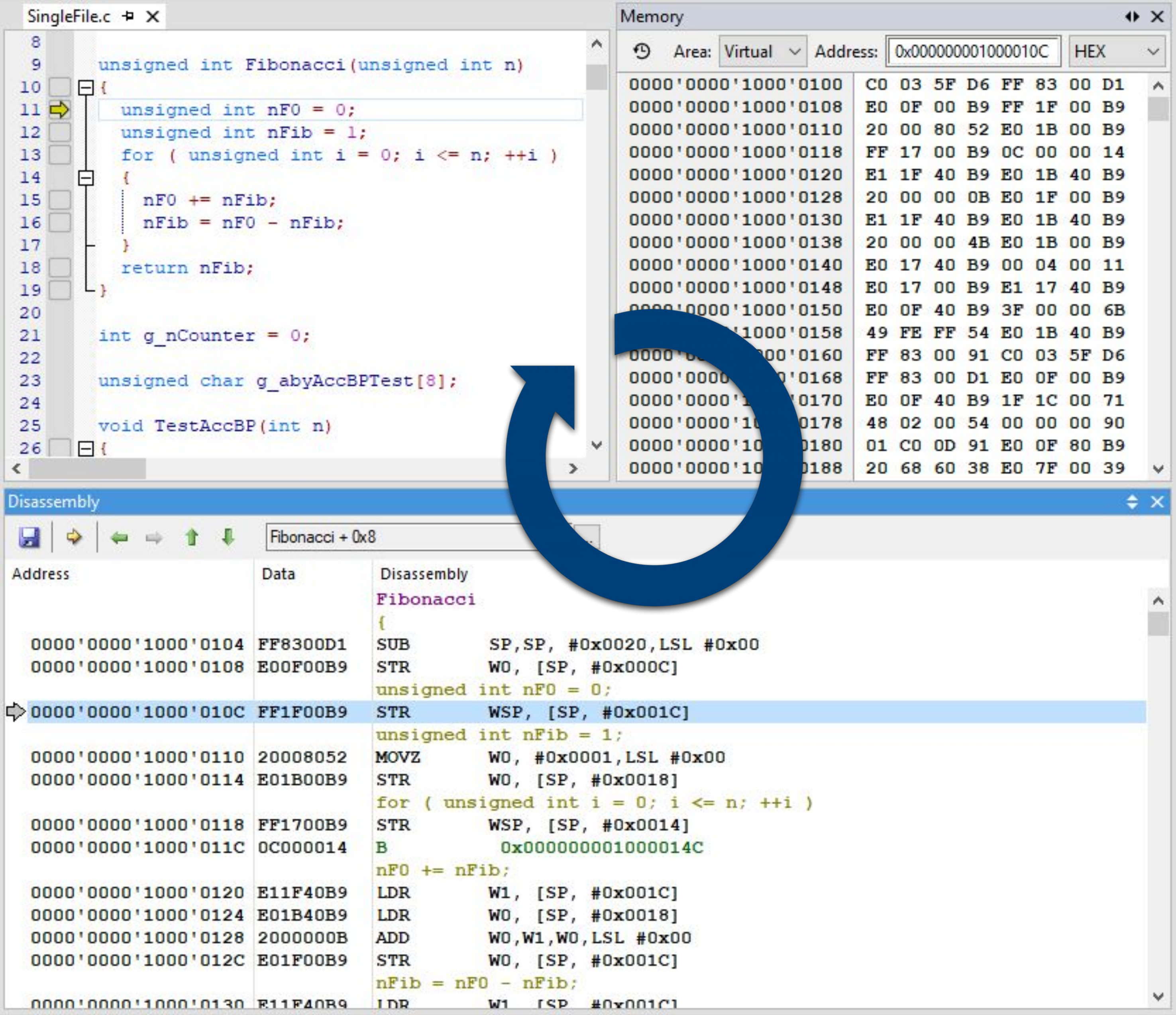
Before main ()

C project setup, compilation and startup based on GCC

Primoz Alic
(Jan 2019)

Goal



SingleFile.c

```
unsigned int Fibonacci(unsigned int n)
{
  unsigned int nF0 = 0;
  unsigned int nFib = 1;
  for ( unsigned int i = 0; i <= n; ++i )
  {
    nF0 += nFib;
    nFib = nF0 - nFib;
  }
  return nFib;
}

int g_nCounter = 0;

unsigned char g_abyAccBPTest[8];

void TestAccBP(int n)
{
```

Memory

Area: Virtual Address: 0x000000001000010C HEX

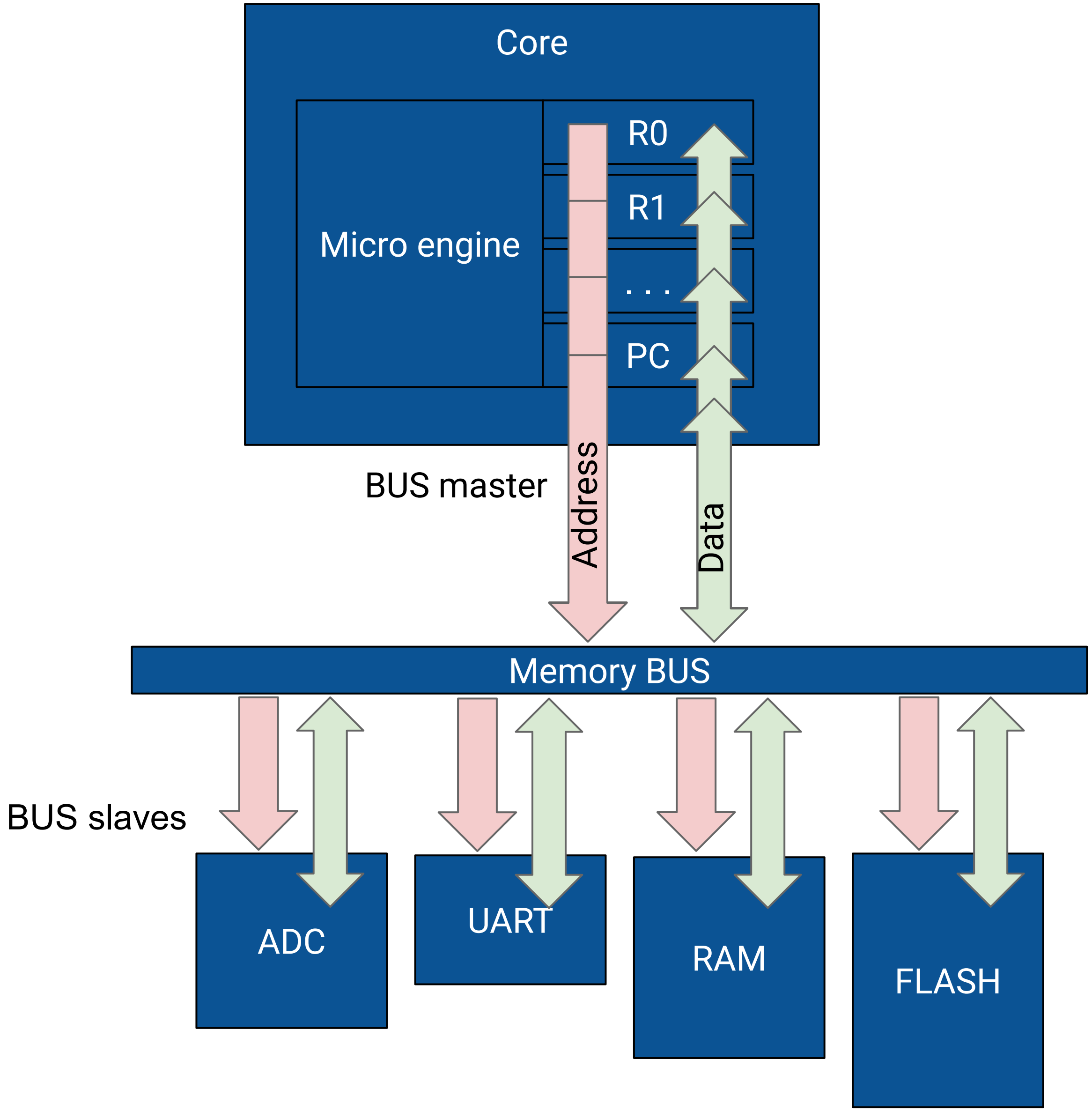
```
0000'0000'1000'0100  C0 03 5F D6 FF 83 00 D1
0000'0000'1000'0108  E0 0F 00 B9 FF 1F 00 B9
0000'0000'1000'0110  20 00 80 52 E0 1B 00 B9
0000'0000'1000'0118  FF 17 00 B9 0C 00 00 14
0000'0000'1000'0120  E1 1F 40 B9 E0 1B 40 B9
0000'0000'1000'0128  20 00 00 0B E0 1F 00 B9
0000'0000'1000'0130  E1 1F 40 B9 E0 1B 40 B9
0000'0000'1000'0138  20 00 00 4B E0 1B 00 B9
0000'0000'1000'0140  E0 17 40 B9 00 04 00 11
0000'0000'1000'0148  E0 17 00 B9 E1 17 40 B9
0000'0000'1000'0150  E0 0F 40 B9 3F 00 00 6B
          '1000'0158  49 FE FF 54 E0 1B 40 B9
0000'        00'0160  FF 83 00 91 C0 03 5F D6
0000'0000      0'0168  FF 83 00 D1 E0 0F 00 B9
0000'0000'1     '0170  E0 0F 40 B9 1F 1C 00 71
0000'0000'10     0178  48 02 00 54 00 00 00 90
0000'0000'10     0180  01 C0 0D 91 E0 0F 80 B9
0000'0000'10     0188  20 68 60 38 E0 7F 00 39
```

Disassembly

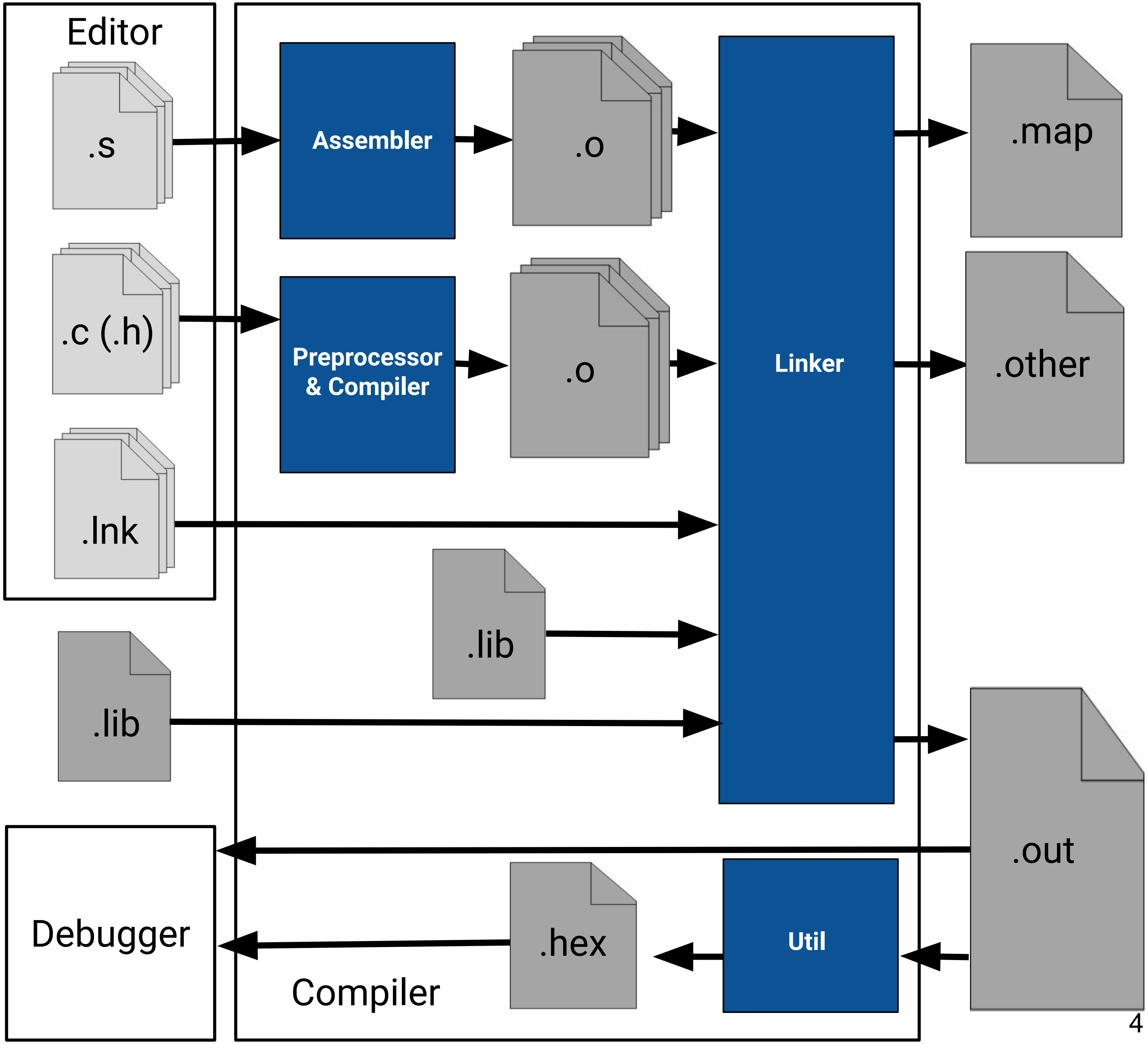
Fibonacci + 0x8

```
Address              Data      Disassembly
                               Fibonacci
                               {
0000'0000'1000'0104  FF8300D1  SUB        SP,SP, #0x0020,LSL #0x00
0000'0000'1000'0108  E00F00B9  STR        W0, [SP, #0x000C]
                               unsigned int nF0 = 0;
0000'0000'1000'010C  FF1F00B9  STR        WSP, [SP, #0x001C]
                               unsigned int nFib = 1;
0000'0000'1000'0110  20008052  MOVZ       W0, #0x0001,LSL #0x00
0000'0000'1000'0114  E01B00B9  STR        W0, [SP, #0x0018]
                               for ( unsigned int i = 0; i <= n; ++i )
0000'0000'1000'0118  FF1700B9  STR        WSP, [SP, #0x0014]
0000'0000'1000'011C  0C000014  B           0x000000001000014C
                               nF0 += nFib;
0000'0000'1000'0120  E11F40B9  LDR        W1, [SP, #0x001C]
0000'0000'1000'0124  E01B40B9  LDR        W0, [SP, #0x0018]
0000'0000'1000'0128  2000000B  ADD        W0,W1,W0,LSL #0x00
0000'0000'1000'012C  E01F00B9  STR        W0, [SP, #0x001C]
                               nFib = nF0 - nFib;
0000'0000'1000'0130  E11F40B9  LDR        W1, [SP, #0x001C]
```

MCU


Core
Micro engine
R0
R1
. . .
PC
Address
Data
BUS master
Memory BUS
BUS slaves
ADC
UART
RAM
FLASH

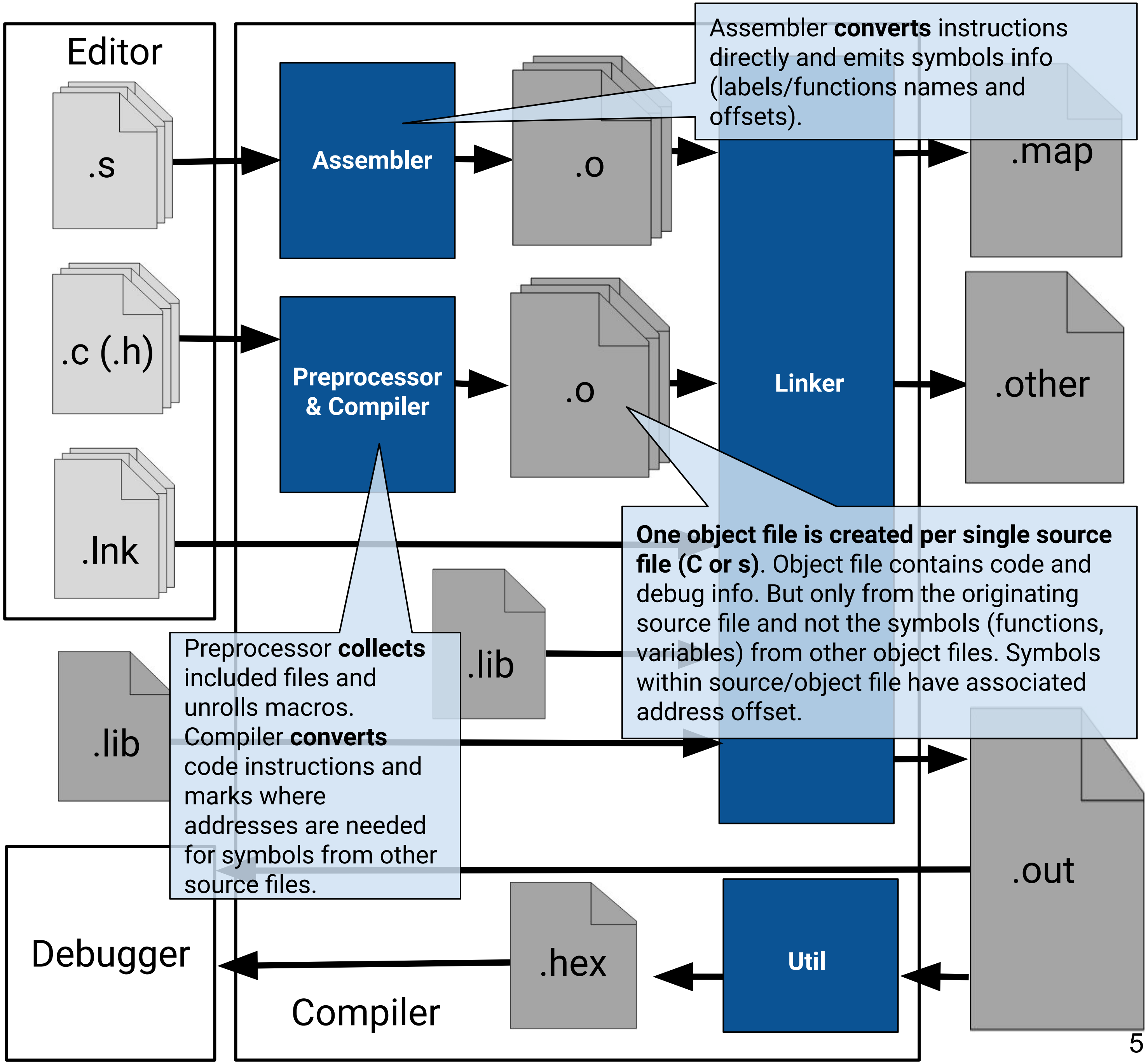
Build


Editor
.s
.c (.h)
.lnk
.lib
Assembler
Preprocessor & Compiler
.o
.o
.lib
Linker
.map
.other
.out
Util
.hex
Debugger
Compiler

Build

Stage I.

COMPILING



Assembler **converts** instructions directly and emits symbols info (labels/functions names and offsets).

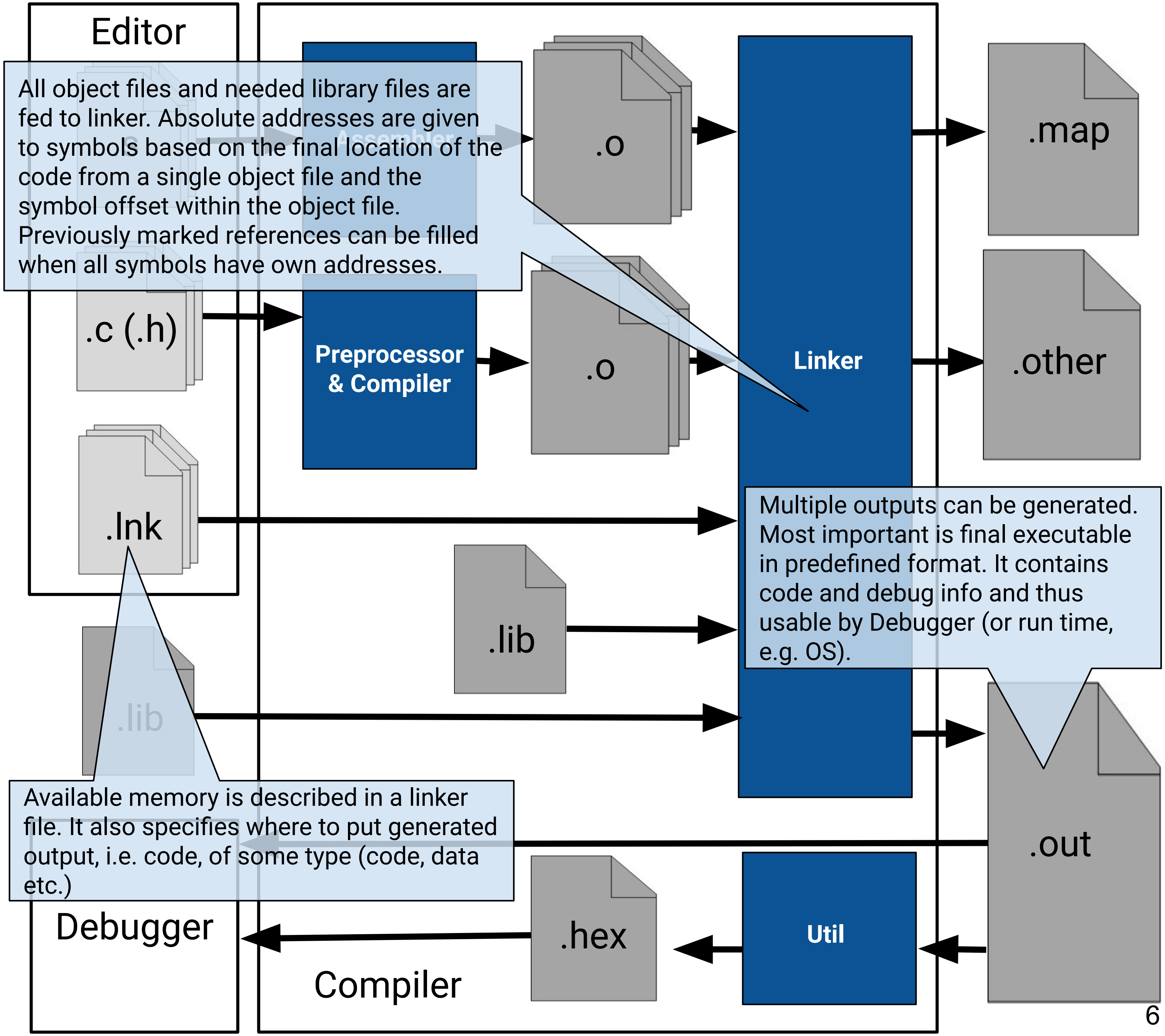
Preprocessor **collects** included files and unrolls macros. Compiler **converts** code instructions and marks where addresses are needed for symbols from other source files.

One object file is created per single source file (C or s). Object file contains code and debug info. But only from the originating source file and not the symbols (functions, variables) from other object files. Symbols within source/object file have associated address offset.

Build

Stage I.

LINKING



All object files and needed library files are fed to linker. Absolute addresses are given to symbols based on the final location of the code from a single object file and the symbol offset within the object file. Previously marked references can be filled when all symbols have own addresses.

Multiple outputs can be generated. Most important is final executable in predefined format. It contains code and debug info and thus usable by Debugger (or run time, e.g. OS).

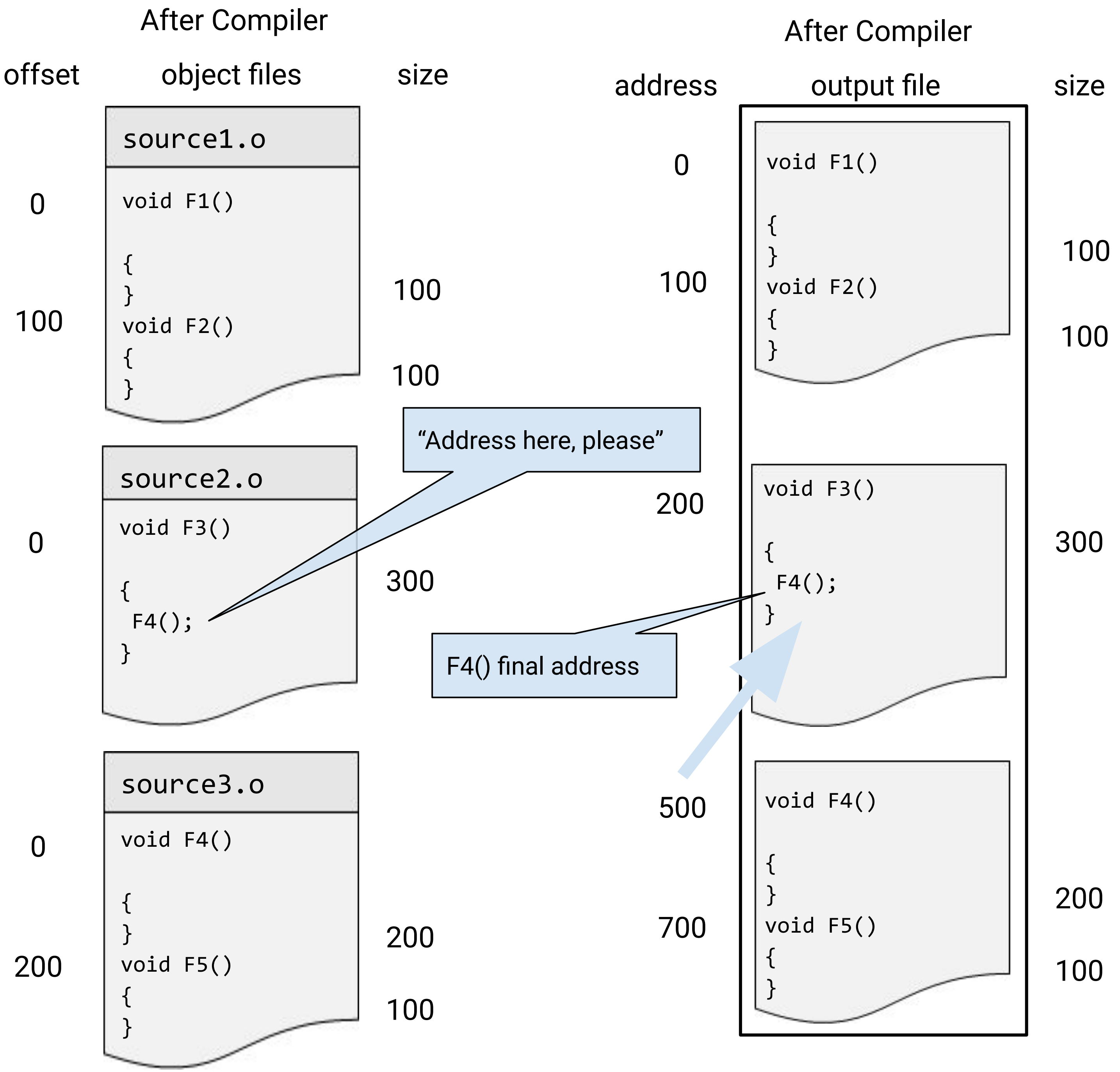
Available memory is described in a linker file. It also specifies where to put generated output, i.e. code, of some type (code, data etc.)

Build

Stage II.

LINKING

Example:
3 object files,
each from one source file



After Compiler

offset | object files | size

source1.o

0

```
void F1()
{
}
```

100

100

```
void F2()
{
}
```

100

source2.o

0

```
void F3()
{
 F4();
}
```

300

source3.o

0

```
void F4()
{
}
```

200

200

```
void F5()
{
}
```

100

"Address here, please"

After Compiler

address | output file | size

0

```
void F1()
{
}
```

100

100

```
void F2()
{
}
```

100

200

```
void F3()
{
 F4();
}
```

300

F4() final address

500

```
void F4()
{
}
```

200

700

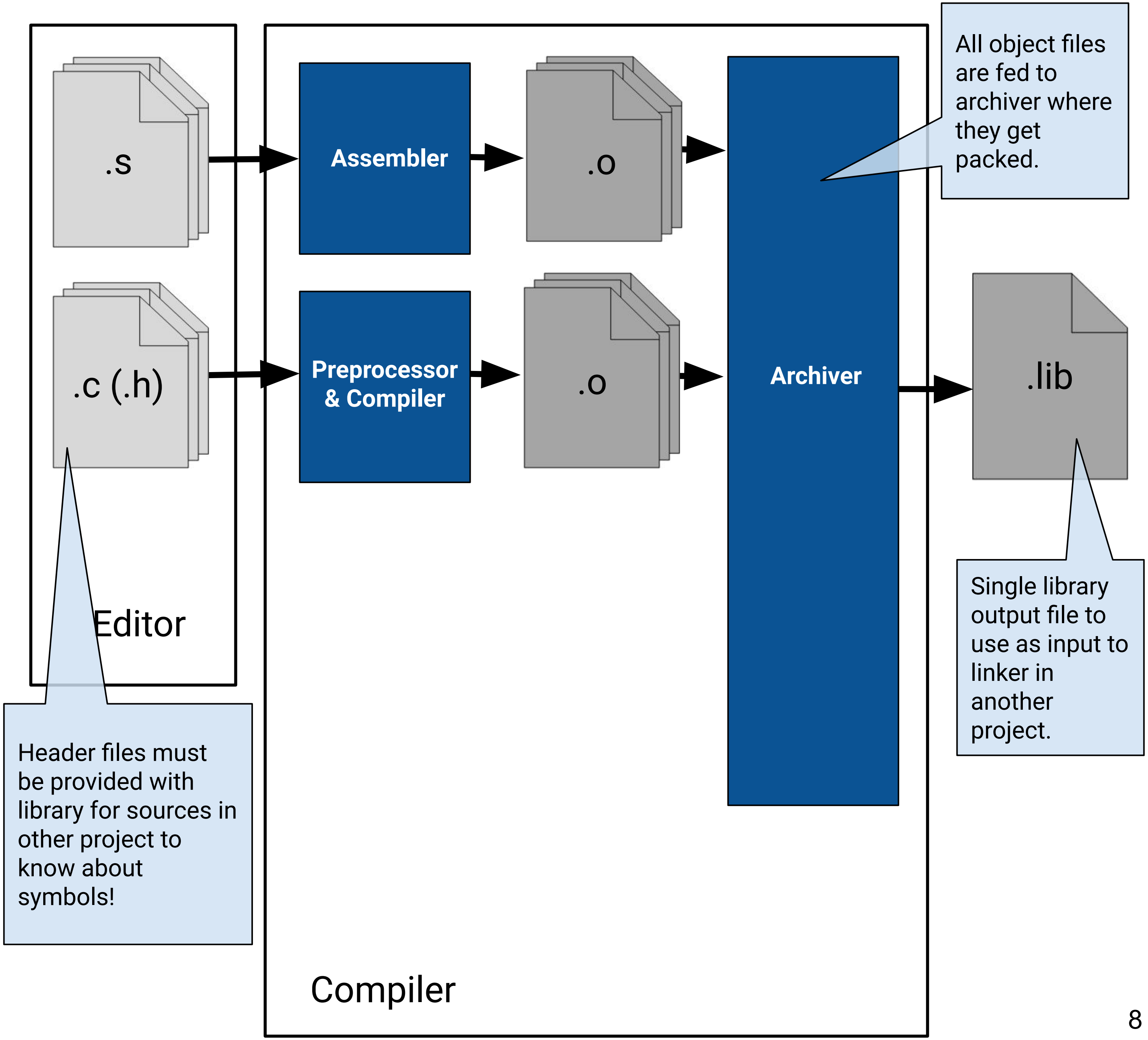
```
void F5()
{
}
```

100

Build

Stage II.

Alternative
ARCHIVING



All object files are fed to archiver where they get packed.

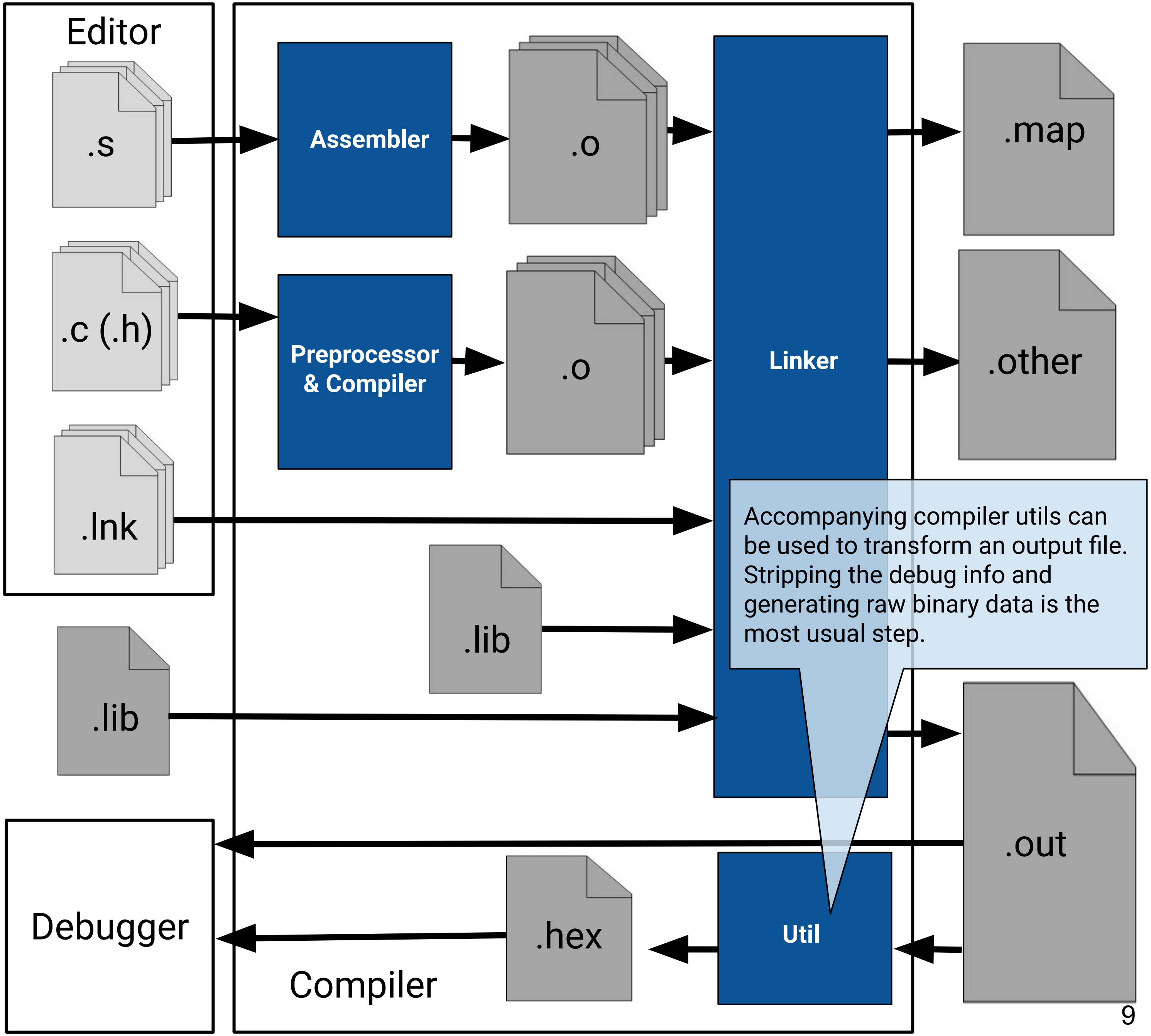
Single library output file to use as input to linker in another project.

Header files must be provided with library for sources in other project to know about symbols!

Build

Stage III.

CONVERTING



Accompanying compiler utils can be used to transform an output file. Stripping the debug info and generating raw binary data is the most usual step.

GCC

GNU Compiler Collection

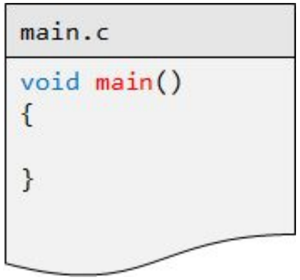
- Multiple purpose executables (compiling: gcc, ar, ld… utilities: objcopy...)
- Multiple levels of executables (similar file names)
- Top executable calls other executables
- Use executables from topmost **bin** folder
- Use **-pipe** to avoid temp files for executables internal data propagation (useful for parallel compilation)
- Cross-compilation means compiling on one architecture for other architecture (e.g. on x86 for ARM)
- File names in form of: architecture-os-calling convention, e.g. arm-none-eabi; none means **no OS support** and not **not eabi** (eabi: Embedded Application Binary Interface)
- Documents usually deep under **share** subfolder (USE THEM!)
- Single file compile:
  - *gcc.exe -march=armv7-m -mthumb -mfpu=vfp -Wimplicit -g3 -O0 -c -o<out_path> <in_path>*
- Link:
  - *gcc.exe -march=armv7-m -mthumb -mfpu=vfp -nostartfiles -Wl,--script=E:\Project\LinkerScript.lnk,--output=<out_path> -nostdlib -nodefaultlibs -fno-exceptions <obj_in_path>... <lib_path>...*
- Single file preprocess only:

  gcc.exe -P -E -o<out_path> <in_path>

Compile Options (usual)

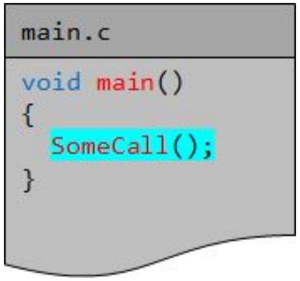
-march=armv7-m	Generate instructions for Cortex M3 or M4 device.
-mthumb	Generate architecture Thumb instructions (Thumb2 for armv7-m).
-mfpu=vfp	Floating point ABI (with -mfloat-abi=hard/soft - soft by default).
-pipe	Use standard input/output to transfer data between stages.
-xc	Force C language (capital '.C' file can be compiled as C++).
-g3	Generate most debug info.
-O0	Don't optimize (others: 0, 1, 2, 3, s, fast, g).
-c	Stop after compilation.
-Wimplicit	Warn when calling undeclared function.
-ffunction-sections	Put each function in own .text section.
-DSOME_SYMBOL	Define 'SOME_SYMBOL' for preprocessor (e.g. _DEBUG).
-IE:\Path\To\Folder\With\Include\Files	Include path when looking for included files (#include "File.h").
-oE:\Project\Debug\Output\File.o	Where to put generated object file.
E:\Path\To\Source\File.c	Source file to compile.

C-H-C



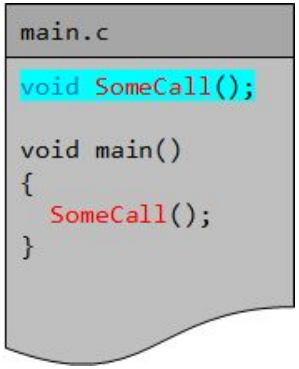
```
main.c

void main()
{

}
```

Compiles and links.



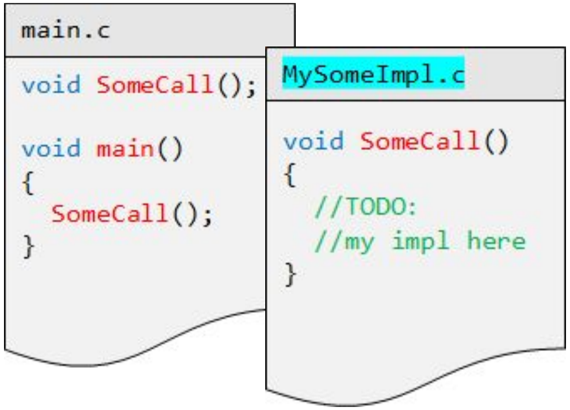
```
main.c

void main()
{
   SomeCall();
}
```

Compiles with warning (implicit function are understood as *int* returning functions without parameters). DOESN'T LINK! Linker can't find *SomeCall* symbol.



```
main.c

void SomeCall();

void main()
{
   SomeCall();
}
```

Compiles without warning – function declared. Still does not link.



```
main.c

void SomeCall();

void main()
{
   SomeCall();
}
```

```
MySomeImpl.c

void SomeCall()
{
   //TODO:
   //my impl here
}
```

Compiles and links.

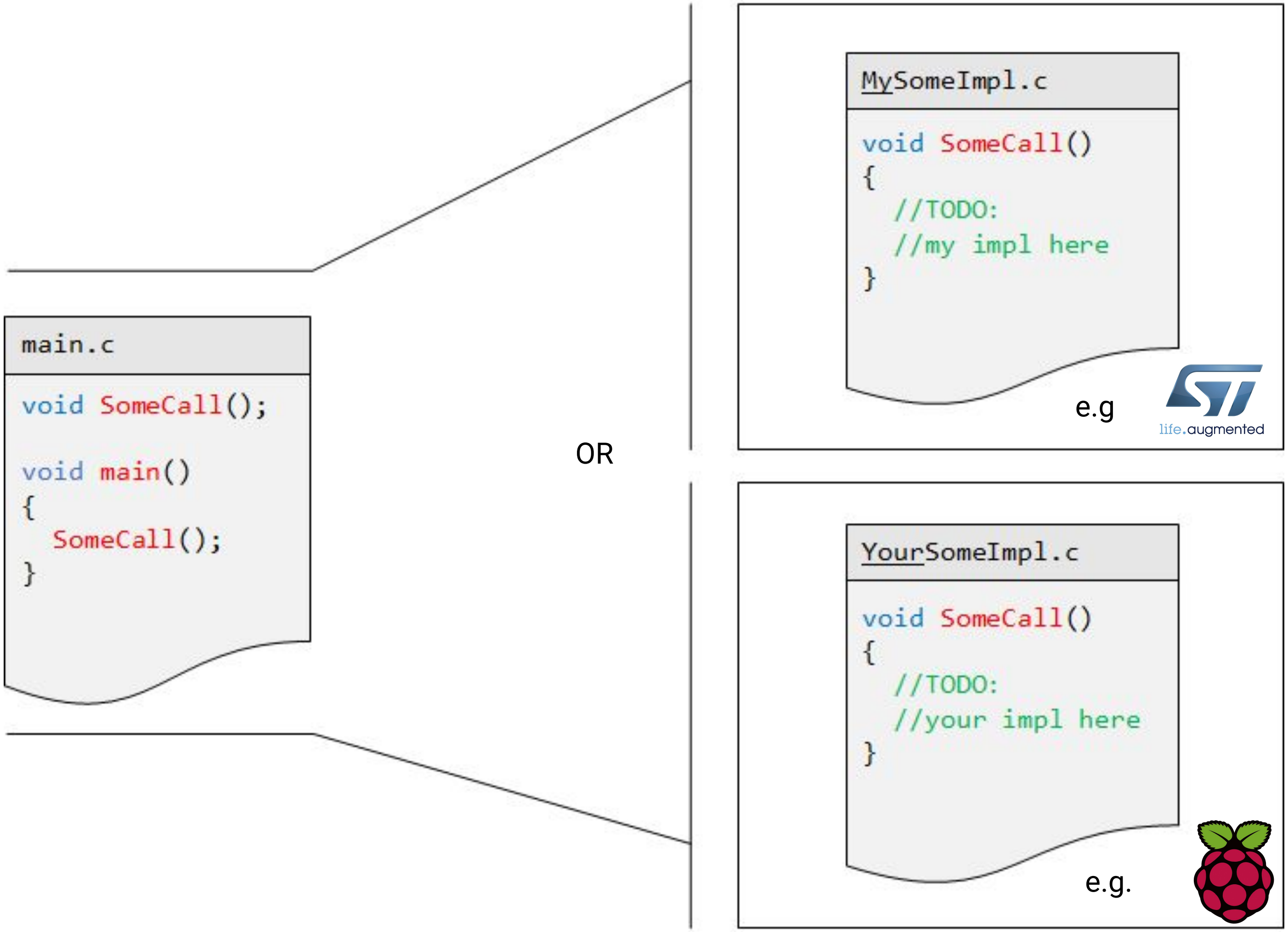
Notice no #include.

Link Options (usual)

--march=armv7-m	Use libraries for Cortex M3 or M4 device.
-mthumb	Use architecture Thumb libraries (Thumb2 for armv7-m).
-mfpu=vfp	Floating point library (with -mfloat-abi=hard/soft soft by default).
-pipe	Use standard input/output to transfer data between stages.
-nostartfiles	Don’t add any startup code.
-LE:\Path\To\Some\Folder\With\Libs	Consider the folder when searching for libraries.
--Wl,	Comma separated list of linker specific options will follow:
--output=E:\Project\Debug\Output.out,	Where to create final output file.
-Map=E:\Project\Debug\Output.map,	Where to create map file.
--script=E:\Project\LinkerScript.lnk,	Additional link options.
--gc-sections	Remove unreferenced section.
-nostdlib	Only use specified libs (-l).
-nodefaultlibs	Don’t use any default libs automatically.
-fno-exceptions	Throwing/catching is not used.
-lSomeLib	Link with ‘libSomeLib.a’.

C-H-C

- Can have multiple Implementations, but linked with only one at the time.



main.c

```
void SomeCall();

void main()
{
  SomeCall();
}
```

MySomeImpl.c

```
void SomeCall()
{
  //TODO:
  //my impl here
}
```

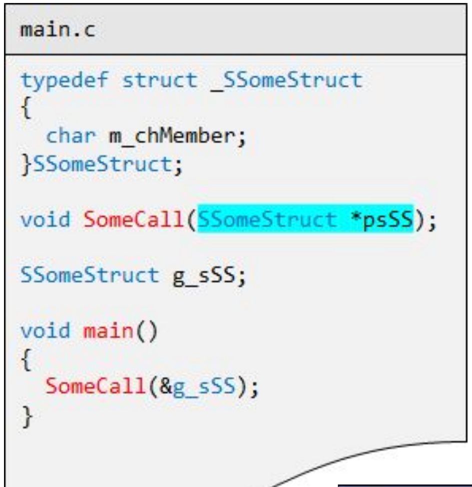
YourSomeImpl.c

```
void SomeCall()
{
  //TODO:
  //your impl here
}
```

SwitchLED(bool bOn) is good practical example. Some library code calls this function to report status. On one type of HW this function is implemented in a file specific to that HW. Implementation is aware of GPIO registers and pin configuration. Same library can run on other type of HW where SwitchLED is implemented in the other source file specific to different HW. HW specific implementations are usually called HAL (Hardware Abstraction Layer), also simply drivers.

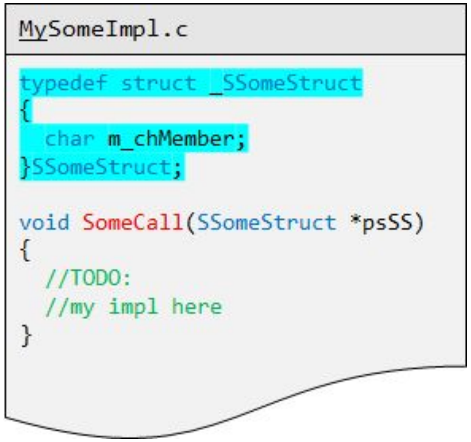
C-H-C

- It is impractical to repeat (copy/paste) code in each file. It leads to errors when desynchronized.



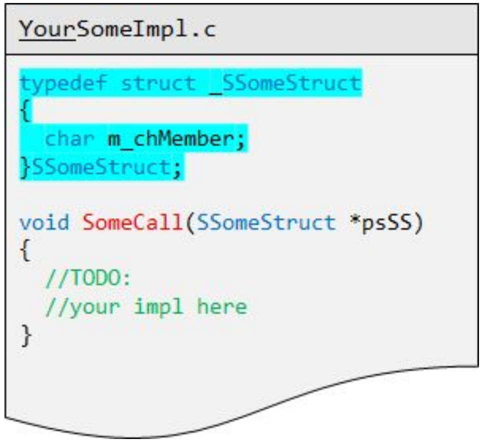
main.c

```
typedef struct _SSomeStruct
{
   char m_chMember;
}SSomeStruct;

void SomeCall(SSomeStruct *psSS);

SSomeStruct g_sSS;

void main()
{
   SomeCall(&g_sSS);
}
```



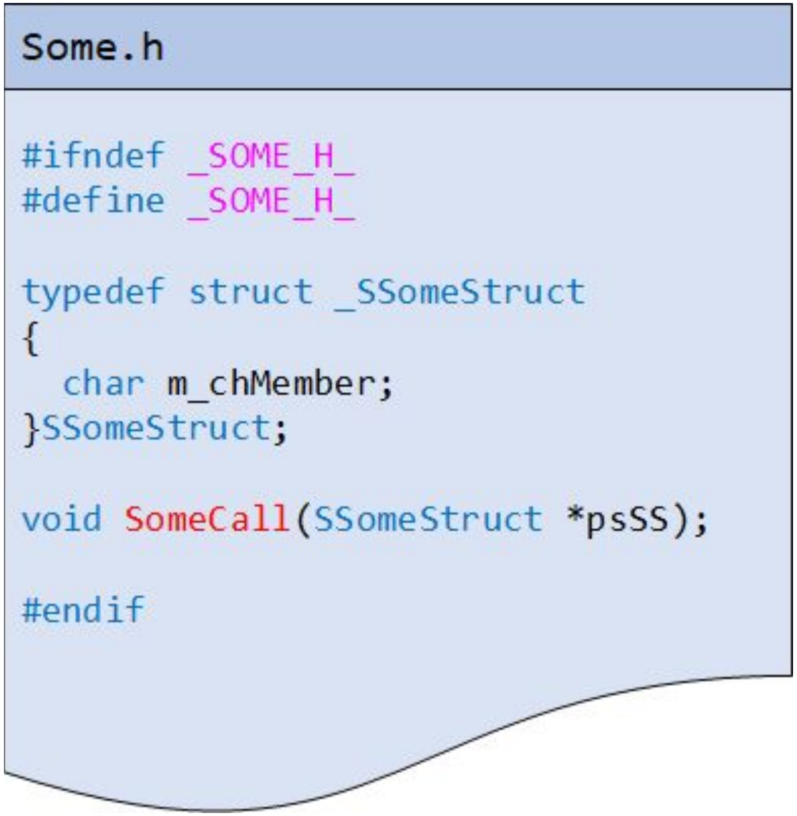
MySomeImpl.c

```
typedef struct _SSomeStruct
{
   char m_chMember;
}SSomeStruct;

void SomeCall(SSomeStruct *psSS)
{
   //TODO:
   //my impl here
}
```



YourSomeImpl.c

```
typedef struct _SSomeStruct
{
   char m_chMember;
}SSomeStruct;

void SomeCall(SSomeStruct *psSS)
{
   //TODO:
   //your impl here
}
```

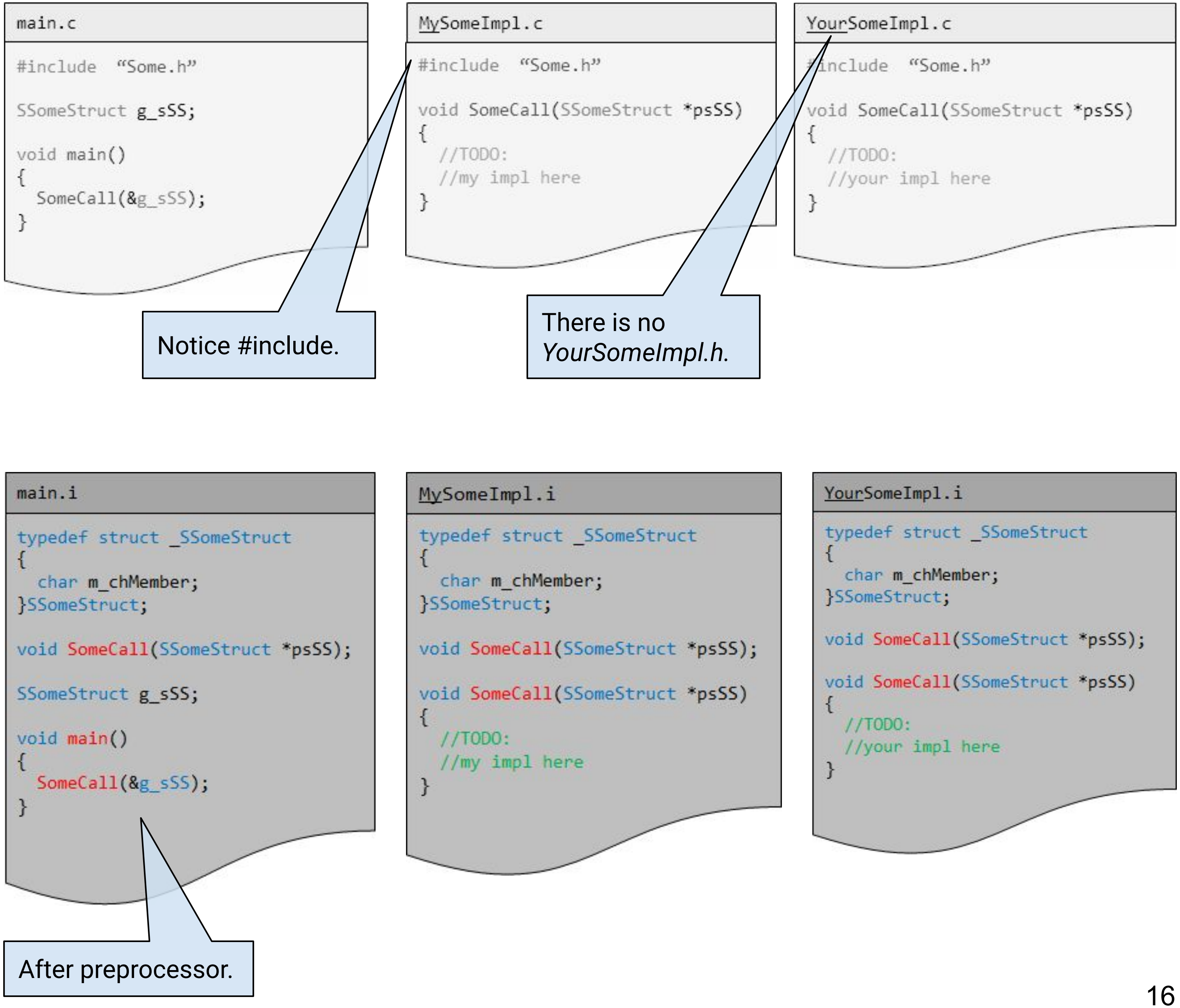


Some.h

```
#ifndef _SOME_H_
#define _SOME_H_

typedef struct _SSomeStruct
{
   char m_chMember;
}SSomeStruct;

void SomeCall(SSomeStruct *psSS);

#endif
```

C-H-C

Files after preprocessor (*.i) are same as previous with copy/pasted code.



main.c

```c
#include   “Some.h”

SSomeStruct g_sSS;

void main()
{
  SomeCall(&g_sSS);
}
```

MySomeImpl.c

```c
#include   “Some.h”

void SomeCall(SSomeStruct *psSS)
{
  //TODO:
  //my impl here
}
```

YourSomeImpl.c

```c
#include   “Some.h”

void SomeCall(SSomeStruct *psSS)
{
  //TODO:
  //your impl here
}
```

Notice #include.

There is no *YourSomeImpl.h.*

main.i

```c
typedef struct _SSomeStruct
{
  char m_chMember;
}SSomeStruct;

void SomeCall(SSomeStruct *psSS);

SSomeStruct g_sSS;

void main()
{
  SomeCall(&g_sSS);
}
```

MySomeImpl.i

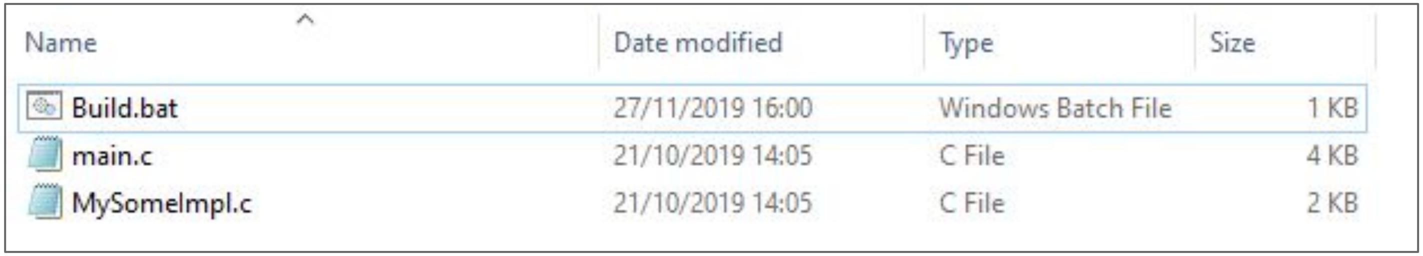
```c
typedef struct _SSomeStruct
{
  char m_chMember;
}SSomeStruct;

void SomeCall(SSomeStruct *psSS);

void SomeCall(SSomeStruct *psSS)
{
  //TODO:
  //my impl here
}
```

YourSomeImpl.i

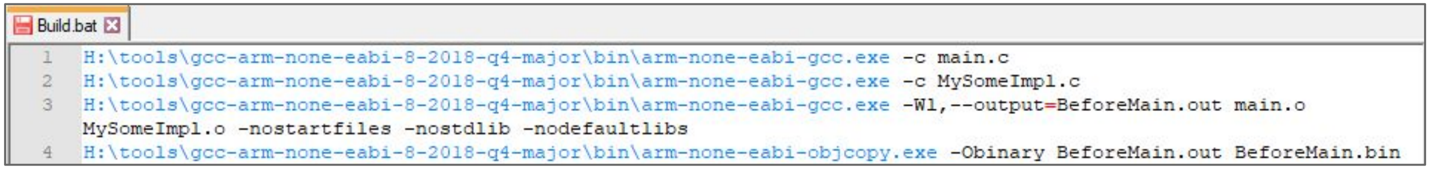
```c
typedef struct _SSomeStruct
{
  char m_chMember;
}SSomeStruct;

void SomeCall(SSomeStruct *psSS);

void SomeCall(SSomeStruct *psSS)
{
  //TODO:
  //your impl here
}
```

After preprocessor.

Lameman's Build manager



Name	Date modified	Type	Size
Build.bat	27/11/2019 16:00	Windows Batch File	1 KB
main.c	21/10/2019 14:05	C File	4 KB
MySomeImpl.c	21/10/2019 14:05	C File	2 KB



```
H:\tools\gcc-arm-none-eabi-8-2018-q4-major\bin\arm-none-eabi-gcc.exe -c main.c
H:\tools\gcc-arm-none-eabi-8-2018-q4-major\bin\arm-none-eabi-gcc.exe -c MySomeImpl.c
H:\tools\gcc-arm-none-eabi-8-2018-q4-major\bin\arm-none-eabi-gcc.exe -Wl,--output=BeforeMain.out main.o MySomeImpl.o -nostartfiles -nostdlib -nodefaultlibs
H:\tools\gcc-arm-none-eabi-8-2018-q4-major\bin\arm-none-eabi-objcopy.exe -Obinary BeforeMain.out BeforeMain.bin
```

Linker file memory layout

Start of memory layout description.

Can read (r) and execute from (x).

Starts at 0x00000000.

Is 128 kBytes in size.

Made up name.

Can read (r) and execute from (x), also write to (w).

```
MEMORY
{
ROM (rx)  : ORIGIN = 0x00000000, LENGTH = 128k
RAM (rwx) : ORIGIN = 0x10000000, LENGTH = 32k
}

_estack = ORIGIN(RAM) + LENGTH(RAM); /* end of RAM */

ENTRY(reset)
```

Made up symbol.

Symbol marked as start point.

Assigned value available as symbol address in source files.

Linker file sections layout

```
SECTIONS
{
  .text_out :                /* Code goes into ROM */
  {
       KEEP(*(.isr_vector))/* Startup code */
       *(.text*)             /* All text sections */
       _etext = .;           /* Code ends here */
  } >ROM

  .rodata_out :              /* Constant data goes into ROM */
  {
       . = ALIGN(4);
       *(.rodata*)           /* All read only sections (constants, strings etc.) */

       . = ALIGN(4);
       _sidata = .;          /* Values for initialized data go here */
  } >ROM

  .data_out :                /* Initialized data sections goes into RAM */
  {
   _sdata = .;               /* Initialized data starts here */
   *(.data*)                 /* All initialized data sections */
      . = ALIGN(4);
   _edata = .;               /* Initialized data ends here */
  } >RAM AT> ROM             /* Loaded in ROM */

  .bss_out :                 /* Uninitialized data section follows in RAM */
  {
     . = ALIGN(4);
   _sbss = .;                /* Uninitialized (zero initialized) data starts here */

  *(.bss*)                   /* All zero initialized data sections (e.g. int n=0;) */

  *(COMMON)                  /* All Uninitialized  data sections (e.g. int n;) */
     . = ALIGN(4);
  _ebss = .;                 /* Uninitialized (zero initialized) data ends here */
  } >RAM
}
```

Start of memory layout description.

Compiler output section name emitted by compiler.

Code example shows how to make custom sections.

Made up name for linker output section.

Current address.

Made up symbol.

Place input section here and do not move or remove for optimization.

Align current address, skip bytes if needed.

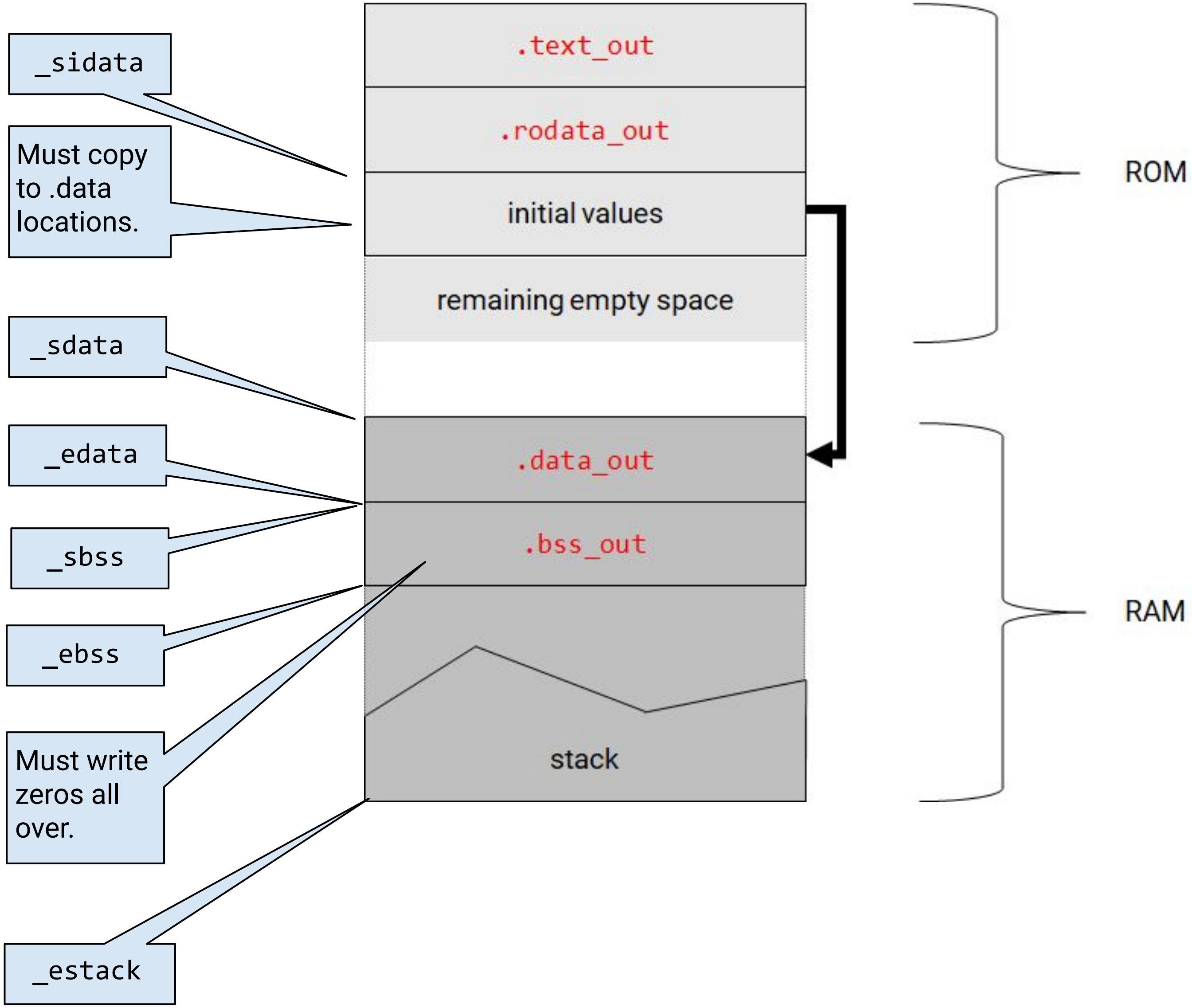
Initialized global variables section.

Accessed at addresses in RAM.

Uninitialized global variables section.

But initial values stored in ROM, at _sidata.

Final layout



Startup

Initialized variable.

Uninitialized variable.

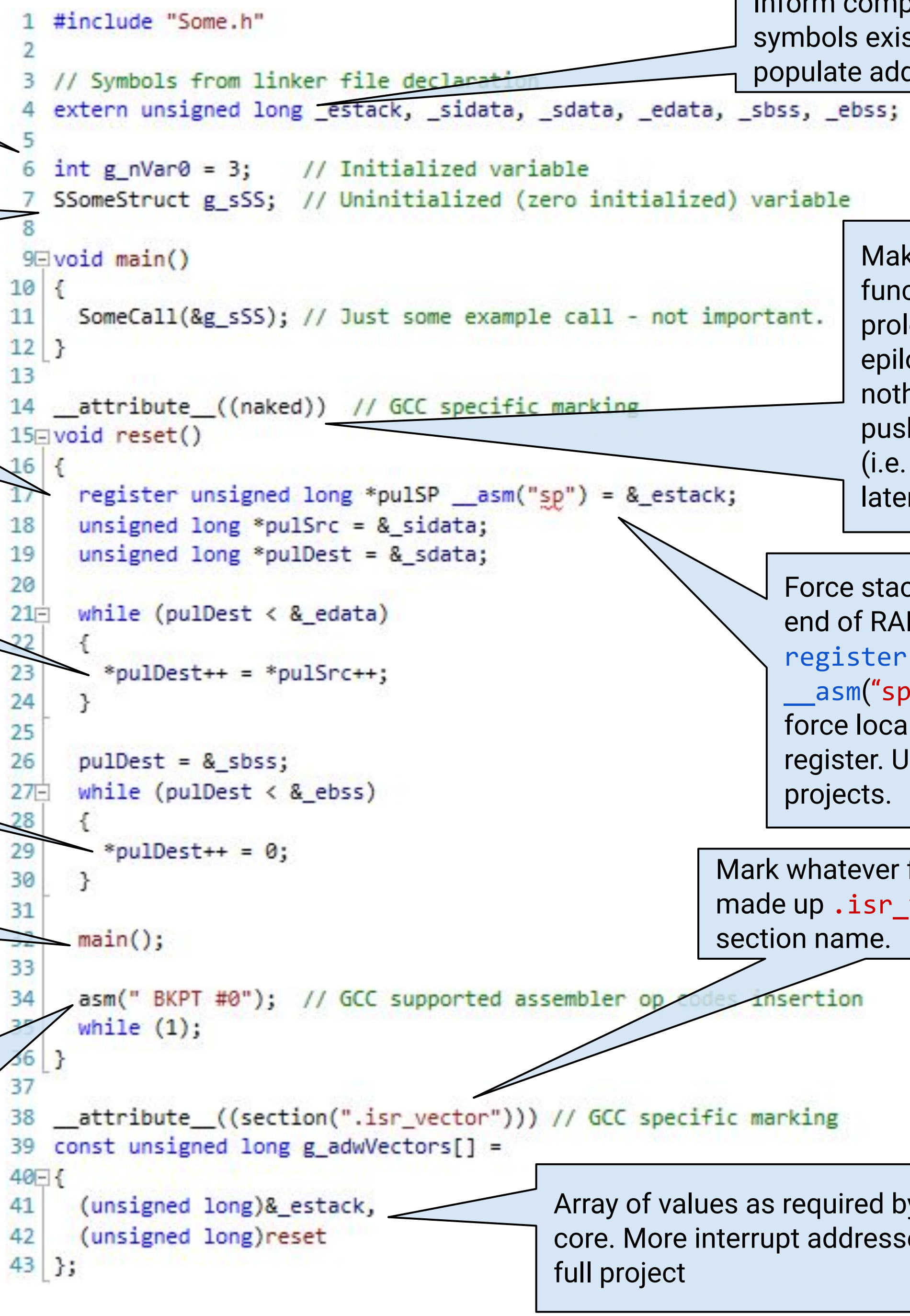
Local variables loaded with addresses.

Copy loop.

Zero init loop.

Finally call `main()`.

Normally main won't return. But just in case force software breakpoint instruction and endless loop.



```c
#include "Some.h"

// Symbols from linker file declaration
extern unsigned long _estack, _sidata, _sdata, _edata, _sbss, _ebss;

int g_nVar0 = 3;    // Initialized variable
SSomeStruct g_sSS;  // Uninitialized (zero initialized) variable

void main()
{
  SomeCall(&g_sSS); // Just some example call - not important.
}

__attribute__((naked))  // GCC specific marking
void reset()
{
  register unsigned long *pulSP __asm("sp") = &_estack;
  unsigned long *pulSrc = &_sidata;
  unsigned long *pulDest = &_sdata;

  while (pulDest < &_edata)
  {
    *pulDest++ = *pulSrc++;
  }

  pulDest = &_sbss;
  while (pulDest < &_ebss)
  {
    *pulDest++ = 0;
  }

  main();

  asm(" BKPT #0");  // GCC supported assembler op codes insertion
  while (1);
}

__attribute__((section(".isr_vector"))) // GCC specific marking
const unsigned long g_adwVectors[] =
{
  (unsigned long)&_estack,
  (unsigned long)reset
};
```

Inform compiler these symbols exist. Linker will populate addresses.

Make following function without prologue and epilogue. So nothing gets pushed to stack (i.e. saved) and later restored.

Force stack pointer to end of RAM. Notice `register` and `__asm("sp")` directive to force local variable to register. Usable in RAM projects.

Mark whatever follows with made up `.isr_vector` section name.

Array of values as required by Cortex-M core. More interrupt addresses follow in full project

Map file

```
Name             Origin             Length             Attributes
ROM              0x00000000         0x00020000         xr
RAM              0x10000000         0x00008000         xrw
*default*        0x00000000         0xffffffff

Linker script and memory map

                0x10008000                _estack = ((ORIGIN (RAM) + 0x8000))

.text_out       0x00000000       0x7c
 *(.isr_vector)
 .isr_vector    0x00000000        0x8 .\Debug\main.o
                0x00000000                g_adwVectors
 *(.text*)
 .text          0x00000008       0x60 .\Debug\main.o
                0x00000008                main
                0x00000018                reset
 .text          0x00000068       0x14 .\Debug\some.o
                0x00000068                SomeCall
                0x0000007c                _etext = .

.rodata_out     0x0000007c        0x0
                0x0000007c                . = ALIGN (0x4)
 *(.rodata*)
                0x0000007c                . = ALIGN (0x4)
                0x0000007c                _sidata = .

.data_out       0x10000000        0x4 load address 0x0000007c
                0x10000000                _sdata = .
 *(.data*)
 .data          0x10000000        0x4 .\Debug\main.o
                0x10000000                g_nVar0
 .data          0x10000004        0x0 .\Debug\some.o
                0x10000004                . = ALIGN (0x4)
                0x10000004                _edata = .

.bss_out        0x10000004        0x4 load address 0x00000080
                0x10000004                . = ALIGN (0x4)
                0x10000004                _sbss = .
 *(.bss*)
 .bss           0x10000004        0x0 .\Debug\main.o
 .bss           0x10000004        0x0 .\Debug\some.o
 *(COMMON)
 COMMON         0x10000004        0x1 .\Debug\main.o
                0x10000004                g_sSS
                0x10000008                . = ALIGN (0x4)
 *fill*         0x10000005        0x3
                0x10000008                _ebss = .
```

Homework

Cortex-R, Cortex-A and older ARM cores start by executing directly from address 0x00000000 instead of first reading the start address from address 0x00000004.

How would .isr_vector code look like for those?

TIP: BL is the assembler instruction for branch on ARM. But you don’t need it.