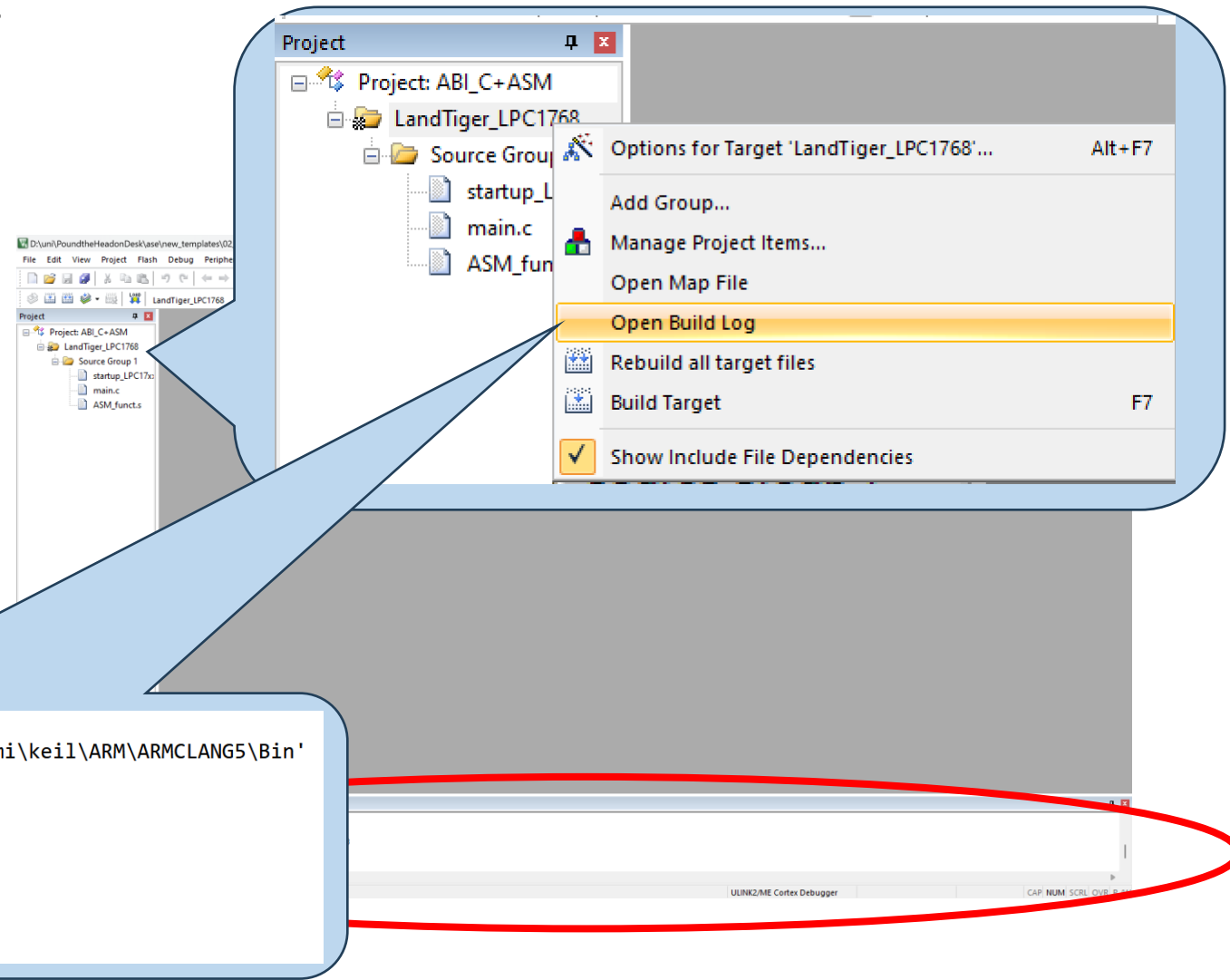
- It is the log of the entire build process for a given project.
- It includes a log of tools version, software packages and components used.





# The build output log

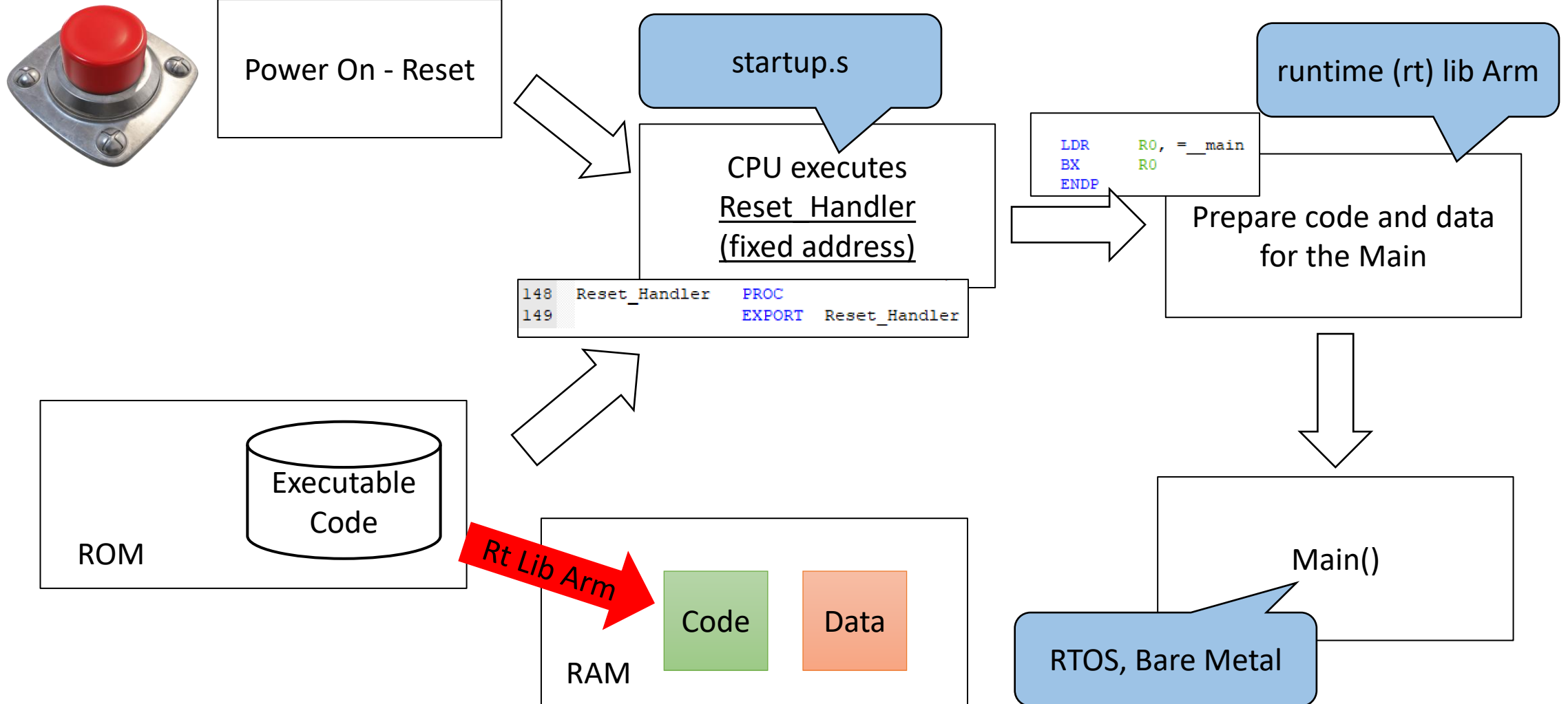
- It is the log of the entire build process for a given project.
- It includes a log of tools version, software packages and components used.



# Outline

- What is a toolchain?
  - The Arm toolchain.
  - Investigating the compilation output files.
- How does a System-on-Chip start the program?
  - The Arm “Magic secret sauce”.

# How does a System-on-Chip start the program?



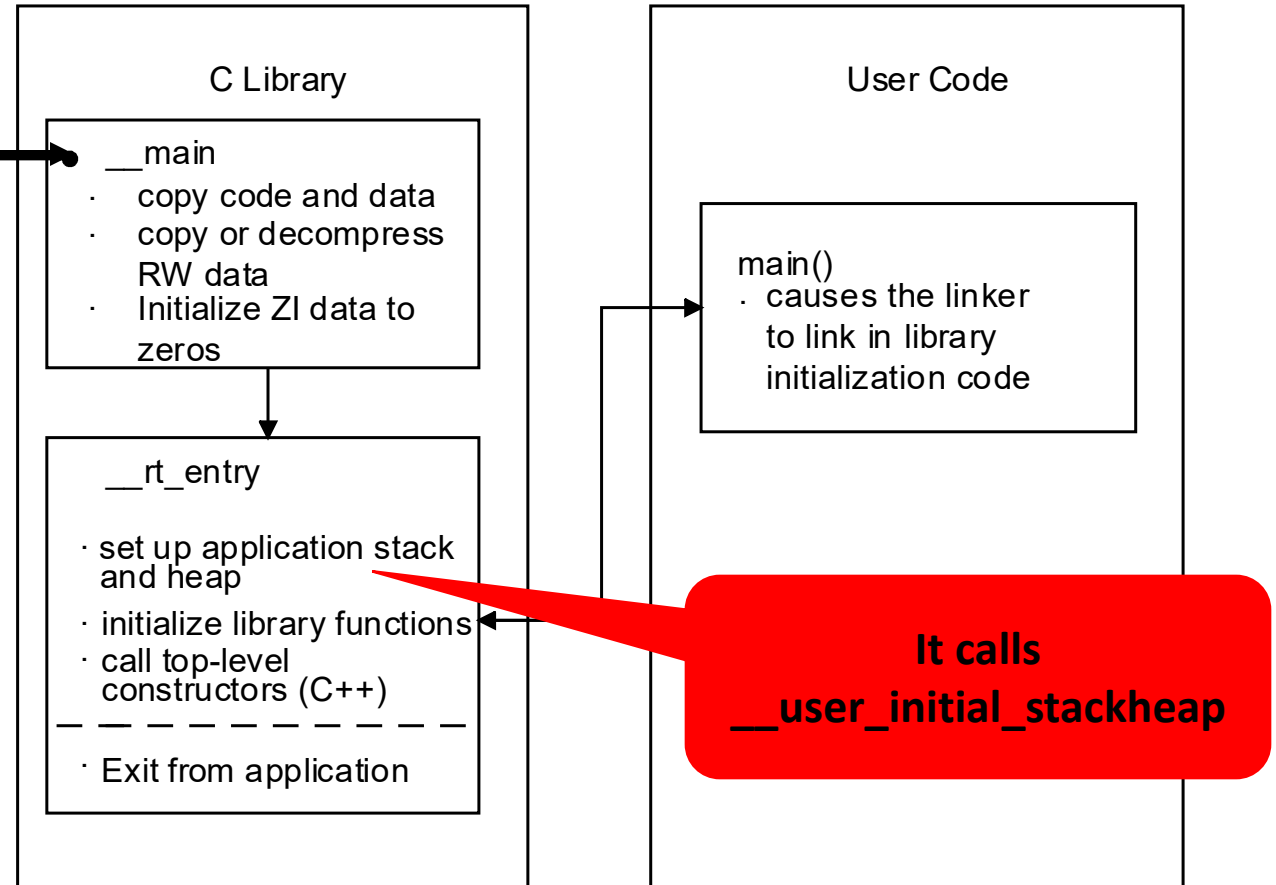
# Outline

- What is a toolchain?
  - The Arm toolchain.
  - Investigating the compilation output files.
- How does a System-on-Chip start the program?
  - The Arm “Magic secret sauce”.

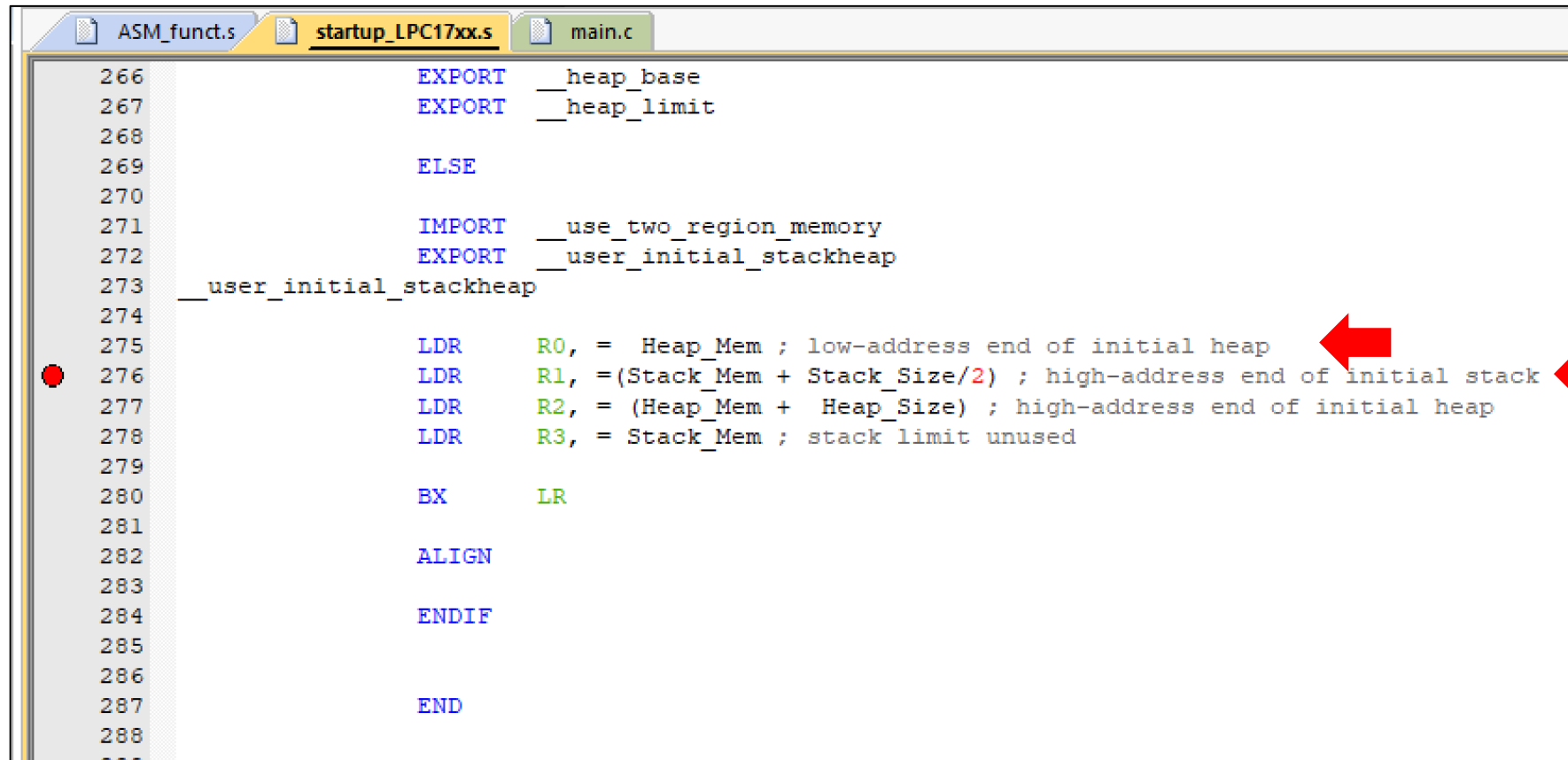
# The Arm “Magic secret sauce”

```
Reset_Handler  PROC
                 EXPORT Reset_Handler            [WEAK]
                 IMPORT  __main
                 LDR     R0, =__main
                 BX      R0
                 ENDP
```

- `__main` responsible for:
  - Setting up the memory code and data.
- `__rt_entry` responsible for:
  - Setting up **stack(s)** and **heap**.
  - Initializing the lib functions and static data.
  - Calling any top level constructors.

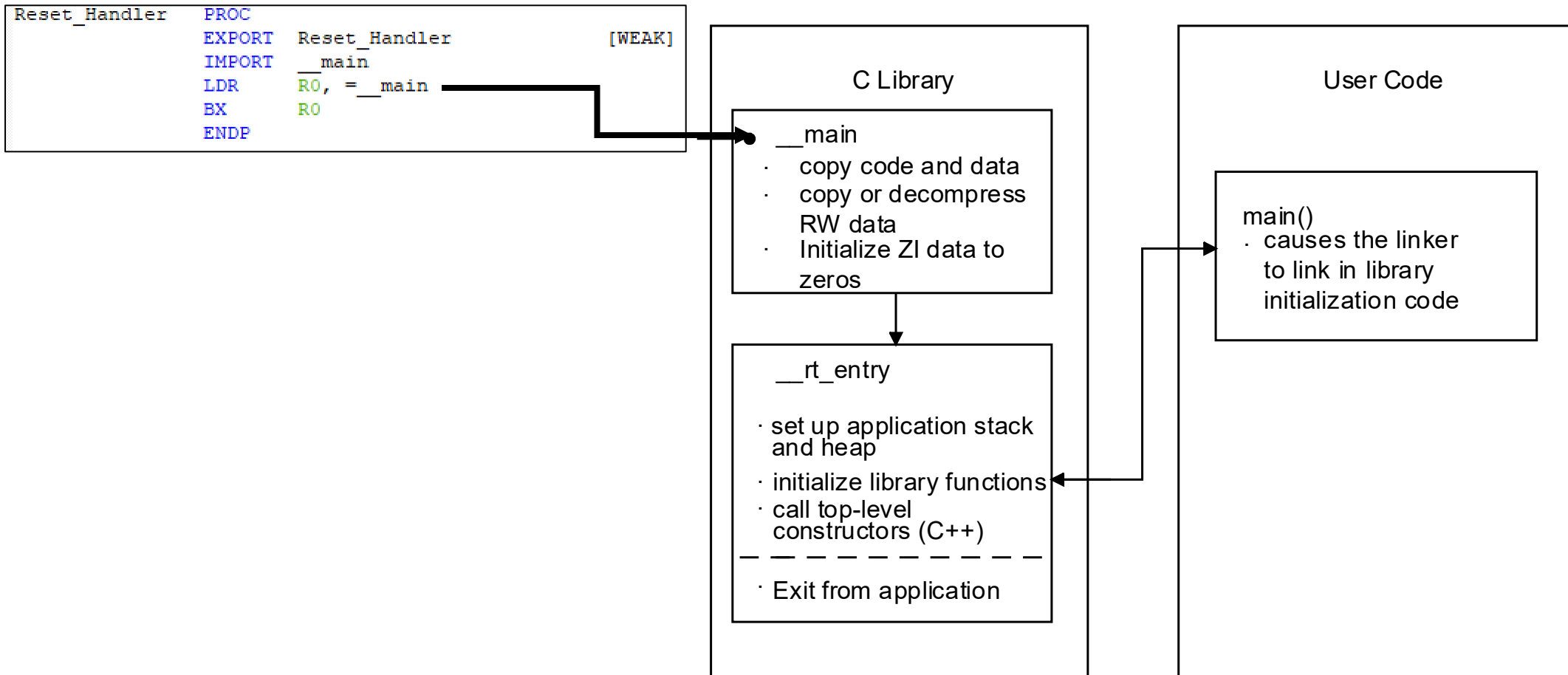


# Setting up stack(s)



```
266      EXPORT __heap_base
267      EXPORT __heap_limit
268
269      ELSE
270
271      IMPORT __use_two_region_memory
272      EXPORT __user_initial_stackheap
273  __user_initial_stackheap
274
275      LDR    R0, = Heap_Mem ; low-address end of initial heap
276      LDR    R1, =(Stack_Mem + Stack_Size/2) ; high-address end of initial stack
277      LDR    R2, = (Heap_Mem + Heap_Size) ; high-address end of initial heap
278      LDR    R3, = Stack_Mem ; stack limit unused
279
280      BX     LR
281
282      ALIGN
283
284      ENDIF
285
286
287      END
288
289
```

# Example – Skipping the “Magic secret sauce”



# Example – Skipping the “Magic secret sauce”

```
Reset_Handler PROC
EXPORT Reset_Handler [WEAK]
IMPORT main
LDR R0, =main
BX R0
ENDP
```

```
Reset_Handler PROC
EXPORT Reset_Handler [WEAK]
IMPORT main
LDR R0, =main
BX R0
ENDP
```

- Compilation without errors.
- See the map file.
- Is it correct?

```
1
2
3  const int pippo[]= {21312,44321};
4  int this_is_zero;
5  short int this_is_not_zero=0xcafe;
6  volatile int my_array[N] = {1,2,3,4,5,6,7,8,9,10};
7
8
9
10 int main(void){
11     int i=0;
12     volatile int value = 0xcafe;
13     volatile int my_var=pippo[0];
14
15     for ( i=0;i<N;i++){
16         value=my_array[i];
17     }
18
19     while(1);
20 }
21
```




# Example – ASM SVC vs C-ASM SVC calling

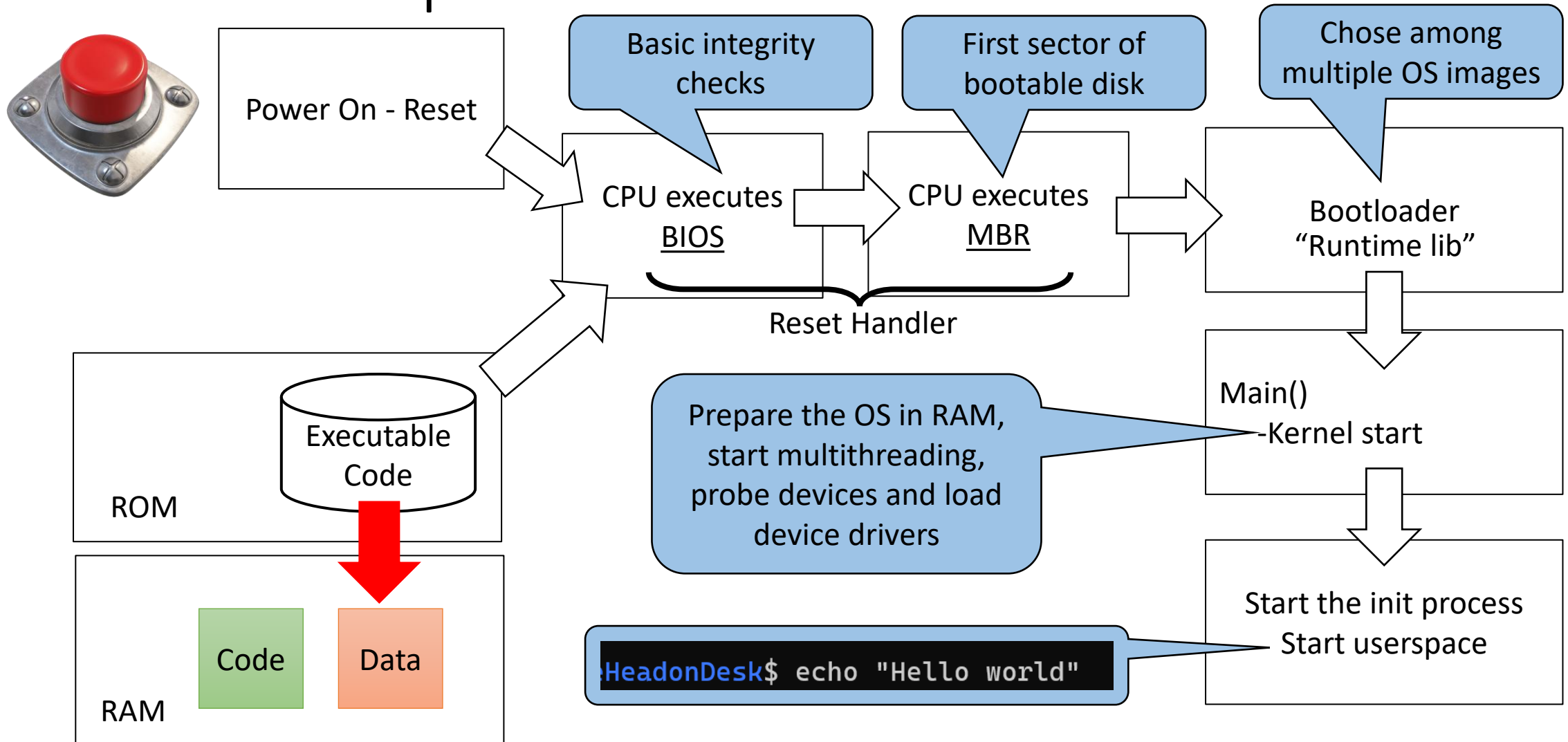
- In assembly, you have full control over the stacks.
- What happens if we call the SVC from C?

```
111 Reset_Handler  PROC
112                 EXPORT Reset_Handler
113                 import __main
114                 ; your code here
115
116                 MOV     R0, #3
117                 MSR     CONTROL, R0
118                 LDR     SP, =Stack_Mem
119
120                 nop
121
122                 SVC     0x10      ;0x000000DA
123
```

```
3  int main (void) {
4
5
6
7
8      __asm volatile("svc 0x10");
9
10     while(1);
11 }
12
```

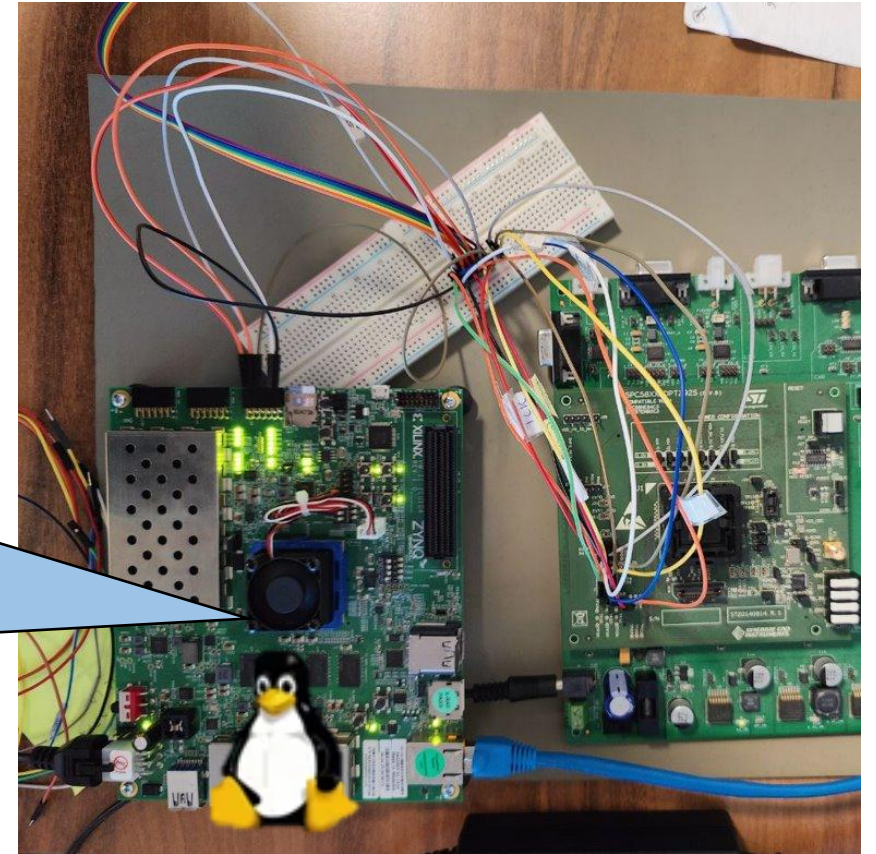
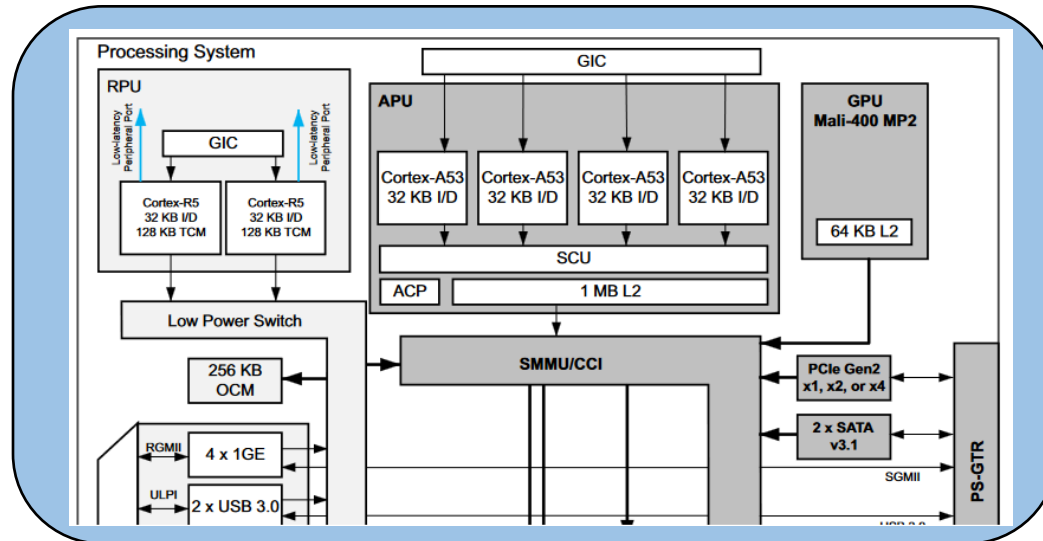


# OS Bootstrap – The linux case



# OS Bootstrap – The linux case

- Xilinx MPSoC zcu 104 Evaluation board.
- Four Cortex-A53 Arm Core.
- Running a custom linux-based operating system.



# The BIOS

- BIOS stands for Basic Input/Output System.
- Performs some system integrity checks.
- Searches, loads, and executes the boot loader program.
- It looks for boot loader in floppy, cd-rom, or hard drive. You can press a key (typically F12 or F2, but it depends on your system) during the BIOS startup to change the boot sequence.
- Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
- The BIOS loads and executes the MBR boot loader.

# The MBR

- MBR stands for Master Boot Record.
- It is located in the 1st sector of the bootable disk. Typically `/dev/hda`, or `/dev/sda`.
- MBR is less than 512 bytes in size. This has three components:
  - Primary boot loader info in 1st 446 bytes
  - Partition table info in next 64 bytes
  - MBR validation check in last 2 bytes.
- It contains information about the bootloader.
- MBR loads and executes the boot loader.

# The Bootloader

- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- A common bootloader is GRUB (Grand Unified Bootloader).
- GRUB displays a splash screen, waits for few seconds, if you don't enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem.
- Grub configuration file is `/boot/grub/grub.conf` (`/etc/grub.conf` is a link to this).
- GRUB just loads and executes Kernel and initrd images.

# The Kernel

- Mounts the root file system as specified in the “root=” in grub.conf.
- Kernel executes the /sbin/init program.
- Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1.
- initrd stands for Initial RAM Disk.
- initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.
- It starts other cores, and probe devices (loading their device drivers).

# Start the user space – the init process

- Looks at the `/etc/inittab` file to decide the Linux run level.
- Following are the available run levels
  - 0 – halt, 1 – Single user mode, 2 – Multiuser, without NFS, 3 – Full multiuser mode, 4 – unused, 5 – X11, 6 – reboot.
- Init identifies the default initlevel from `/etc/inittab` and uses that to load all appropriate program.
- Execute `'grep initdefault /etc/inittab'` on your system to identify the default run level.
- If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
- Typically you would set the default run level to either 3 or 5.



# References

- [Clang and the LLVM project](#)
- [Arm Compiler](#)
- [Arm Embedded Software Development](#)
- [Arm Image Structure and Generation](#)
- [Debugging With Arbitrary Record Formats \(DWARF\)](#) and [Executable and Linkable Format \(ELF\)](#)
- [Linux Boot Process](#)