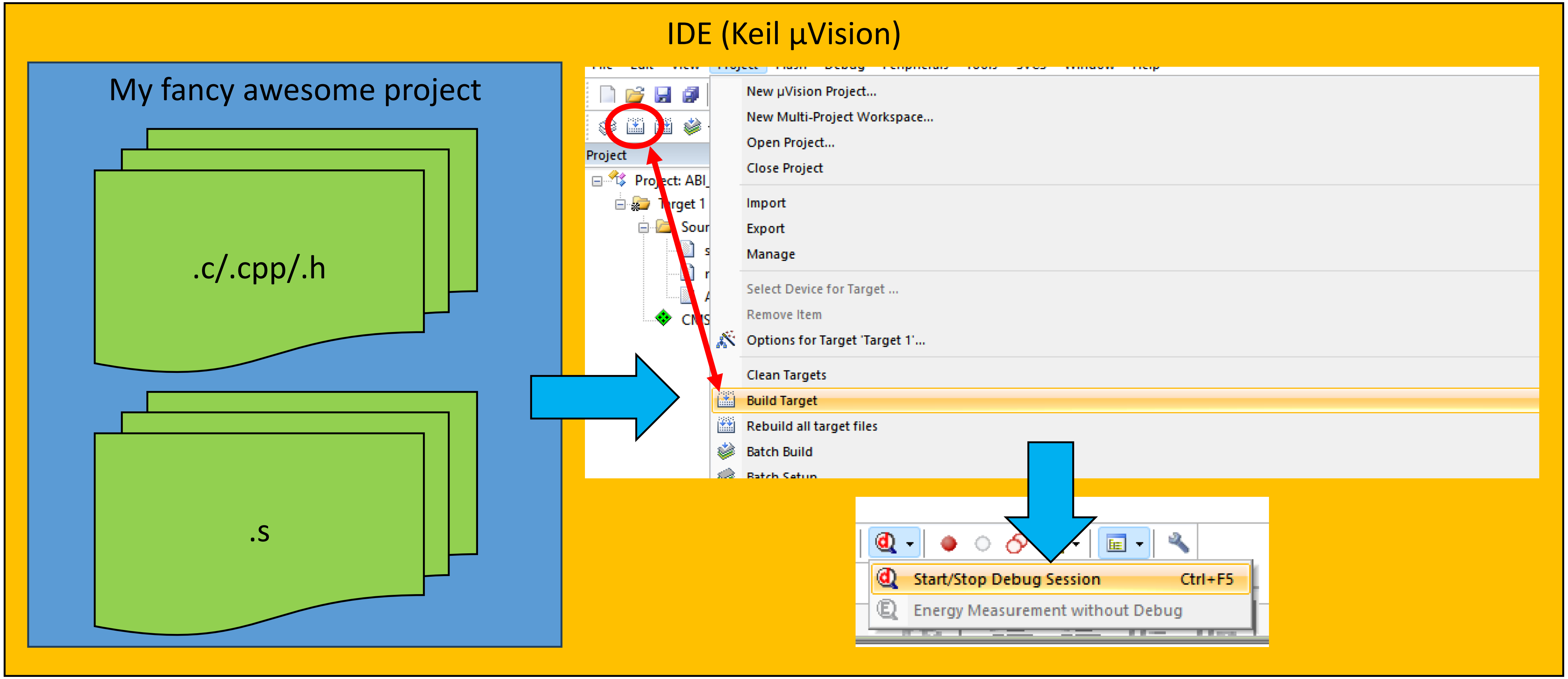
From Source code to Executable
The Arm Toolchain for Embedded Systems

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

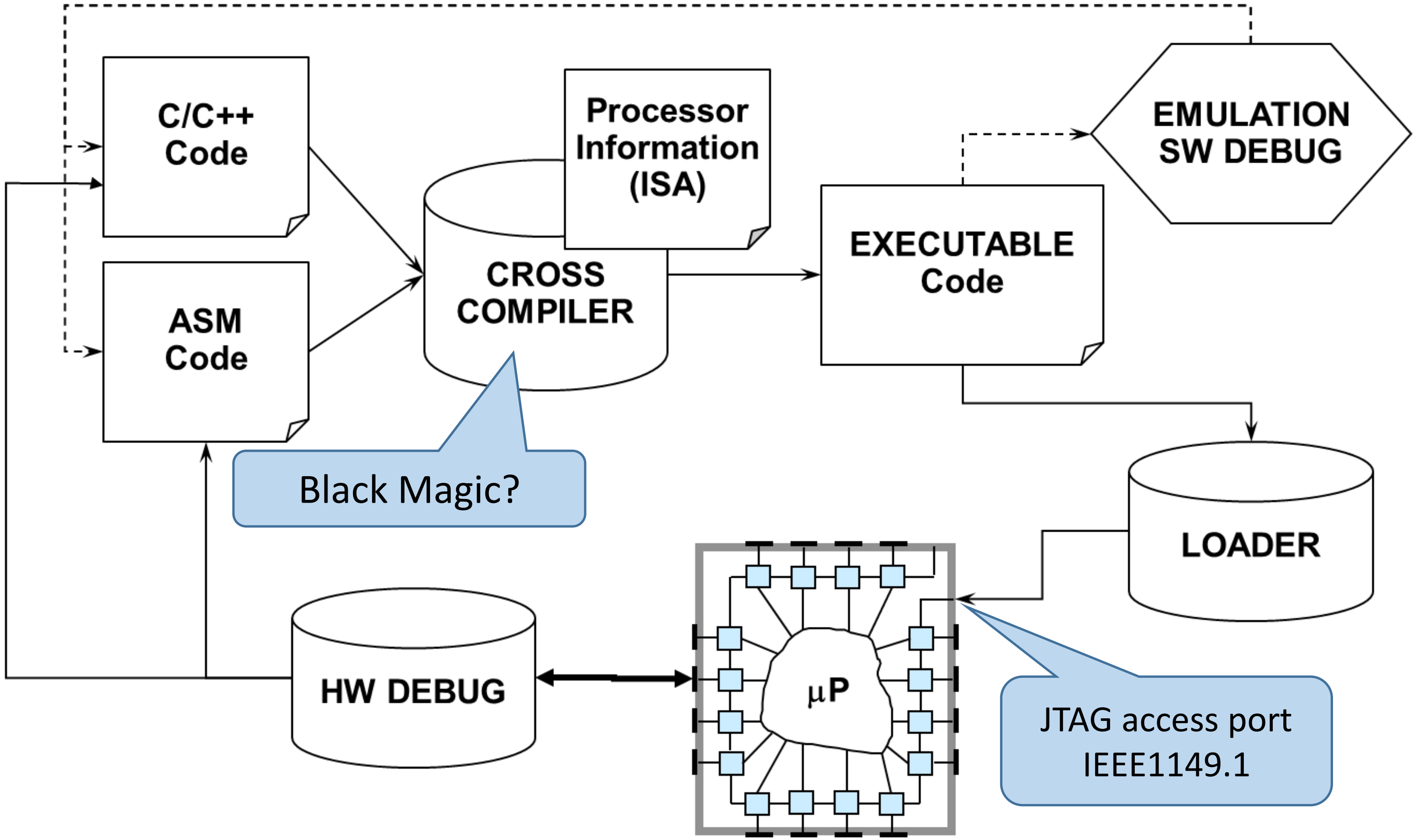
- What is a toolchain?
  - The Arm toolchain.
  - Investigating the compilation output files.
- How does a System-on-Chip start the program?
  - The Arm “Magic secret sauce”.

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”.

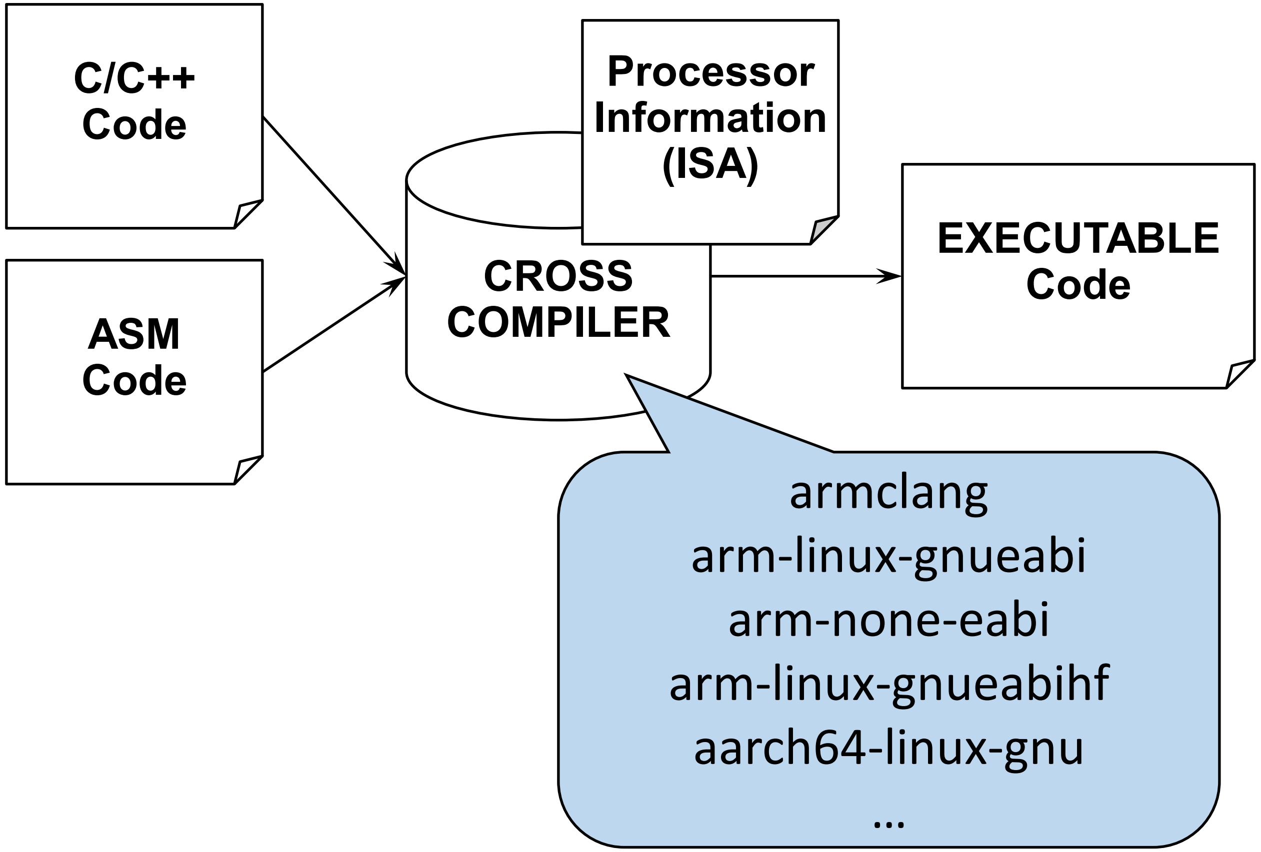
What is a toolchain?

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



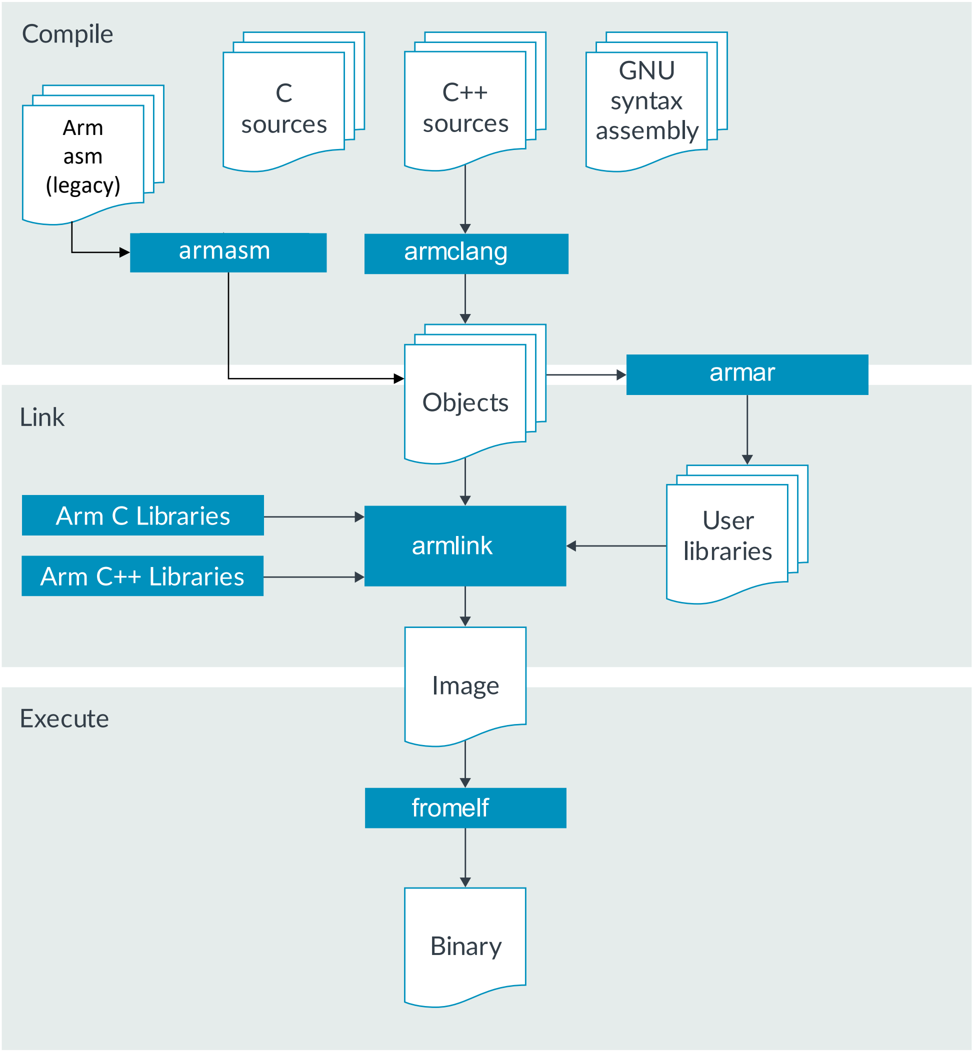
What does the cross-compiler?

- 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

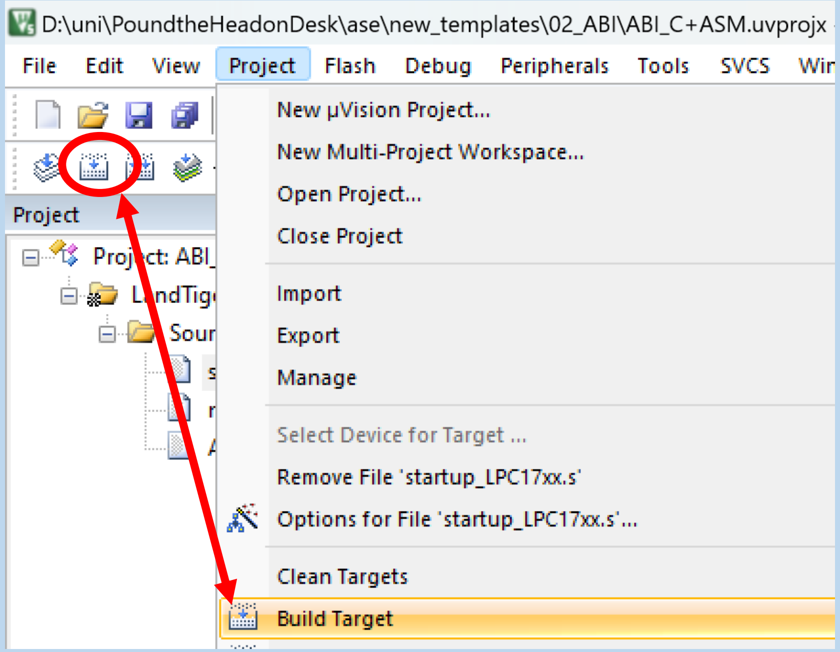
- Based on an enhanced version of *clang* (frontend) and *llvm project*, with proprietary customizations.
- The toolchain is composed of 3 different phases:
  - Preprocessing and compilation phase (armasm for legacy support).
  - Link phase.
  - Execute phase.

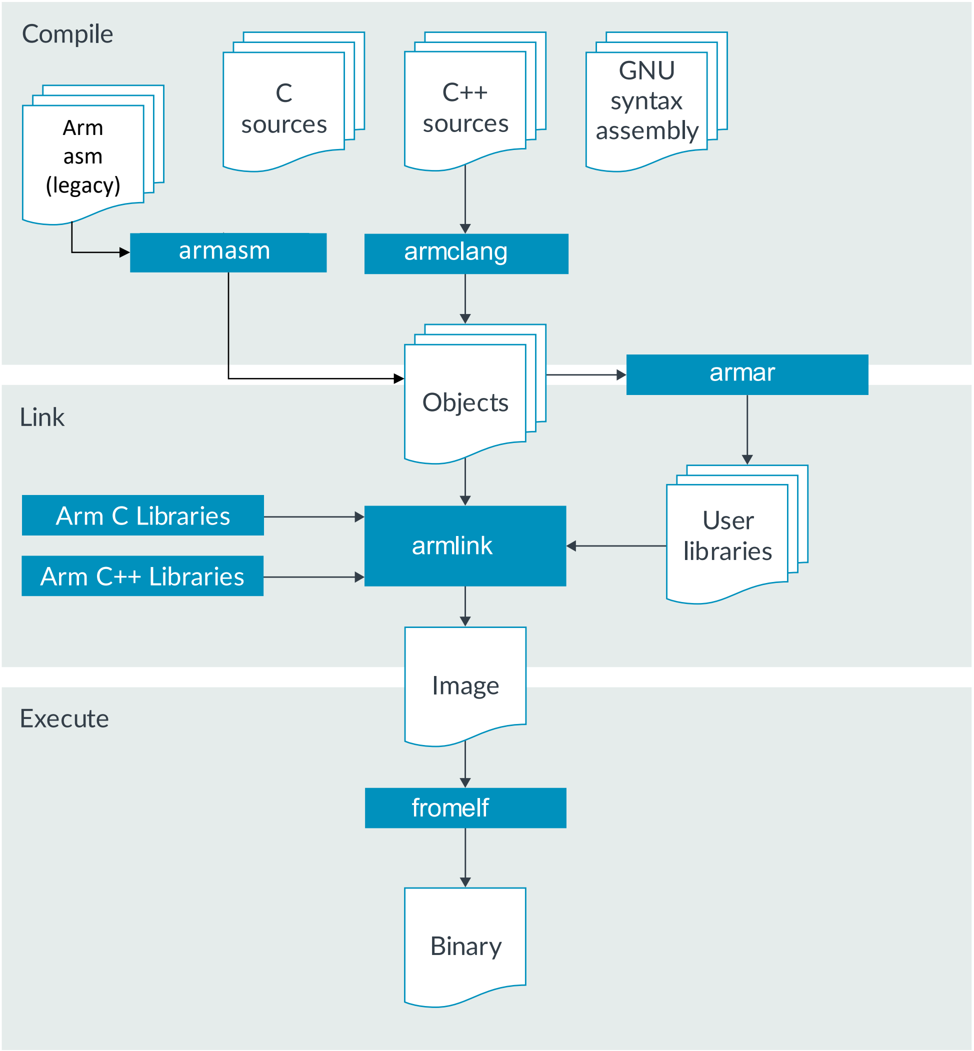


The Arm Toolchain

- Based on an enhanced version of
  cla
  w
- Th
  d

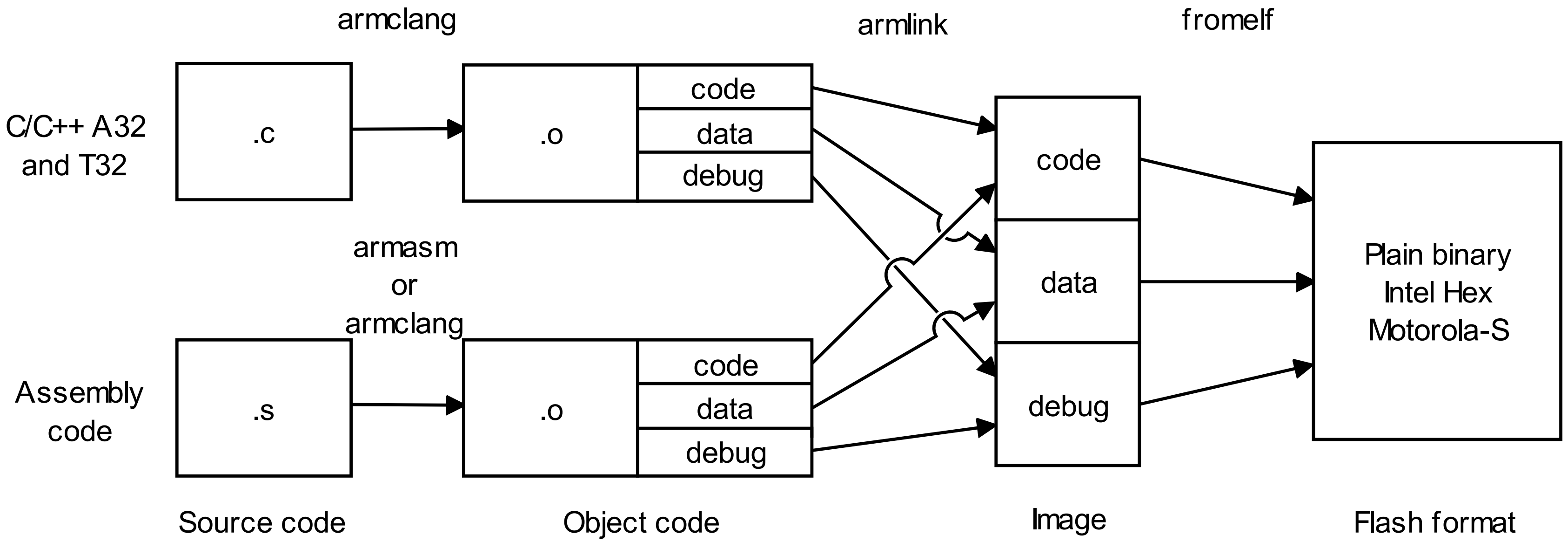
All the three phases included in:



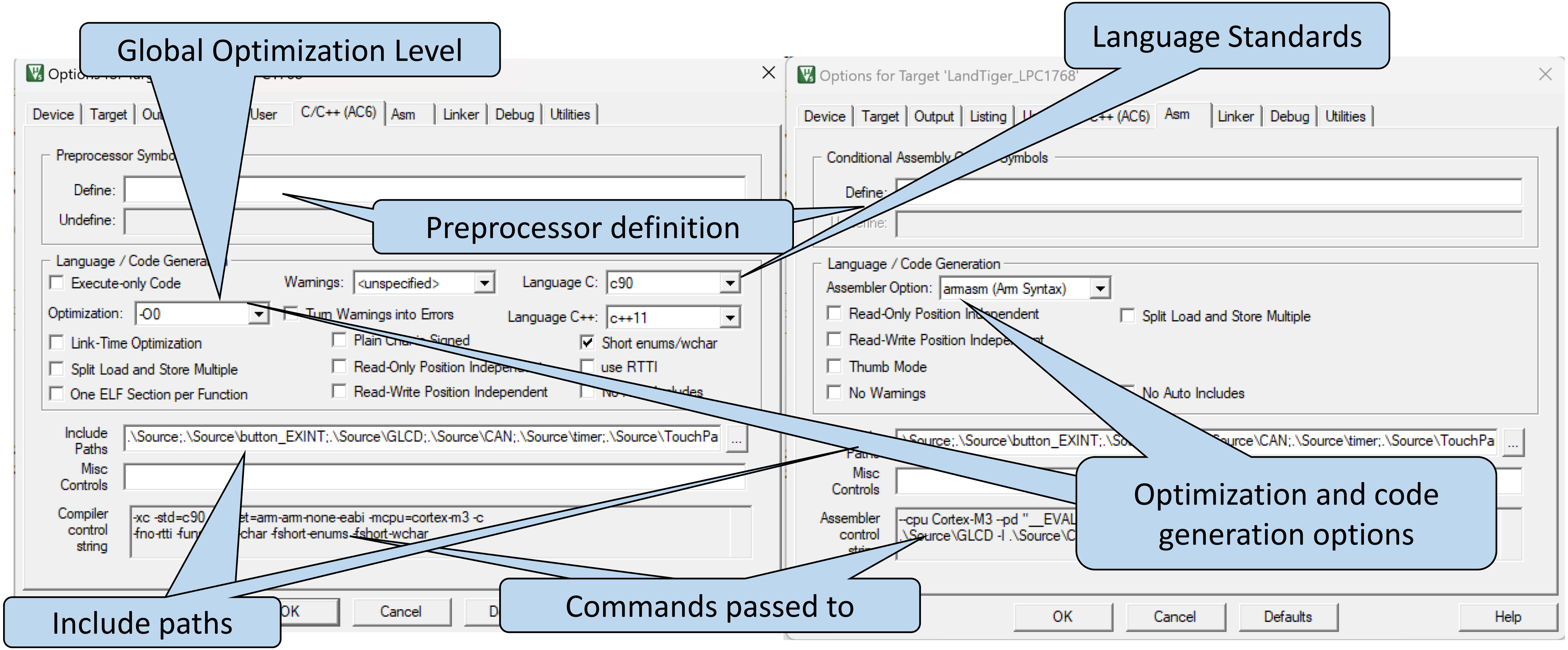


The compile phase – armclang and armasm

- 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.

```c
volatile int my_array[N];
int main(void){

  int i;

  for (i=0;i<N;i++){
  my_array[i]=i*3;
  }
  while(1);
}
```

```c
volatile int my_array[N];
int main(void){

  for ( int i=0;i<N;i++){
  my_array[i]=i*3;
  }

  while(1);
}
```

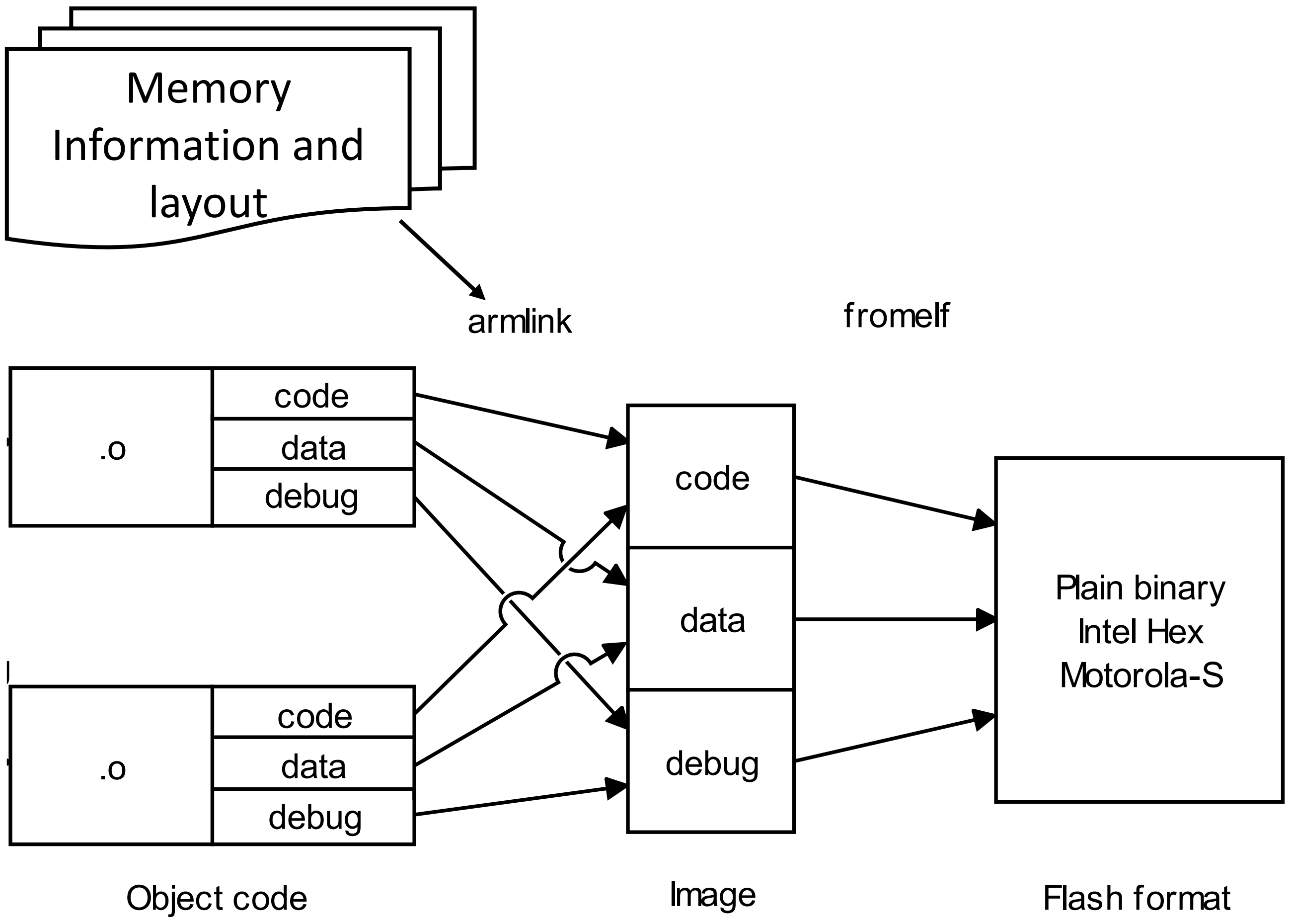
```
*** Using Compiler 'V6.22', folder: 'D:\programmi\keil\ARM\ARMCLANG\Bin'
Rebuild target 'LandTiger_LPC1768'
assembling ASM_funct.s...
assembling startup_LPC17xx.s...
creating list file for main.c...
creating preprocessor file for main.c...
Source/main.c(8): warning: GCC does not allow variable declarations 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).

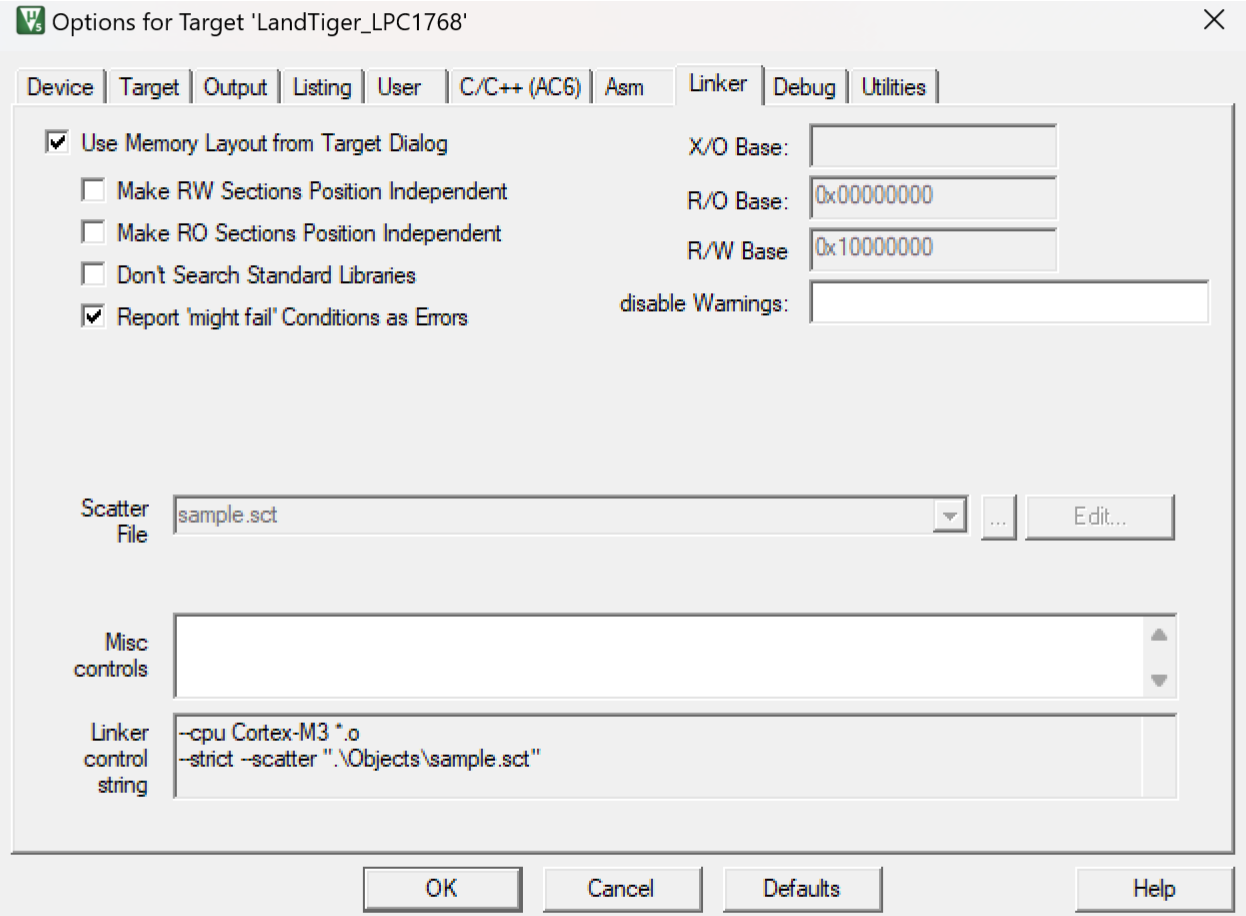
The link phase – armlink

- **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.



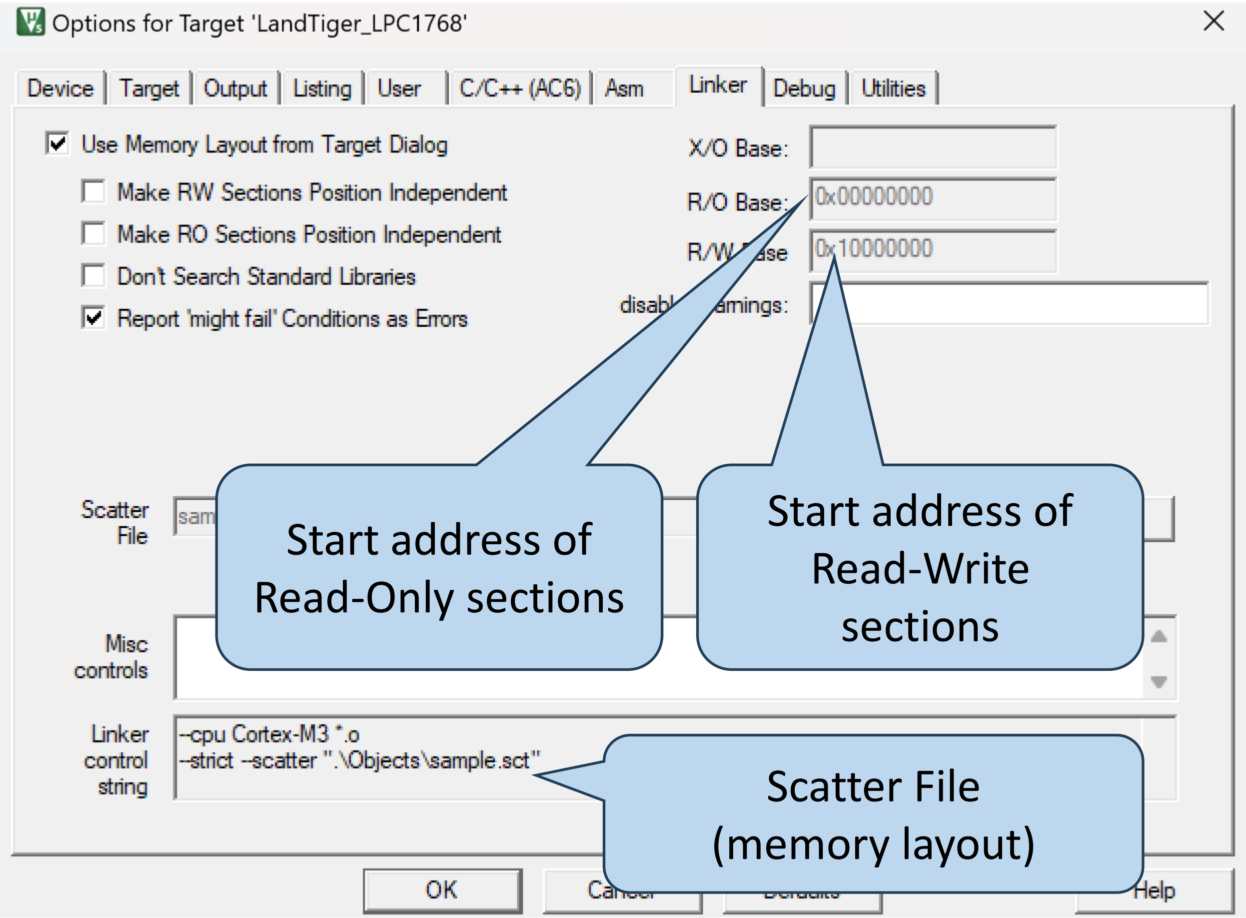
Memory Information and layout

- You can sp nt at the startu called at the syst
- the m ory
  - By co line usi armlink tool.
  - By pass a scatter fi .
- You can sp fy additional custom code and da a se ions.

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

Start address of Zero Init sections

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



Start address of Read-Only sections

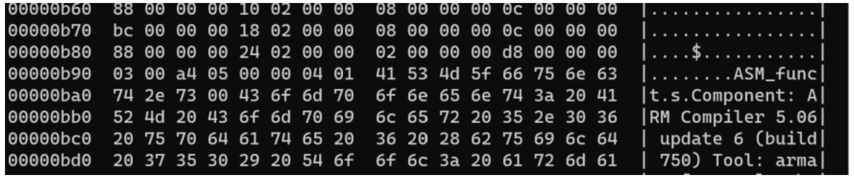
Start address of Read-Write sections

Scatter File (memory layout)

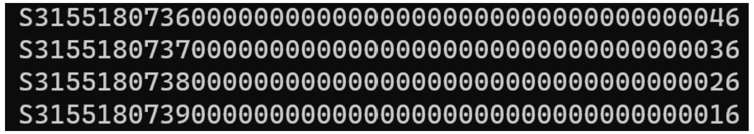
The execute phase - fromelf

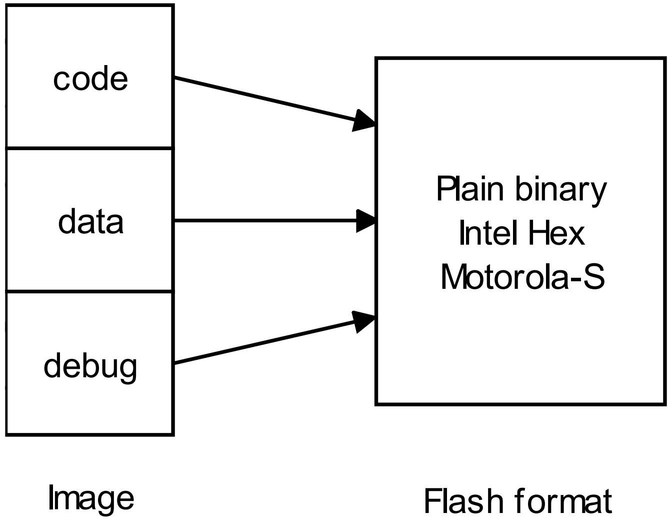
- Process object and image files.
- Convert ELF images into other formats for use by ROM tools or for direct loading into memory. The formats available are:
    - Plain binary.
    - Motorola 32-bit S-record.
    - Intel Hex-32.
    - Byte oriented hexadecimal.
- Display information about the input file,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

Binary



Motorola



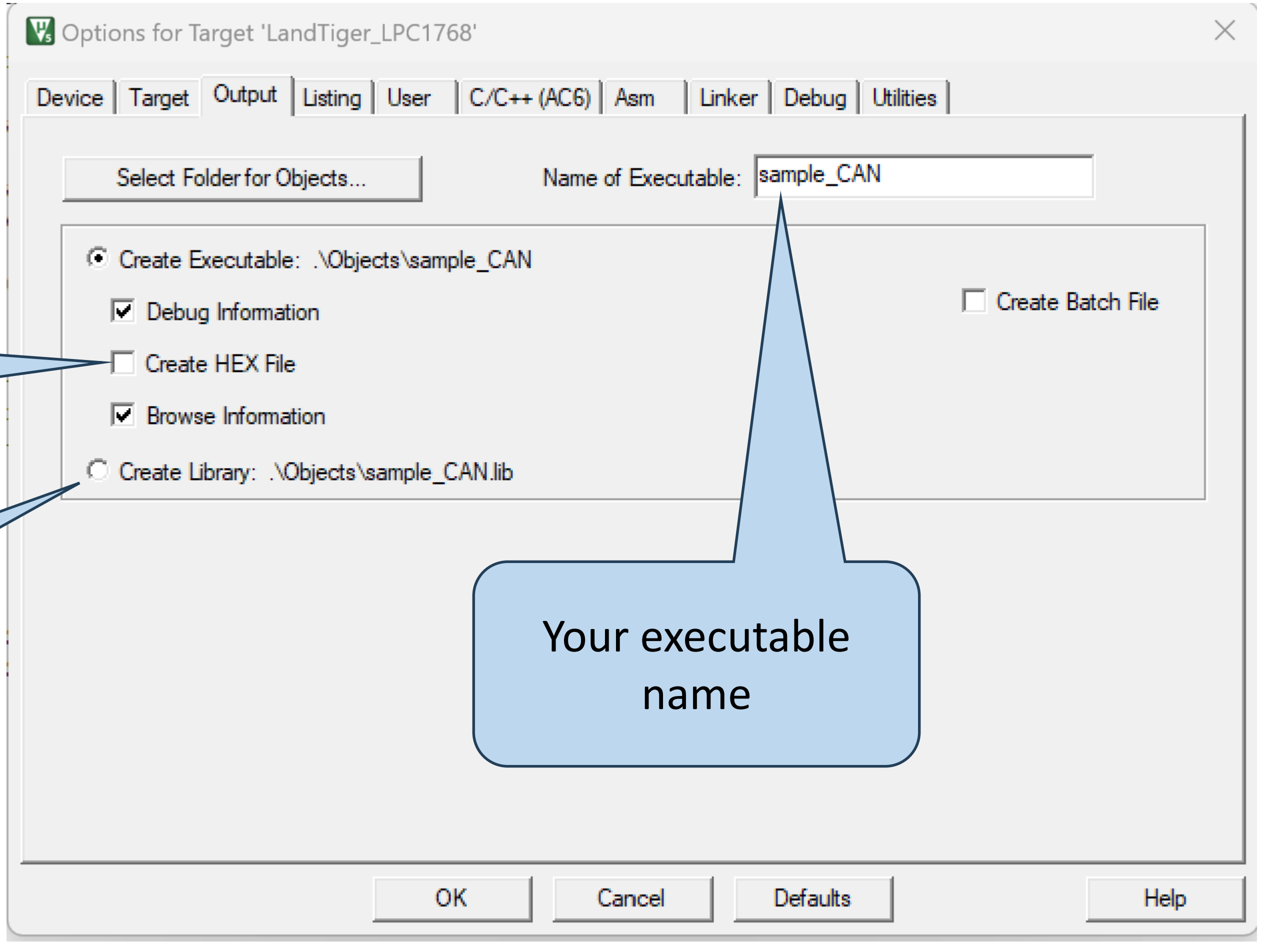


The execute phase - fromelf

- Process object and image files.
- Convert ELF images into other formats for use … for direct loading i… formats availa…
    - Plain binary.
    - Motorola 32-bit S-record.
    - Intel Hex-32.
    - Byte orien…
- Display inform… …ut file, for exampl… …ut or symbol listin…

If you want a hex file

If you want a library to be included into another project



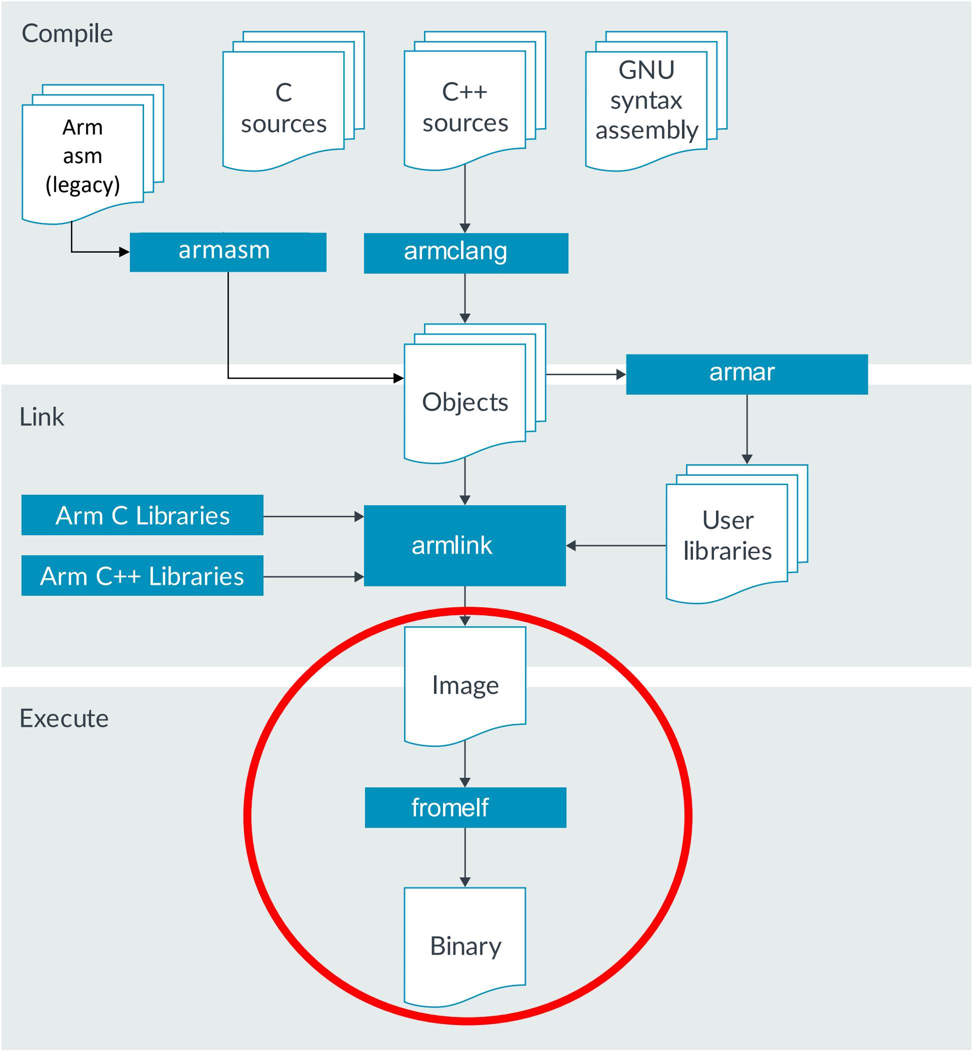
Your executable name

Outline

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

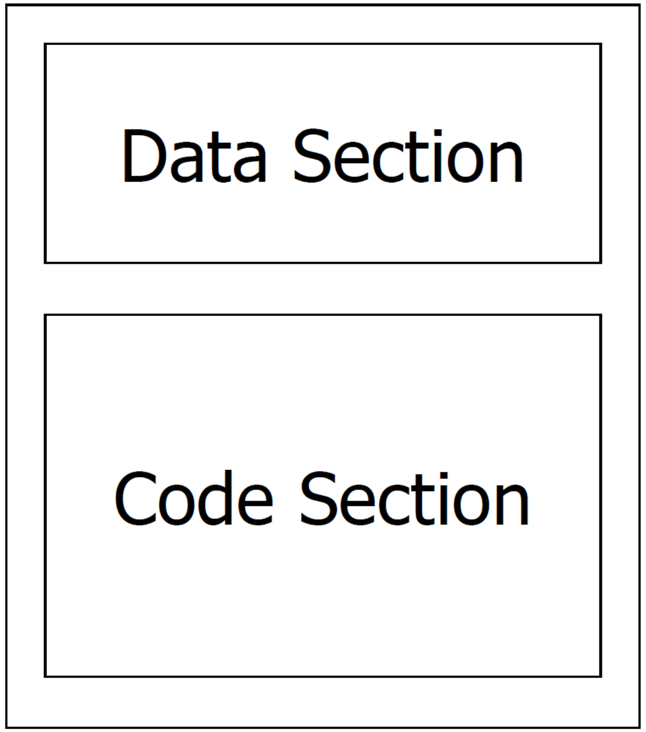
Investigating the compilation output files.

- 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.

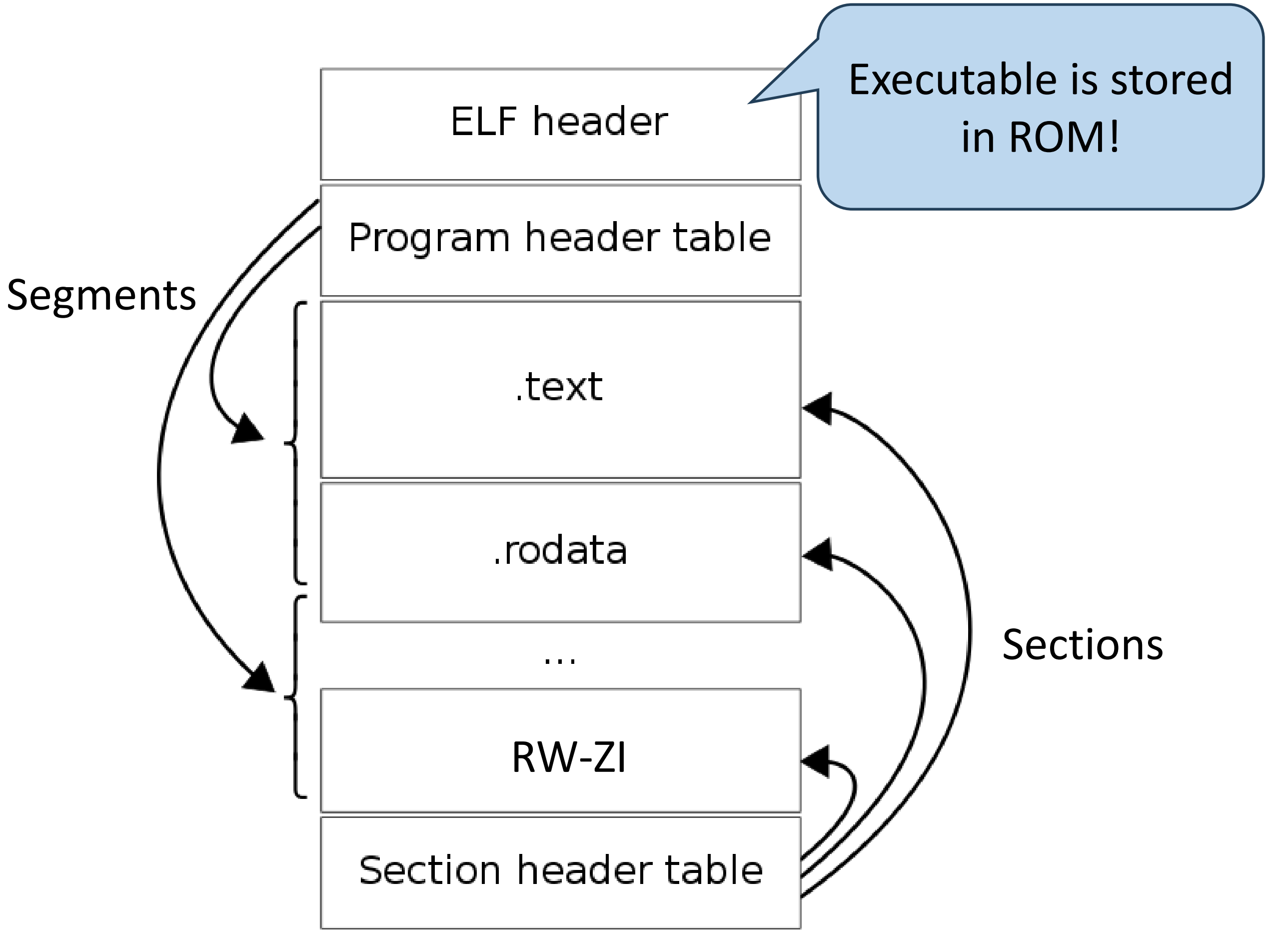


My fancy awesome Image program

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

The executable – Load view

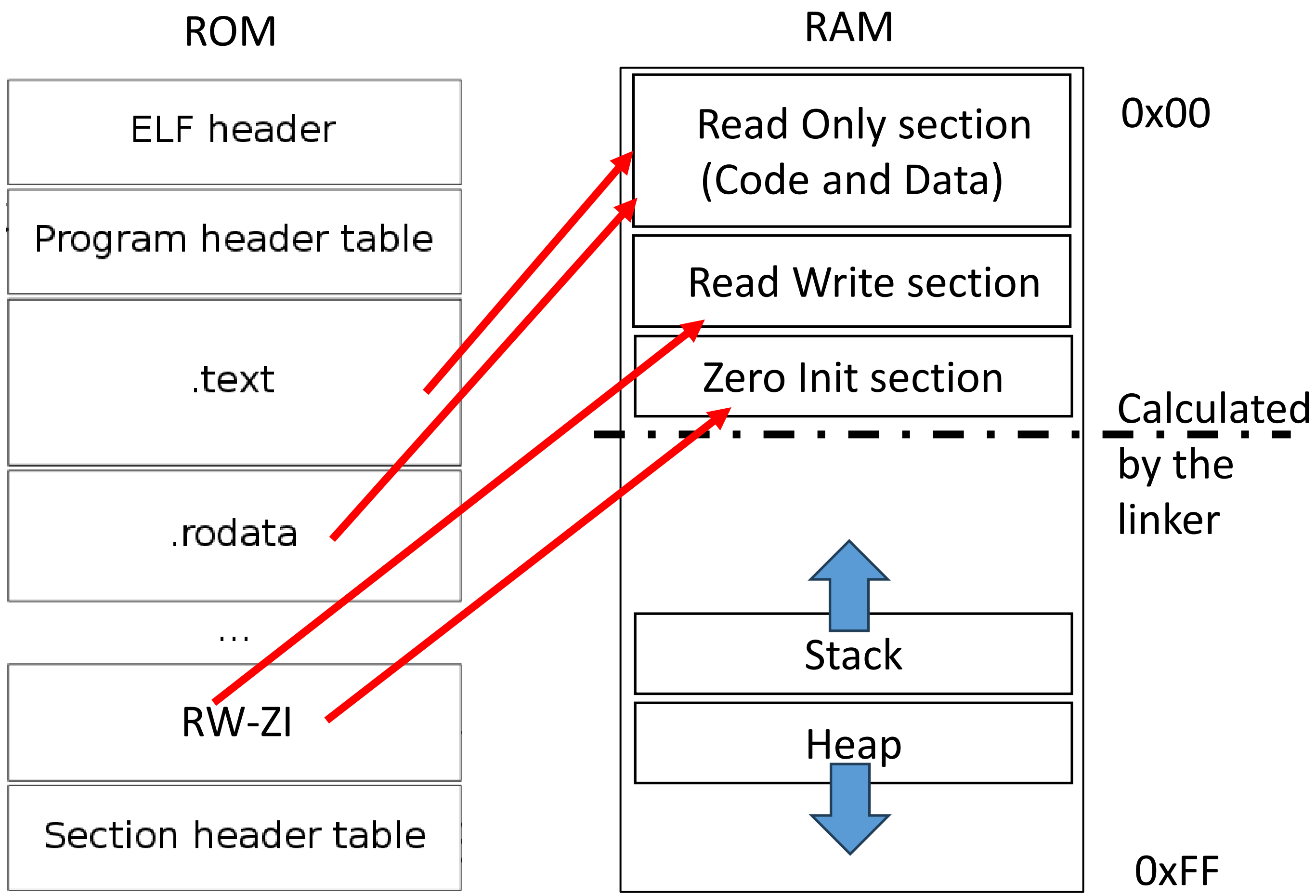
- The overall image (from the source code) is converted into an executable (.exe, .elf , .axf for Arm).
- Data and Code sections are in the executable.
- Composed of:
  - Entry address.
  - Stack and heap information.
  - Sections, used by the linker.
  - Segments, used by the loader (at runtime).



My fancy awesome Image program (.axf)

The executable – Execution view

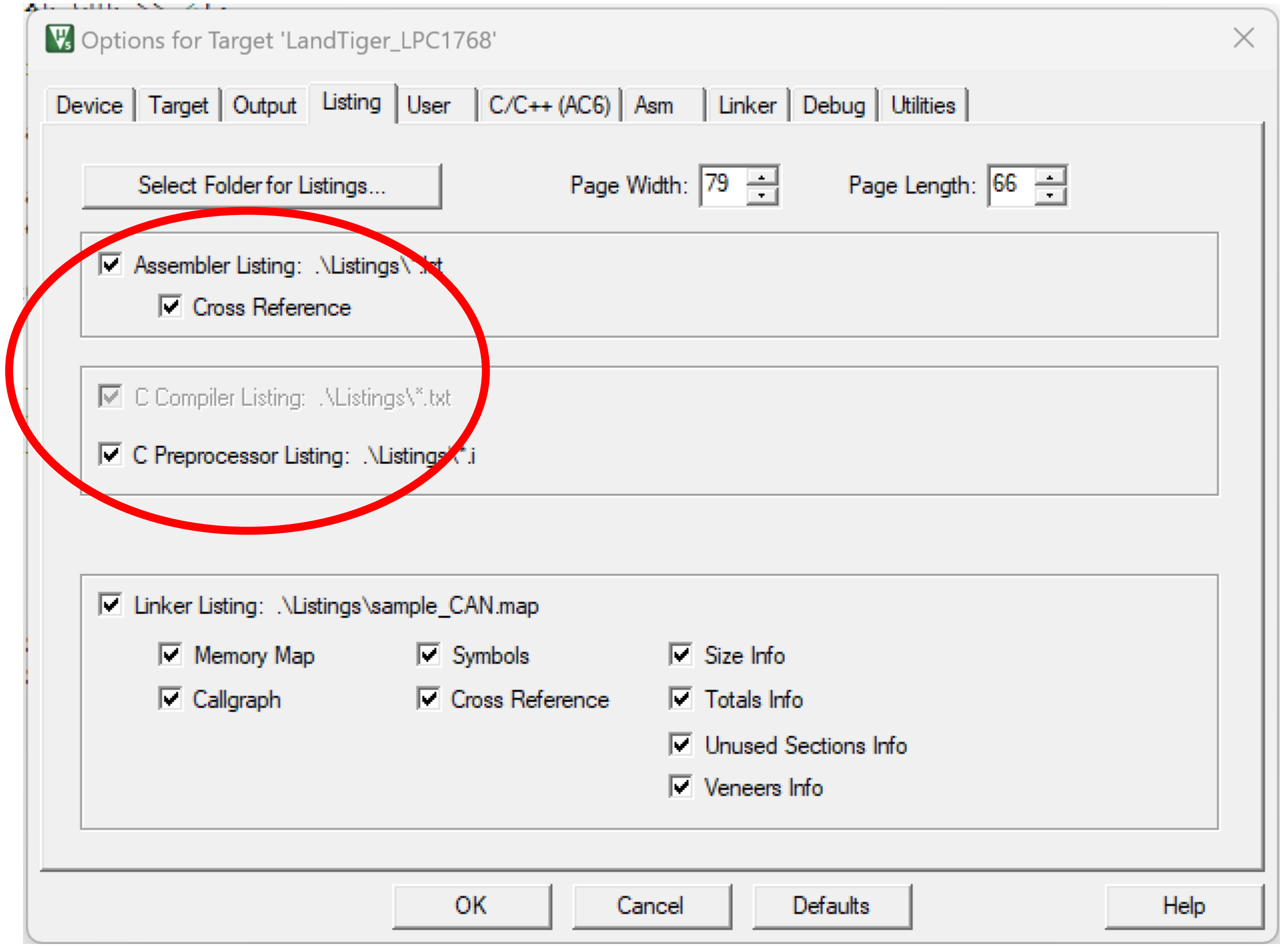
- Before the image is executed:
  - Move executable segments from ROM to their execution addresses in RAM.
  - RW data must be copied from its load address in the ROM to its execution address in the RAM.
- Runtime memory layout information is calculated offline:
  - Stack and heap execution address and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



My fancy awesome Image program (.axf)

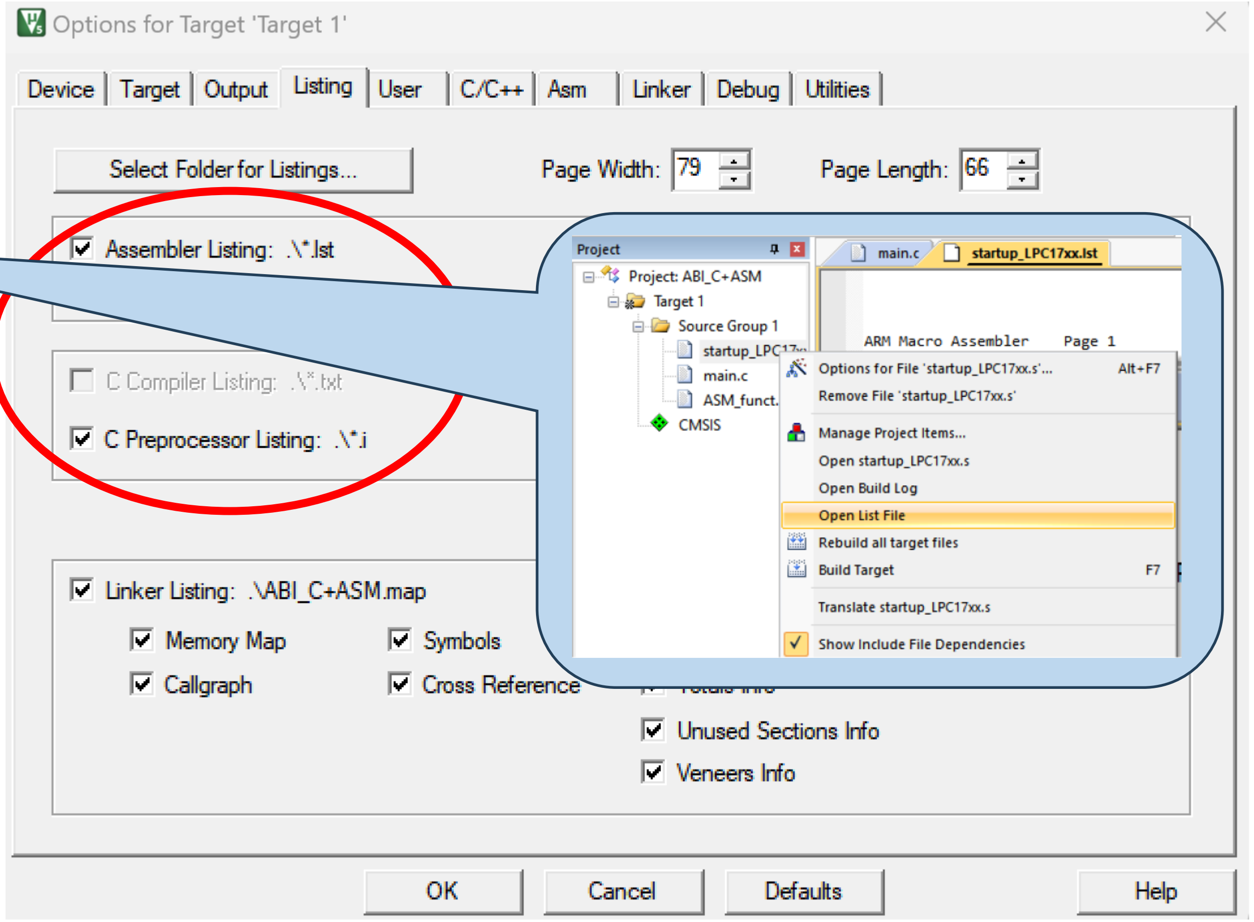
The Listing and dependencies files

- 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.



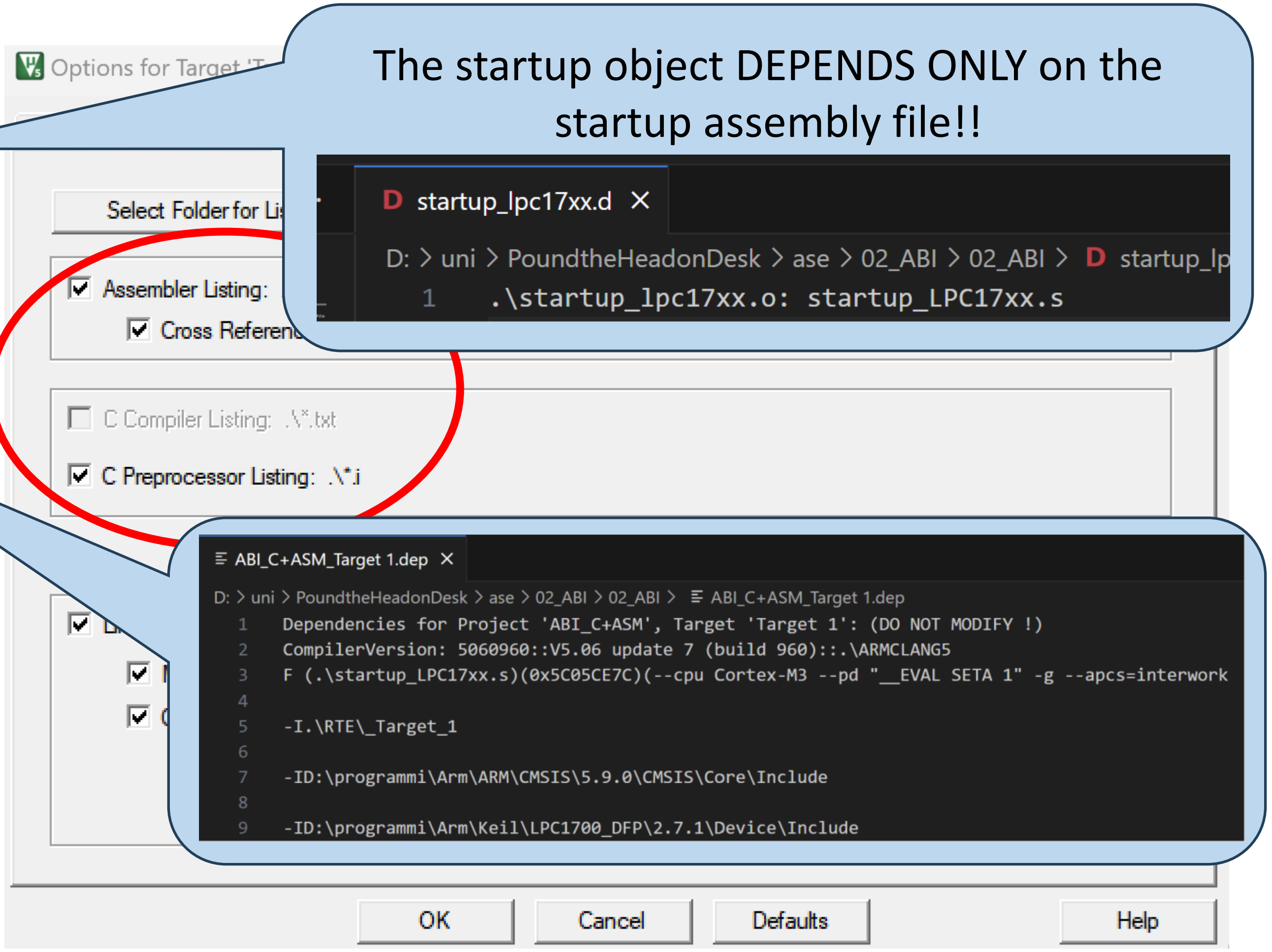
The Listing and dependencies files

- 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.



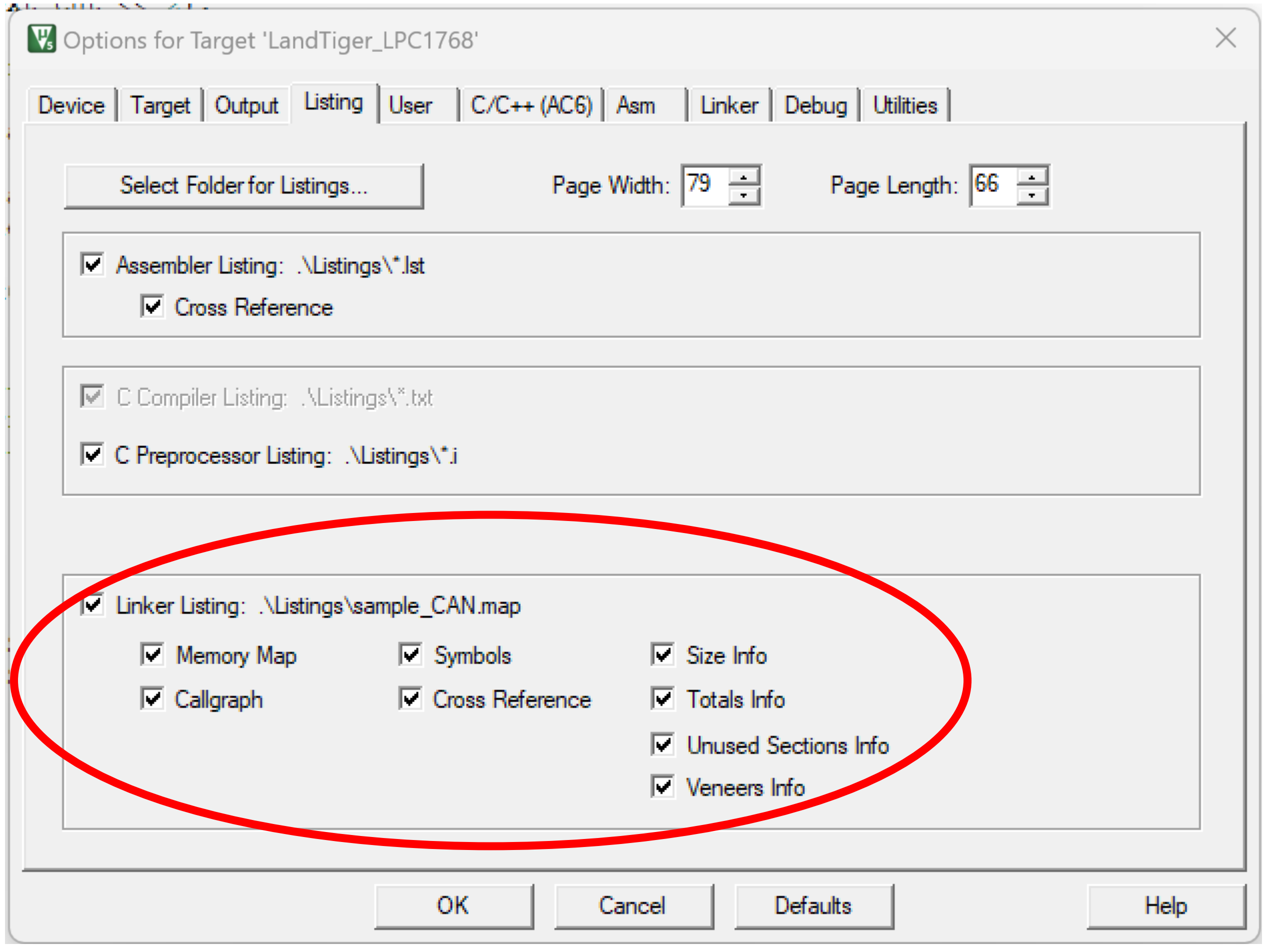
The Listing and dependencies files

- 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.



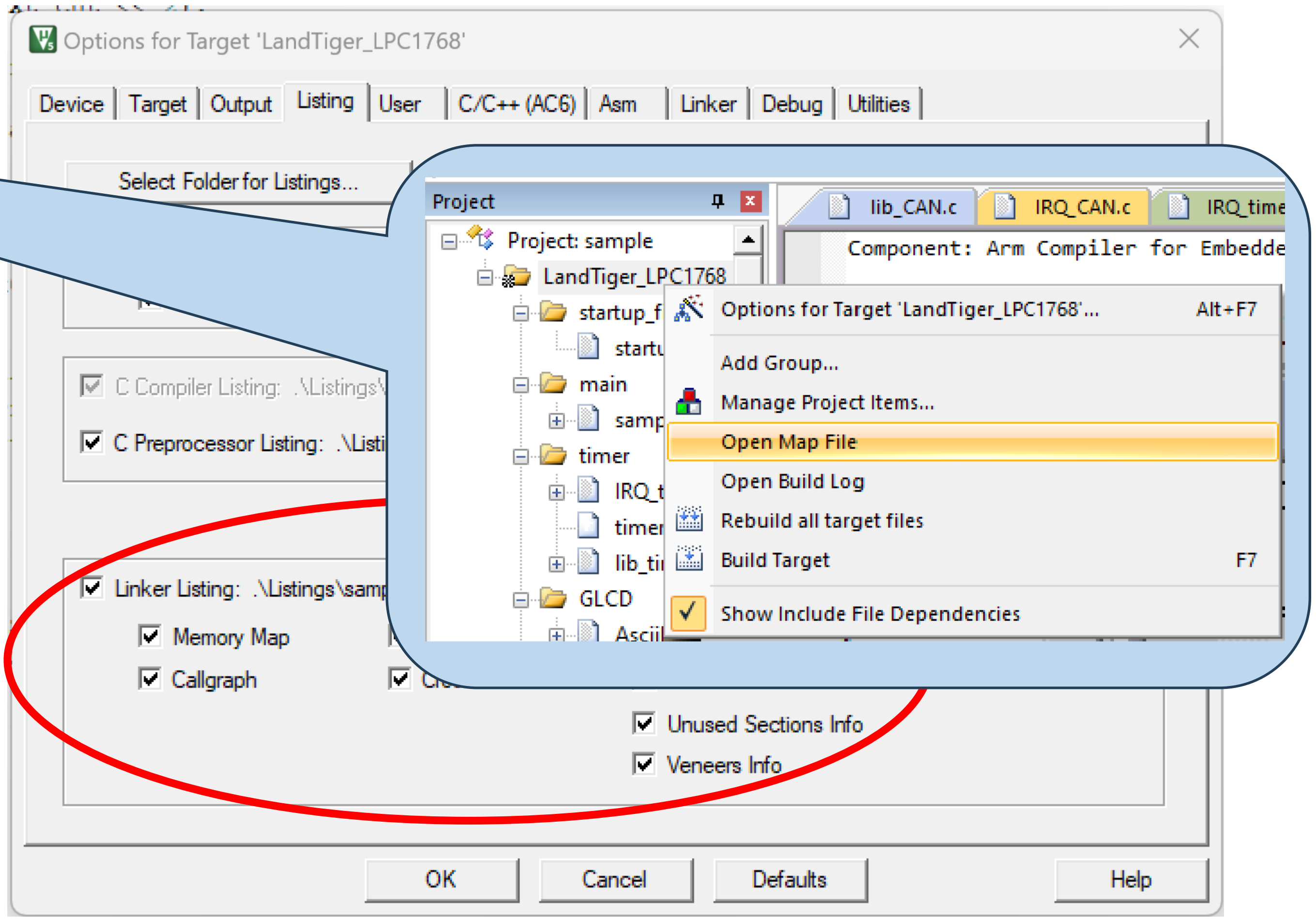
The map

- 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.



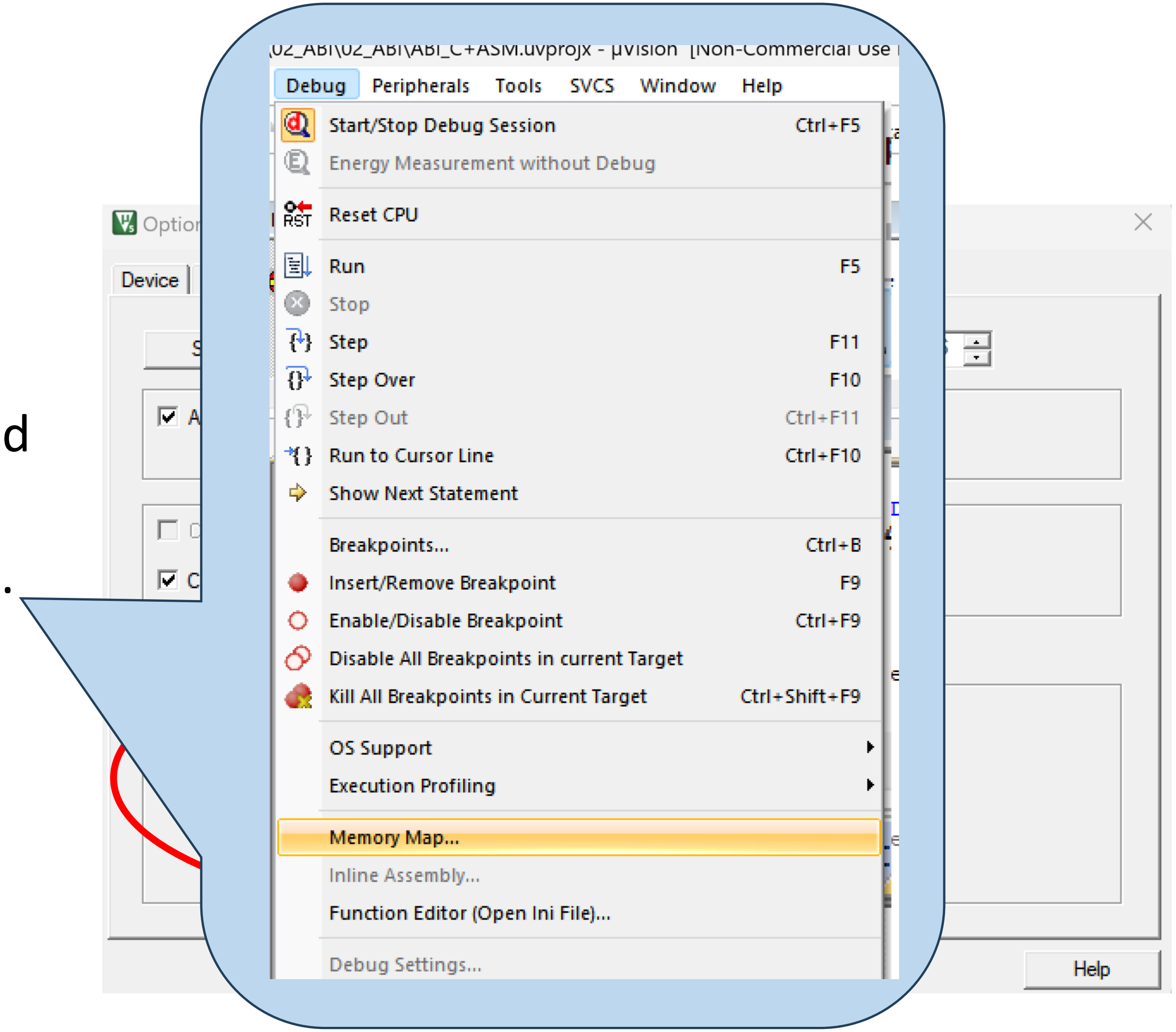
The map

- 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?

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

```
Image Symbol Table

    Local Symbols

    Symbol Name                                   Value      Ov Type          Size   Object(Section)
```

```
    Global Symbols

    Symbol Name                                   Value      Ov Type          Size   Object(Section)
```

```c


const int pippo[]= {21312,44321};
int this_is_zero;
short int this_is_not_zero=0xcafe;
volatile int my_array[N];



int main(void){
   int i=0;

   volatile int value=pippo[0];

   for ( i=0;i<N;i++){
   my_array[i]=i*3;
   }

   while(1);
}

```

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:

ABI_C+ASM.axf
ABI_C+ASM.build_log.htm
ABI_C+ASM.htm

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_funct

Mutually Recursive functions

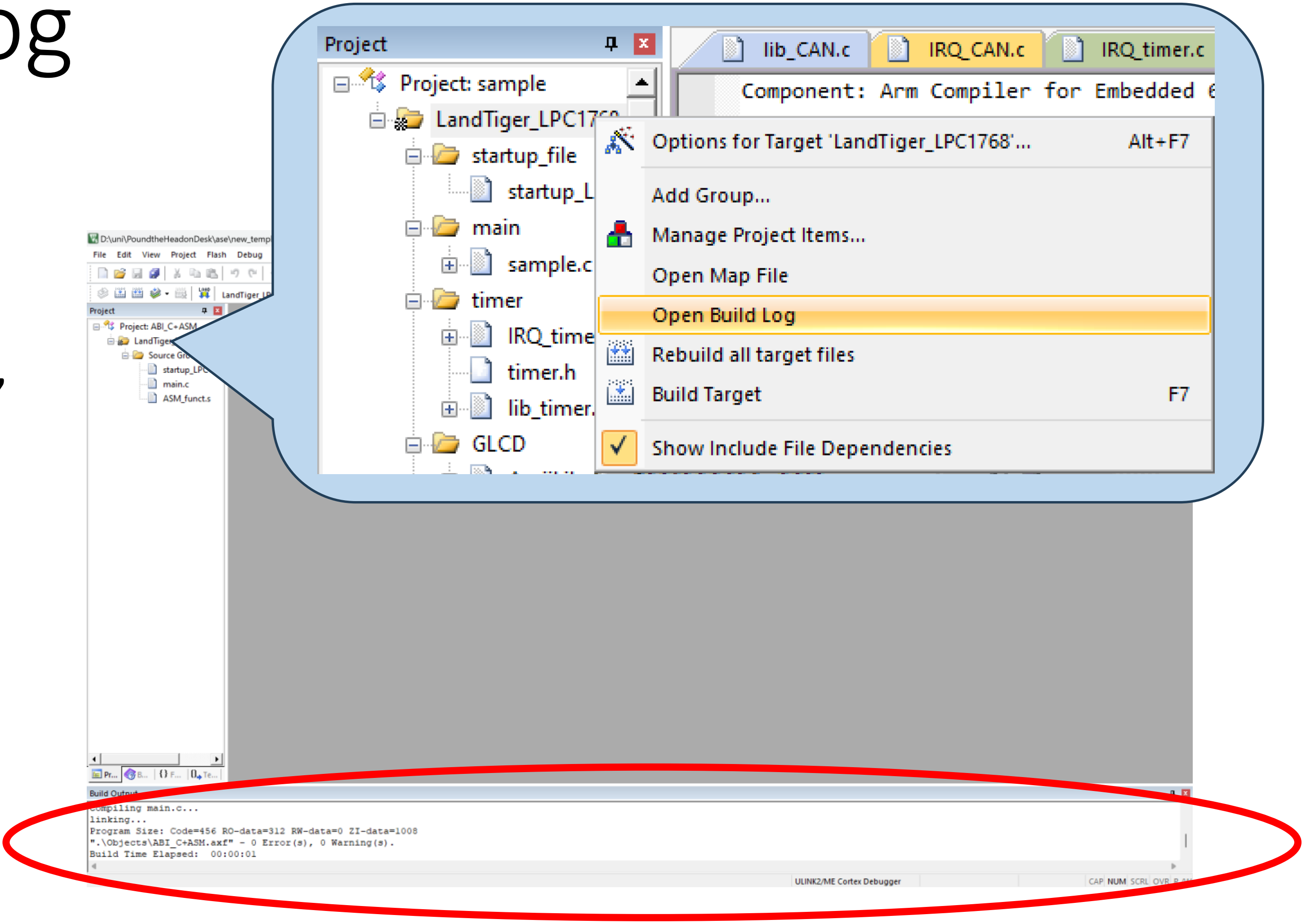
- NMI_Handler ⇒ NMI_Handler
- HardFault_Handler ⇒ HardFault_Handler
- MemManage_Handler ⇒ MemManage_Handler
- BusFault_Handler ⇒ BusFault_Handler
- UsageFault_Handler ⇒ UsageFault_Handler
- SVC_Handler ⇒ SVC_Handler
- DebugMon_Handler ⇒ DebugMon_Handler
- PendSV_Handler ⇒ PendSV_Handler
- SysTick_Handler ⇒ SysTick_Handler
- ADC_IRQHandler ⇒ ADC_IRQHandler

Function Pointers

- ADC_IRQHandler from startup_lpc17xx.o(.text) referenced from startup_lpc17xx.o(RESET)
- ASM_funct from asm_funct.o(asm_functions) referenced from asm_funct.o(asm_functions)
- BOD_IRQHandler from startup_lpc17xx.o(.text) referenced from startup_lpc17xx.o(RESET)

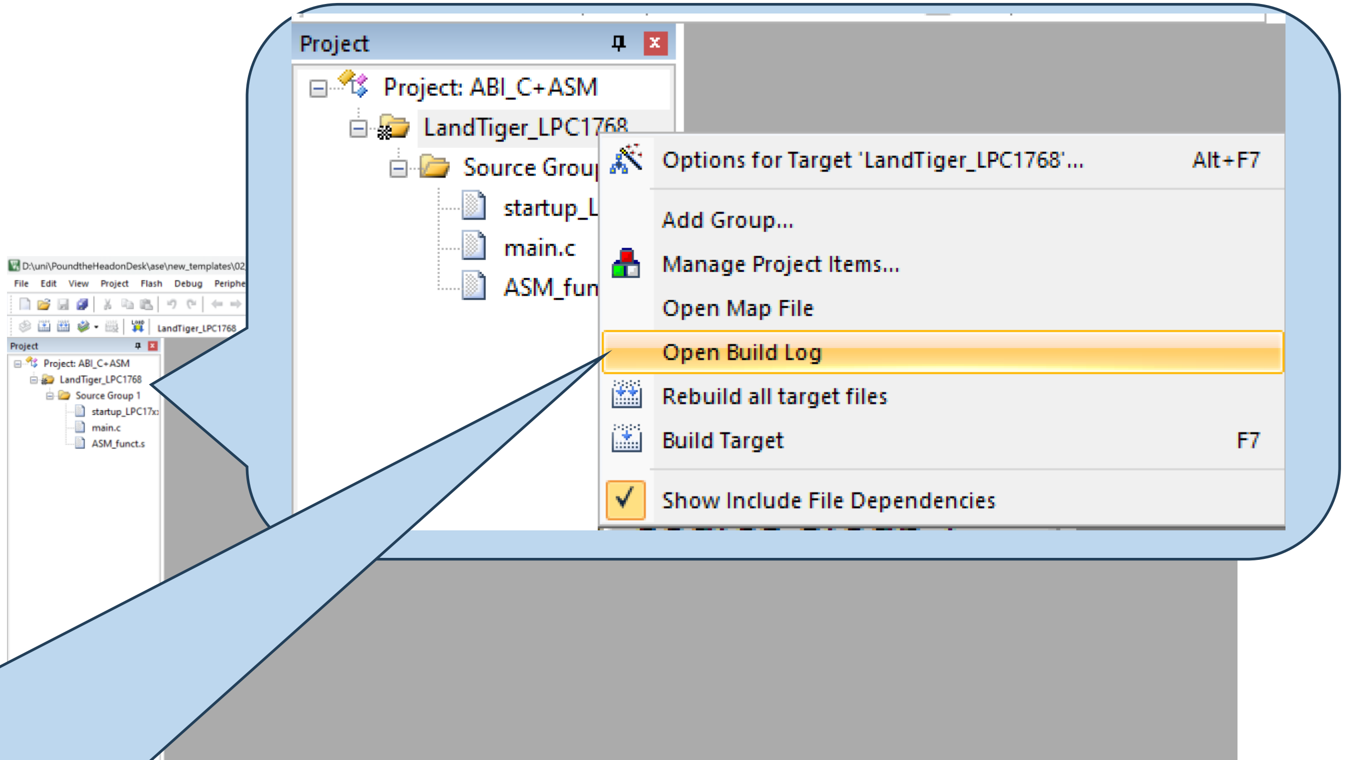
The build output log

- 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.

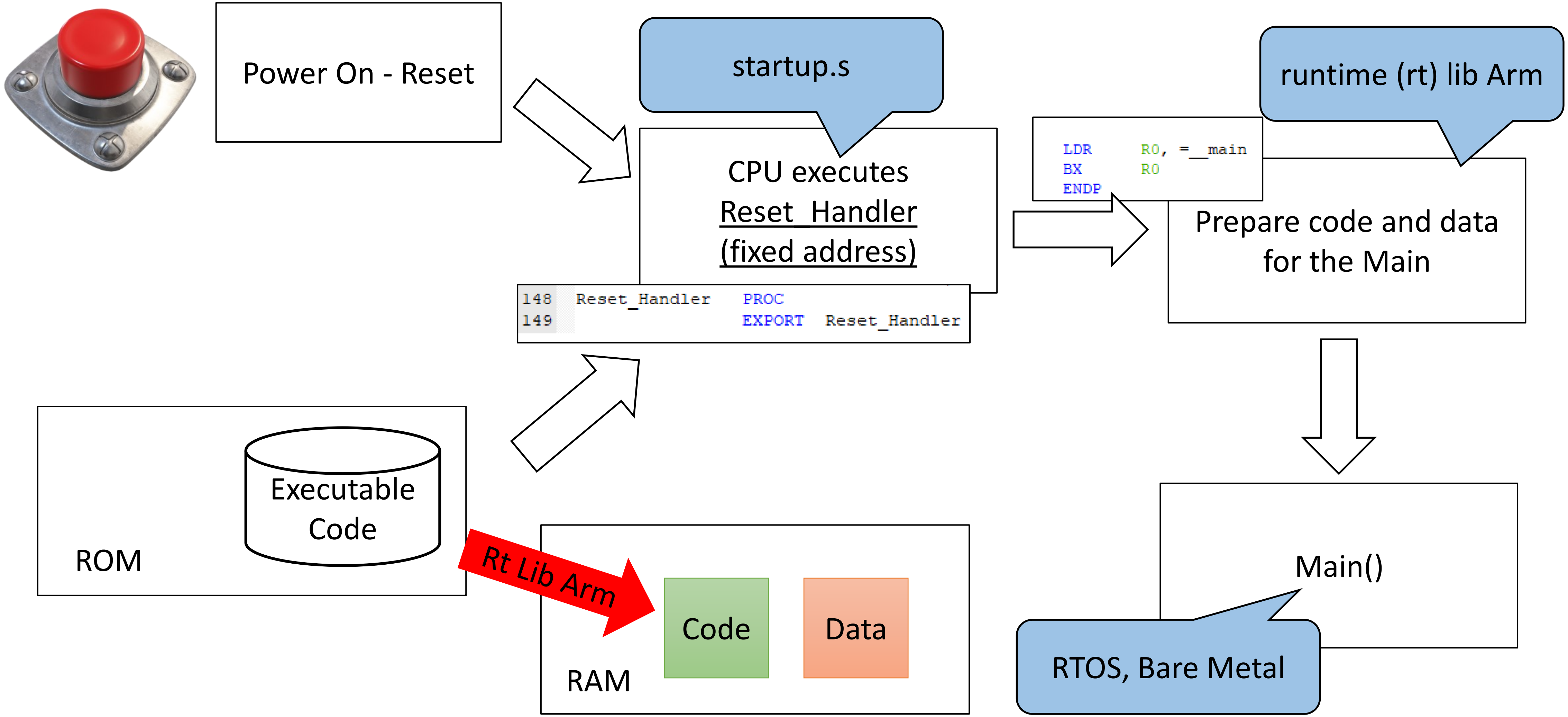



```
*** Using Compiler 'V5.06 update 7 (build 960)', folder: 'D:\programmi\keil\ARM\ARMCLANG5\Bin'
Rebuild target 'Target 1'
assembling ASM_funct.s...
compiling main.c...
assembling startup_LPC17xx.s...
linking...
Program Size: Code=384 RO-data=384 RW-data=0 ZI-data=608
".\ABI_C+ASM.axf" - 0 Error(s), 0 Warning(s).
```

Outline

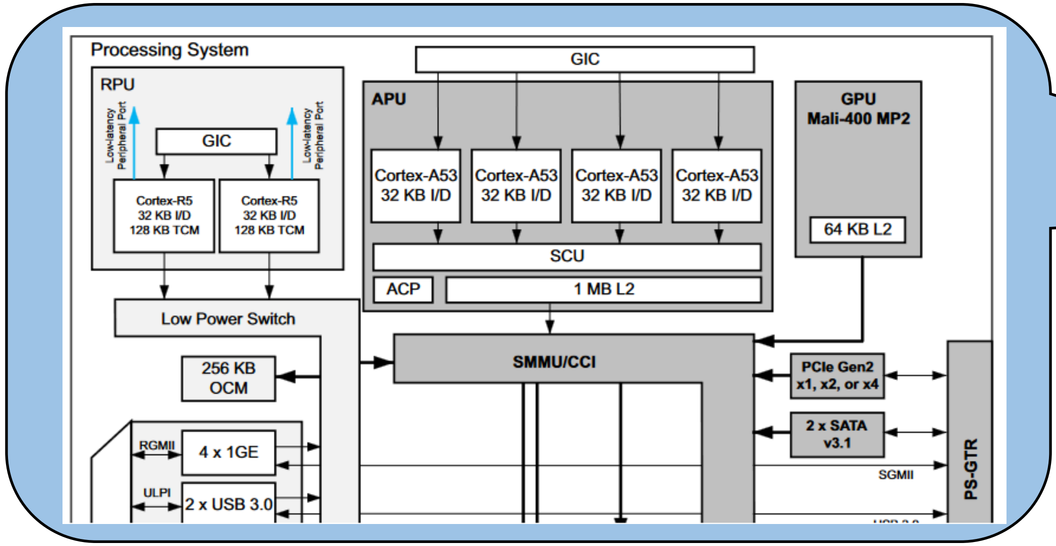
- What is a toolchain?
  - The Arm toolchain.
  - Investigating the compilation output files.
- How does a System-on-Chip start the program?
  - The Arm “Magic secret sauce”.

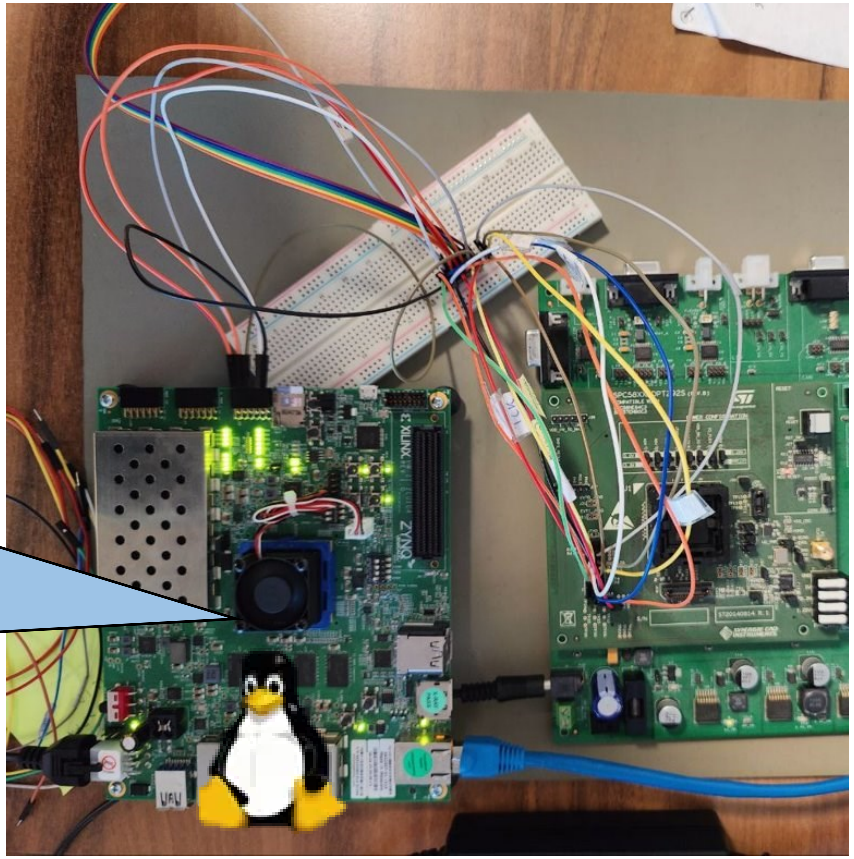
How does a System-on-Chip start the program?



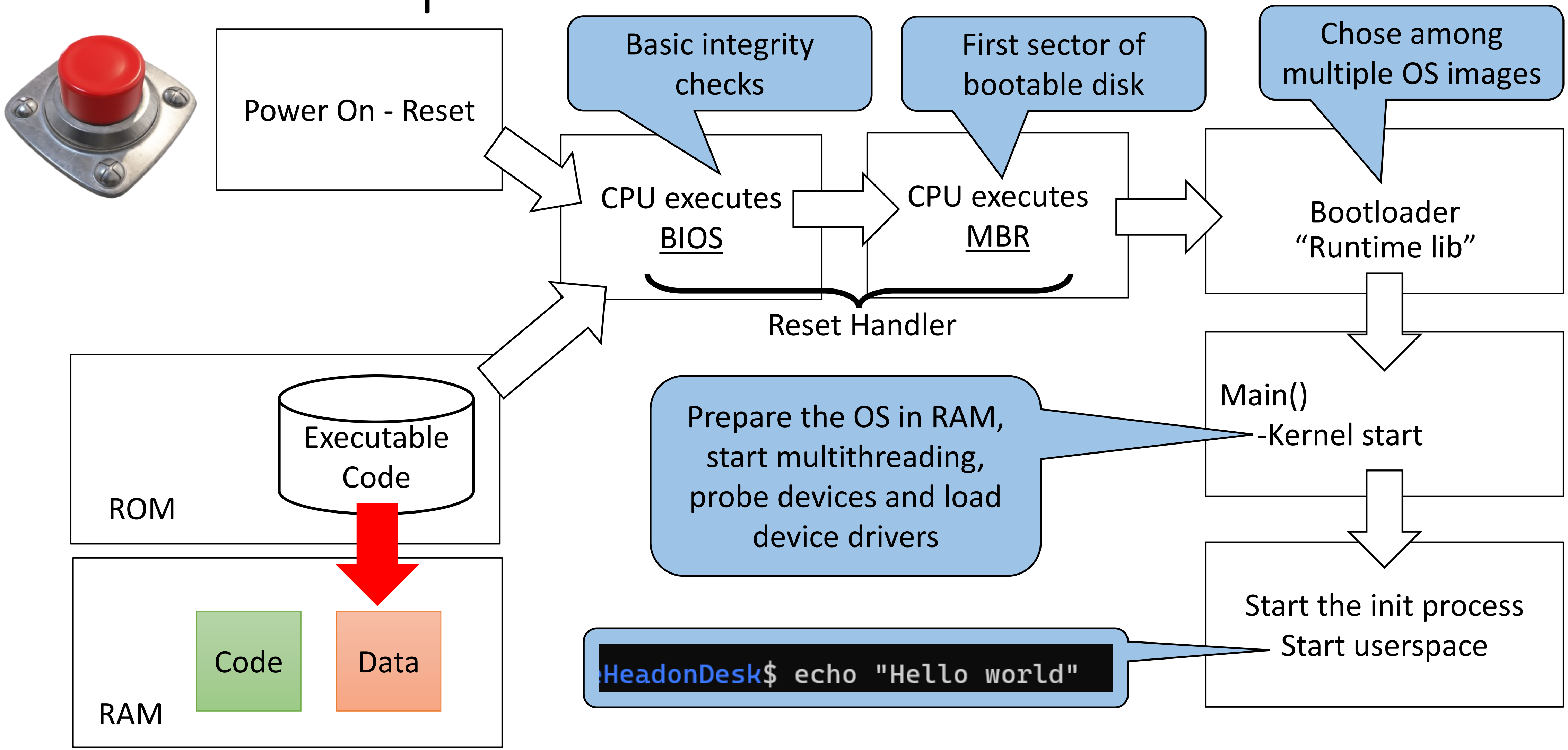
OS Bootstrap – The linux case

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





OS Bootstrap – The linux case



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 of 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

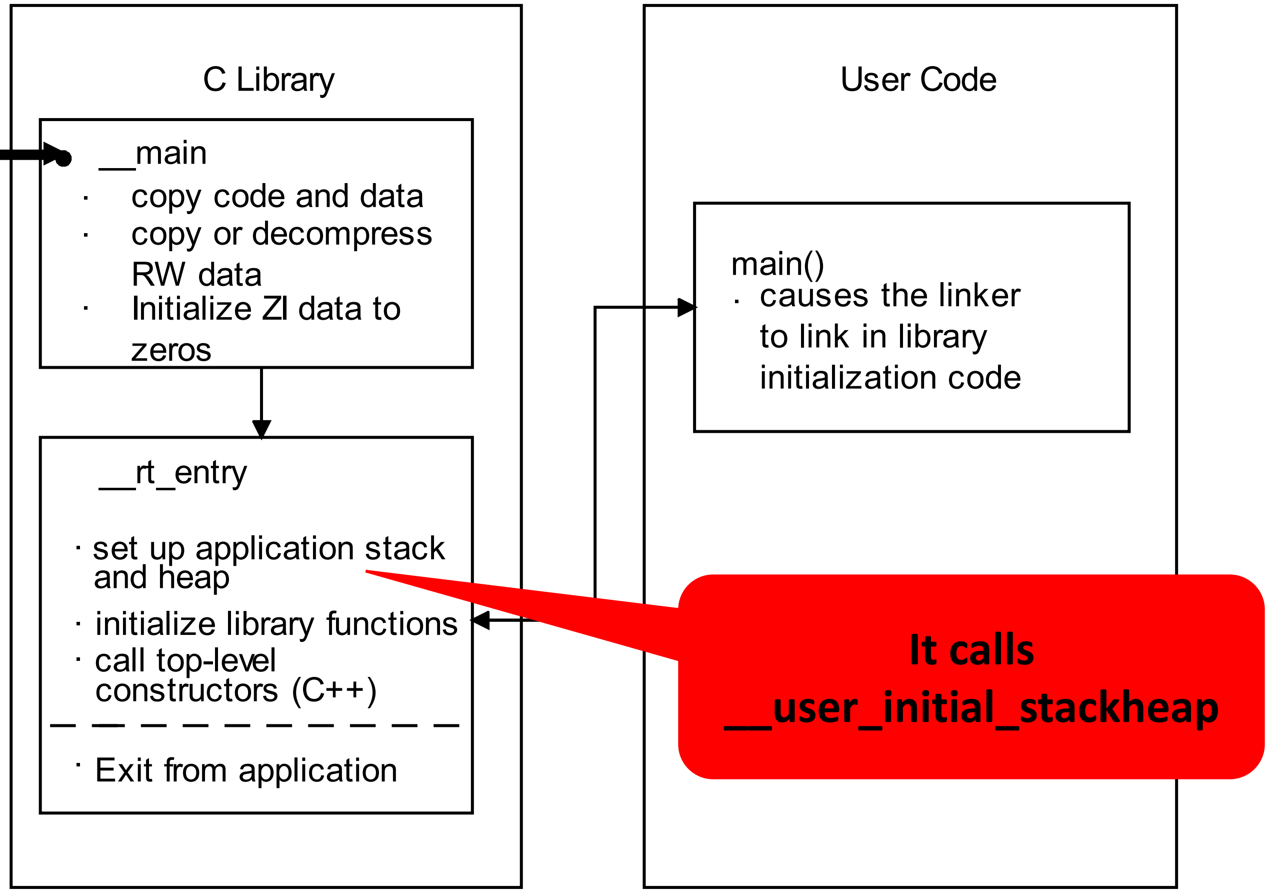
- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
  - 0 – halt, 1 – Single user mode, 2 – Multiuser, without NFS, 3 – Full multiuser mode, 4 – unused, 5 – X11, 6 – reboot.
- Init identifies the default initlevel from /etc/inittab and uses that to 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.

Outline

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

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.

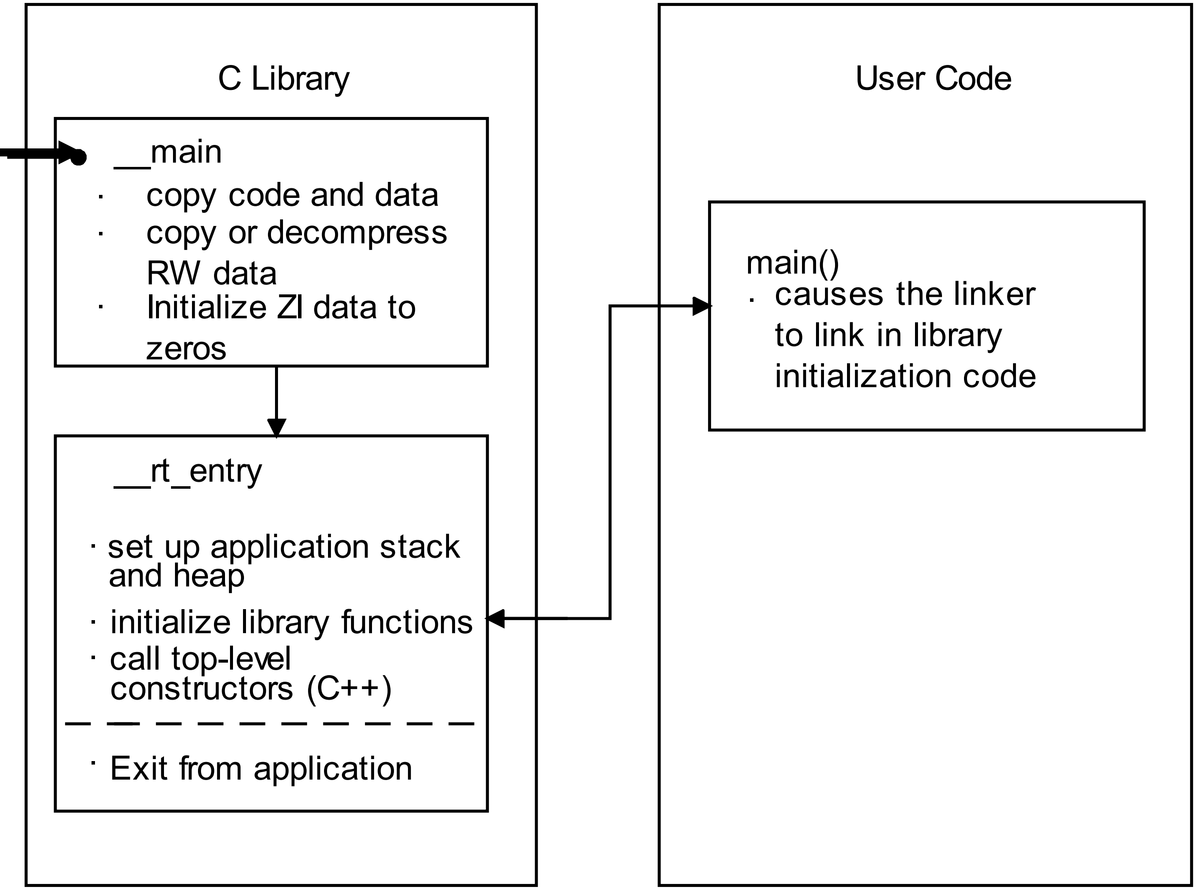
Setting up stack(s)

ASM_funct.s | startup_LPC17xx.s | main.c

```
                EXPORT  __heap_base
                EXPORT  __heap_limit

                ELSE

                IMPORT  __use_two_region_memory
                EXPORT  __user_initial_stackheap
__user_initial_stackheap

                LDR     R0, =  Heap_Mem ; low-address end of initial heap
                LDR     R1, =(Stack_Mem + Stack_Size/2) ; high-address end of initial stack
                LDR     R2, = (Heap_Mem +  Heap_Size) ; high-address end of initial heap
                LDR     R3, = Stack_Mem ; stack limit unused

                BX      LR

                ALIGN

                ENDIF


                END
```

Example – Skipping the "Magic secret sauce"

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



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
```

```c

const int pippo[]= {21312,44321};
int this_is_zero;
short int this_is_not_zero=0xcafe;
volatile int my_array[N] = {1,2,3,4,5,6,7,8,9,10};



int main(void){
  int i=0;
  volatile int value = 0xcafe;
  volatile int my_var=pippo[0];

  for ( i=0;i<N;i++){
  value=my_array[i];
  }

  while(1);
}

```

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

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?

```
Reset_Handler   PROC
                EXPORT  Reset_Handler
                import  __main
                ; your code here

                MOV     R0, #3
                MSR     CONTROL, R0
                LDR     SP, =Stack_Mem

                nop

                SVC     0x10     ;0x000000DA

```

```
int main (void){




    __asm volatile("svc 0x10");

    while(1);
}

```

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