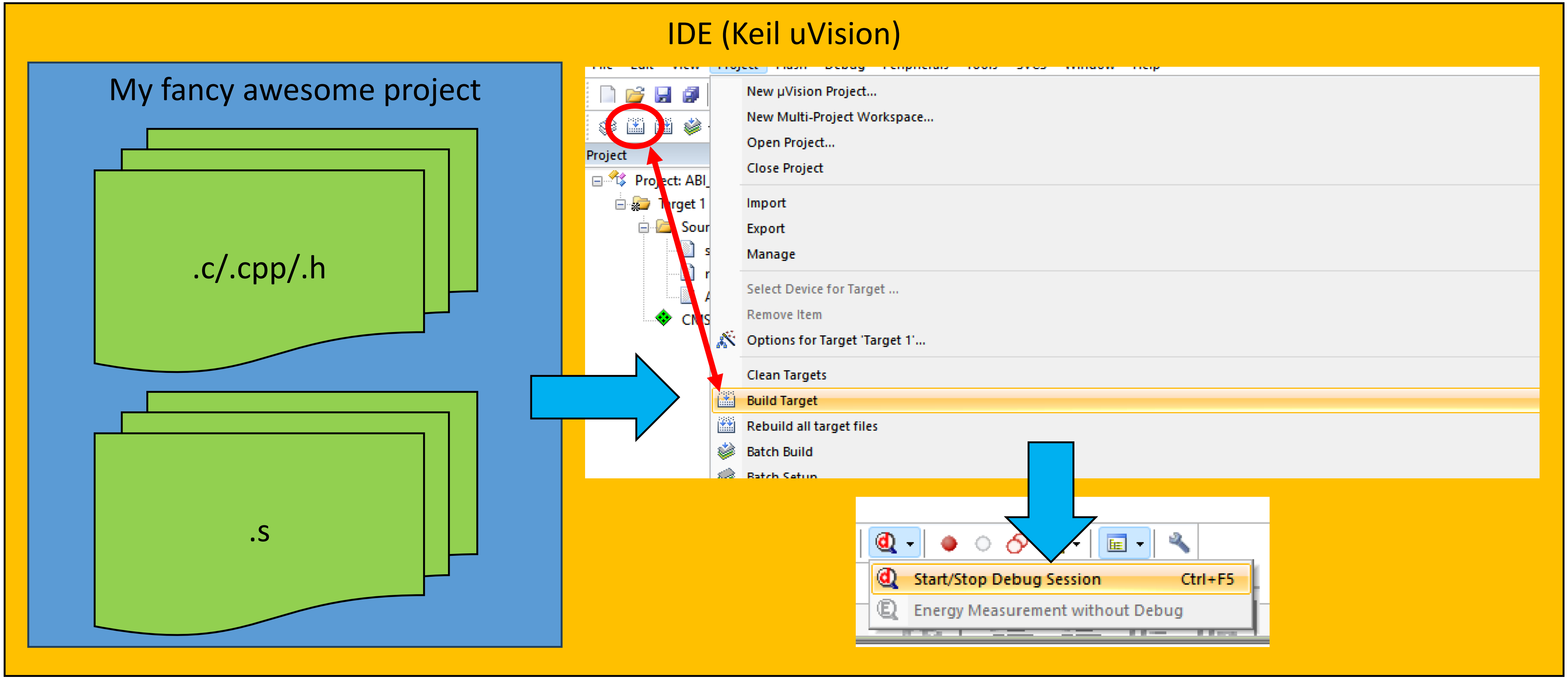
From Source code to Executable The Arm Toolchain for Embedded Systems

Francesco Angione, Paolo Bernardi

How is an executable produced from the source code?



Outline

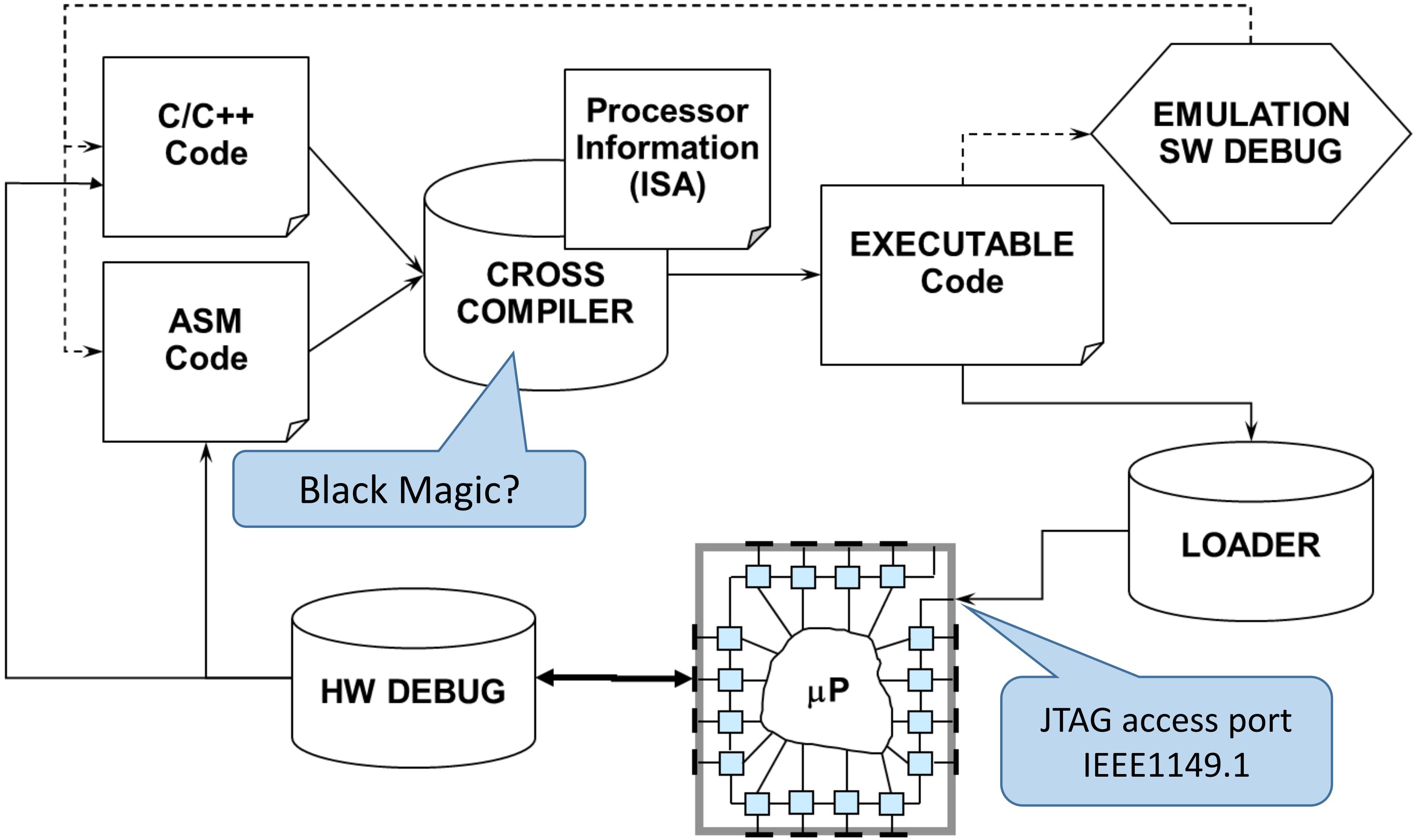
- What is a toolchain?
    - The Arm toolchain.
- Investigating the compilation output files.
- How does a System-on-Chip start the program?
- The Arm “Magic secret sauce”.
- HowTo and HowNotTo - Examples.

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”.
- HowTo and HowNotTo - Examples.

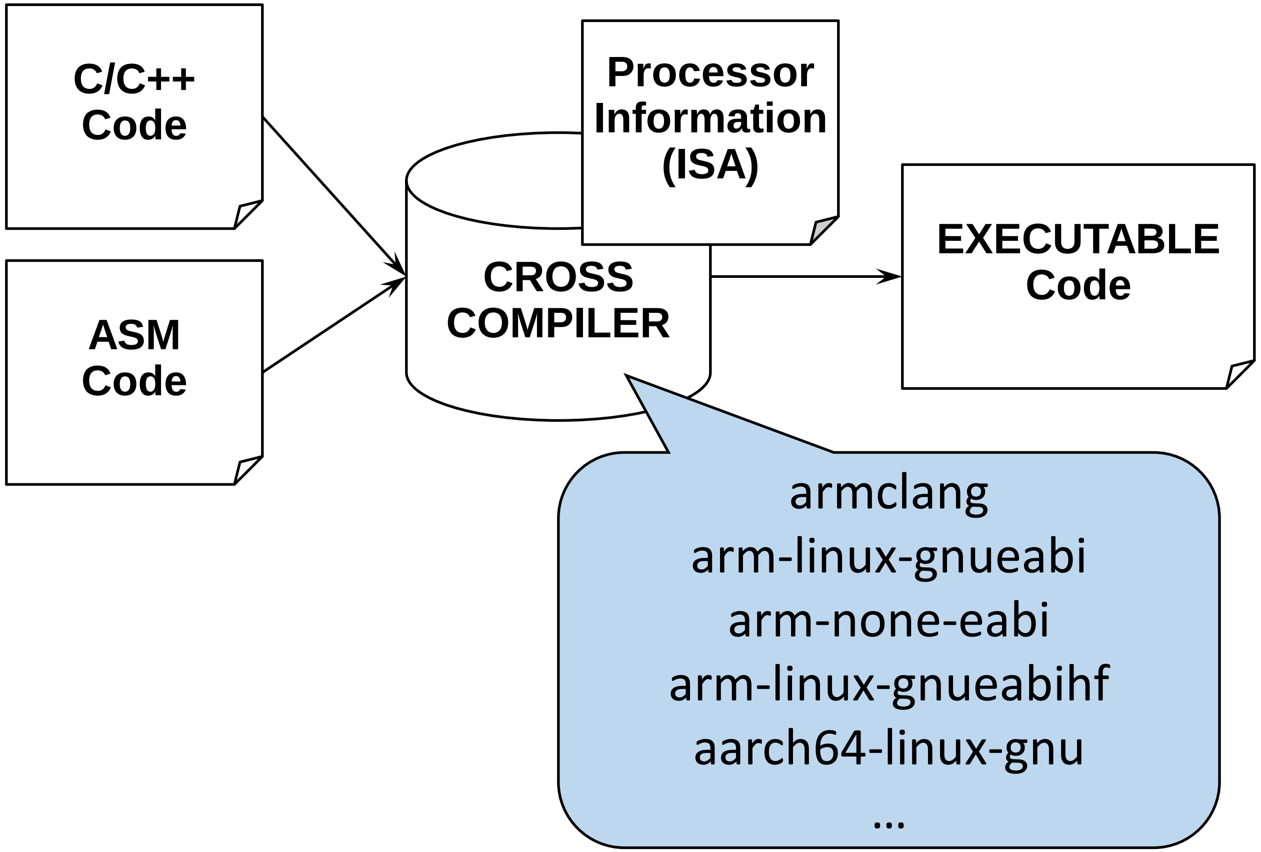
What is a toolchain?

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



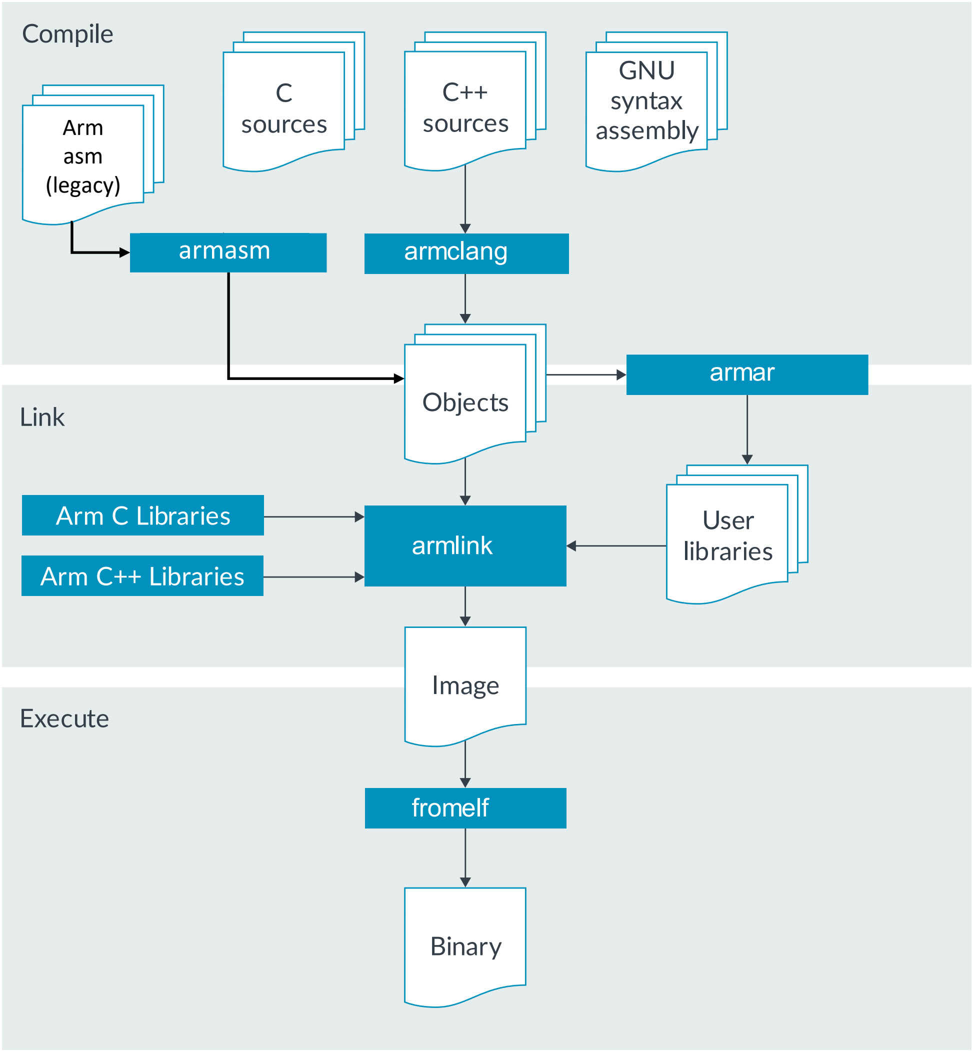
What does the cross-compiler?

- 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

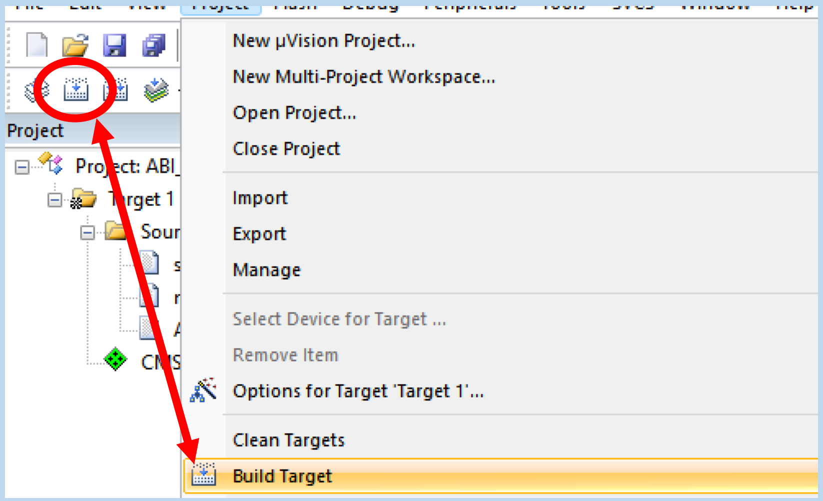
- Based on an enhanced version of *clang* (frontend) and *llvm project*, with proprietary customizations.
- The toolchain is composed of 3 different phases:
    - Preprocessing and compilation phase (armasm for legacy support).
    - Link phase.
    - Execute phase.

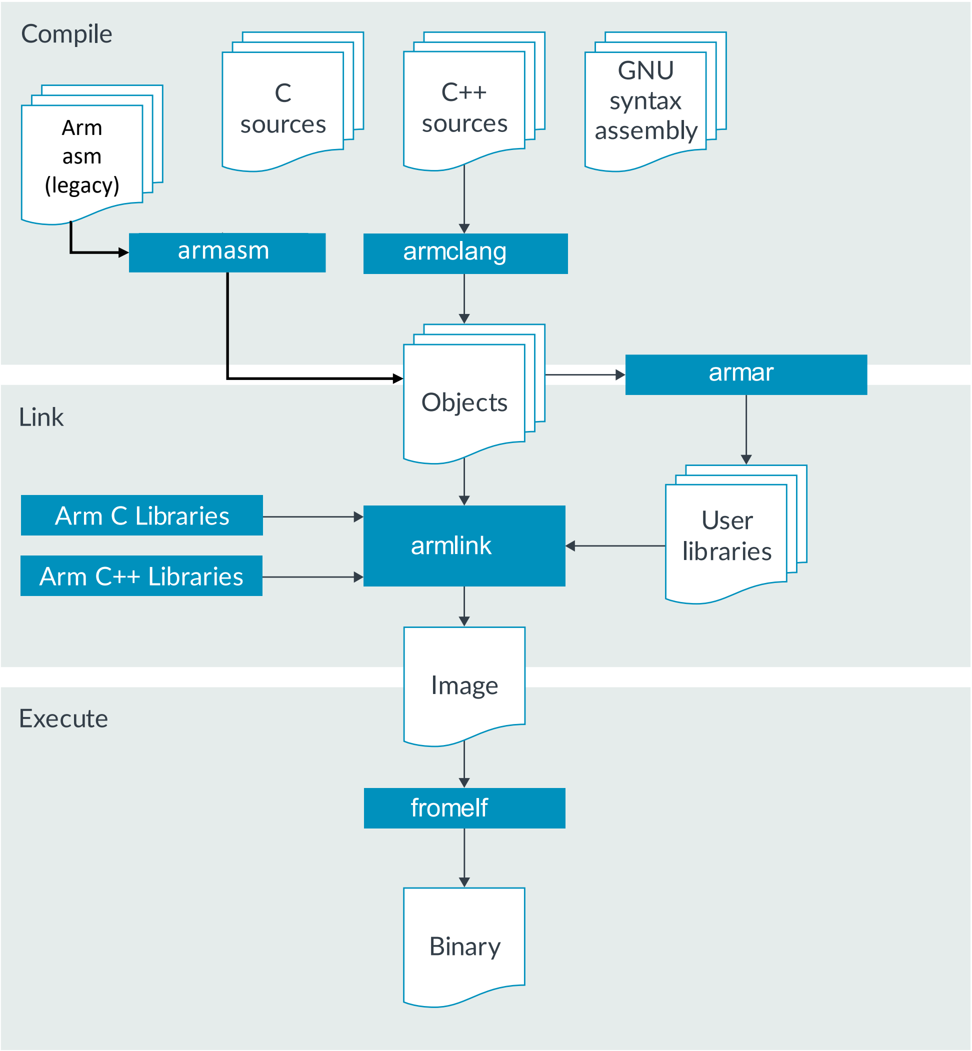


The Arm Toolchain

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

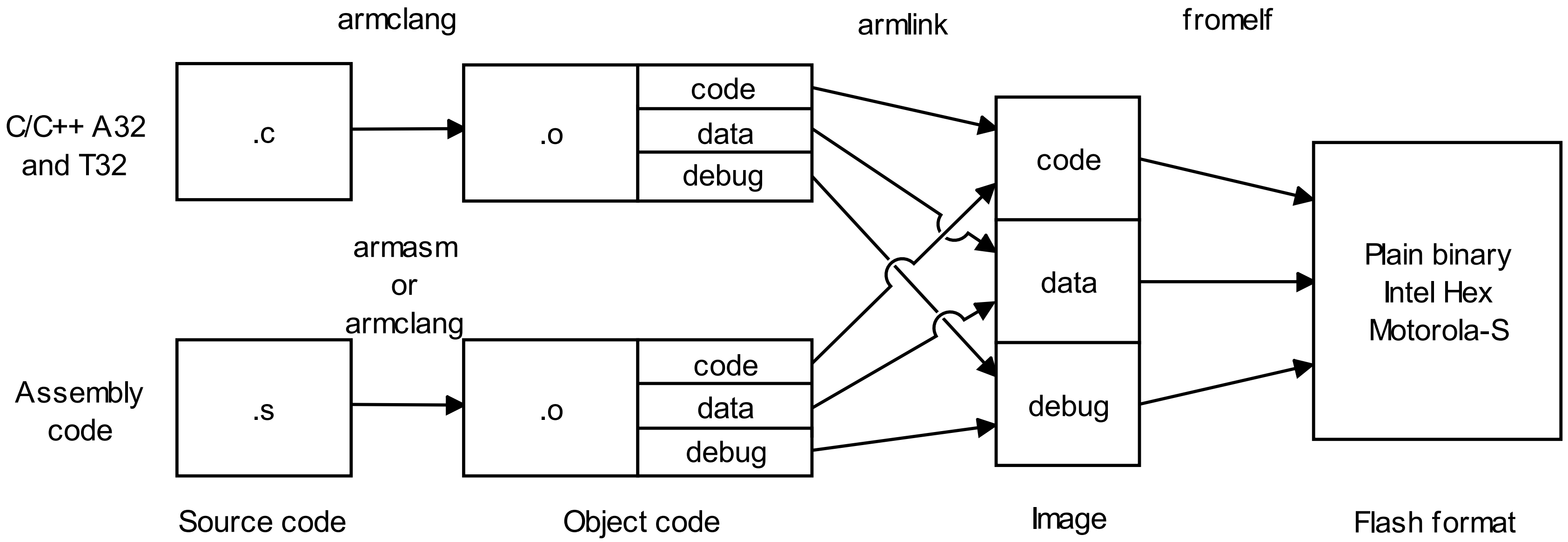
All the three phases included in:



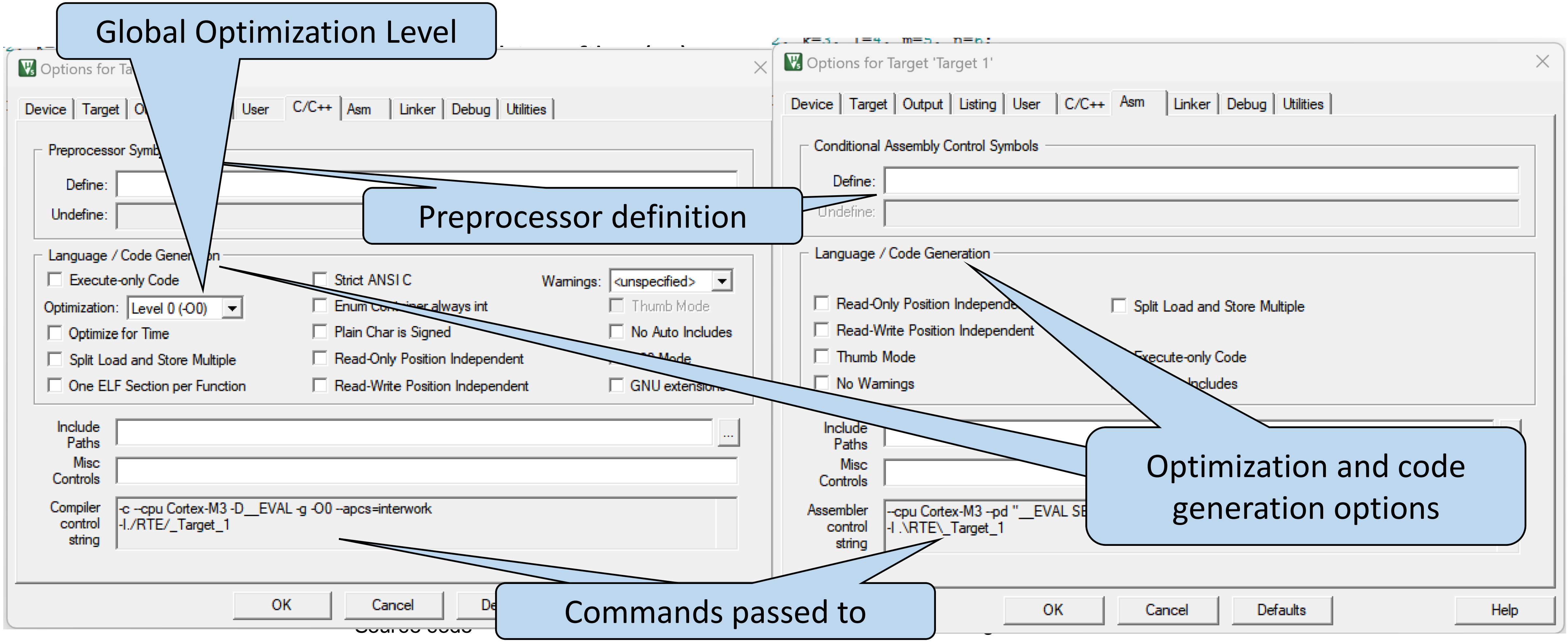


The compile phase – armclang and armasm

- 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.

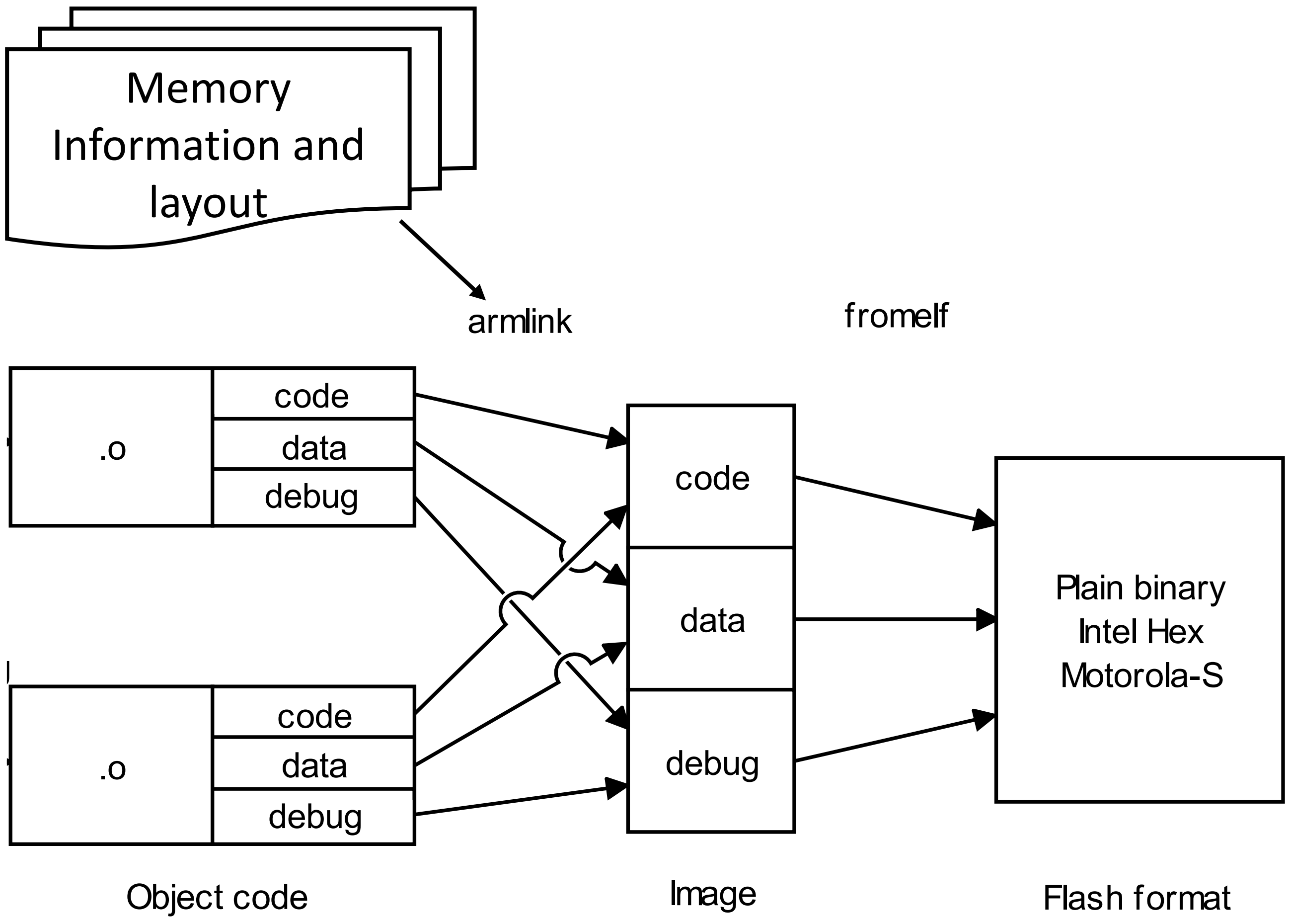


The compile phase – armclang and armasm



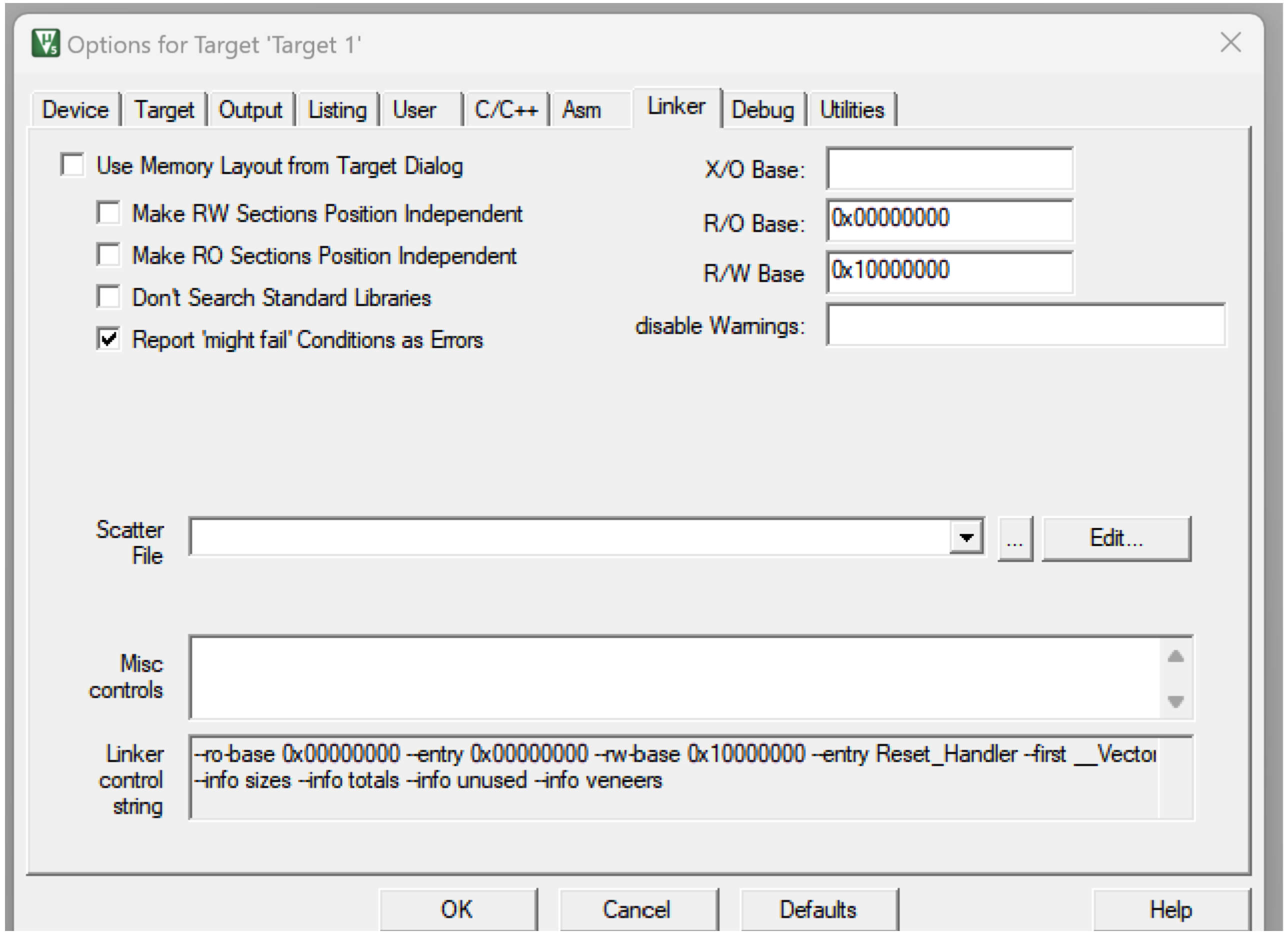
The link phase – armlink

- **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.



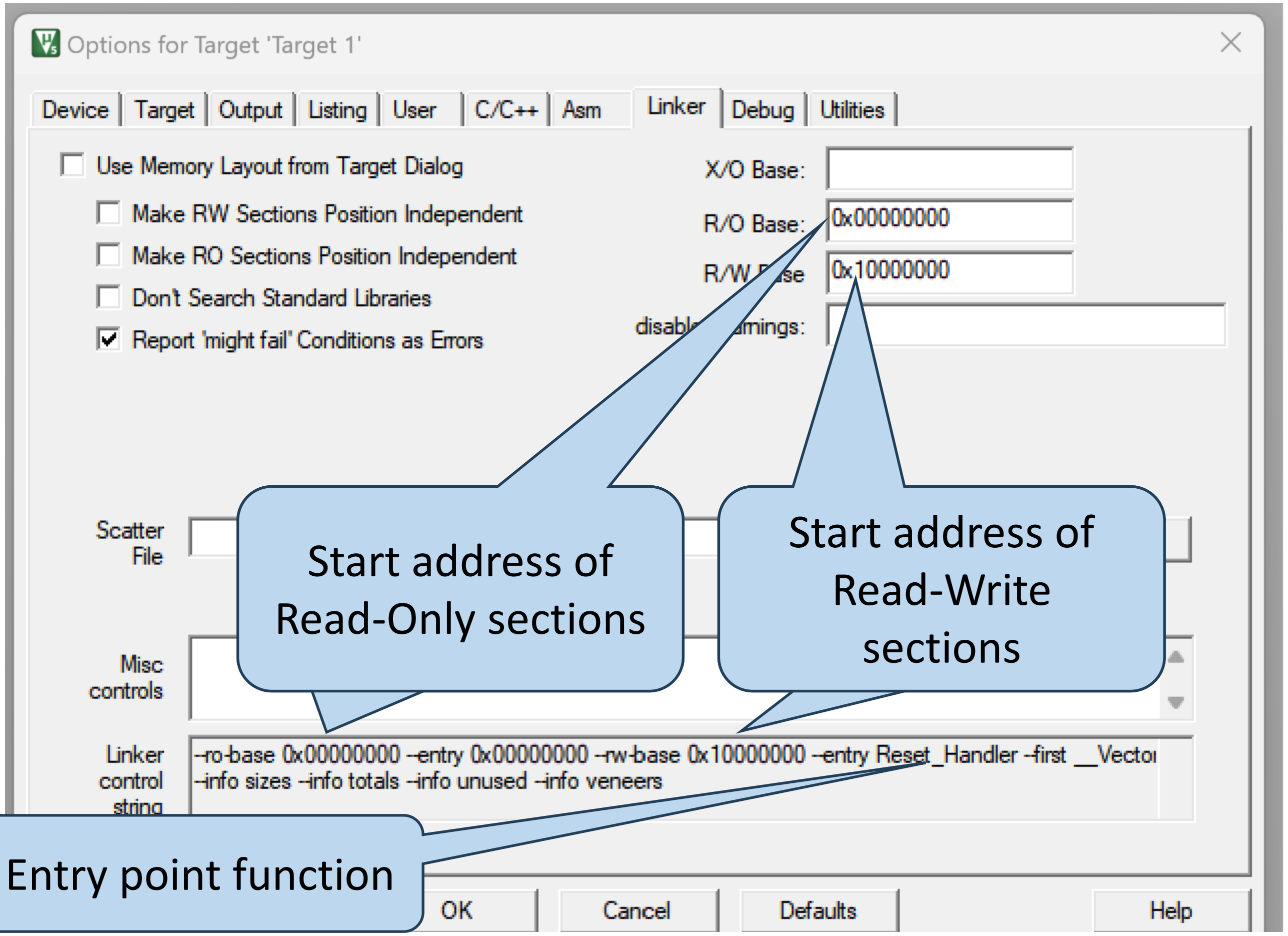
Memory Information and layout

- You can sp... nt at the startu... called at the syst...
- ... the m... ory
  - ...y co... ...d 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)
}
```



The execute phase - fromelf

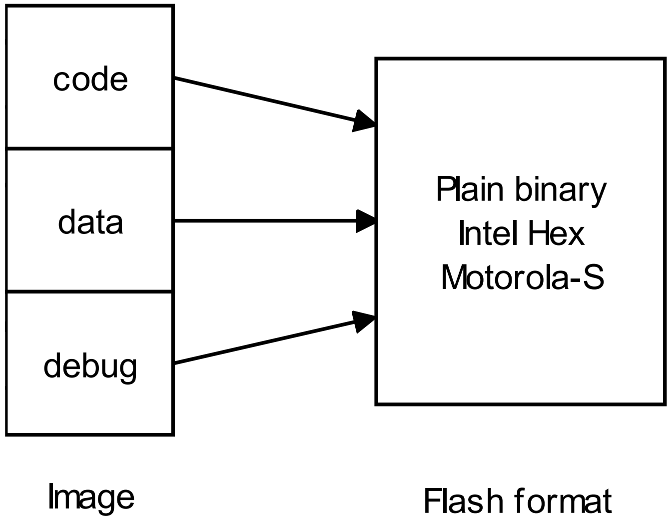
- Process object and image files.
- Convert ELF images into other formats for use by ROM tools or for direct loading into memory. The formats available are:
    - Plain binary.
    - Motorola 32-bit S-record.
    - Intel Hex-32.
    - Byte oriented hexadecimal.
- Display information about the input file,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

Binary

```
00000b60  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 a4 05 00 00 04 01  41 53 4d 5f 66 75 6e 63  |........ASM_func|
00000ba0  74 2e 73 00 43 6f 6d 70  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  |RM Compiler 5.06|
00000bc0  20 75 70 64 61 74 65 20  36 20 28 62 75 69 6c 64  | update 6 (build|
00000bd0  20 37 35 30 29 20 54 6f  6f 6c 3a 20 61 72 6d 61  | 750) Tool: arma|
```

Motorola

```
S315518073600000000000000000000000000000000046
S315518073700000000000000000000000000000000036
S315518073800000000000000000000000000000000026
S315518073900000000000000000000000000000000016
```

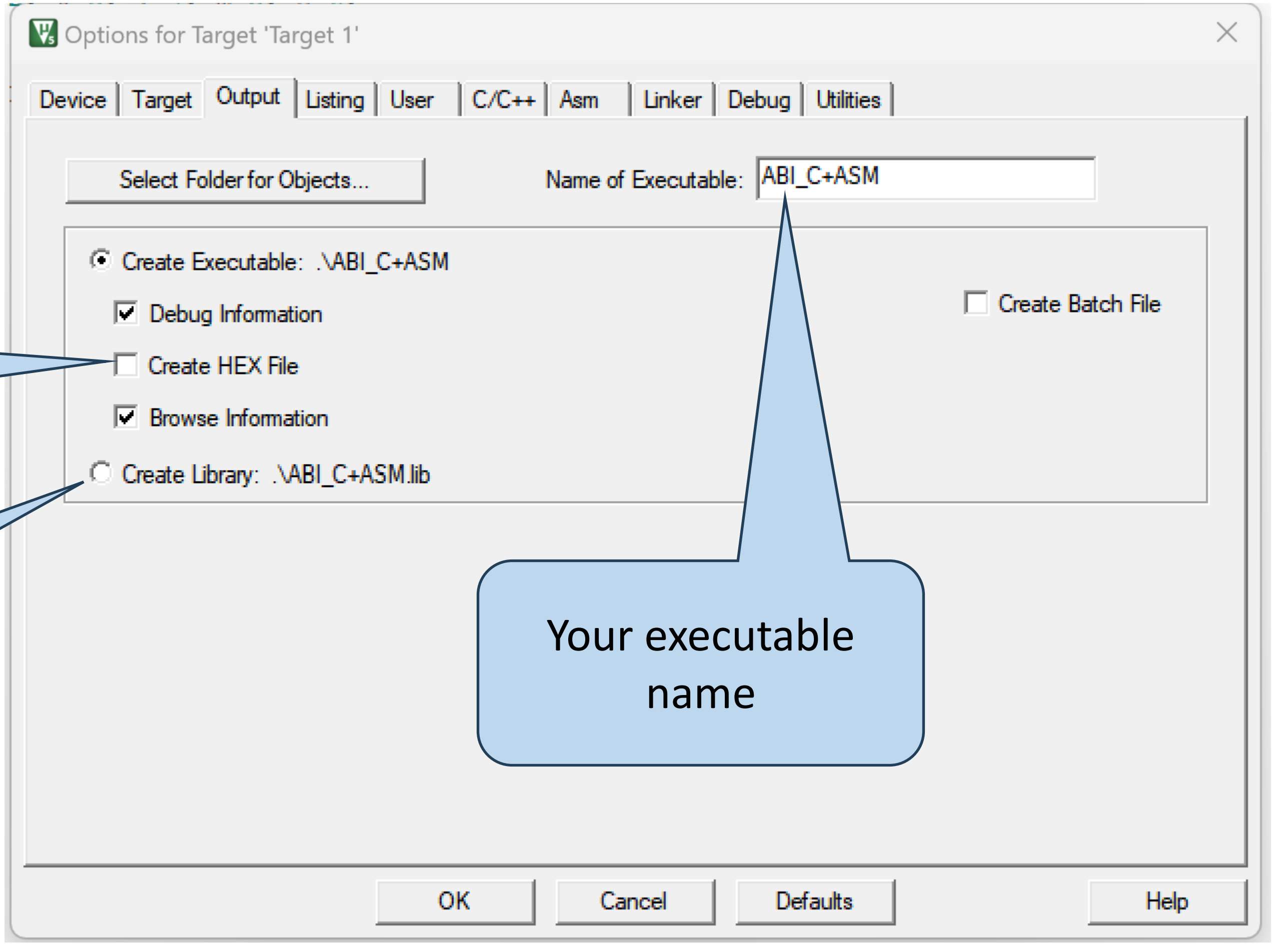


The execute phase - fromelf

- 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



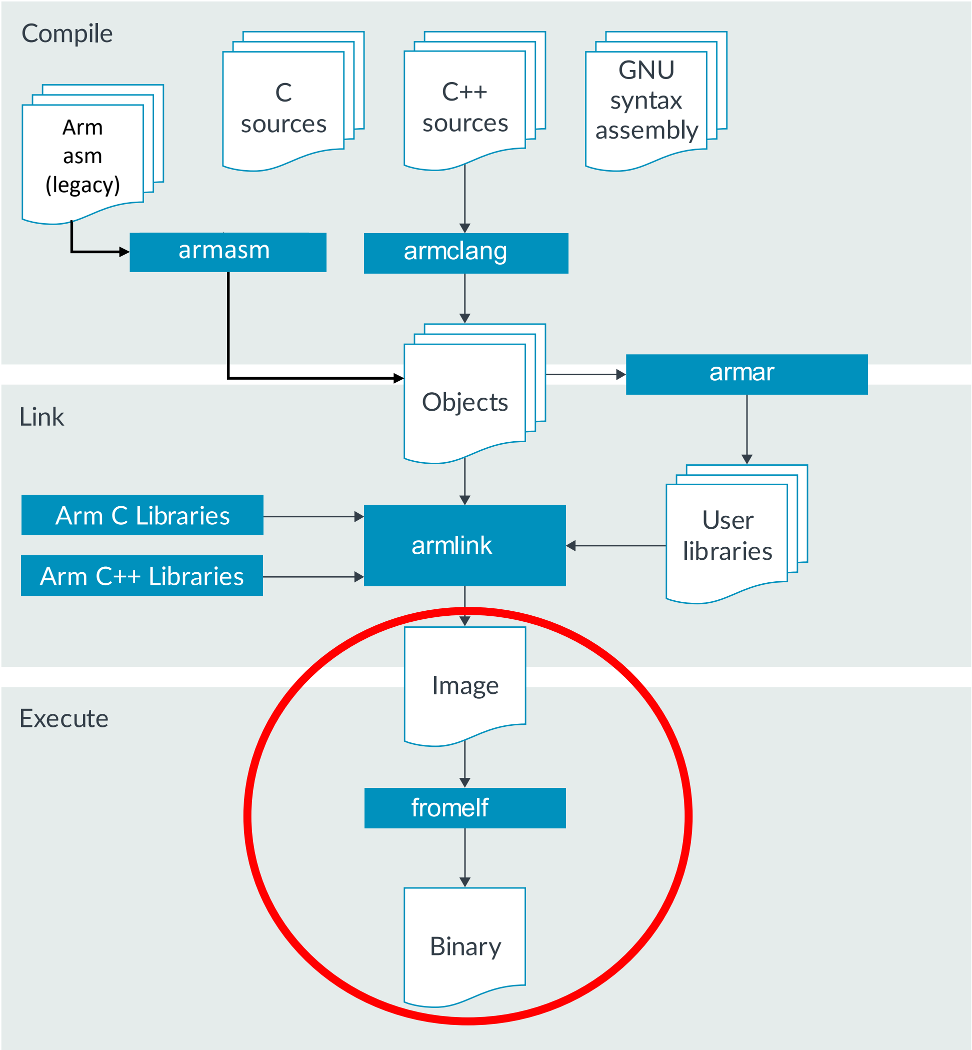
Your executable name

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”.
- HowTo and HowNotTo - Examples.

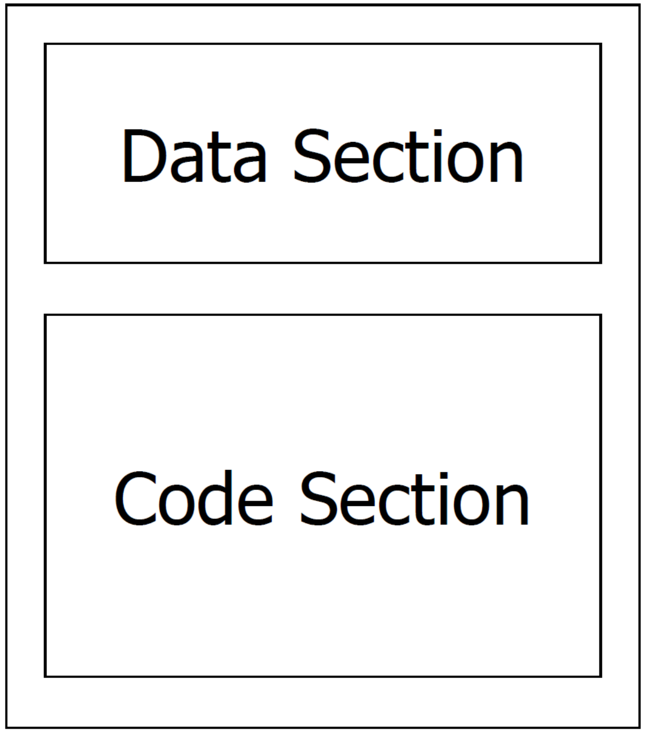
Investigating the compilation output files.

- 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.

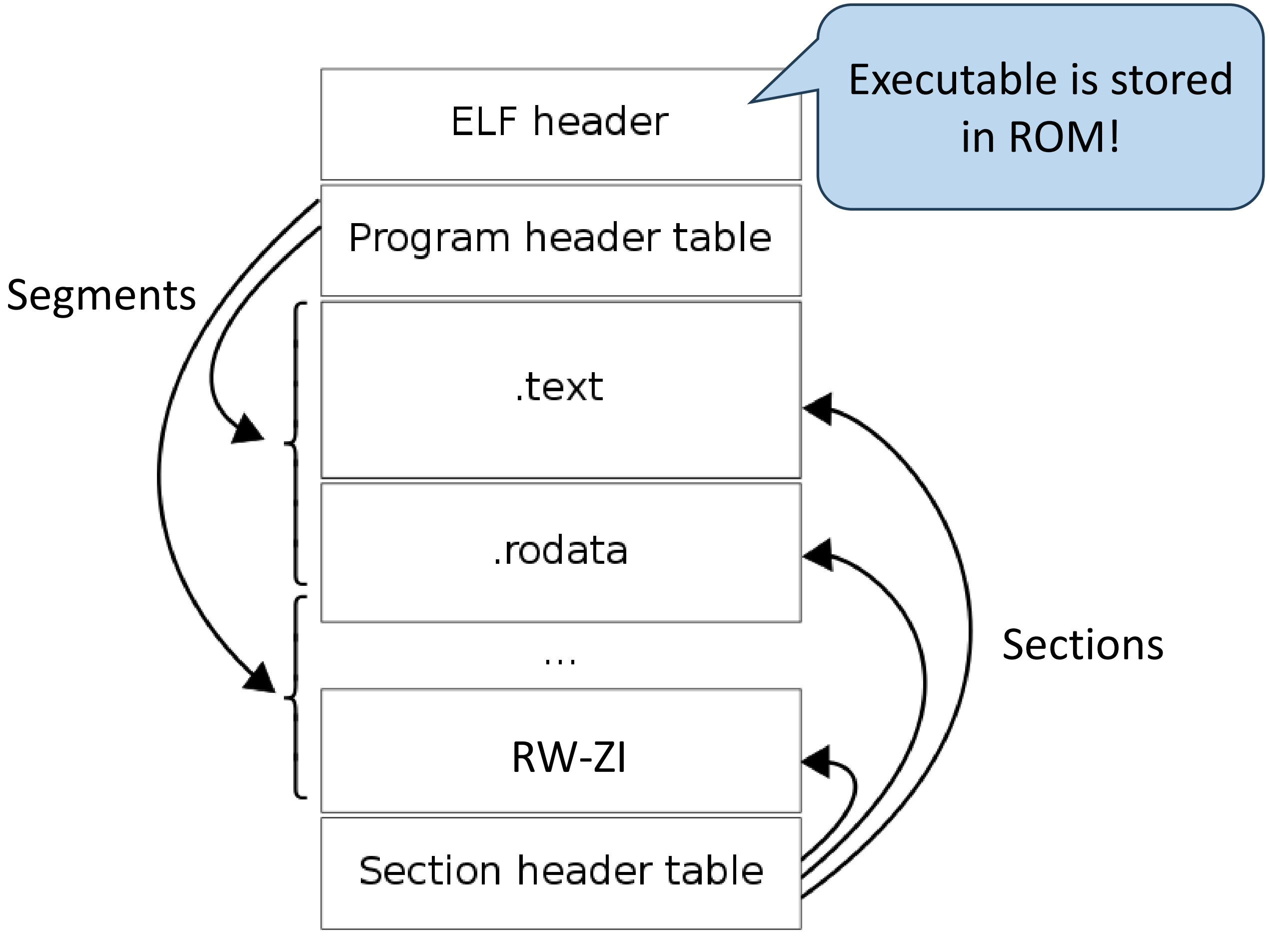


My fancy awesome Image program

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

The executable – Load view

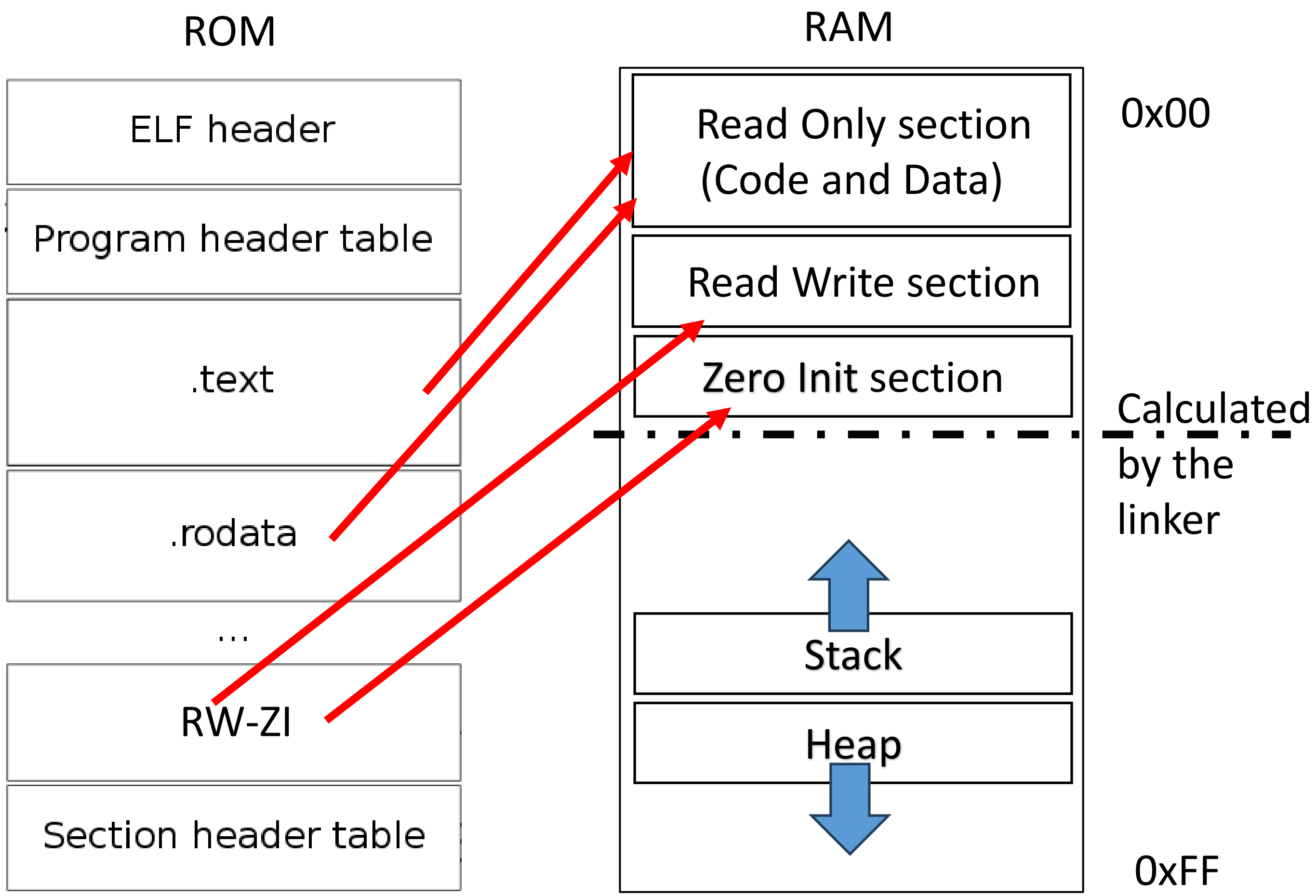
- The overall image (from the source code) is converted into an executable (.exe, .elf , .axf for Arm).
- Data and Code sections are in the executable.
- Composed of:
  - Entry address.
  - Stack and heap information.
  - Sections, used by the linker.
  - Segments, used by the loader (at runtime).



My fancy awesome Image program (.axf)

The executable – Execution view

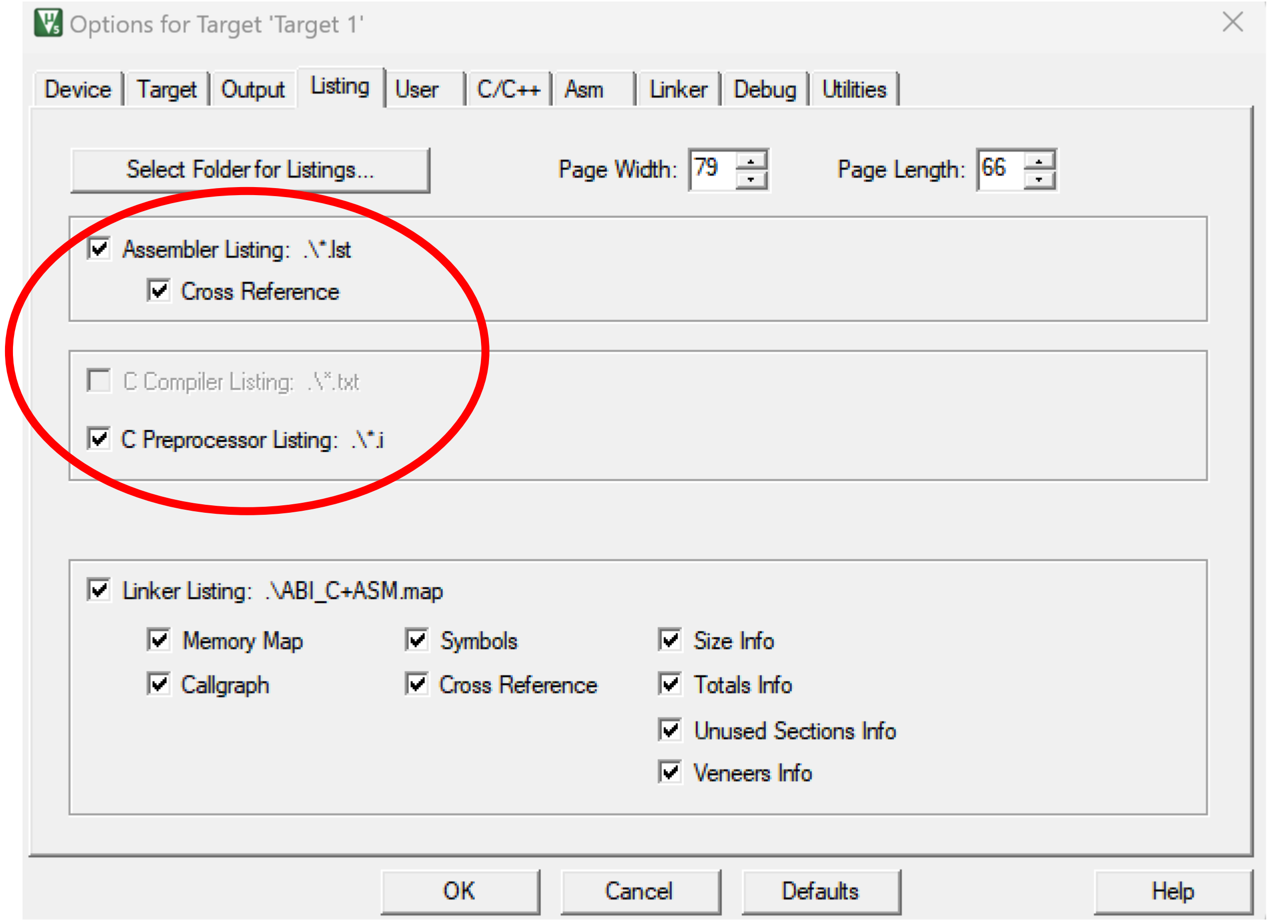
- Before the image is executed:
  - Move executable segments from ROM to their execution addresses in RAM.
  - RW data must be copied from its load address in the ROM to its execution address in the RAM.
- Runtime memory layout information is calculated offline:
  - Stack and heap execution address and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



My fancy awesome Image program (.axf)

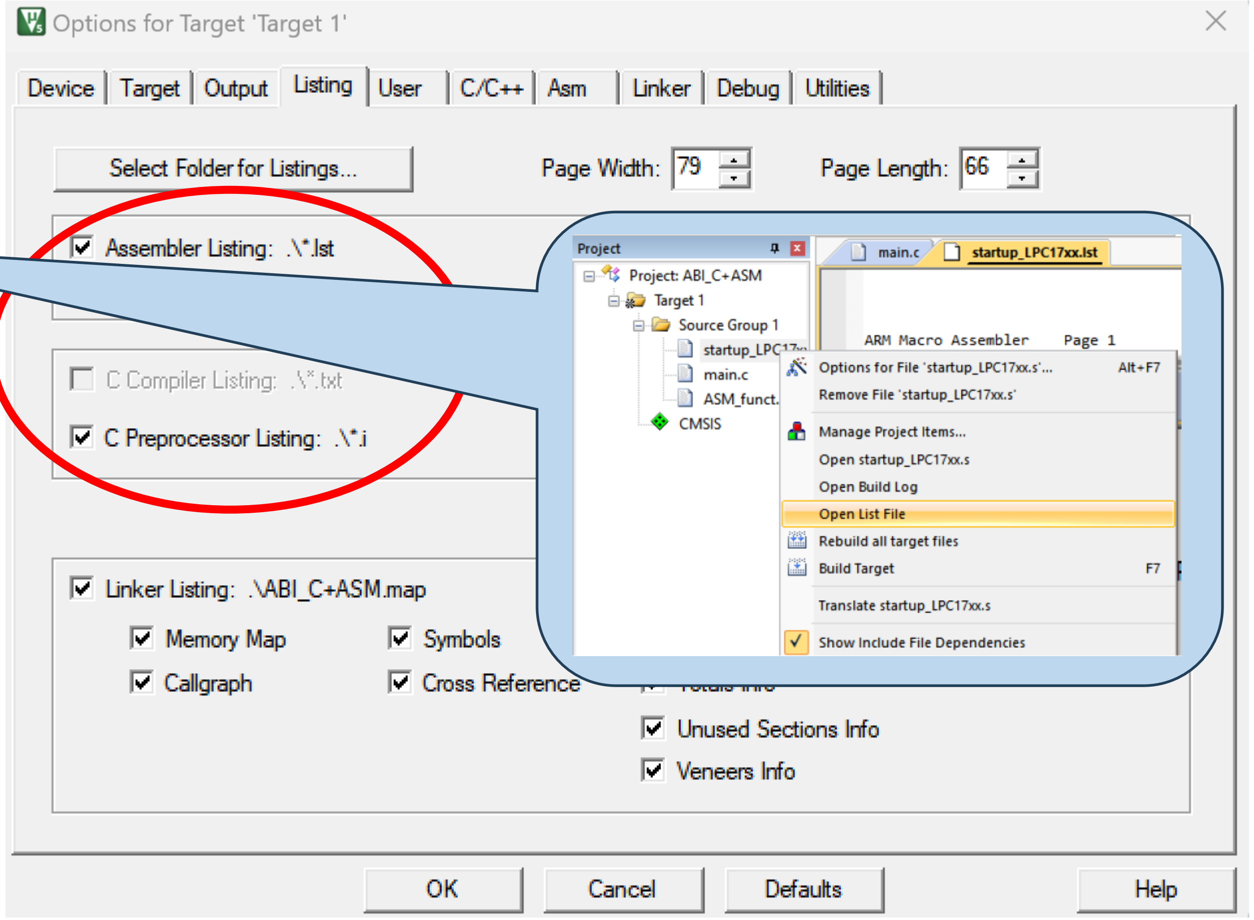
The Listing and dependencies files

- 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.



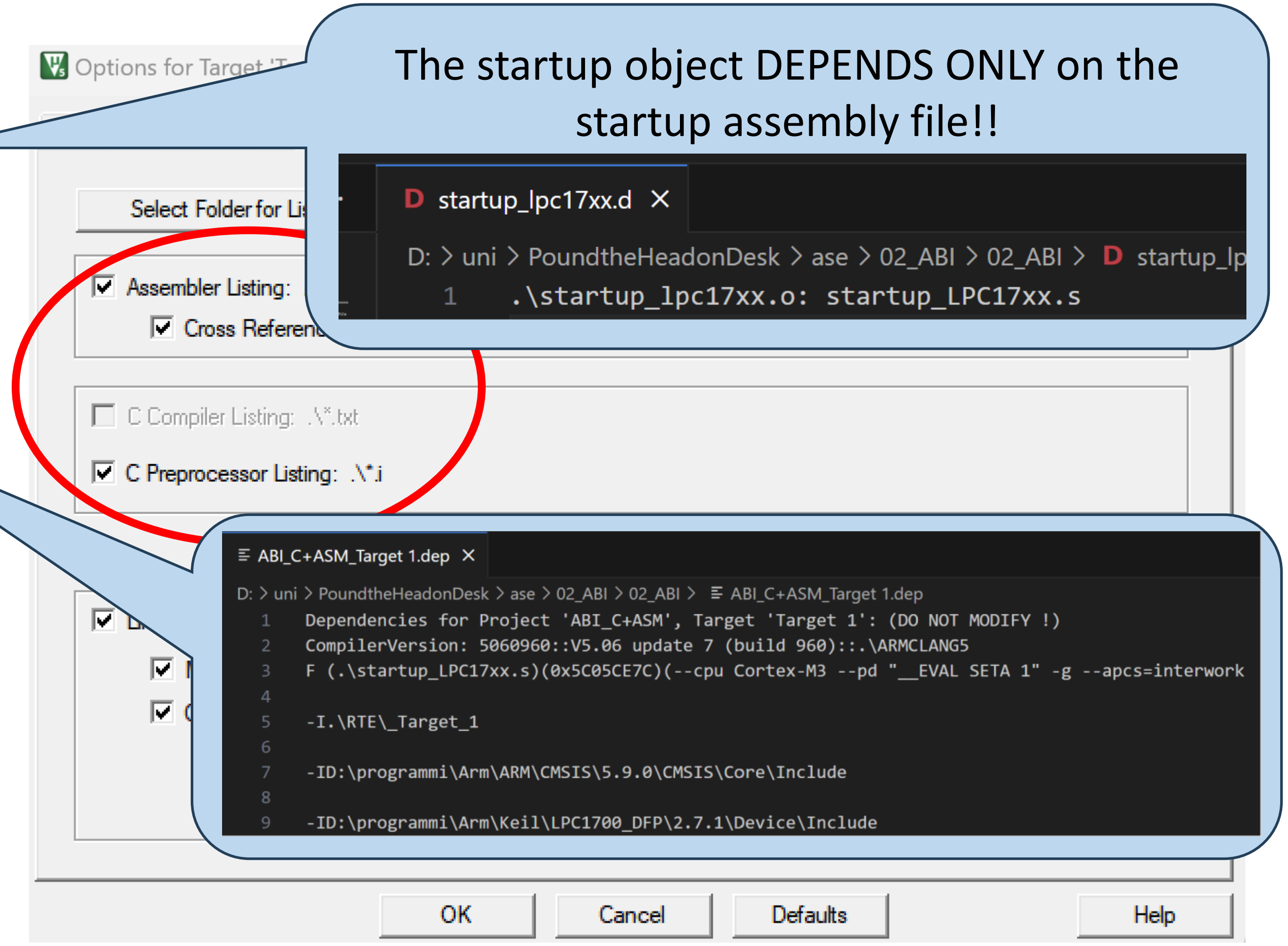
The Listing and dependencies files

- 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.



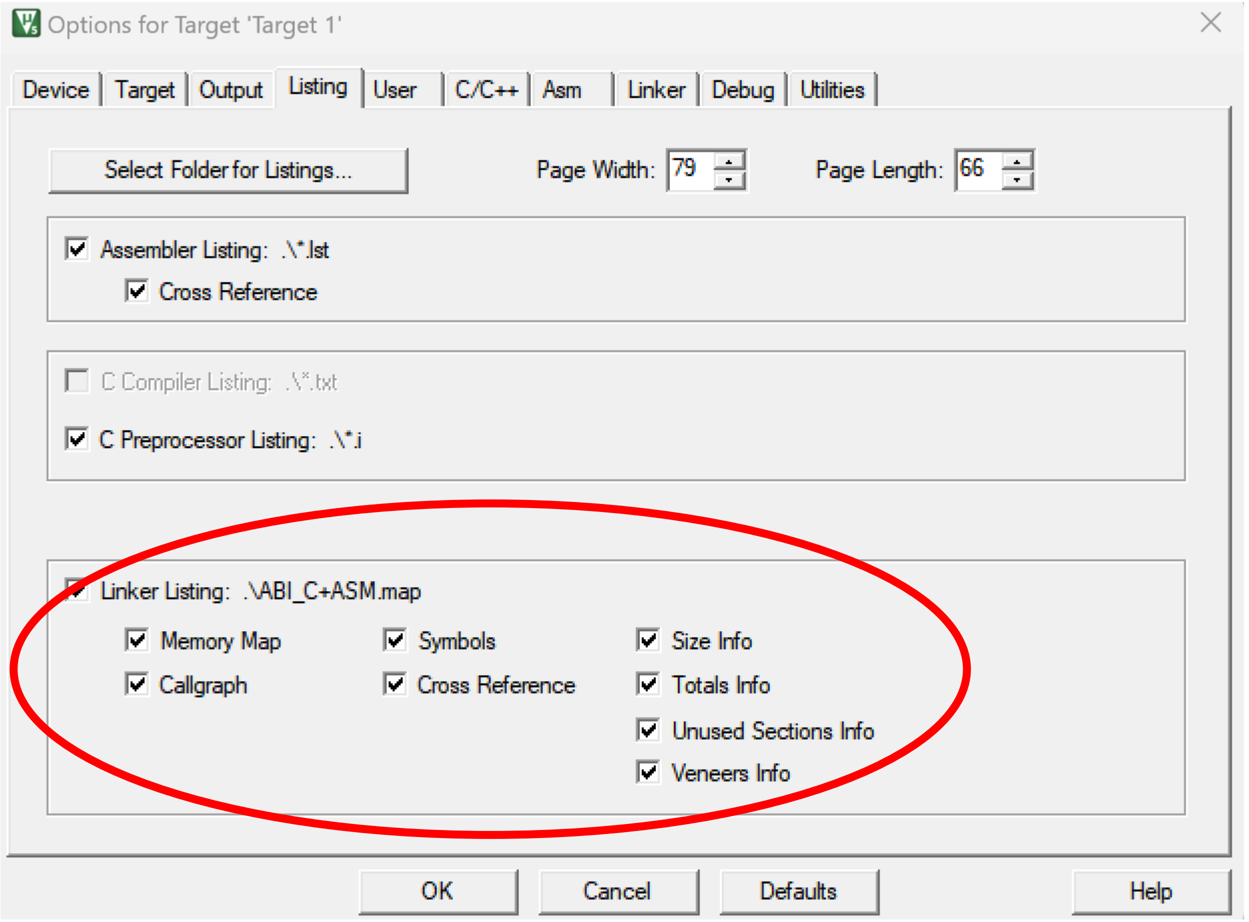
The Listing and dependencies files

- 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.



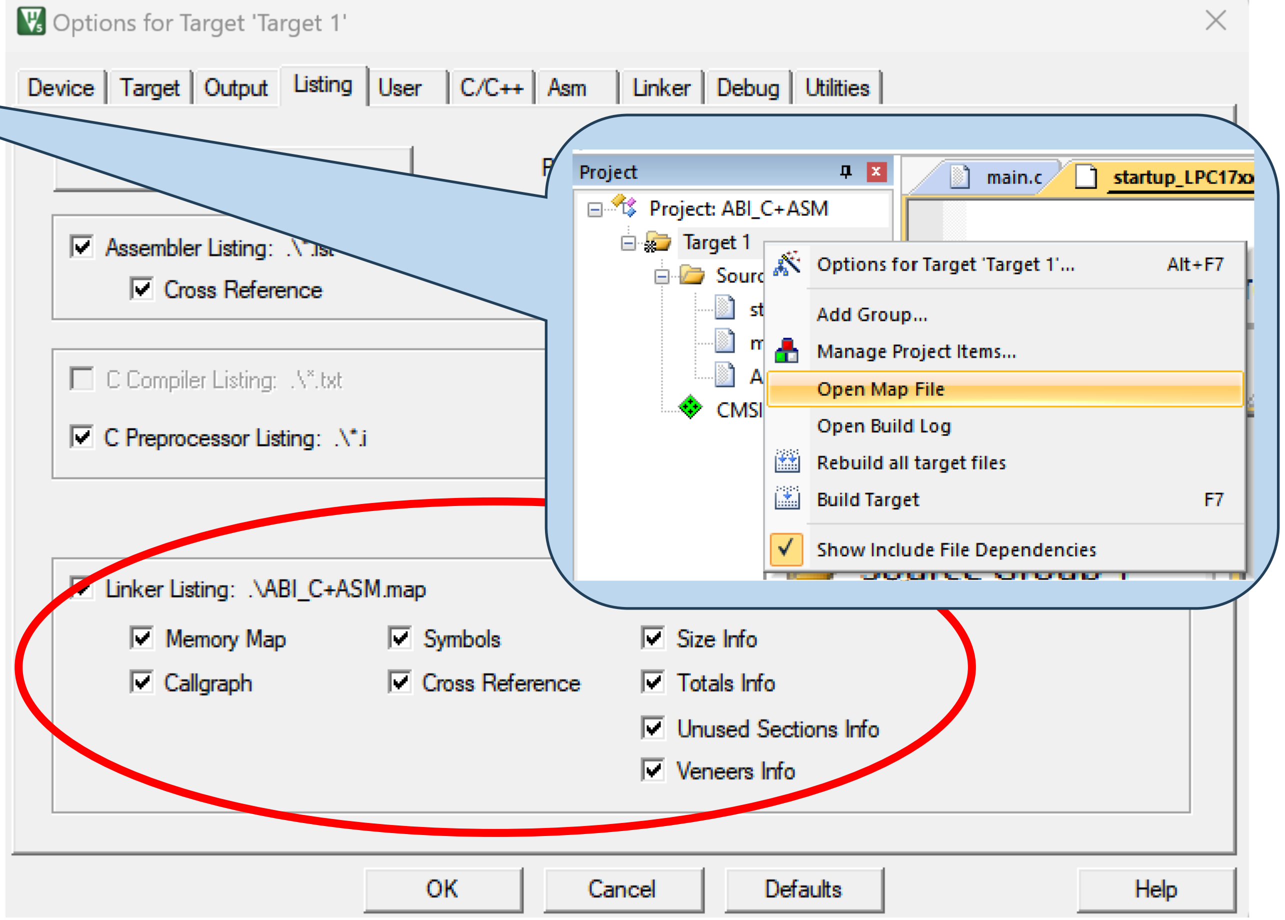
The map

- 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.



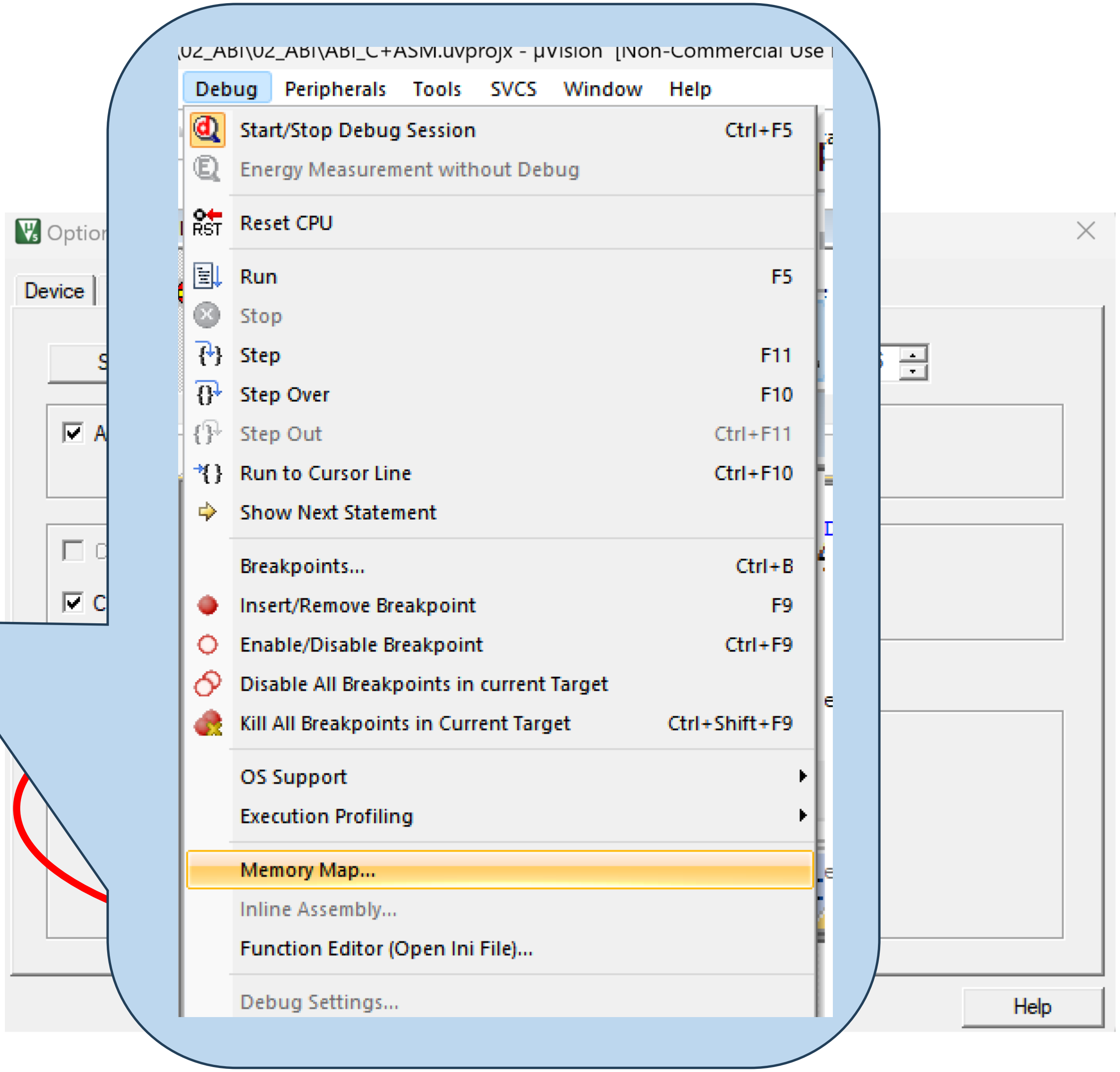
The map

- 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.



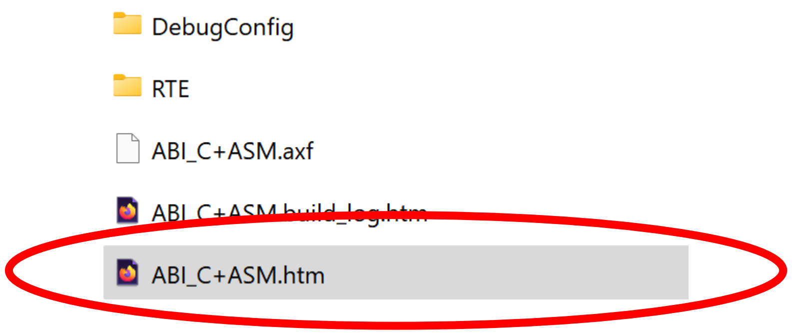
The map

- 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.



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.



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

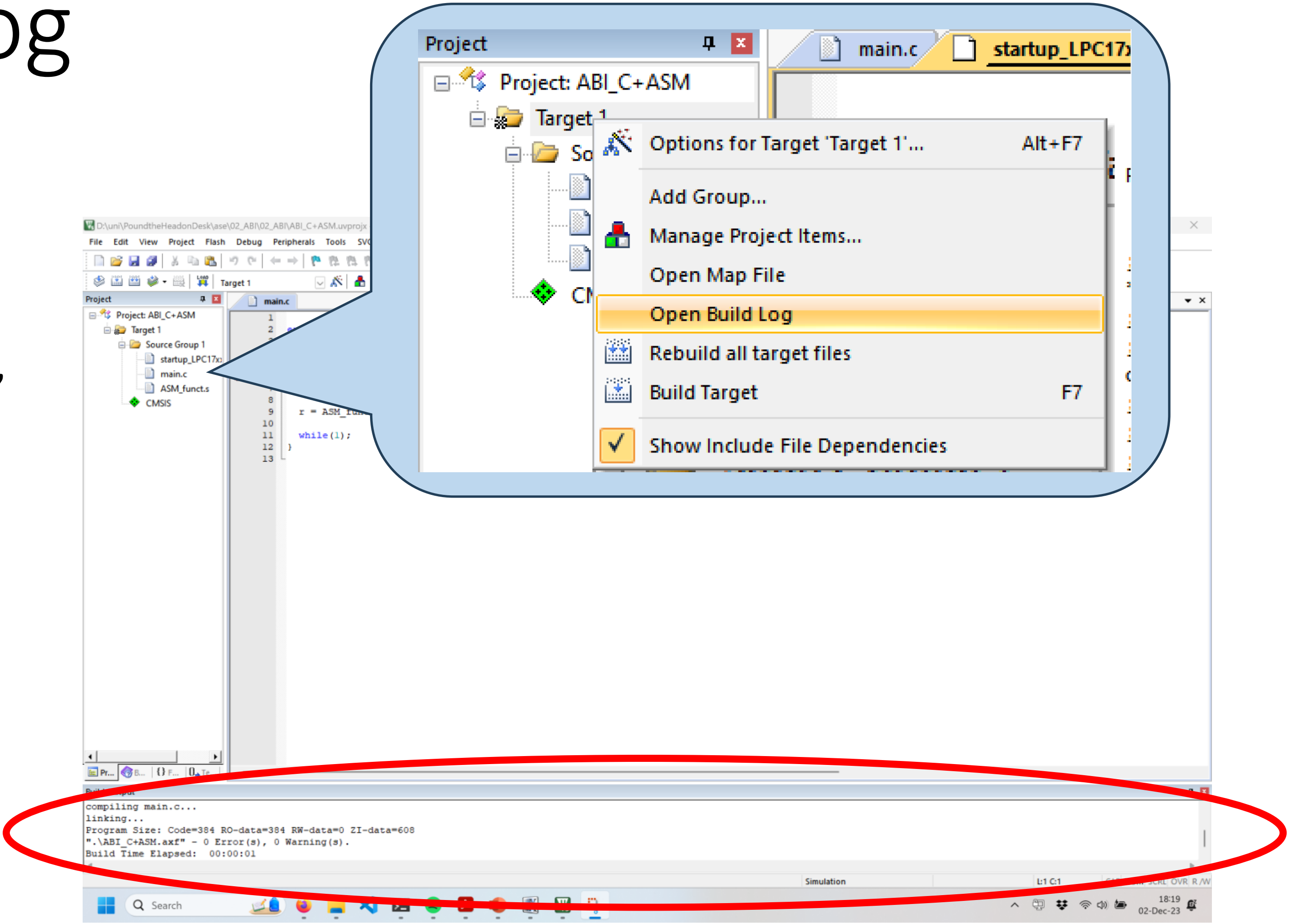
- NMI_Handler ⇒ NMI_Handler
- HardFault_Handler ⇒ HardFault_Handler
- MemManage_Handler ⇒ MemManage_Handler
- BusFault_Handler ⇒ BusFault_Handler
- UsageFault_Handler ⇒ UsageFault_Handler
- SVC_Handler ⇒ SVC_Handler
- DebugMon_Handler ⇒ DebugMon_Handler
- PendSV_Handler ⇒ PendSV_Handler
- SysTick_Handler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

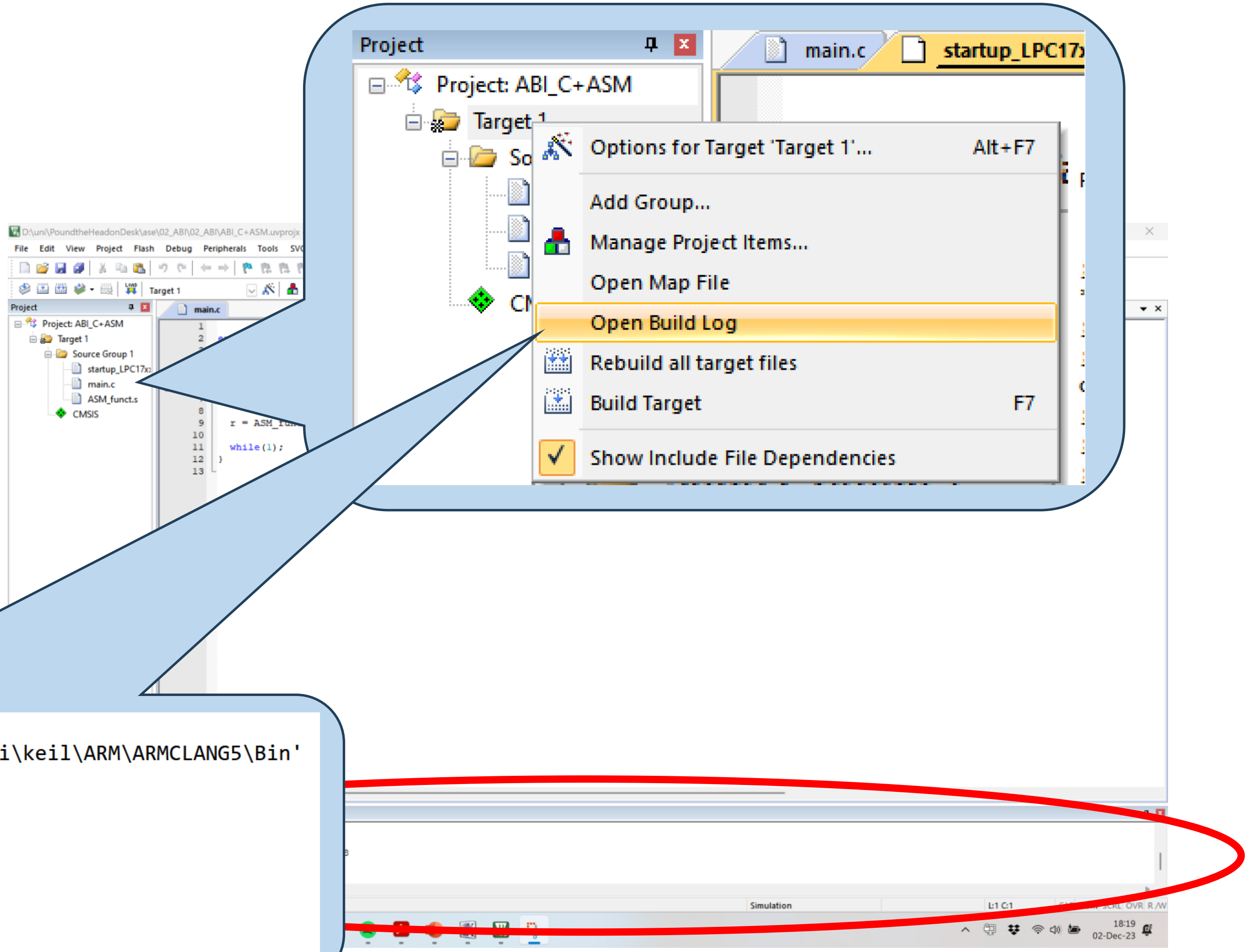
The build output log

- 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.

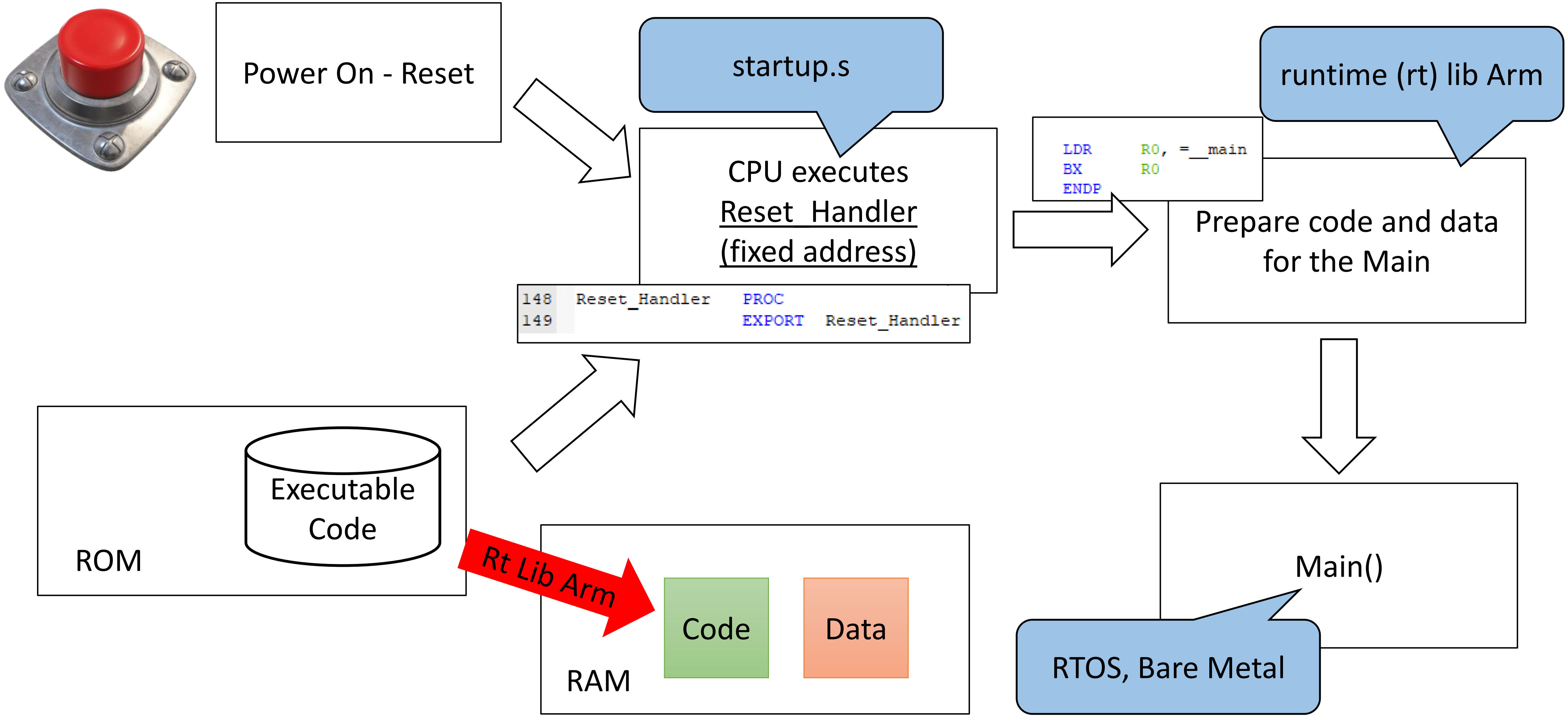



```
*** Using Compiler 'V5.06 update 7 (build 960)', folder: 'D:\programmi\keil\ARM\ARMCLANG5\Bin'
Rebuild target 'Target 1'
assembling ASM_funct.s...
compiling main.c...
assembling startup_LPC17xx.s...
linking...
Program Size: Code=384 RO-data=384 RW-data=0 ZI-data=608
".\ABI_C+ASM.axf" - 0 Error(s),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

Outline

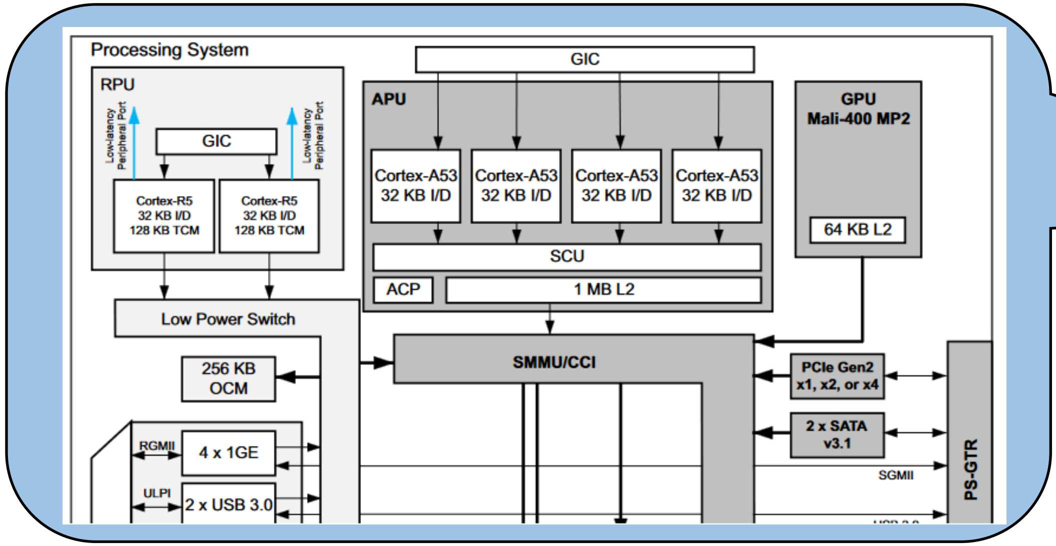
- What is a toolchain?
  - The Arm toolchain.
- Investigating the compilation output files.
- <u>How does a System-on-Chip start the program?</u>
- The Arm “Magic secret sauce”.
- HowTo and HowNotTo - Examples.

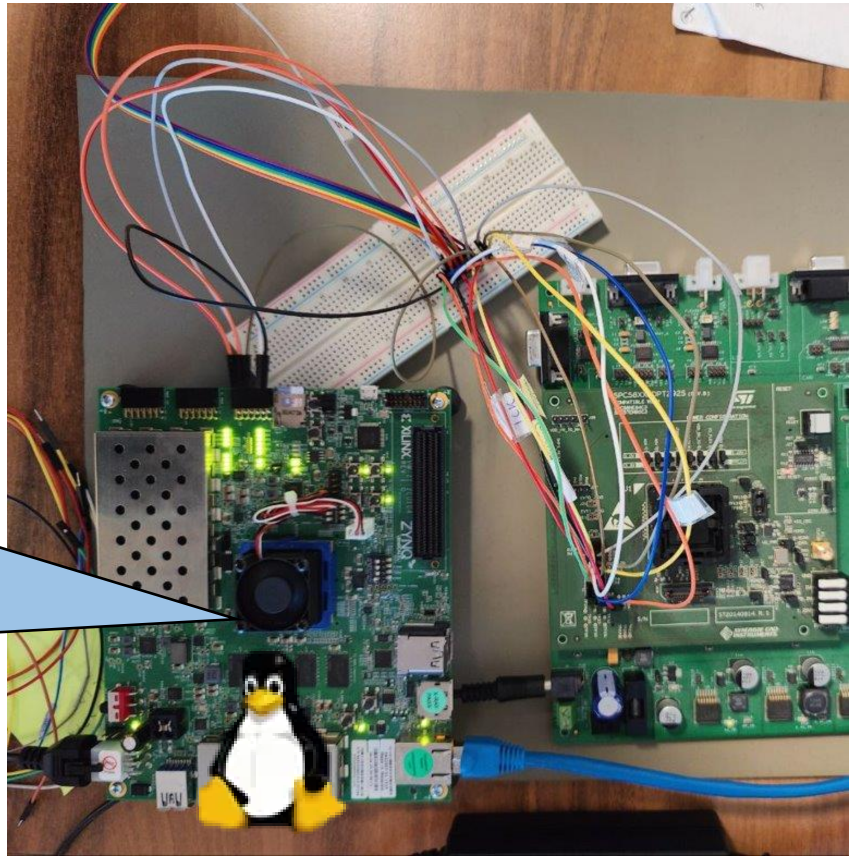
How does a System-on-Chip start the program?



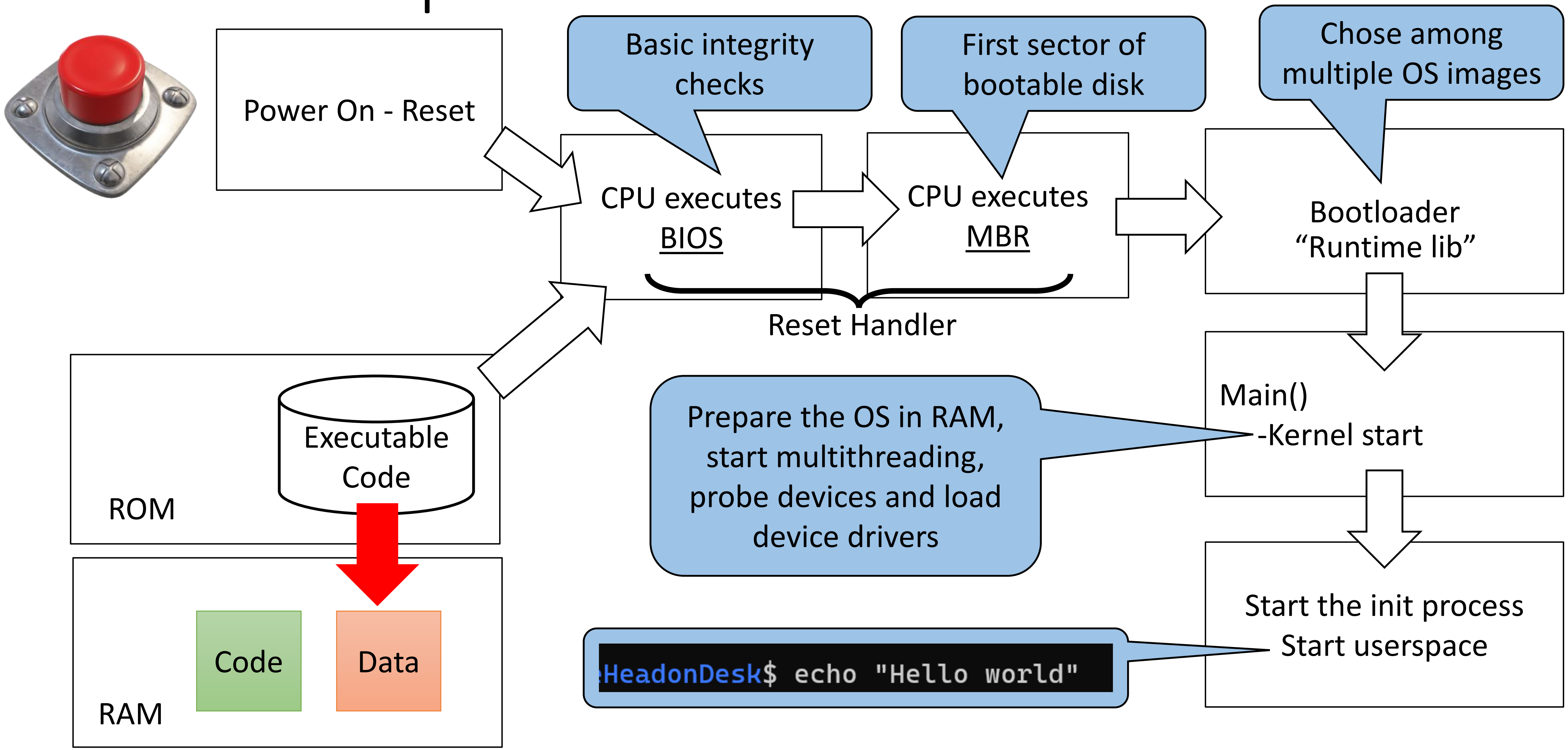
OS Bootstrap – The linux case

- 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

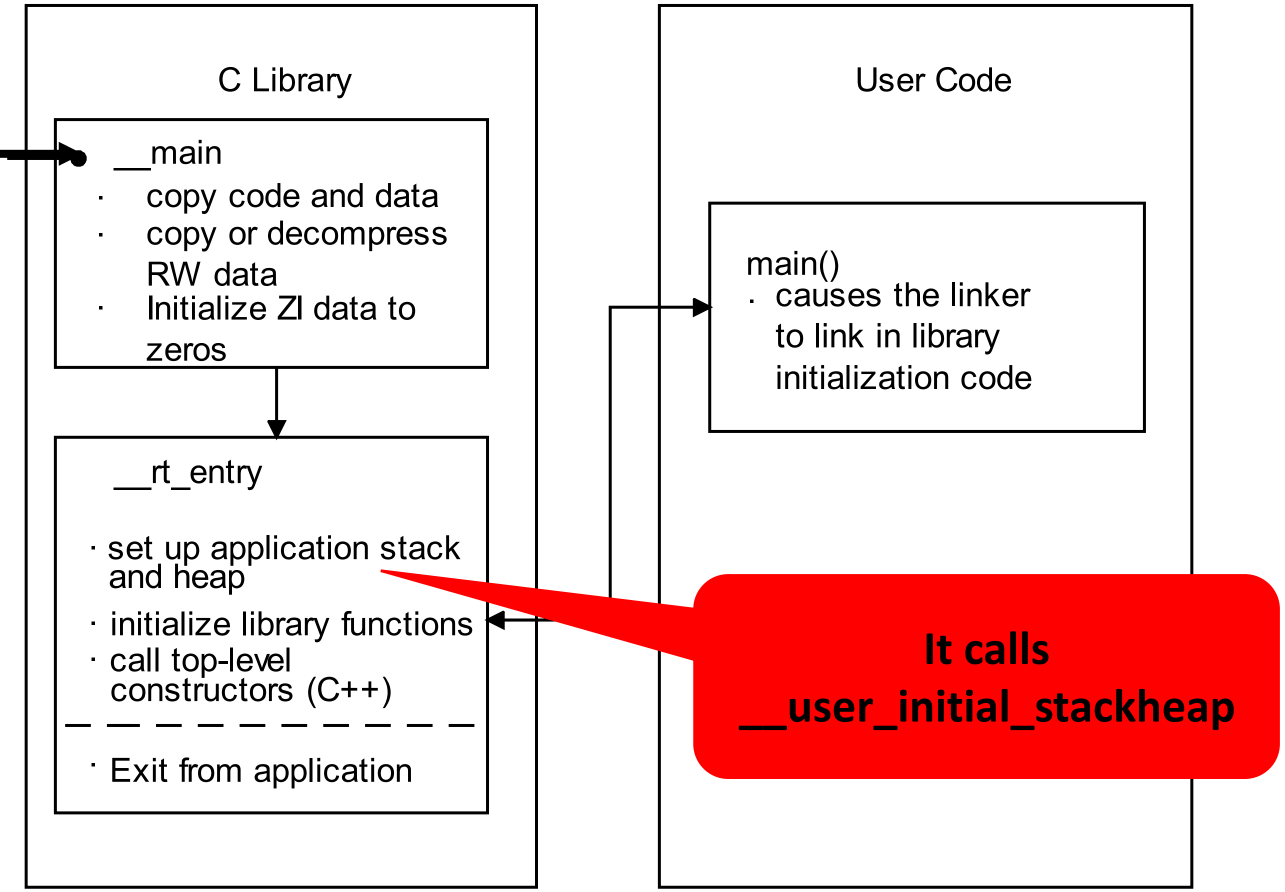
- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
  - 0 – halt, 1 – Single user mode, 2 – Multiuser, without NFS, 3 – Full multiuser mode, 4 – unused, 5 – X11, 6 – reboot.
- Init identifies the default initlevel from /etc/inittab and uses that to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

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)

main.c ASM_funct.s startup_LPC17xx.s

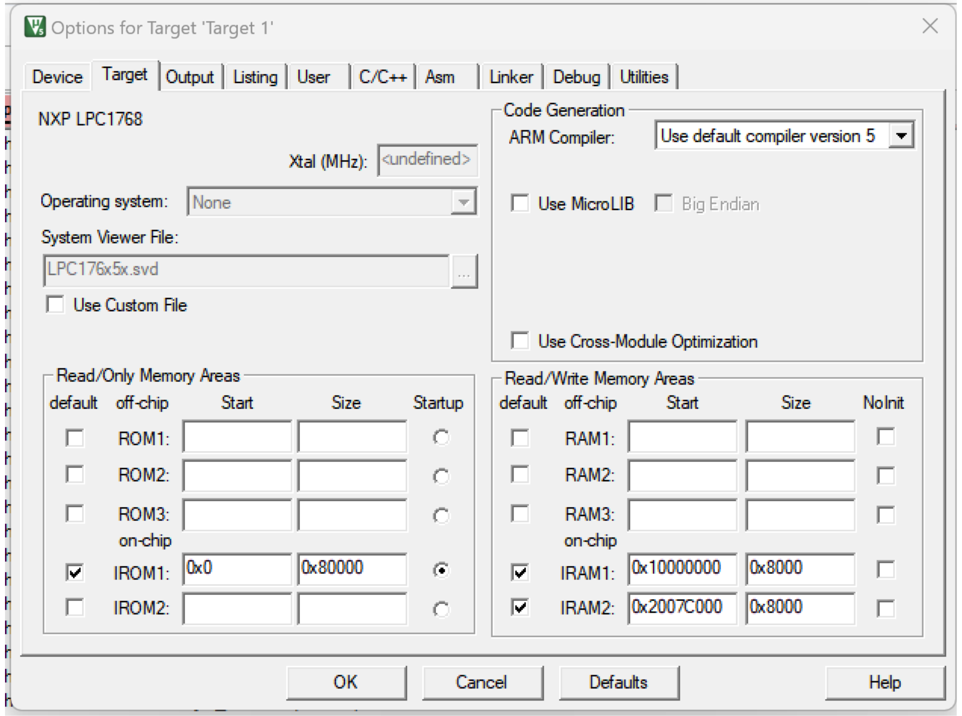
```
                IF      :DEF:__MICROLIB

                EXPORT  __initial_sp
                EXPORT  __heap_base
                EXPORT  __heap_limit

                ELSE

                IMPORT  __use_two_region_memory
                EXPORT  __user_initial_stackheap
__user_initial_stackheap

                LDR     R0, =  Heap_Mem ;; heap start
                LDR     R1, =(Stack_Mem); + Stack_Size/2) ; msp
                LDR     R2, = (Heap_Mem +  Heap_Size) ;; heap end
                LDR     R3, = (Stack_Mem + Stack_Size/2) ; psp
                BX      LR

                ALIGN

                ENDIF


                END
```

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”.
- <u>HowTo and HowNotTo - Examples.</u>

Example – Where do they belong?

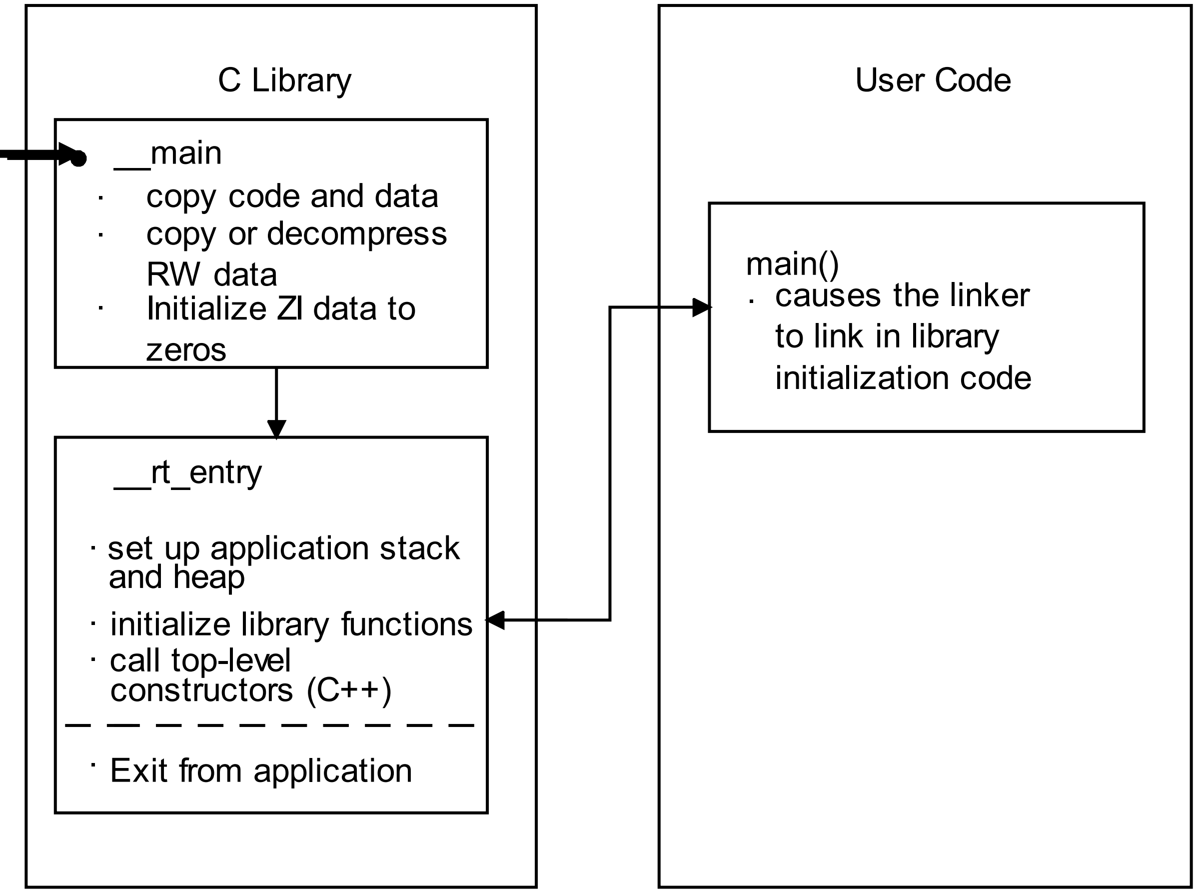
- See the map file.




```c

const int pippo[]={21,32};
int this_is_zero;
int catched=234;
volatile  int array[N] = {1,2,3,4,5,6,7,8,9,10};

void my_fancy_function(int * a ){
  if (a==0) {
    return ;
  }
  (*a)++;
}

int main(void){


  int i=0;
  volatile int value=0;
  volatile int my_var=pippo[0];

  my_fancy_function(&my_var);

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




  while(1);
}
```

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[]={21,32};
int this_is_zero;
int catched=234;
volatile  int array[N] = {1,2,3,4,5,6,7,8,9,10};

void my_fancy_function(int * a ){
  if (a==0) {
    return ;
  }
  (*a)++;
}

int main(void){


  int i=0;
  volatile int value=0;
  volatile int my_var=pippo[0];

  my_fancy_function(&my_var);

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



  while(1);
}
```

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

Example – On the fly variable declaration

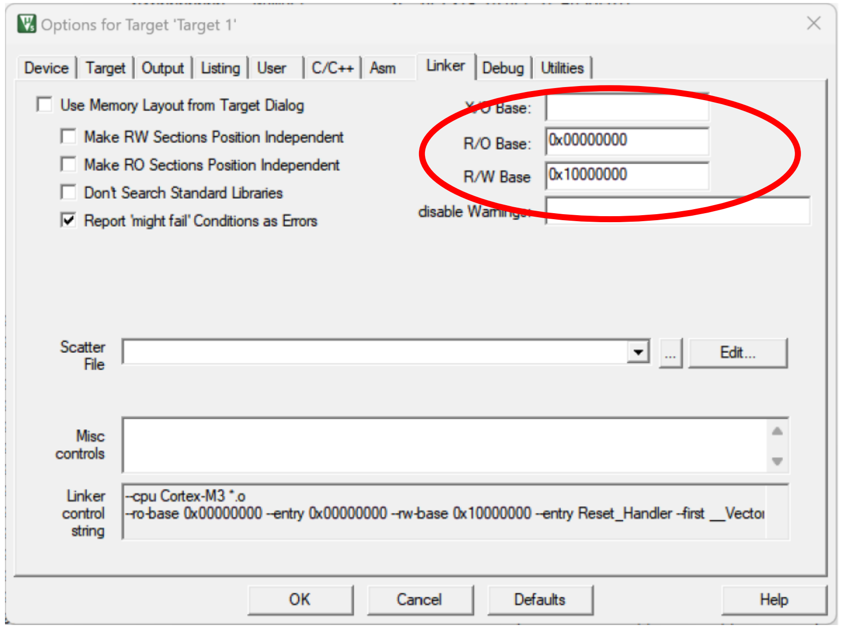
- See build log.

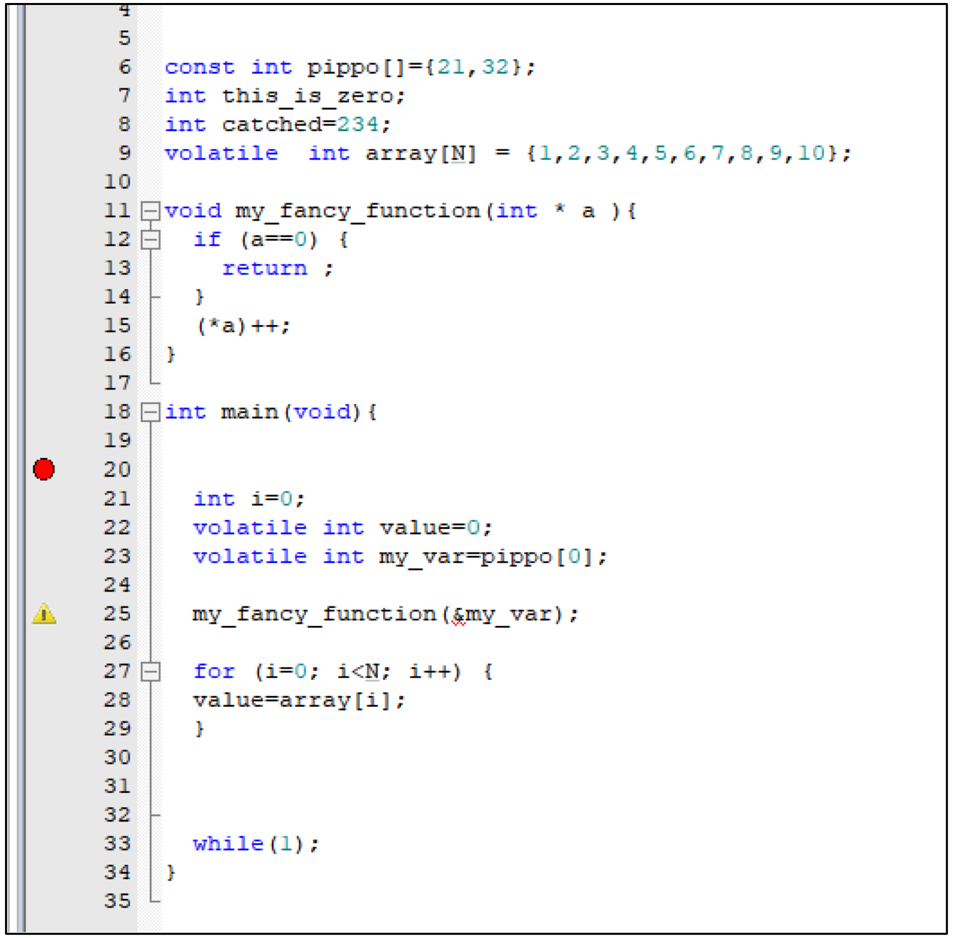
```
*** Using Compiler 'V5.06 update 7 (build 960)', folder: 'D:\programmi\keil\ARM\ARMCLANG5\Bin'
Build target 'Target 1'
compiling main.c...
main.c(25): warning:  #167-D: argument of type "volatile int *" is incompatible with parameter of typ
        my_fancy_function(&my_var);
main.c(27): error:  #29: expected an expression
        for (int i=0; iSoftware Packages used:

Package Vendor: ARM
                http://www.keil.com/pack/ARM.CMSIS.5.9.0.pack
                ARM.CMSIS.5.9.0
                CMSIS (Common Microcontroller Software Interface Standard)
   * Component: CORE Version: 5.6.0

Package Vendor: Keil
                http://www.keil.com/pack/Keil.LPC1700_DFP.2.7.1.pack
                Keil.LPC1700_DFP.2.7.1
                NXP LPC1700 Series Device Support, Drivers and Examples for MCB1700 and LPC1788-32
```

```c

const int pippo[]={21,32};
int this_is_zero;
int catched=234;
volatile  int array[N] = {1,2,3,4,5,6,7,8,9,10};

void my_fancy_function(int * a ){
  if (a==0) {
    return ;
  }
  (*a)++;
}

int main(void){


  int i=0;
  volatile int value=0;
  volatile int my_var=pippo[0];

  my_fancy_function(&my_var);

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



  while(1);
}

```

Example - Changing the Section start address

- See the map file.
- Let’s put the IROM1 start address at 0x00100.
- Same for RW base address (see array in map file).





Example – Adding a custom section

- See the map file.

```
Removing Unused input sections from the image.

    Removing asm_funct.o(my_asm_functions), (32 bytes).

1 unused section(s) (total 32 bytes) removed from the image.
```

- Let’s call the function from the main.

```

                AREA my_asm_functions, CODE, READONLY
                EXPORT  ASM_funct
ASM_funct
                ; save current SP for a faster access
                ; to parameters in the stack
                MOV   r12, sp
                ; save volatile registers
                STMFD sp!,{r4-r8,r10-r11,lr}
                ; extract argument 4 and 5 into R4 and R5
                LDR   r4, [r12]
                LDR   r5, [r12,#4]

                LDR r12, =ASM_funct
                STR r5, [r12]

                ; setup a value for R0 to return
                MOV   r0, r5
                ; restore volatile registers
                LDMFD sp!,{r4-r8,r10-r11,pc}

                END
```

Example – Adding a custom section

- See the map file.

```
Removing Unused input sections from the image.

    Removing asm_funct.o(my_asm_functions), (32 bytes).

1 unused section(s) (total 32 bytes) removed from the image.
```

- Let's call the function from the main.

main.c | startup_LPC17xx.s | ASM_funct.s | ABI_C+ASM.map

```
    .text                          0x00000364   Section
    .text                          0x00000378   Section
    my_asm_functions               0x00000384   Section
    .constdata                     0x000003a4   Section
    .data                          0x10000000   Section
    .bss                           0x10000030   Section
    HEAP                           0x10000090   Section
```

```
                AREA my_asm_functions, CODE, READONLY
                EXPORT  ASM_funct
ASM_funct
                ; save current SP for a faster access
                ; to parameters in the stack
```

```
int main(void){

    int azeroth=0xFFFFFFFF, kalimdor=2,
      outlands=3, northrend=4,
      pandaria=5, shadowlands=6;
    volatile int r=0;
    int i=0;
    volatile int value=0;
    volatile int my_var=pippo[0];

    //my_fancy_function(&my_var);

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


    r = ASM_funct(azeroth, kalimdor,
    outlands, northrend,
    pandaria, shadowlands);

    while(1);
}
```

Example – Adding a custom section

- You can also use __attribute__((section(".ARM.__at_<address>"))) to specify the absolute address of a variable or a section name.
- The linker help us.

This is unsafe!!!

```
volatile  int array[N] = {1,2,3,4,5,6,7,8,9,10};
```

```
#if TEST_UNSAFE
  __attribute__ ((section(".ARM.__at_0x10000008"))) int fancy_value[]={
                                                    0xcafecafe,0xc1a0c1a0};
  #endif
```

```
    int *p=(int *)0x10000008;
    *p=101010;


    my_fancy_function(&my_var);

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

```

- It directly handles all the data placement during the link phase.

startup_LPC17xx.s | ASM_funct.s | main.c | ABI_C+ASM.map

fancy_value	0x10000008	Data	8	main.o(.ARM.__at_0x10000008)
this_is_zero	0x10000010	Data	4	main.o(.data)
catched	0x10000014	Data	4	main.o(.data)
array	0x10000018	Data	40	main.o(.data)
__libspace_start	0x10000040	Data	96	libspace.o(.bss)
__temporary_stack_top$libspace	0x100000a0	Data	0	libspace.o(.bss)

Example – Exporting data from asm to C

- Data can be exported from asm to C (like functions).
- Why in my_amazing_asm_vector do we have the first data?

```
            AREA my_amazing_data, DATA, READWRITE
            EXPORT _my_asm_vector
_my_asm_vector DCD 0x1234,0x3214
```

```
extern unsigned int  * _my_asm_vector;
```

```
volatile unsigned int * my_amazing_asm_vector=_my_asm_vector;
```

Example – Exporting data from asm to C

- Data can be exported from asm to C (like functions).
- Why in my_amazing_asm_vector do we have the first data?
- Let's have a look at the asm_func.lst file.

```
ARM Macro Assembler    Page 1 Alphabetic symbol ordering
Relocatable symbols

_my_asm_vector 00000000

Symbol: _my_asm_vector
  Definitions
    At line 28     file ASM_funct.s
  Uses
    At line 27     le ASM_funct.s
Comment: _my_asm        used once
my_amazing_data
```

```
  25 0000001C
  26 0000001C 00000000        AREA           my_amazing_data, DATA, READWRIT
E
  27 00000000                 EXPORT         _my_asm_vector
  28 00000000 00001234
                00003214 _my_asm_vector
                              DCD            0x1234,0x3214
  29 00000008
  30 00000008
```

It is a symbol (label) to the first value!!

Example – Exporting data from asm to C

- The linker is in charge of substituting the symbols with addresses during the link phase.
- A linker symbol is not equivalent to a variable declaration in high level language, it is instead a symbol that does not have a value.
- The label _my_asm_vector is a linker symbol (it is directly defined in assembly and used in C), when the linker resolves the symbols it retrieves the first value of the vector (independently from the data type).
- The correct way to export data defined in assembly is to force the use of the address.

```
            AREA my_amazing_data, DATA, READWRITE
            EXPORT _my_asm_vector
_my_asm_vector DCD 0x1234,0x3214
```

```
extern unsigned int  * _my_asm_vector;
```

```
extern unsigned int  _my_asm_vector;
```

```
volatile unsigned int * my_amazing_asm_vector=&_my_asm_vector;
```

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);
}

```

Remainder!!

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

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