Fundamentals of Embedded and Real Time Systems

MODULE 07

TAMER AWAD

Review Module 06

Module 07

C Types
- stdint.h
- Mixing types
- typedef

Structures in C
- Syntax
- Access to struct members
- Layout
- Nested structures
- Pointers and structures
- Access in assembly

Cortex-M Software Interface Standard (CMSIS)
- What is CMSIS?
- CMSIS Standardization
- Organization of CMSIS
- How to use CMSIS?
- GPIO_TypeDef structure
- Demo: Blinking LED using CMSIS

Startup Code
- Before “main”
- Standard startup code
- Linker Map file
- Data initialization
- The reset sequence
- Vector Table
- Fault Handler

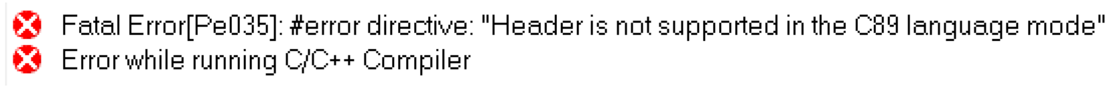
Assignment
- Assignment 06

C Types

- stdint.h
- #include with quotes vs brackets
- Mixing types
- typedef

<stdint.h>

- The C standard does not define the size of the basic data types.
  - *For example:* “int” can occupy 32-bit on one machine and 16-bit or 8-bit on another.
- The C standard specifies that the size of “short” must not be bigger than “int” and “int” must be not bigger than “long”.
- It is often important to know exactly the size and sign of the variables so that the code runs the same way regardless of the target processor architecture.
- The C99 standard specifies the header <**stdint.h**> which declares sets of integer types with standard names, specified widths and defines corresponding sets of macros (see section 7.18 in the C99 standard for details)
- Compiler vendors are responsible for providing the appropriate typedefs inside that standard library header file.
- The IAR compiler supports the C99 standard (in the project options settings).
- **Note:** If you switch your project to C89, you will get a compiler error including stdint.h



```
Fatal Error[Pe035]: #error directive: "Header is not supported in the C89 language mode"
Error while running C/C++ Compiler
```

typedef

- Each **typedef** statement defines a new type.
- Hint: Typedef definitions should be read from right to left (similar to pointers)
- EX> **typedef int uint32_t:**
  - Reads: “uint32_t” is typedef name for an int data type
- int* b;
  - Reads: “b” is a pointer to an “int” variable type.
- typedef vs #define:
  - https://www.geeksforgeeks.org/typedef-versus-define-c/

<stdint.h>

- The file uses typedefs to define the following fixed-width integer types (among other things):
  - **int8_t**: for signed 8-bit integer
  - **uint8_t**: for unsigned 8-bit integer
  - **int16_t**: for signed 16-bit integer
  - **uint16_t**: for unsigned 16-bit integer
  - **int32_t**: for signed 32-bit integer
  - **uint32_t**: for unsigned 32-bit integer
- The value of the **stdint** header file is in standardizing the type names and the fact that it is the responsibility of the compiler vendor to provide the right definitions for the CPU.
- It's done by means of macros, such as __INT32_T_TYPE__, which in turn are defined in terms of the built-in types.

Size & Range of Data Types in ARM

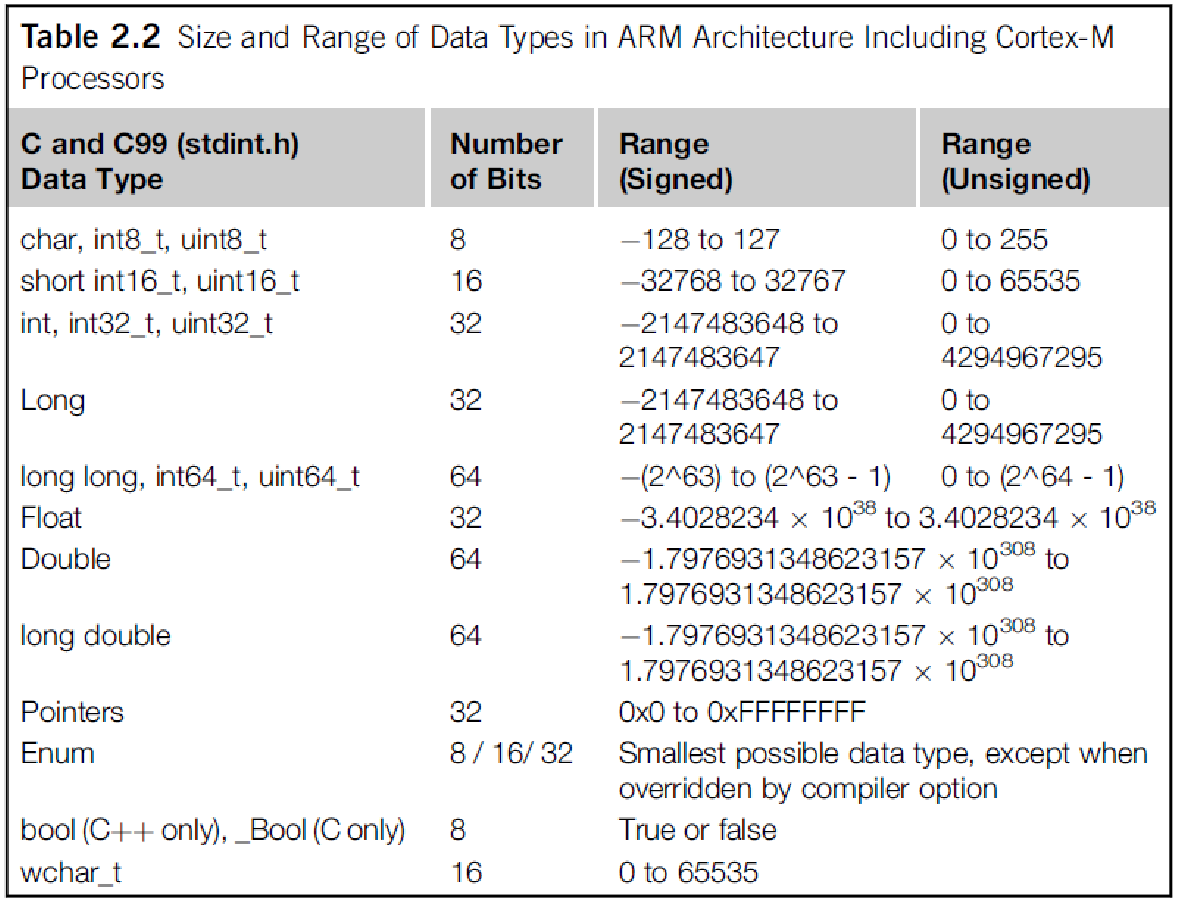


Table 2.2 Size and Range of Data Types in ARM Architecture Including Cortex-M Processors

C and C99 (stdint.h) Data Type	Number of Bits	Range (Signed)	Range (Unsigned)
char, int8_t, uint8_t	8	–128 to 127	0 to 255
short int16_t, uint16_t	16	–32768 to 32767	0 to 65535
int, int32_t, uint32_t	32	–2147483648 to 2147483647	0 to 4294967295
Long	32	–2147483648 to 2147483647	0 to 4294967295
long long, int64_t, uint64_t	64	–(2^63) to (2^63 - 1)	0 to (2^64 - 1)
Float	32	$-3.4028234 \times 10^{38}$ to 3.4028234×10^{38}	
Double	64	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
long double	64	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
Pointers	32	0x0 to 0xFFFFFFFF	
Enum	8 / 16/ 32	Smallest possible data type, except when overridden by compiler option	
bool (C++ only), _Bool (C only)	8	True or false	
wchar_t	16	0 to 65535	

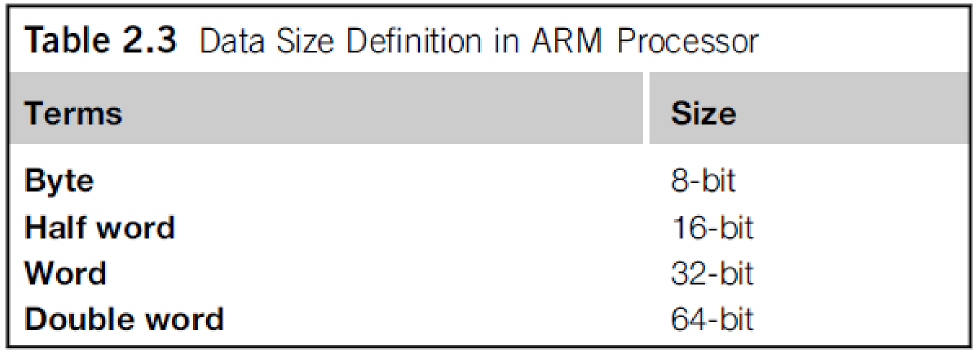


Table 2.3 Data Size Definition in ARM Processor

Terms	Size
Byte	8-bit
Half word	16-bit
Word	32-bit
Double word	64-bit

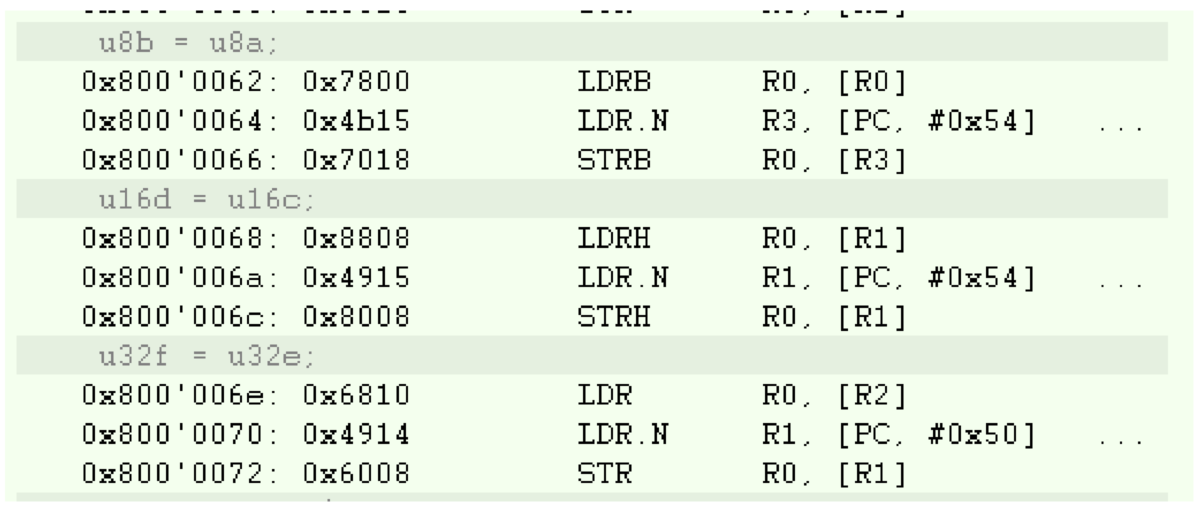
#include with "quotes" vs <brackets>

- C99 Spec section 6.10.2:
  - A preprocessing directive of the form **# include <***h-char-sequence***>**
    - Searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the **<** and **>** delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.
  - A preprocessing directive of the form **# include "***q-char-sequence***“**
    - Causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the **"** delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read **# include <***h-char-sequence***>** *new-line*

- In summary: It's compiler dependent, but in general:
  - Using *"quotes"* prioritizes headers in the current working directory over system headers.
  - Using *<brackets>* is often used for system headers

Assignment of different types

```
uint8_t u8a, u8b;
uint16_t u16c, u16d;
uint32_t u32e, u32f;

u8a = 0xa1u;
u16c = 0xc1c2u;
u32e = 0xe0e1e2e3;

u8b = u8a;
u16d = u16c;
u32f = u32e;
```



```
 u8b = u8a;
0x800'0062: 0x7800          LDRB      R0, [R0]
0x800'0064: 0x4b15          LDR.N     R3, [PC, #0x54]    ...
0x800'0066: 0x7018          STRB      R0, [R3]
 u16d = u16c;
0x800'0068: 0x8808          LDRH      R0, [R1]
0x800'006a: 0x4915          LDR.N     R1, [PC, #0x54]    ...
0x800'006c: 0x8008          STRH      R0, [R1]
 u32f = u32e;
0x800'006e: 0x6810          LDR       R0, [R2]
0x800'0070: 0x4914          LDR.N     R1, [PC, #0x50]    ...
0x800'0072: 0x6008          STR       R0, [R1]
```

Mixing types and implicit conversion

- ***C always automatically promotes smaller-size integers to the built-in 'int' or 'unsigned int' type before performing computations.***
- The precision in which the computation is performed does not depend on the left-hand side of the assignment.
- Enforce promotion at least of one of the operands.
- The computation is performed at the largest precision of the involved operands.
- When mixing signed and unsigned operands, both are promoted to 'unsigned int’ and the result is 'unsigned int'.

Demo: stdint.h & Mixing Types

- Include “stdint.h”
- Failure to compile with C89
- Open stdint.h and show location as “system” file.
- Look inside stdint.h
- sizeof(<stdint_types>)
- Mixing types issues

Mixing types: Example 1 - Issue

```
uint32_t u32e, u32f;

uint64_t u64z;


u32e = 4000000000u;

u32f = 3000000000u;


u64z = u32e + u32f;
```

- On a 64-bit machine, the promotion will be to 64-bit, because this is the size of 'int' on that machine.
- However, on 32-bit machine, no promotion happens, because the type 'int’ is only 32-bit wide.
  - 4000,000,000 + 7000,000,000 will overflow
  - The result of the computation is eventually assigned to a 64-bit wide number, which has enough range to represent 7000,000,000.
- This example shows that *the precision in which the computation is performed does not depend on the left-hand side of the assignment*.

How do you fix this?

Mixing types: Example 1 - Solution

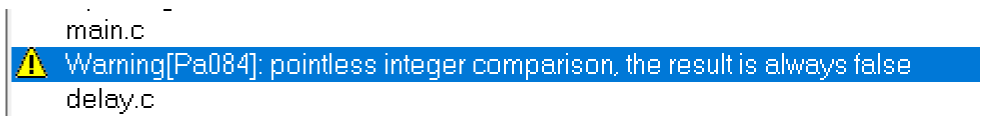
```
uint32_t u32e, u32f;

uint64_t u64z;



u32e = 4000000000u;

u32f = 3000000000u;



u64z = u32e + u32f;                          //Not Portable

u64z = (uint64_t)u32e + u32f;                // Portable

u64z = (uint64_t)u32e + (uint64_t)u32f;      // Portable
```

- Enforce promotion to a 64-bit precision of at least one of the operands.
- According to the implicit conversion rule of the C language: ***The computation is performed at the largest precision of the involved operands*.**
- If one of the operands is 64-bit wide, the other will be promoted to 64-bits and the whole computation will be performed at 64-bit.

Mixing types: Example 2 - Issue

```
uint32_t u32e = 1000;
if (u32e > -1)
 {
      u8a = 1u;
 }
 else
 {
      u8a = 0u;
 }
```

Remember that in a mixed sign operation, the C standard will promote the signed operand to unsigned int.



```
main.c
Warning[Pa084]: pointless integer comparison, the result is always false
delay.c
```

How do you fix this?

Mixing types: Example 2 - Solution

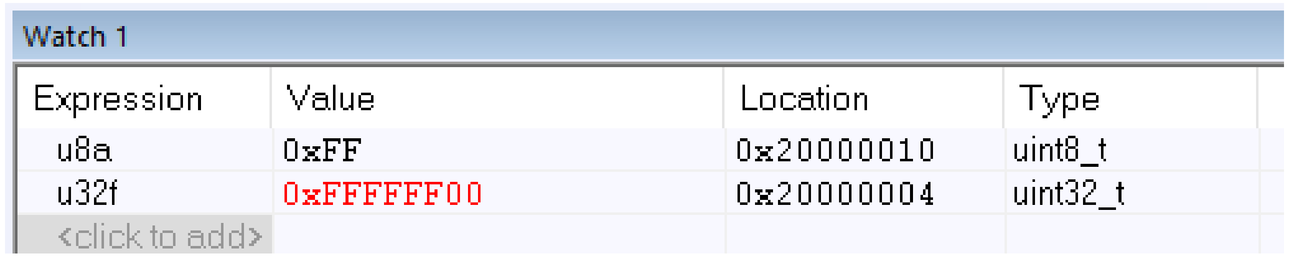
```
uint32_t u32e = 1000;
if ((int32_t)u32e > -1)
{
     u8a = 1u;
}
else
{
     u8a = 0u;
}
```

Mixing types: Example 3 - Issue

```
uint8_t u8a;

uint32_t u32f;

u8a = 0xffu;

if (~u8a == 0x00u)

{

    u8b = 1u;

}

u32f = ~u8a;
```

What's the problem?

u8a will be promoted to int (32-bit), so the most significant bytes will be all 0, and the inversion will make them all ones.

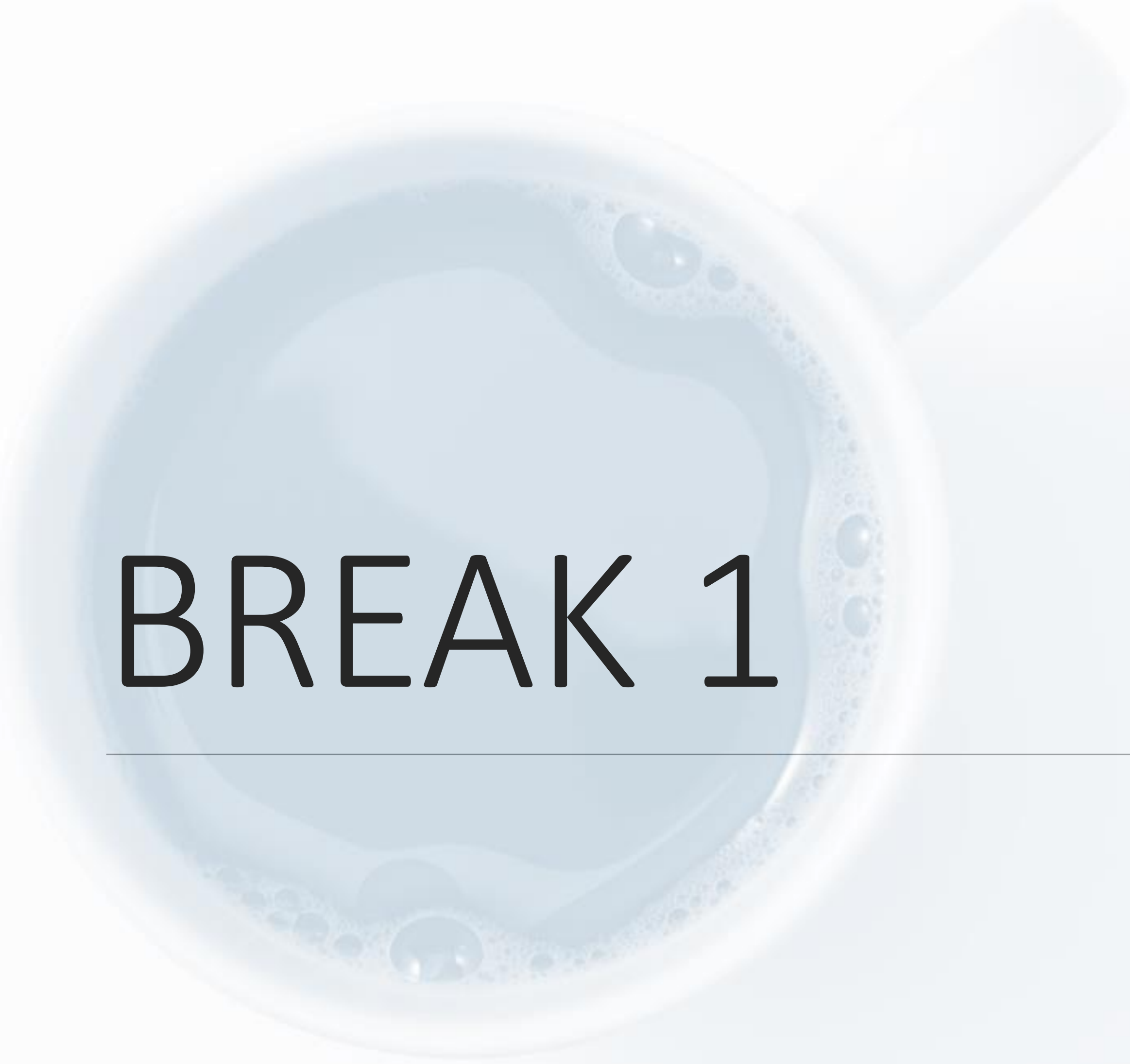


Watch 1

Expression	Value	Location	Type
u8a	0xFF	0x20000010	uint8_t
u32f	0xFFFFFF00	0x20000004	uint32_t
<click to add>			

Mixing types: Example 3 - Solution

```
uint8_t u8a;
uint32_t u32f;
u8a = 0xffu;
if ((uint8_t)(~u8a) == 0x00u)
{
    u8b = 1u;
}
u32f = ~u8a;
```



BREAK 1

C - Structures

- Syntax
- Access to struct members
- Layout
- Nested structures
- Pointers and structures
- Access in assembly

Structures in C

- Structures in C offer a way to group together variables, possibly of different types.
- The benefit of structures is that they permit a group of related variables to be treated as a unit instead of separate entities.
- In embedded systems, structures also permit you to access hardware in an elegant and intuitive way (CMSIS)
- <u>According to C99 standard:</u>

A structure type describes a sequentially allocated nonempty set of member objects

each of which has an optionally specified name and possibly distinct type.

Syntax 1: struct definition & declaration with tags

```
struct <optional_tag> {
   type1 member_1;
   type2 member_2;
 } <optional_declaration>;
```

```
struct  Point {
   uint16_t x;
   uint8_t y;
 } p1, p2;
```

Syntax 2: struct definition & declaration without tags

```
struct <optional_tag> {
   type1 member_1;
   type2 member_2;
 } <optional_declaration>;
```

```
struct {
   uint16_t x;
   uint8_t y;
 } p1, p2;
```

Syntax 3: struct declaration after definition

```
struct <optional_tag> {
    type1 member_1;
    type2 member_2;
 } <optional_declaration>;
```

```
struct Point {
   uint16_t x;
   uint8_t y;
 };

struct  Point p1, p2;
```

NOTE:
Must repeat the “struct” keyword in front of the tag. Unlike C++ where “struct” and “class” are not needed before each declaration.

Syntax 4: typedef after struct definition

```
struct <optional_tag> {
    type1 member_1;
    type2 member_2;
 } <optional_declaration>;
```

```
struct  Point {
   uint16_t x;
   uint8_t y;
 };
                        Tag Name     Typedef Name
 typedef struct Point Point;


 Point p1, p2;
```

Note:
Tag names in C occupy a different namespace than typedef names, variable names, and function names; hence, can have Tag "Point" followed by variable name "Point" as shown above.

Syntax 5: typedef before struct definition

```
struct <optional_tag> {
    type1 member_1;
    type2 member_2;
 } <optional_declaration>;
```

```
typedef struct Point Point;

struct  {
   uint16_t x;
   uint8_t y;
 };

 Point p1, p2;
```

Note:
Can place the typedef before the struct.

Syntax 6: Untagged struct inside typedef

```
typedef struct {
   type1 member_1;
   type2 member_2;
 } TypeDefName;
```

```
typedef struct {
   uint16_t x;
   uint8_t y;
 } Point;

 Point p1, p2;
```

Note:
Struct tag names are almost never needed. The exception being the self-referential structures, such as nodes of linked lists or trees.

Member Access

```
typedef struct {
   uint16_t x;
   uint8_t y;
 } Point;
```

```
Point p1, p2;

p1.x = sizeof(Point);

P1.y = 42;
```

Question:

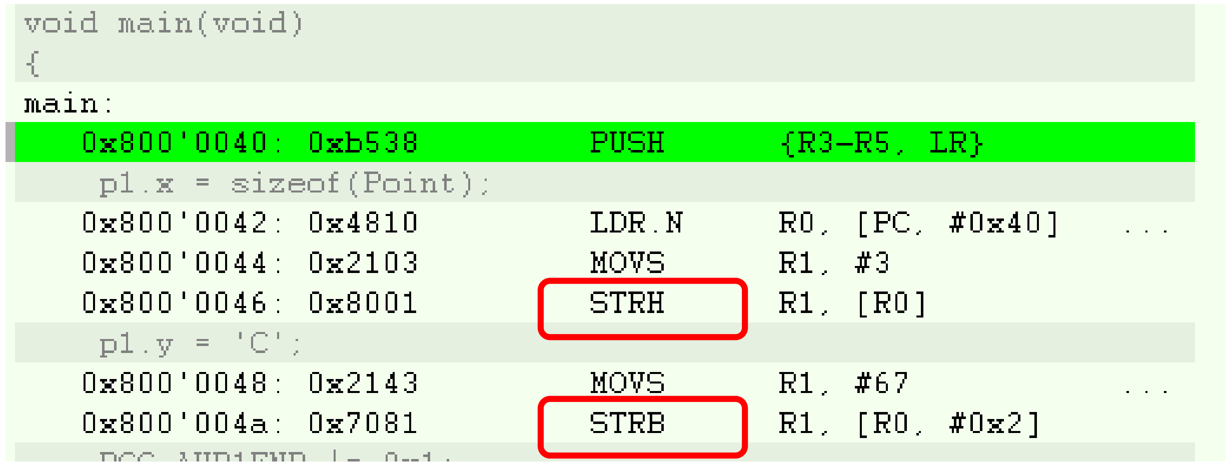
What is the value of p1.x?

Demo - Structures

1. Definition and declaration.
2. Show memory layout, addressing, and assembly:
    1. typedef struct {uint16_t x; uint8_t y;} Point;
    2. typedef struct {uint8_t y; uint16_t x;} Point;
3. Use __packed extended-keyword
4. Switch to Cortex-M0 and show layout, addressing, and assembly.

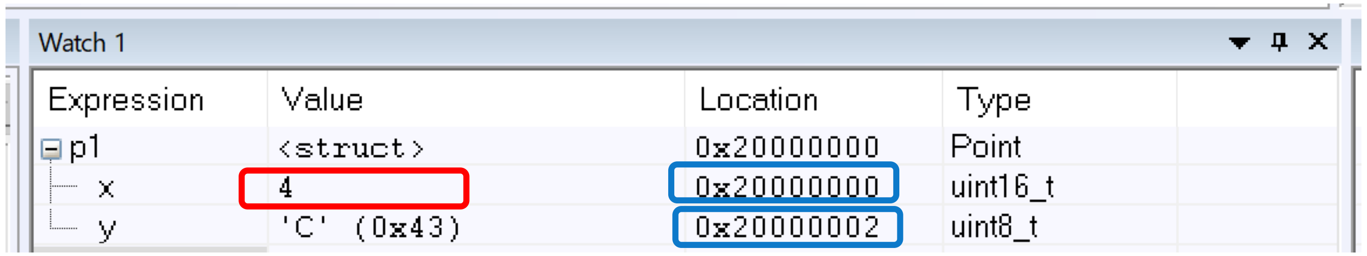
Layout: Size & Padding

```
typedef struct {
    uint16_t x;
    uint8_t y;
 } Point;


 Point p1, p2;
 p1.x = sizeof(Point);
 P1.y = ‘C’;
```

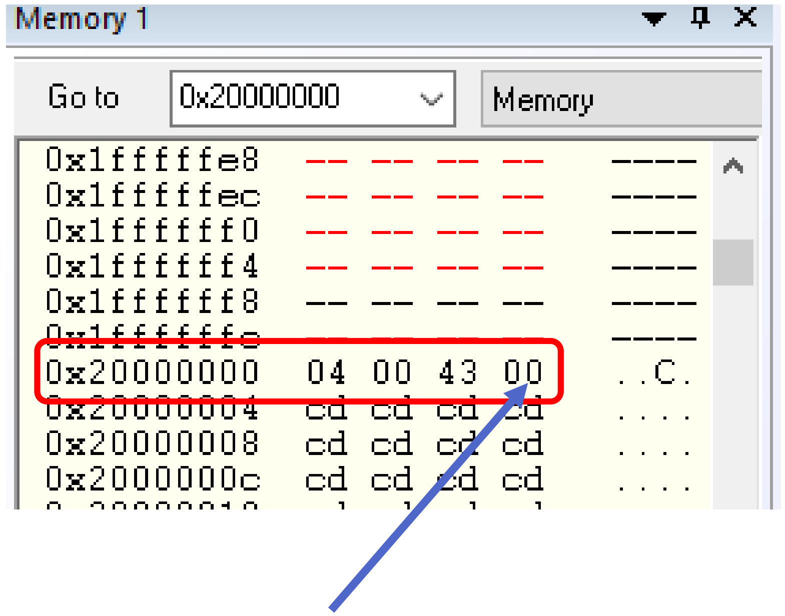


```
void main(void)
{
main:
    0x800'0040: 0xb538        PUSH      {R3-R5, LR}
     p1.x = sizeof(Point);
    0x800'0042: 0x4810        LDR.N     R0, [PC, #0x40]    ...
    0x800'0044: 0x2103        MOVS      R1, #3
    0x800'0046: 0x8001        STRH      R1, [R0]
     p1.y = 'C';
    0x800'0048: 0x2143        MOVS      R1, #67            ...
    0x800'004a: 0x7081        STRB      R1, [R0, #0x2]
```

Layout: Size & Padding



Expression	Value	Location	Type
p1	<struct>	0x20000000	Point
x	4	0x20000000	uint16_t
y	'C' (0x43)	0x20000002	uint8_t

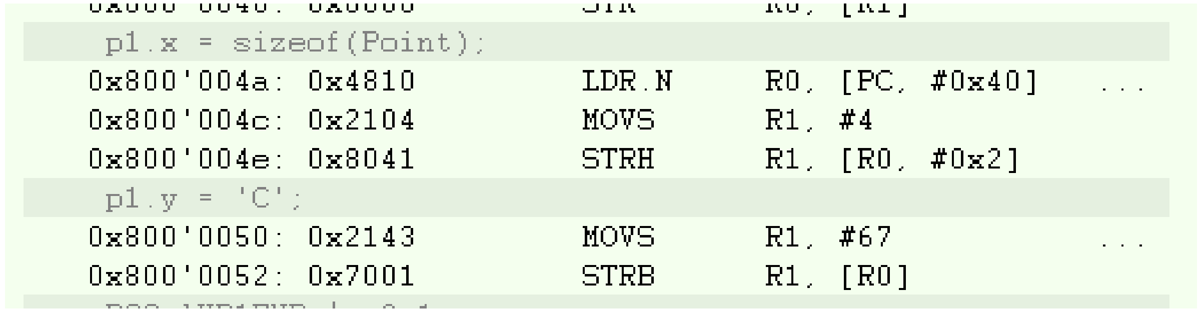


```
0x1fffffe8  -- -- -- --   ----
0x1fffffec  -- -- -- --   ----
0x1ffffff0  -- -- -- --   ----
0x1ffffff4  -- -- -- --   ----
0x1ffffff8  -- -- -- --   ----
0x1ffffffc  -- -- -- --   ----
0x20000000  04 00 43 00   ..C.
0x20000004  cd cd cd cd   ....
0x20000008  cd cd cd cd   ....
0x2000000c  cd cd cd cd   ....
```

The compiler padded the structure by one byte to avoid “odd” addresses

Layout: Changing order of struct members

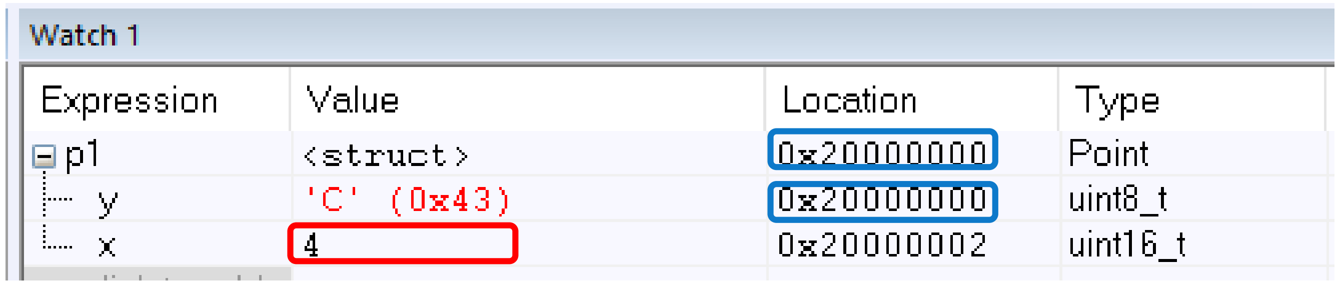
```
typedef struct {
   uint8_t y;
  uint16_t x;
 } Point;
```



```
 p1.x = sizeof(Point);
0x800'004a: 0x4810          LDR.N     R0, [PC, #0x40]     ...
0x800'004c: 0x2104          MOVS      R1, #4
0x800'004e: 0x8041          STRH      R1, [R0, #0x2]
 p1.y = 'C';
0x800'0050: 0x2143          MOVS      R1, #67             ...
0x800'0052: 0x7001          STRB      R1, [R0]
```

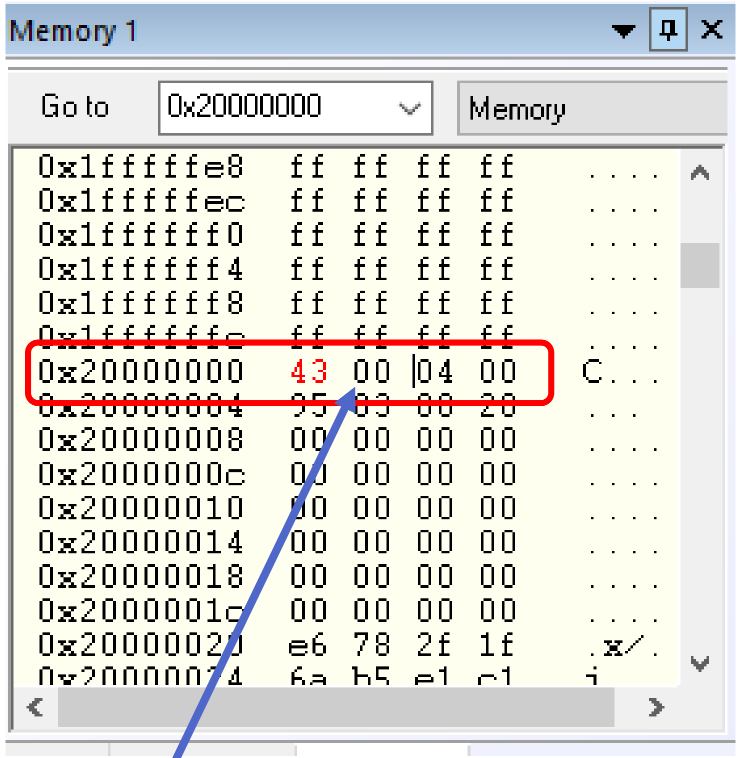
```
 Point p1, p2;
 p1.x = sizeof(Point);
 P1.y = ‘C’;
```

Layout: Changing order of struct members



Watch 1

Expression	Value	Location	Type
p1	<struct>	0x20000000	Point
y	'C' (0x43)	0x20000000	uint8_t
x	4	0x20000002	uint16_t



Memory 1

Go to 0x20000000 Memory

```
0x1fffffe8  ff ff ff ff  ....
0x1fffffec  ff ff ff ff  ....
0x1ffffff0  ff ff ff ff  ....
0x1ffffff4  ff ff ff ff  ....
0x1ffffff8  ff ff ff ff  ....
0x1ffffffc  ff ff ff ff  ....
0x20000000  43 00 04 00  C...
0x20000004  95 03 00 20  ....
0x20000008  00 00 00 00  ....
0x2000000c  00 00 00 00  ....
0x20000010  00 00 00 00  ....
0x20000014  00 00 00 00  ....
0x20000018  00 00 00 00  ....
0x2000001c  00 00 00 00  ....
0x20000020  e6 78 2f 1f  .x/.
0x20000024  6a b5 e1 c1  j
```

The compiler padded the structure by one byte

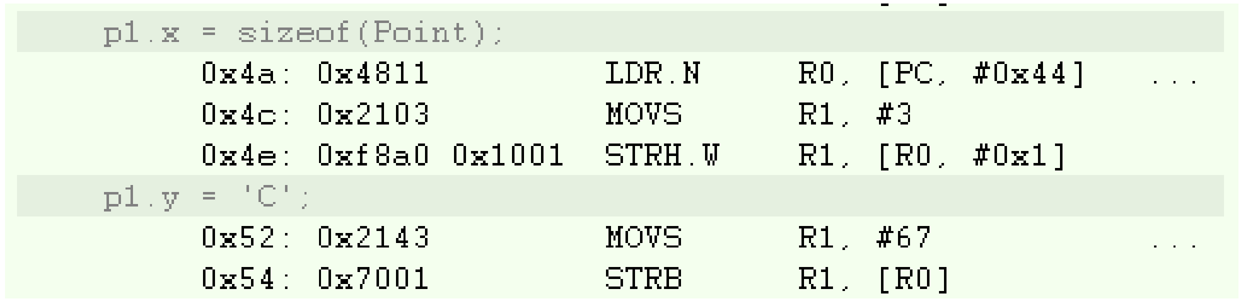
Layout: Using “__*packed*”

```
typedef __packed struct {
    uint16_t x;
    uint8_t y;
 } Point;

 Point p1, p2;
 p1.x = sizeof(Point);
 P1.y = 42;
```

- **Not in C standard**
- **Most embedded compilers provide some non-standard extension to pack the structure members.**
- **The IAR compiler provides the keyword “__packed”, which is placed in front of the struct keyword.**
- **Question:**
  - **So what is the value of p1.x?**

Layout: Odd addressing

typedef __*packed* struct {
 uint8_t y;
 uint16_t x;
} Point;

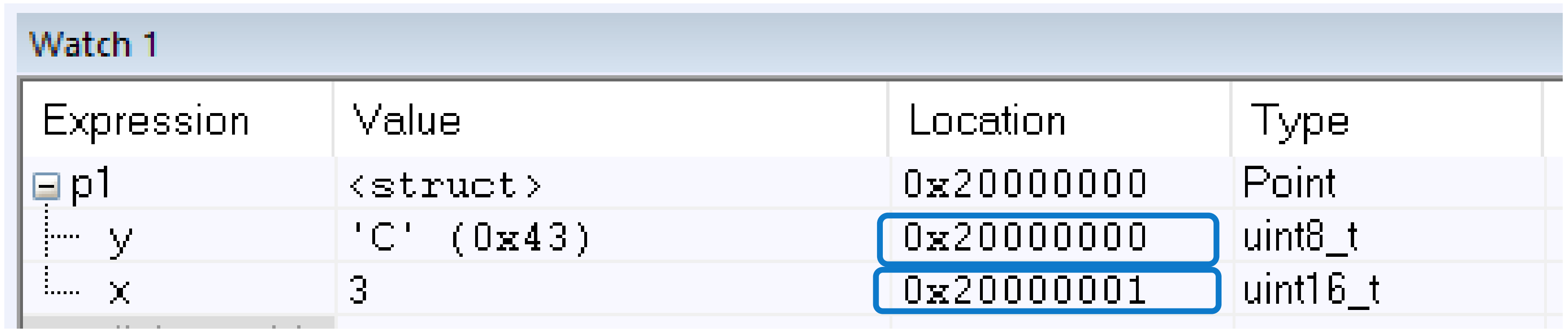


```
    p1.x = sizeof(Point);
        0x4a: 0x4811         LDR.N     R0, [PC, #0x44]    ...
        0x4c: 0x2103         MOVS      R1, #3
        0x4e: 0xf8a0 0x1001  STRH.W    R1, [R0, #0x1]
    p1.y = 'C';
        0x52: 0x2143         MOVS      R1, #67            ...
        0x54: 0x7001         STRB      R1, [R0]
```

Point p1, p2;

p1.x = sizeof(Point);

P1.y = 0x43;

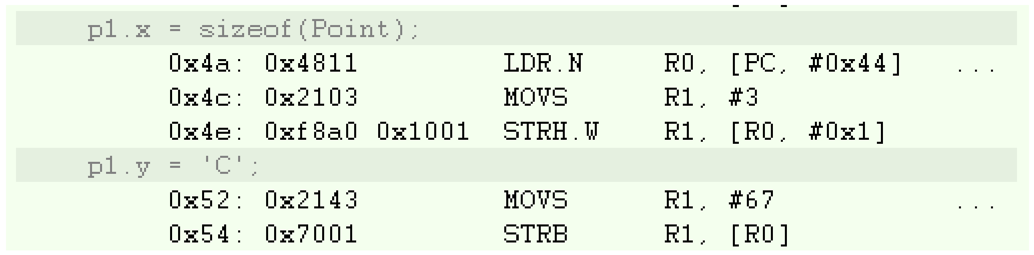


Watch 1

Expression	Value	Location	Type
⊟ p1	<struct>	0x20000000	Point
y	'C' (0x43)	0x20000000	uint8_t
x	3	0x20000001	uint16_t

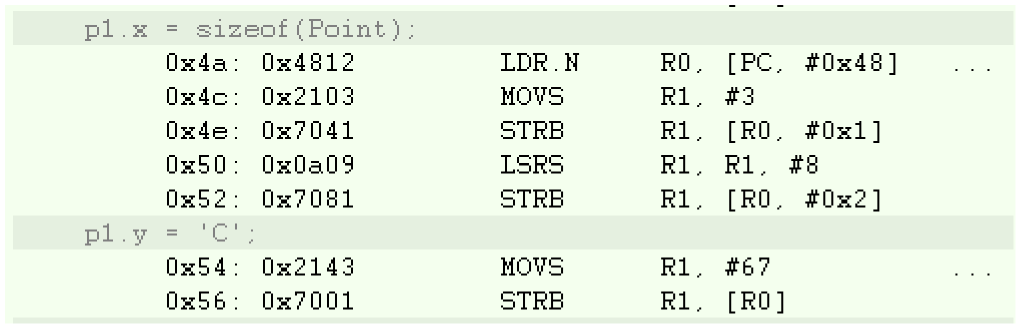
Layout: Misalignment- Cortex M4 vs M0

CORTEX M4



```
   p1.x = sizeof(Point);
        0x4a: 0x4811           LDR.N      R0, [PC, #0x44]    ...
        0x4c: 0x2103           MOVS       R1, #3
        0x4e: 0xf8a0 0x1001    STRH.W     R1, [R0, #0x1]
   p1.y = 'C';
        0x52: 0x2143           MOVS       R1, #67            ...
        0x54: 0x7001           STRB       R1, [R0]
```

CORTEX M0



```
   p1.x = sizeof(Point);
        0x4a: 0x4812           LDR.N      R0, [PC, #0x48]    ...
        0x4c: 0x2103           MOVS       R1, #3
        0x4e: 0x7041           STRB       R1, [R0, #0x1]
        0x50: 0x0a09           LSRS       R1, R1, #8
        0x52: 0x7081           STRB       R1, [R0, #0x2]
   p1.y = 'C';
        0x54: 0x2143           MOVS       R1, #67            ...
        0x56: 0x7001           STRB       R1, [R0]
```

Why Padding?

- The code for the assignment of p1.x is bigger on Cortex M0 than Cortex M4.
- The compiled code for Cortex-M0 consists of two STRB instructions plus a logical-shift-right instruction; whereas Cortex-M4 achieved the same effect with just one STRH instruction.
- While Cortex-M0 has the STRH instruction; unlike on Cortex-M4, it cannot access a half-word allocated at an odd address.
- **The compiler prefers to keep the data aligned instead of wasting the CPU cycles to access the mis-aligned data.**
- Note: Packed structures might not be always be as efficient to access as an un-packed structures.

Nested structures

```
typedef struct {
    uint16_t x;
    uint8_t y;
} Point;

typedef struct {
    Point bottom_left;
    Point top_right;
} Rectangle;

typedef struct {
    Point corners[3];
} Triangle;
```

```
Rectangle s1, s2;
Triangle t1, t2;

s1.bottom_left.x = 1;
s1.bottom_left.y = 1;
s1.top_right.x = 5;
s1.top_right.y = 5;

t1.corners[0].x = 1;
t1.corners[0].y = 1;

t1.corners[1].x = 3;
t1.corners[1].y = 4;

t1.corners[2].x = 5;
t1.corners[2].y = 1;
```

Pointers to structures

Complex structures can occupy considerable memory.

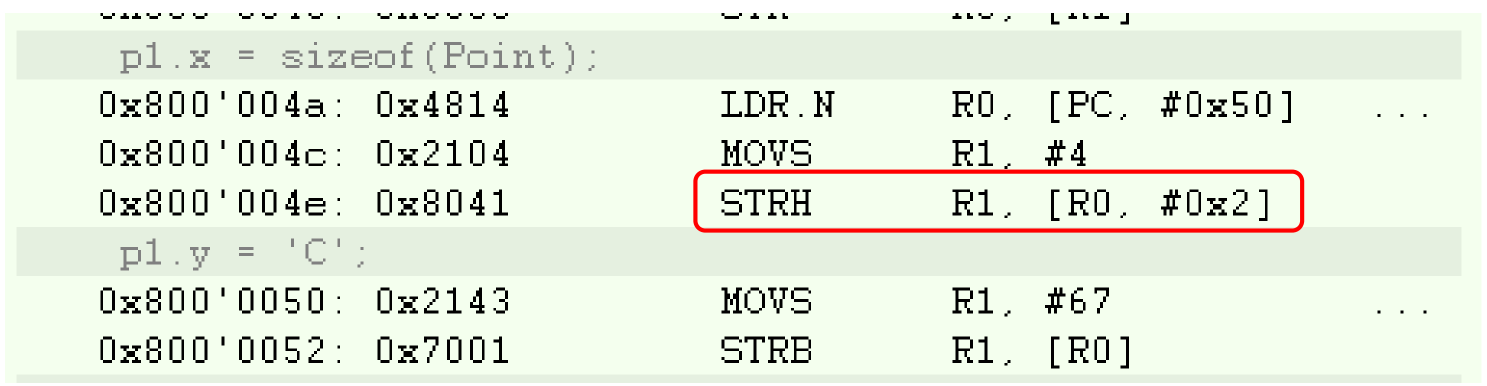
So structure assignment can mean copying a large size of memory from one variable to another.

It is more efficient to use “pointers” to structures and avoid copying structures where ever possible.

```
typedef struct {
    uint16_t x;
    uint8_t y;
 } Point;

 Point p1;
 p1.x = sizeof(Point);
 P1.y = 42;

 Point *p3;
 p3 = &p1;
 p3->x = p1.x;
 (*p3).y = 42;
```

Access in assembly

The compiler accesses the structure by using the offset addressing from the beginning of the structure.



```
 p1.x = sizeof(Point);
0x800'004a: 0x4814        LDR.N      R0, [PC, #0x50]    ...
0x800'004c: 0x2104        MOVS       R1, #4
0x800'004e: 0x8041        STRH       R1, [R0, #0x2]
 p1.y = 'C';
0x800'0050: 0x2143        MOVS       R1, #67            ...
0x800'0052: 0x7001        STRB       R1, [R0]
```

Demo - Structures

1. Nested Structures.
2. Pointers to Structures.
3. View data member access in disassembly.

CMSIS

- What is CMSIS
- CMSIS Standardization
- Organization of CMSIS-Core
- How to use CMSIS?
- GPIO_TypeDef structure
- Demo: Blinking LED using CMSIS

Cortex Microcontroller Software Interface Standard (CMSIS)

- With a significant amount of hardware components being identical, a large portion of the Hardware Abstraction Layer (HAL) can also be identical.
- However, reality has shown that lacking a common standard we find a variety of HAL/driver libraries for different devices that do the same thing.
- ARM has recognized that there is a need to create a standard to access these hardware components and put effort into a standard.
- The result of that effort is CMSIS.

Cortex Microcontroller Software Interface Standard (CMSIS)

- The CMSIS is a vendor-independent hardware abstraction layer for microcontrollers that are based on Arm Cortex processors.
- CMSIS is a framework that is implemented by vendors.
  - It provides a common API (Application Programming Interface) for core specific components.
  - And defines convention on how the device specific portions should be implemented.
- In general, most microcontroller vendors provide C header files and driver libraries for their microcontrollers.
- In most cases, these files are developed with the Cortex Microcontroller Software Interface Standard (CMSIS).
- ***CMSIS enables the use of structures to access hardware in Cortex-M microcontrollers.***

Benefits of CMSIS

- Portability between Cortex-M microcontroller-based devices.
  - Peripheral setup and access code will need to be modified, but processor core access functions are based on the same CMSIS source code and do not require changes.
- Reduces the learning curve for microcontroller developers
- Improves time to market.
- Tested by many silicon vendors and software developers

CMSIS Standardization

The CMSIS-Core standardizes a number of areas:

- Standard definitions for the processor's peripherals.
- Standard functions for accessing special instructions easily.
- Standard function names for system exception handlers
- Standard functions for system initialization
- Standard software variables for clock speed information

Organization of CMSIS

- The CMSIS files are integrated into device-driver library packages from microcontroller vendors.
- Some are prepared by ARM and are common to various microcontroller vendors (ex: core_cm4.h)
- Other files are vendor/device specific ("stm32f401xe.h" & "system_stm32f4xx.h")
- The aim of CMSIS is to provide a common starting point, and the microcontroller vendors can add additional functions if they prefer.
- Software using these added functions will need porting if the software design is to be reused on another microcontroller product.

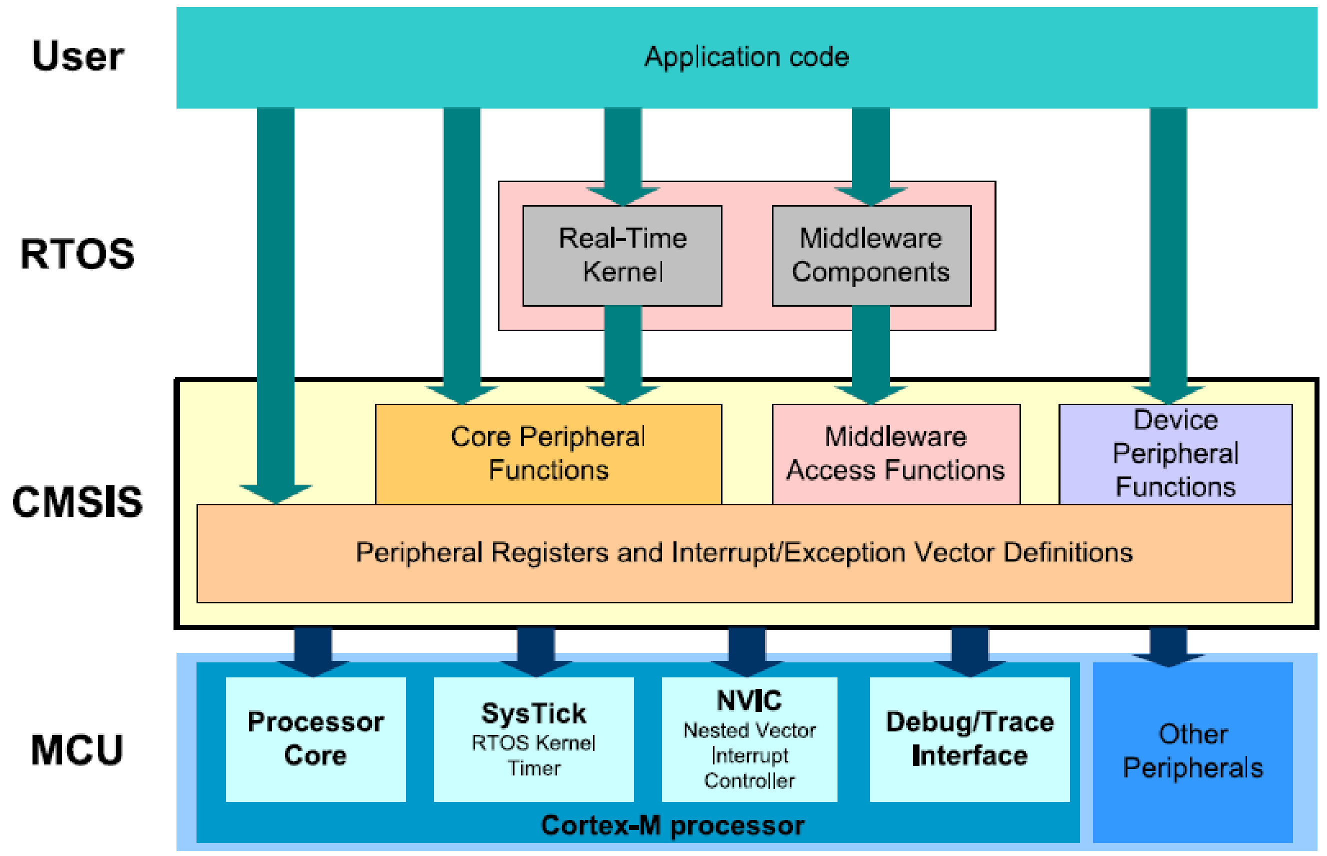

User
Application code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Registers and Interrupt/Exception Vector Definitions
MCU
Processor Core
SysTick
RTOS Kernel Timer
NVIC
Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals
Cortex-M processor

FIGURE 2.13

CMSIS-Core structure

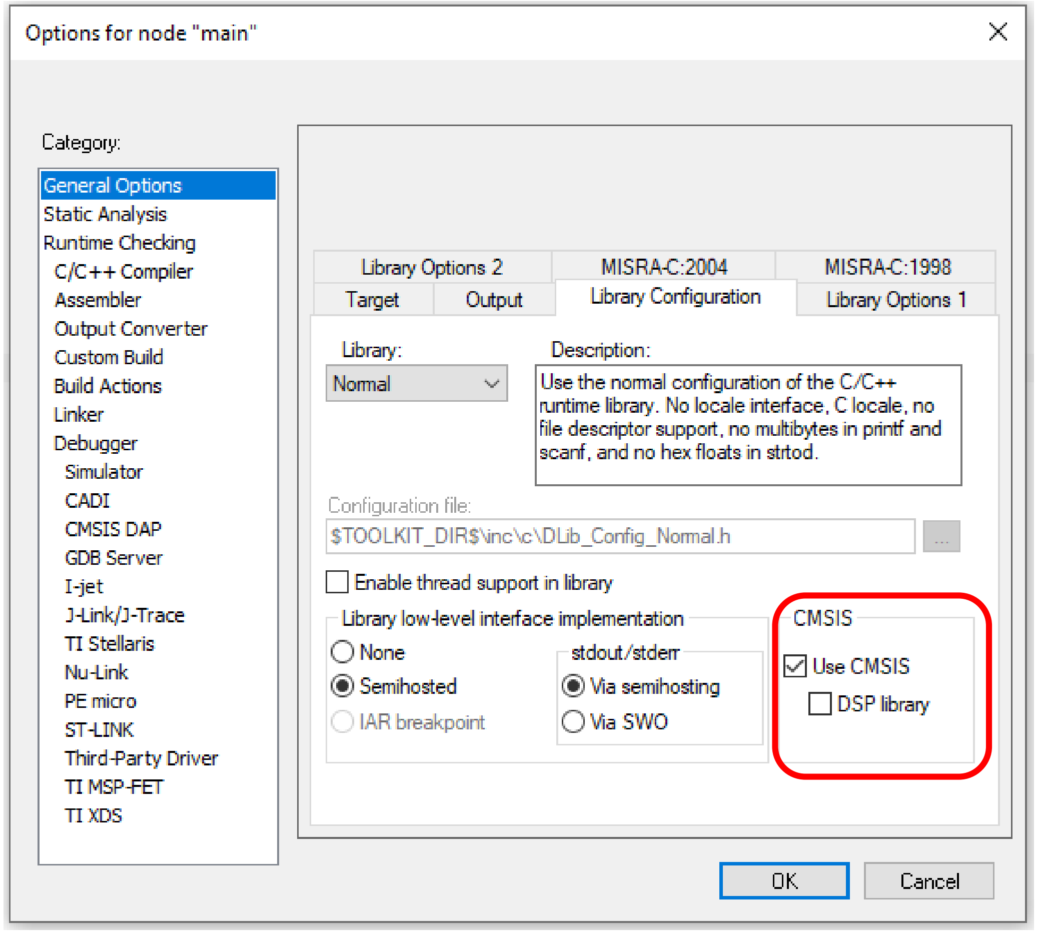
Core Structure

<u>Source:</u> "The Definitive guide to ARM Cortex-M3 & M4 Processors", by Joseph Yiu

How to use CMSIS - Setup

- Get the <device.h> header file for your board **"stm32f401xe.h"**
  - Contains peripheral registers definitions and interrupt assignment definitions.
- Get the “system_<device>.h” header file for your board **"system_stm32f4xx.h"**
  - Contains functions in device initialization code
- Include the device-specific header file in your application code, which will automatically include additional header files
- Therefore, you will need to set up the project search path for the header files in order to compile the project correctly.
- IAR provides the ability to do that with one click which provides a pointer to the path of the CMSIS header files that came with the IDE.
- One of those files is the **core_cm4.h** header file
  - This is part of the CMSIS industry standard and is a generic file for all microcontroller vendors.
  - IAR distributes it as an integral part of the toolset.

Enable CMSIS usage in Project Options


Options for node "main"

Category:

- General Options
- Static Analysis
- Runtime Checking
  - C/C++ Compiler
  - Assembler
  - Output Converter
  - Custom Build
  - Build Actions
  - Linker
  - Debugger
    - Simulator
    - CADI
    - CMSIS DAP
    - GDB Server
    - I-jet
    - J-Link/J-Trace
    - TI Stellaris
    - Nu-Link
    - PE micro
    - ST-LINK
    - Third-Party Driver
    - TI MSP-FET
    - TI XDS

Library Options 2 | MISRA-C:2004 | MISRA-C:1998
Target | Output | Library Configuration | Library Options 1

Library: Normal

Description:
Use the normal configuration of the C/C++ runtime library. No locale interface, C locale, no file descriptor support, no multibytes in printf and scanf, and no hex floats in strtod.

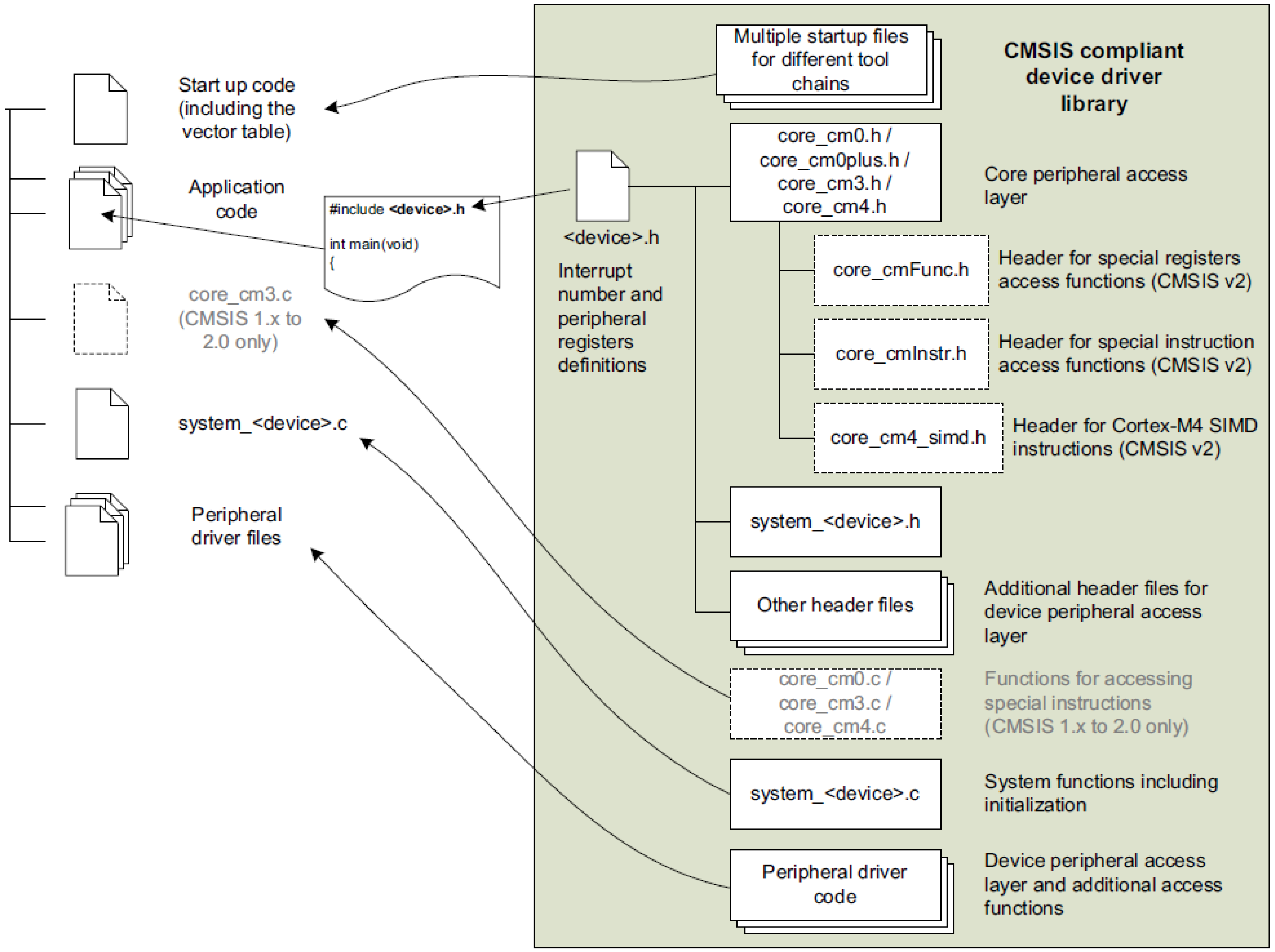
Configuration file:
$TOOLKIT_DIR$\inc\c\DLib_Config_Normal.h

☐ Enable thread support in library

Library low-level interface implementation
- ○ None
- ● Semihosted
- ○ IAR breakpoint

stdout/stderr
- ● Via semihosting
- ○ Via SWO

CMSIS
- ☑ Use CMSIS
- ☐ DSP library


Start up code (including the vector table)
Application code
core_cm3.c (CMSIS 1.x to 2.0 only)
system_<device>.c
Peripheral driver files
#include <device>.h
int main(void)
{
<device>.h
Interrupt number and peripheral registers definitions
Multiple startup files for different tool chains
CMSIS compliant device driver library
core_cm0.h / core_cm0plus.h / core_cm3.h / core_cm4.h
Core peripheral access layer
core_cmFunc.h
Header for special registers access functions (CMSIS v2)
core_cmInstr.h
Header for special instruction access functions (CMSIS v2)
core_cm4_simd.h
Header for Cortex-M4 SIMD instructions (CMSIS v2)
system_<device>.h
Other header files
Additional header files for device peripheral access layer
core_cm0.c / core_cm3.c / core_cm4.c
Functions for accessing special instructions (CMSIS 1.x to 2.0 only)
system_<device>.c
System functions including initialization
Peripheral driver code
Device peripheral access layer and additional access functions

CMSIS use in a project

Key files:

- <device>.h

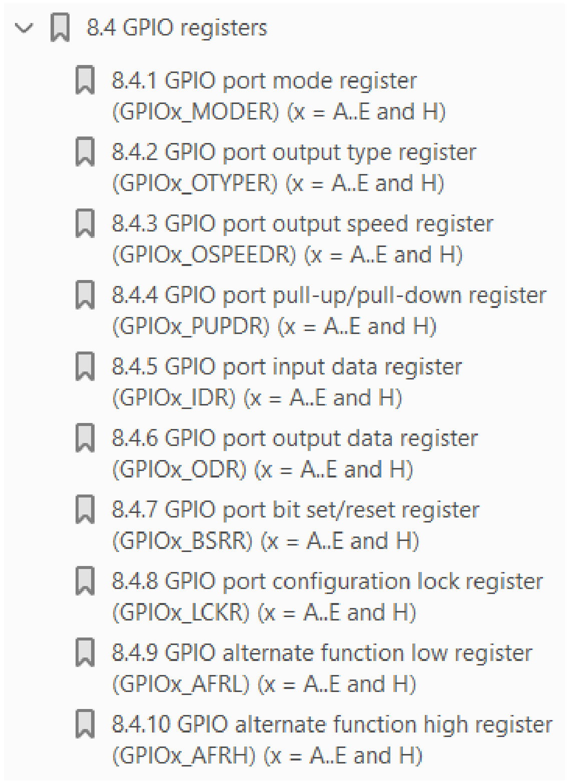
- system_<device>.h

- core_cm4.h

Source: "The Definitive guide to ARM Cortex-M3 & M4 Processors", by Joseph Yiu

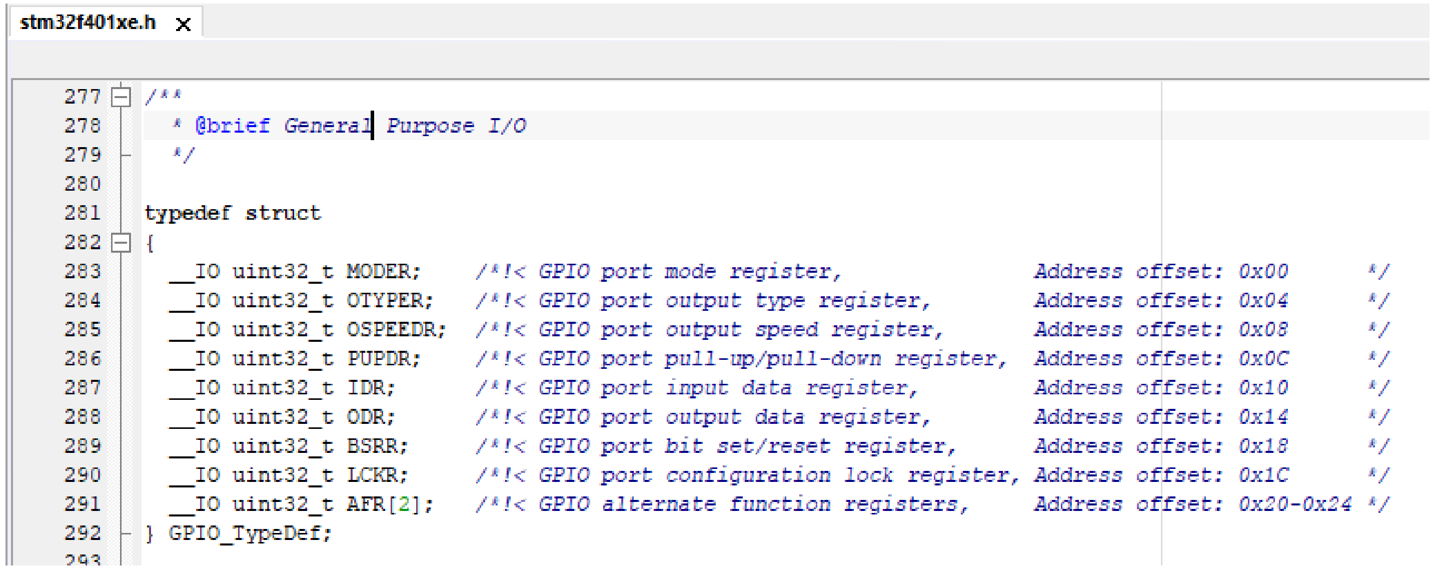
GPIO_TypeDef structure

STM32F401 – REFERENCE MANUAL



- 8.4 GPIO registers
  - 8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..E and H)
  - 8.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A..E and H)
  - 8.4.3 GPIO port output speed register (GPIOx_OSPEEDR) (x = A..E and H)
  - 8.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A..E and H)
  - 8.4.5 GPIO port input data register (GPIOx_IDR) (x = A..E and H)
  - 8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..E and H)
  - 8.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..E and H)
  - 8.4.8 GPIO port configuration lock register (GPIOx_LCKR) (x = A..E and H)
  - 8.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A..E and H)
  - 8.4.10 GPIO alternate function high register (GPIOx_AFRH) (x = A..E and H)

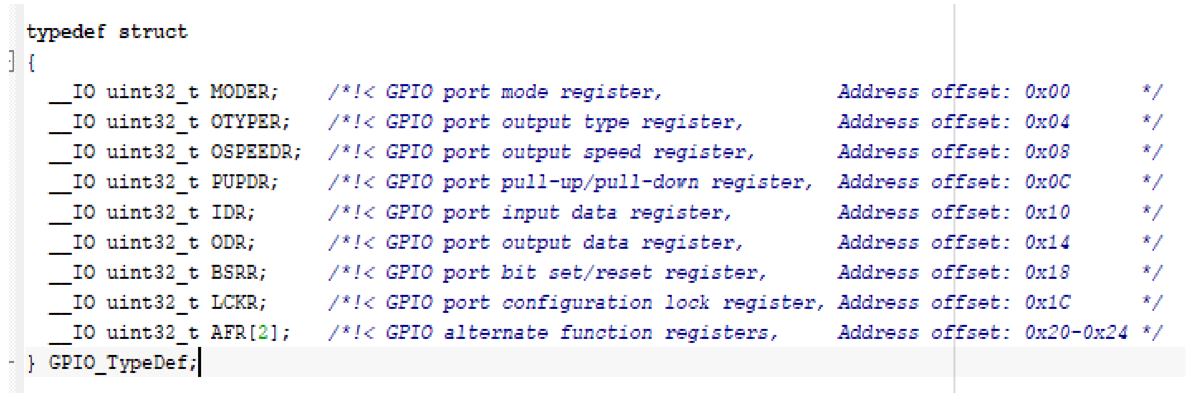
STM32F401XE.H (<DEVICE.H> CMSIS FILE)



```
/**
  * @brief General Purpose I/O
  */

typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;      /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint32_t BSRR;     /*!< GPIO port bit set/reset register,      Address offset: 0x18      */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

***GPIO_TypeDef** is a C structure designed in such a way that its data members correspond to all the registers within a given hardware block, such as the GPIO Registers.*

GPIO_TypeDef structure



```
typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;      /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint32_t BSRR;     /*!< GPIO port bit set/reset register,      Address offset: 0x18      */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

- Since all registers are 32-bit wide, the struct uses uint32_t datatype for its members.
- The __IO, __I, and __O identifiers are preprocessor macros defined in the Cortex Microcontroller Software Interface Standard (CMSIS), which the “core_cm4.h” header file is part of.
- __IO == Read/Write
- __I == Read-Only
- __O == Write-Only

GPIO_TypeDef structure

- Need to make sure that the GPIO structure is at the right base address.
- We have only created instances of Point, Rectangle, and Triangle structures, where the compiler controlled their placement in memory.
- Similar to type casting the GPIO addresses to pointers, we can use pointers to structures initialized to the hard-coded base address for that GPIO.
- This is what is done in the "stm32f401xe.h" header:
  - **#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)**
- The GPIOA macro defines a pointer to the GPIO_TypeDef structure, which is hard-coded to the GPIO_BASE address.

Replace registers with structures

1. Replace every register access using the structure pointer form.
2. For example, to replace the first register access:
   1. Take the RCC pointer and append the member access operator.
   2. IAR will help with intelli-sense displaying all the members of this structure.
   3. Choose the appropriate register from the list.
3. The same for the other registers (Assignment).

```
// RCC Base Address: 0x40023800
// RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)
// Address offset: 0x30
// Set bit[0] to 1
// 1. Enable clock to Peripheral
*((unsigned int*)(0x40023800+0x30)) |= 0x1;

// GPIOA Base Address: 0x40020000
// GPIO port mode register (GPIOx_MODER) (x = A..E and H)
// Address offset: 0x00
// Set bit[11:10] to 0x01  so --> 0x400 // To enable Port5 as output
*((unsigned int*)(0x40020000+0x00)) |= 0x400;

 // GPIOA Base Address: 0x40020000
 // GPIO port output data register (GPIOx_ODR) (x = A..E and H)
 // Address offset: 0x14
 // Set bit[5] to 1 --> 0x20; // Turn LED ON
 // Set bit[5] to 0 --> 0x00; // Turn LED OFF

 while(1)
   {
      delay(1000000);
      *((unsigned int*)(0x40020000+0x14)) |= (1<<5);

      delay(1000000);
      *((unsigned int*)(0x40020000+0x14)) &= ~(1<<5);
   }
```

Demo – Blinking LED using CMSIS

1. Create a new project
2. Bring in the files “stm32f401xe.h” & “system_stm32f4xx.h”
3. Show compiler failures
4. Enable use of CMSIS in project
5. Add code for Blinking LED using GPIO addresses type-casted to pointers (as shown here)
6. Convert usage of type-casted addresses to CMSIS structures

To learn more about CMSIS

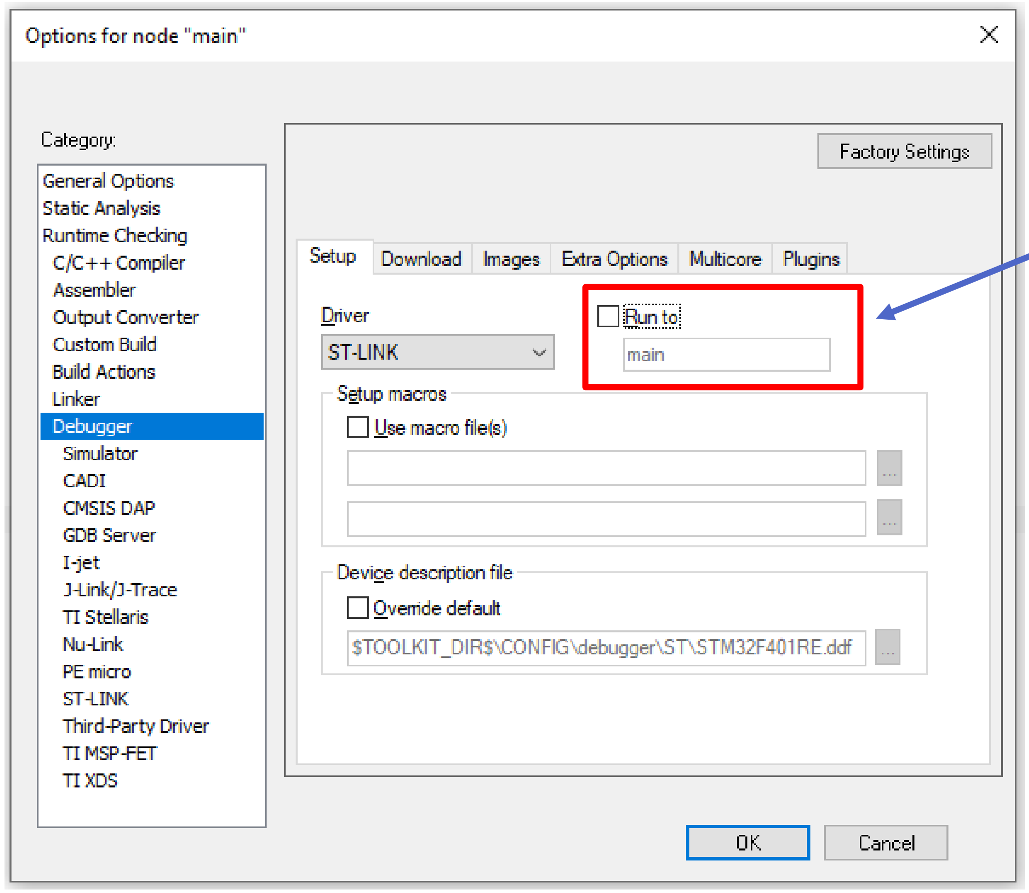
- https://developer.arm.com/tools-and-software/embedded/cmsis
- https://github.com/ARM-software/CMSIS_5
- https://www.st.com/en/embedded-software/stm32-standard-peripheral-libraries.html

BREAK 2

Startup Code

- Before “main”
- Standard startup code
- Linker Map file
- Data initialization
- The reset sequence
- Vector Table
- Fault Handler

Before "main"



Options for node "main"

Category:

- General Options
- Static Analysis
- Runtime Checking
- C/C++ Compiler
- Assembler
- Output Converter
- Custom Build
- Build Actions
- Linker
- Debugger
- Simulator
- CADI
- CMSIS DAP
- GDB Server
- I-jet
- J-Link/J-Trace
- TI Stellaris
- Nu-Link
- PE micro
- ST-LINK
- Third-Party Driver
- TI MSP-FET
- TI XDS

Driver: ST-LINK

☐ Run to: main

Setup macros

☐ Use macro file(s)

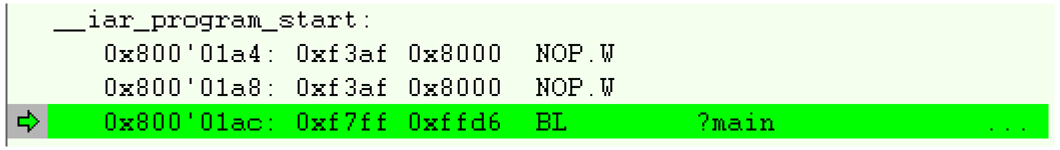
Device description file

☐ Override default

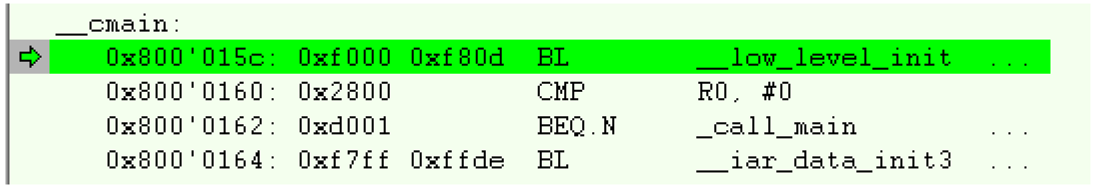
$TOOLKIT_DIR$\CONFIG\debugger\ST\STM32F401RE.ddf

Uncheck box to see the world before main.

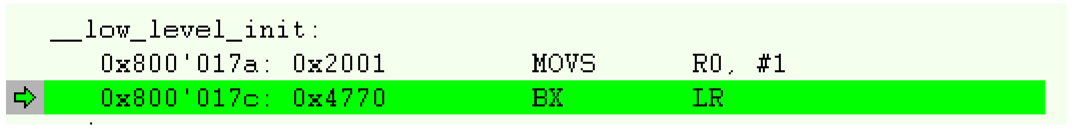
Standard startup code from IAR



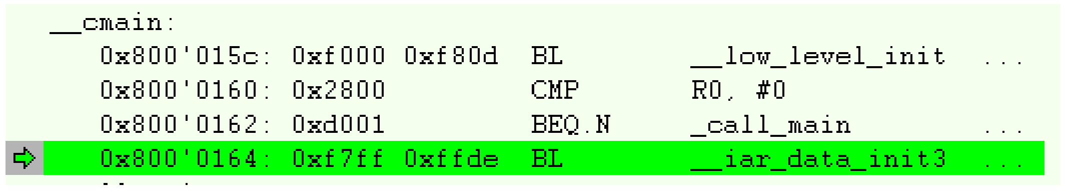
```
__iar_program_start:
    0x800'01a4: 0xf3af 0x8000  NOP.W
    0x800'01a8: 0xf3af 0x8000  NOP.W
    0x800'01ac: 0xf7ff 0xffd6  BL          ?main                    ...
```



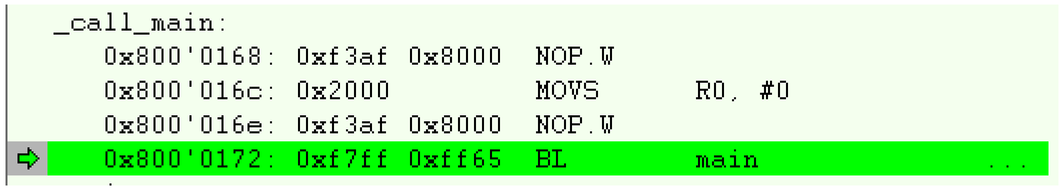
```
__cmain:
    0x800'015c: 0xf000 0xf80d  BL          __low_level_init   ...
    0x800'0160: 0x2800         CMP         R0, #0
    0x800'0162: 0xd001         BEQ.N       _call_main         ...
    0x800'0164: 0xf7ff 0xffde  BL          __iar_data_init3   ...
```



```
__low_level_init:
    0x800'017a: 0x2001         MOVS        R0, #1
    0x800'017c: 0x4770         BX          LR
```



```
__cmain:
    0x800'015c: 0xf000 0xf80d  BL          __low_level_init   ...
    0x800'0160: 0x2800         CMP         R0, #0
    0x800'0162: 0xd001         BEQ.N       _call_main         ...
    0x800'0164: 0xf7ff 0xffde  BL          __iar_data_init3   ...
```



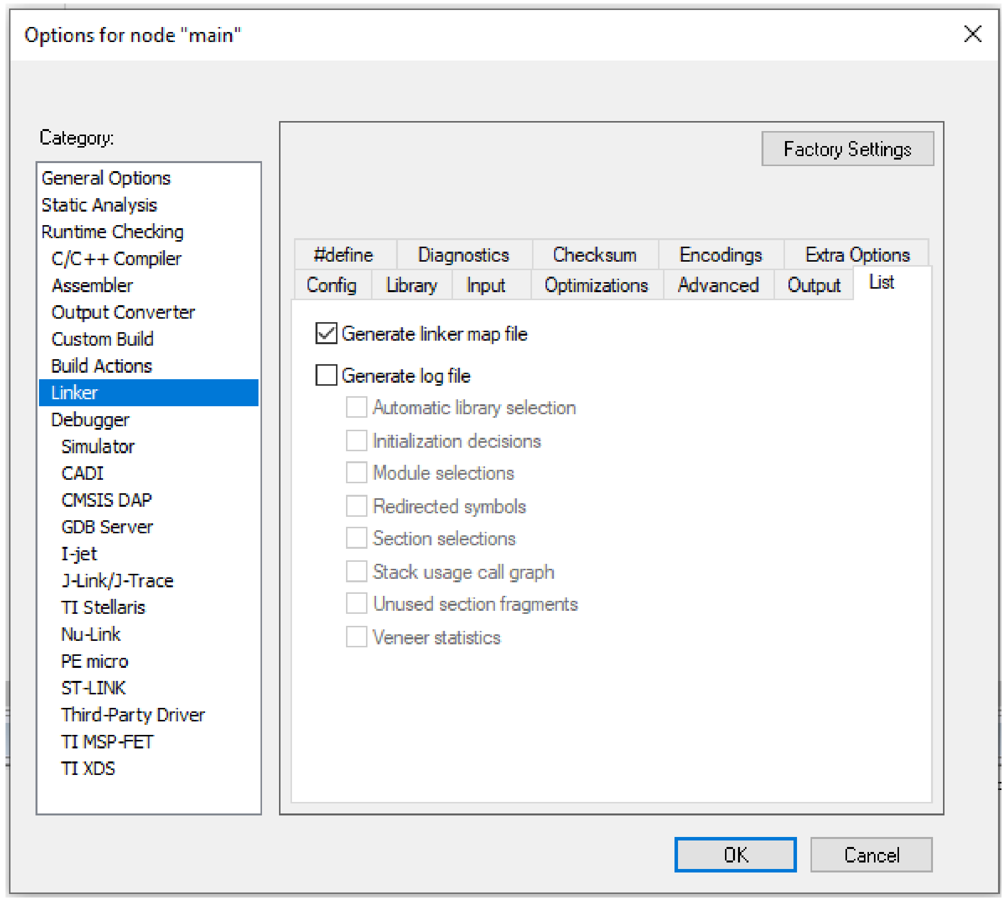
```
_call_main:
    0x800'0168: 0xf3af 0x8000  NOP.W
    0x800'016c: 0x2000         MOVS        R0, #0
    0x800'016e: 0xf3af 0x8000  NOP.W
    0x800'0172: 0xf7ff 0xff65  BL          main                     ...
```

Standard startup code from IAR

- __iar_program_start:

- ?main :
  - This is an IAR specific startup code. IAR decided to call it “?main”.
- __low_level_init:
  - Function intended to perform a customized initialization of the hardware that must occur very early on.

Linker Map File



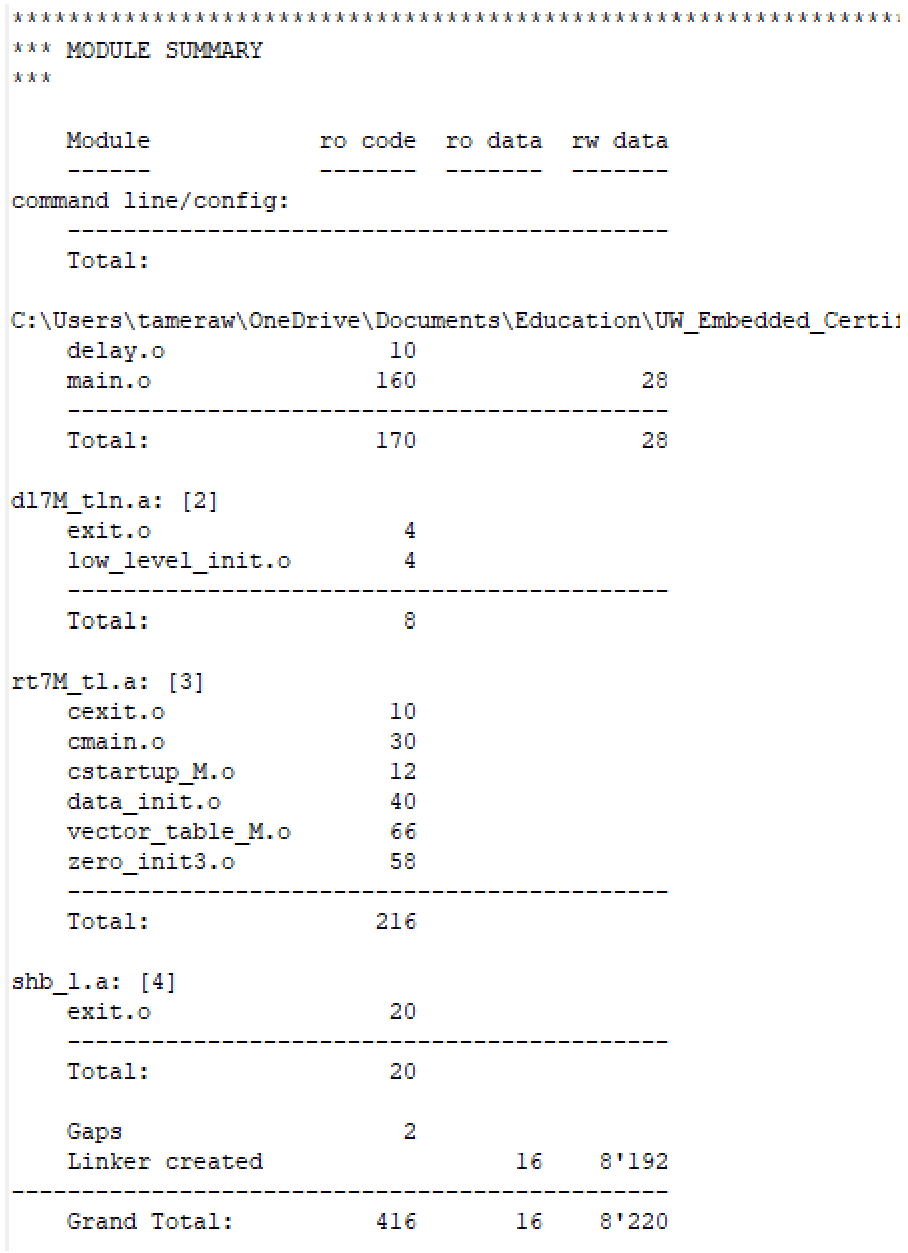
Options for node "main"

- [✓] Generate linker map file
- [ ] Generate log file
  - [ ] Automatic library selection
  - [ ] Initialization decisions
  - [ ] Module selections
  - [ ] Redirected symbols
  - [ ] Section selections
  - [ ] Stack usage call graph
  - [ ] Unused section fragments
  - [ ] Veneer statistics

Map file sections: MODULE SUMMARY

- Should always know how big your program is in terms of code space in ROM and data space in ROM and RAM. To find out, you scroll to the "Module Summary" section.
- Information broken down by read-only code, read-only data, and read-write data, as well as the object modules
- At the bottom of that section you find the total used.
- The largest contributor is section “Linker created” which is generated by the linker for the stack.
  - EX: 8192 which is 0x2000, which is what was specified in the project options.

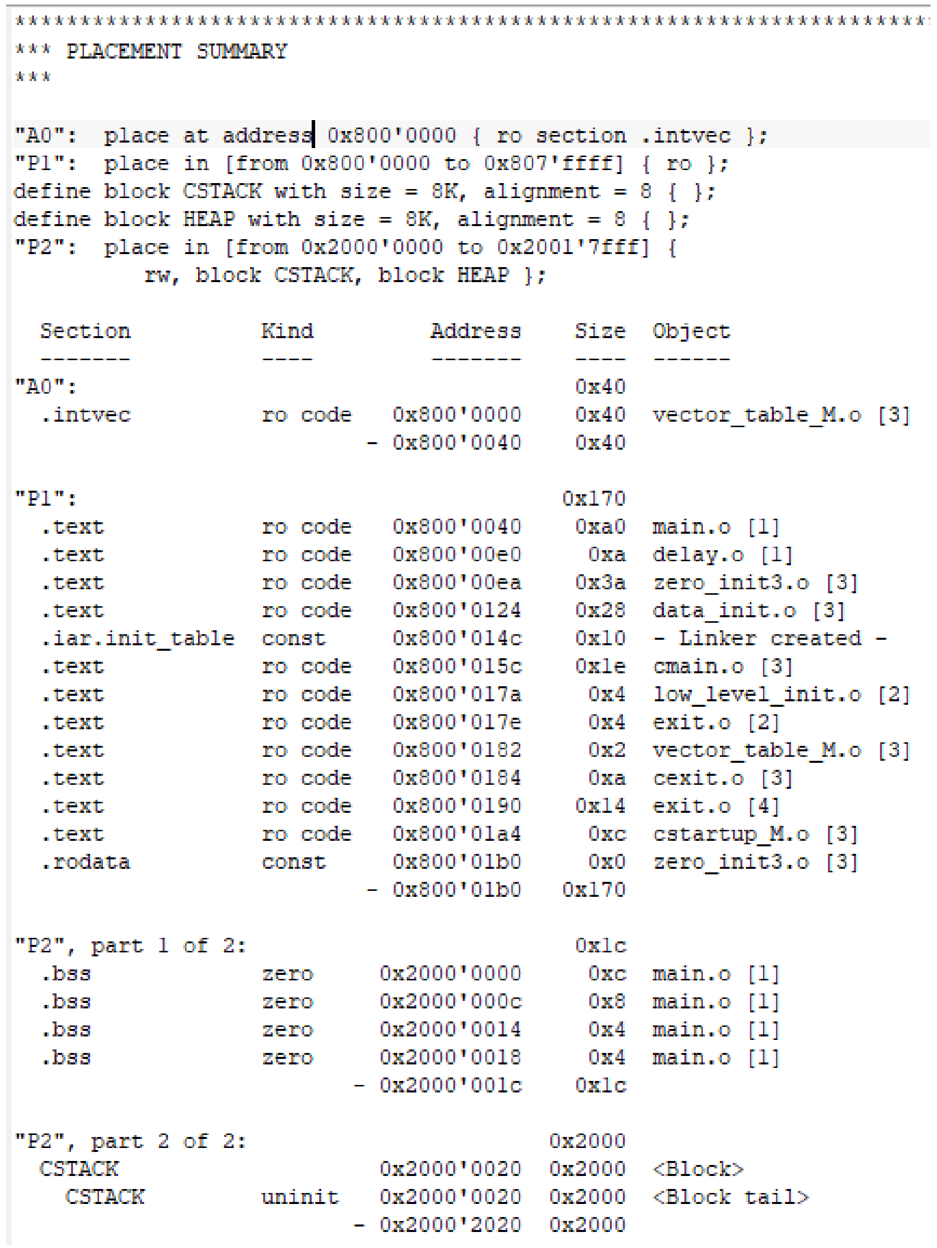
Map File Sections: “Module Summary”



```
*******************************************************************************
*** MODULE SUMMARY
***

    Module              ro code  ro data  rw data
    ------              -------  -------  -------
command line/config:
    -------------------------------------------
    Total:

C:\Users\tameraw\OneDrive\Documents\Education\UW_Embedded_Certi
    delay.o                  10
    main.o                  160                28
    -------------------------------------------
    Total:                  170                28

dl7M_tln.a: [2]
    exit.o                    4
    low_level_init.o          4
    -------------------------------------------
    Total:                    8

rt7M_tl.a: [3]
    cexit.o                  10
    cmain.o                  30
    cstartup_M.o             12
    data_init.o              40
    vector_table_M.o         66
    zero_init3.o             58
    -------------------------------------------
    Total:                  216

shb_l.a: [4]
    exit.o                   20
    -------------------------------------------
    Total:                   20

    Gaps                      2
    Linker created                    16    8'192
-----------------------------------------------
    Grand Total:            416       16    8'220
```

Map file sections: PLACEMENT SUMMARY

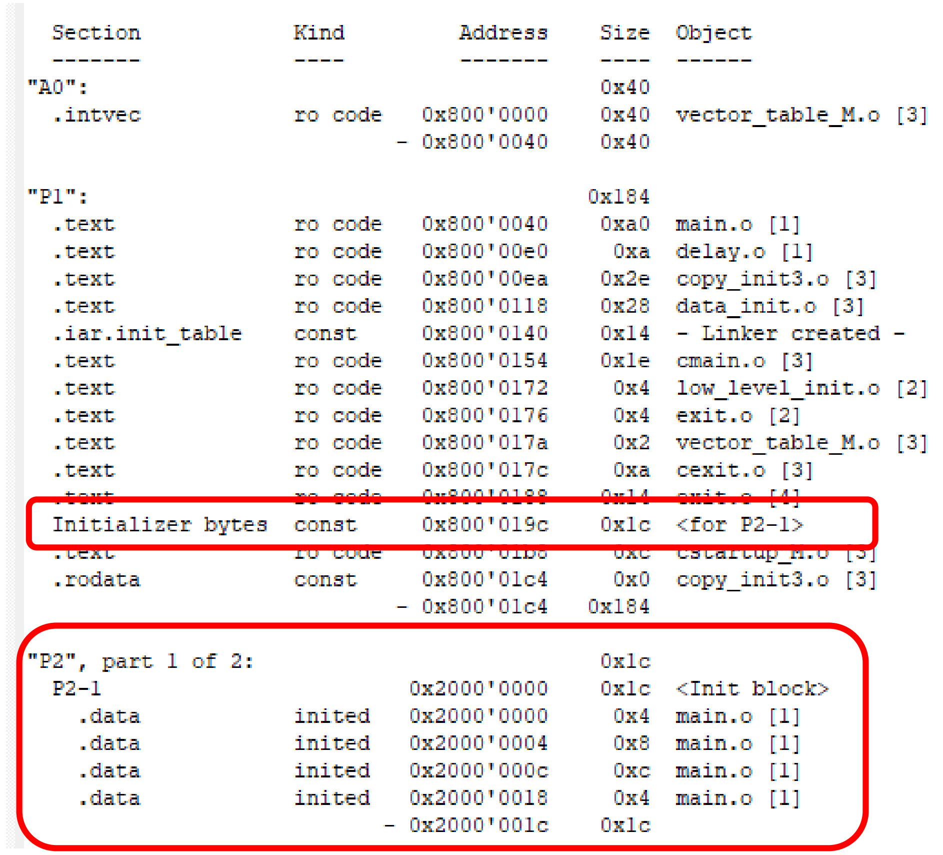
- Lists all the program sections
- A program section is a contiguous chunk of memory that has a symbolic name.
  - .intvec: for interrupts (ROM address range)
  - .text: for the code (ROM address range)
  - .rodata: for read-only data (ROM address range)
  - .bss: holds uninitialized data that need to be set to **zero** during the system startup
  - CSTACK: holds the stack and is left **uninitialized** during startup.



```
***********************************************************************
*** PLACEMENT SUMMARY
***

"A0":  place at address 0x800'0000 { ro section .intvec };
"P1":  place in [from 0x800'0000 to 0x807'ffff] { ro };
define block CSTACK with size = 8K, alignment = 8 { };
define block HEAP with size = 8K, alignment = 8 { };
"P2":  place in [from 0x2000'0000 to 0x2001'7fff] {
          rw, block CSTACK, block HEAP };

  Section            Kind          Address    Size  Object
  -------            ----          -------    ----  ------
"A0":                                         0x40
  .intvec            ro code    0x800'0000    0x40  vector_table_M.o [3]
                              - 0x800'0040    0x40

"P1":                                        0x170
  .text              ro code    0x800'0040    0xa0  main.o [1]
  .text              ro code    0x800'00e0     0xa  delay.o [1]
  .text              ro code    0x800'00ea    0x3a  zero_init3.o [3]
  .text              ro code    0x800'0124    0x28  data_init.o [3]
  .iar.init_table    const      0x800'014c    0x10  - Linker created -
  .text              ro code    0x800'015c    0x1e  cmain.o [3]
  .text              ro code    0x800'017a     0x4  low_level_init.o [2]
  .text              ro code    0x800'017e     0x4  exit.o [2]
  .text              ro code    0x800'0182     0x2  vector_table_M.o [3]
  .text              ro code    0x800'0184     0xa  cexit.o [3]
  .text              ro code    0x800'0190    0x14  exit.o [4]
  .text              ro code    0x800'01a4     0xc  cstartup_M.o [3]
  .rodata            const      0x800'01b0     0x0  zero_init3.o [3]
                              - 0x800'01b0   0x170

"P2", part 1 of 2:                            0x1c
  .bss               zero      0x2000'0000     0xc  main.o [1]
  .bss               zero      0x2000'000c     0x8  main.o [1]
  .bss               zero      0x2000'0014     0x4  main.o [1]
  .bss               zero      0x2000'0018     0x4  main.o [1]
                             - 0x2000'001c    0x1c

"P2", part 2 of 2:                          0x2000
  CSTACK                       0x2000'0020  0x2000  <Block>
    CSTACK           uninit    0x2000'0020  0x2000  <Block tail>
                             - 0x2000'2020  0x2000
```

Placement Summary section

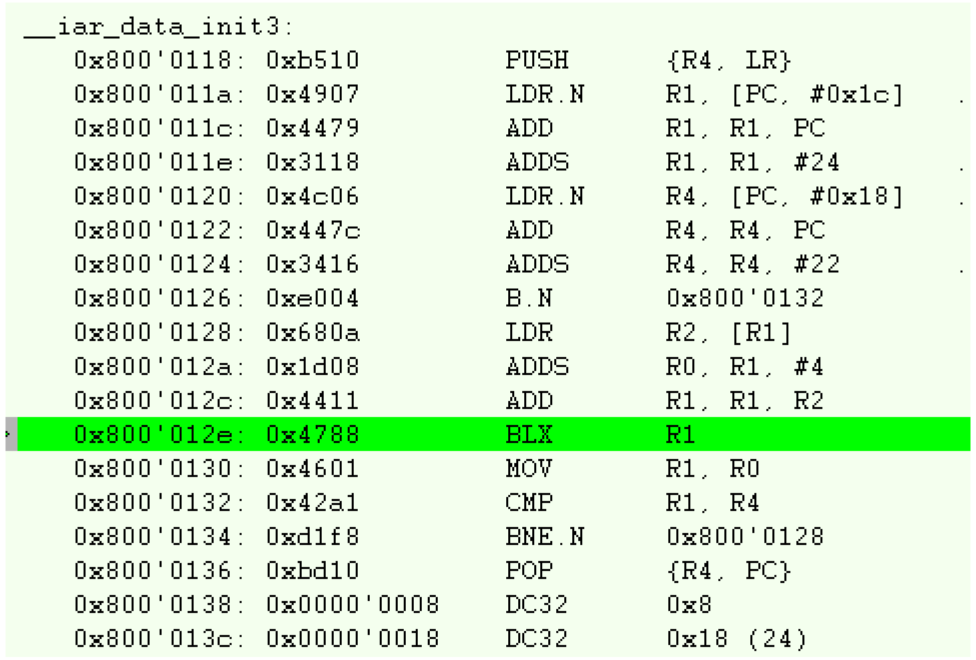
Map file: Data initilization



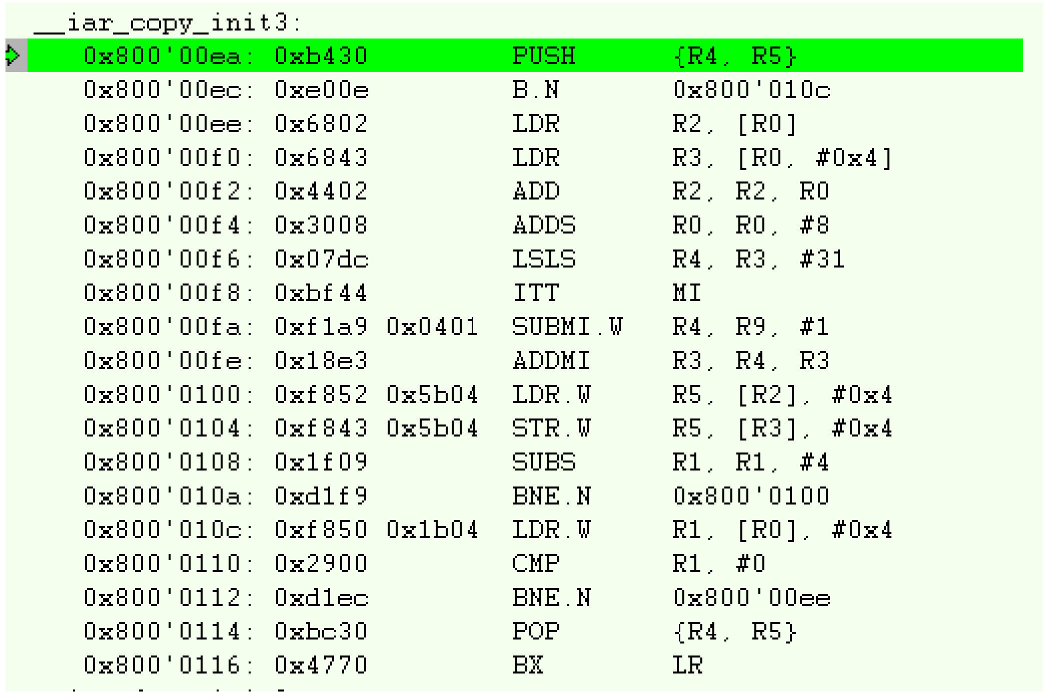
```
  Section             Kind         Address     Size  Object
  -------             ----         -------     ----  ------
"A0":                                          0x40
  .intvec             ro code    0x800'0000    0x40  vector_table_M.o [3]
                               - 0x800'0040    0x40

"P1":                                         0x184
  .text               ro code    0x800'0040    0xa0  main.o [1]
  .text               ro code    0x800'00e0     0xa  delay.o [1]
  .text               ro code    0x800'00ea    0x2e  copy_init3.o [3]
  .text               ro code    0x800'0118    0x28  data_init.o [3]
  .iar.init_table     const      0x800'0140    0x14  - Linker created -
  .text               ro code    0x800'0154    0x1e  cmain.o [3]
  .text               ro code    0x800'0172     0x4  low_level_init.o [2]
  .text               ro code    0x800'0176     0x4  exit.o [2]
  .text               ro code    0x800'017a     0x2  vector_table_M.o [3]
  .text               ro code    0x800'017c     0xa  cexit.o [3]
  .text               ro code    0x800'0188    0x14  exit.o [4]
  Initializer bytes   const      0x800'019c    0x1c  <for P2-1>
  .text               ro code    0x800'01b8     0xc  cstartup_M.o [3]
  .rodata             const      0x800'01c4     0x0  copy_init3.o [3]
                               - 0x800'01c4   0x184

"P2", part 1 of 2:                             0x1c
  P2-1                          0x2000'0000    0x1c  <Init block>
    .data             inited    0x2000'0000     0x4  main.o [1]
    .data             inited    0x2000'0004     0x8  main.o [1]
    .data             inited    0x2000'000c     0xc  main.o [1]
    .data             inited    0x2000'0018     0x4  main.o [1]
                              - 0x2000'001c    0x1c
```

- “Initializer bytes” section is created in ROM
- “.data” section is created in RAM
- Both of equal size
- The startup code copies the "Initializer bytes“ section from ROM to the “.data” sections in RAM.

Data initialization



```
__iar_data_init3:
    0x800'0118: 0xb510          PUSH      {R4, LR}
    0x800'011a: 0x4907          LDR.N     R1, [PC, #0x1c]
    0x800'011c: 0x4479          ADD       R1, R1, PC
    0x800'011e: 0x3118          ADDS      R1, R1, #24
    0x800'0120: 0x4c06          LDR.N     R4, [PC, #0x18]
    0x800'0122: 0x447c          ADD       R4, R4, PC
    0x800'0124: 0x3416          ADDS      R4, R4, #22
    0x800'0126: 0xe004          B.N       0x800'0132
    0x800'0128: 0x680a          LDR       R2, [R1]
    0x800'012a: 0x1d08          ADDS      R0, R1, #4
    0x800'012c: 0x4411          ADD       R1, R1, R2
    0x800'012e: 0x4788          BLX       R1
    0x800'0130: 0x4601          MOV       R1, R0
    0x800'0132: 0x42a1          CMP       R1, R4
    0x800'0134: 0xd1f8          BNE.N     0x800'0128
    0x800'0136: 0xbd10          POP       {R4, PC}
    0x800'0138: 0x0000'0008     DC32      0x8
    0x800'013c: 0x0000'0018     DC32      0x18 (24)
```



```
__iar_copy_init3:
    0x800'00ea: 0xb430          PUSH      {R4, R5}
    0x800'00ec: 0xe00e          B.N       0x800'010c
    0x800'00ee: 0x6802          LDR       R2, [R0]
    0x800'00f0: 0x6843          LDR       R3, [R0, #0x4]
    0x800'00f2: 0x4402          ADD       R2, R2, R0
    0x800'00f4: 0x3008          ADDS      R0, R0, #8
    0x800'00f6: 0x07dc          LSLS      R4, R3, #31
    0x800'00f8: 0xbf44          ITT       MI
    0x800'00fa: 0xf1a9 0x0401   SUBMI.W   R4, R9, #1
    0x800'00fe: 0x18e3          ADDMI     R3, R4, R3
    0x800'0100: 0xf852 0x5b04   LDR.W     R5, [R2], #0x4
    0x800'0104: 0xf843 0x5b04   STR.W     R5, [R3], #0x4
    0x800'0108: 0x1f09          SUBS      R1, R1, #4
    0x800'010a: 0xd1f9          BNE.N     0x800'0100
    0x800'010c: 0xf850 0x1b04   LDR.W     R1, [R0], #0x4
    0x800'0110: 0x2900          CMP       R1, #0
    0x800'0112: 0xd1ec          BNE.N     0x800'00ee
    0x800'0114: 0xbc30          POP       {R4, R5}
    0x800'0116: 0x4770          BX        LR
```

The C-Standard initialization sequence

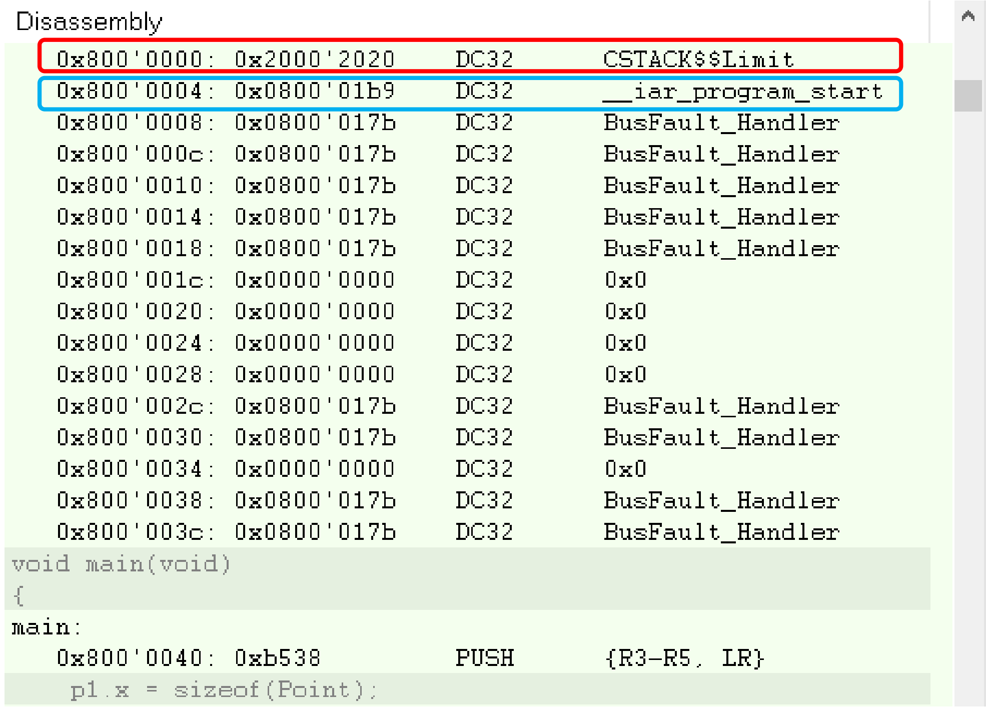
- IAR implements standard-compliant startup code.
- By the time main() is called, the C standard requires that:
  - All initialized variables get their initial values.
  - And all uninitialized variables to be set to zero.
- Some vendors are not compliant with this standard, so should test the start up code to verify.
- If the .bss sections are not cleared, then one might need to explicitly initialize all previously uninitialized variables to zero.
- Note that this would not be optimal:
  - We would be converting the .bss sections to .data sections, which require a matching "Initializer bytes“ section in ROM. In other words, we would be taking space in ROM for a bunch of zeros.

Starting from "Reset"

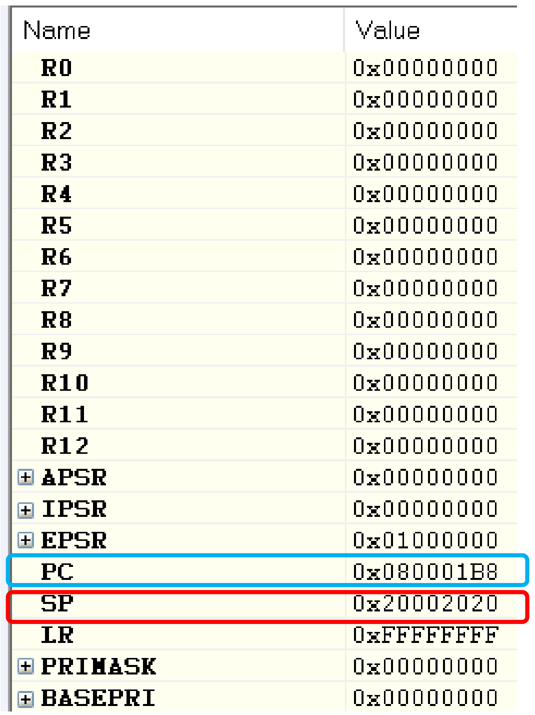
- How does the SP (stack pointer) get its initial value?
- How does the PC (program counter) end up at the function ***__iar_program_start?***

Starting from "Reset"

- The ARM Cortex-M is hardwired after reset to:
  - Copy the bits from address 0 to the SP register.
  - Copy all bits (except the least-significant-bit) from address 0x4 to the PC register.
- Recall:
  - The LSb of any value loaded to the PC must be one, because this bit indicates Thumb mode which is the only mode supported by Cortex-M.

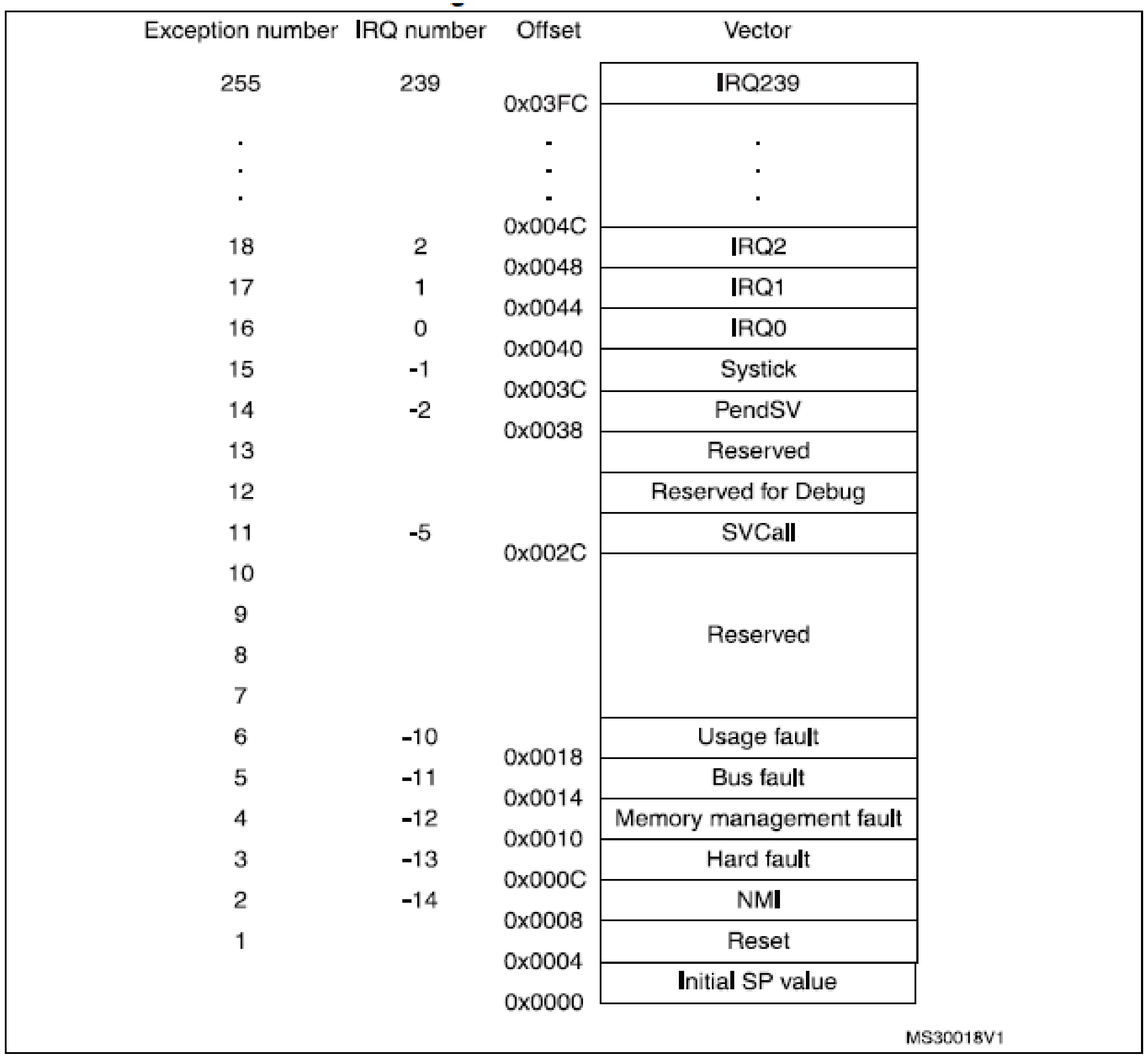


```
Disassembly
   0x800'0000: 0x2000'2020    DC32        CSTACK$$Limit
   0x800'0004: 0x0800'01b9    DC32        __iar_program_start
   0x800'0008: 0x0800'017b    DC32        BusFault_Handler
   0x800'000c: 0x0800'017b    DC32        BusFault_Handler
   0x800'0010: 0x0800'017b    DC32        BusFault_Handler
   0x800'0014: 0x0800'017b    DC32        BusFault_Handler
   0x800'0018: 0x0800'017b    DC32        BusFault_Handler
   0x800'001c: 0x0000'0000    DC32        0x0
   0x800'0020: 0x0000'0000    DC32        0x0
   0x800'0024: 0x0000'0000    DC32        0x0
   0x800'0028: 0x0000'0000    DC32        0x0
   0x800'002c: 0x0800'017b    DC32        BusFault_Handler
   0x800'0030: 0x0800'017b    DC32        BusFault_Handler
   0x800'0034: 0x0000'0000    DC32        0x0
   0x800'0038: 0x0800'017b    DC32        BusFault_Handler
   0x800'003c: 0x0800'017b    DC32        BusFault_Handler
void main(void)
{
main:
   0x800'0040: 0xb538         PUSH        {R3-R5, LR}
     p1.x = sizeof(Point);
```



Name	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
⊞ APSR	0x00000000
⊞ IPSR	0x00000000
⊞ EPSR	0x01000000
PC	0x080001B8
SP	0x20002020
LR	0xFFFFFFFF
⊞ PRIMASK	0x00000000
⊞ BASEPRI	0x00000000

<u>Note:</u> These are not machine instructions. These are simply words in memory.



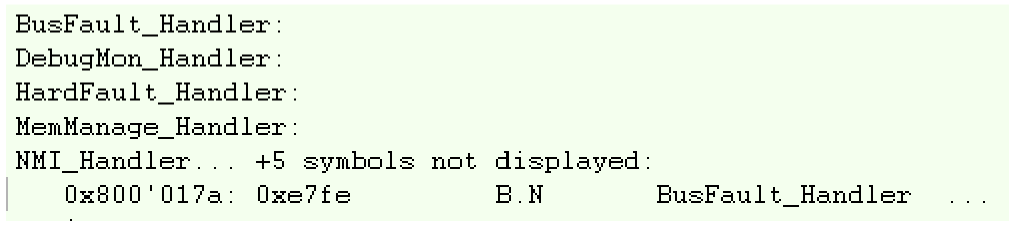
Exception number	IRQ number	Offset	Vector
255	239	0x03FC	IRQ239
.		.	.
.		.	.
.		.	.
		0x004C	
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			
9			Reserved
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

MS30018V1

Vector Table

- The vector table contains the reset value of the stack pointer SP and the start address of the PC.
- It also contains the exception and interrupt vectors that the processor can handle.
- *Source:*
    - *PM0214-Programming Manual for STM32 Cortex-M4*

Fault Handler



```
BusFault_Handler:
DebugMon_Handler:
HardFault_Handler:
MemManage_Handler:
NMI_Handler... +5 symbols not displayed:
   0x800'017a: 0xe7fe         B.N        BusFault_Handler   ...
```

- The IAR startup code defines all exception handlers, such as BusFault, DebugMonitor, HardFault, MemoryManager, and Non-Maskable-Interrupt, but they all point to the same piece of code.
- The IAR code associated with all these exception handlers is a single branch instruction, which jumps to itself.
- So an occurrence of any of the exceptions ends up tying the CPU in a tight endless loop, which is good for debugging, because when you break into the code, you will find it looping inside an exception handler.
- However, this is not good for production as the device will appear to be completely locked and unresponsive.
- Should consider implementing some built-in recovery mechanism upon hitting these exceptions (reset device for example).

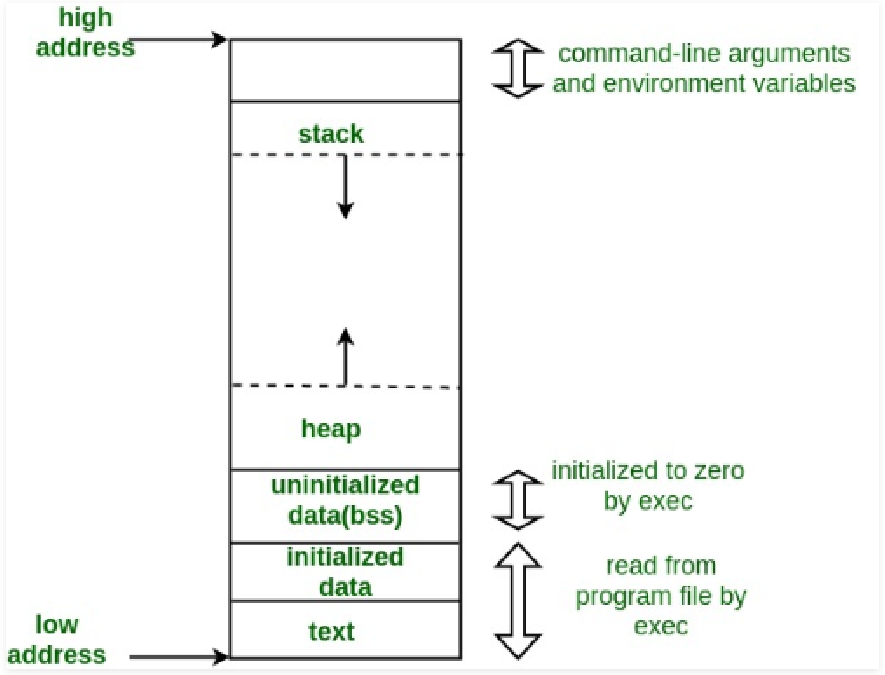
Memory layout of C Programs

https://www.geeksforgeeks.org/memory-layout-of-c-program/

Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap


high address
stack
heap
uninitialized data(bss)
initialized data
text
low address
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

Assignment 06

Suggested Reading

- ***"The C Programming Language" By Brian Kernighan & Dennis Ritchie (Second Edition)***
  - ??
- ***"The Cortex-M4 Device Generic User Guide"***
  - 2.6: Data types in C programming
  - 2.9: The Cortex microcontroller software interface standard
- ***"The Definitive Guide to ARM Cortex M3 & M4" by Joseph Yiu (Third Edition)***
  - ??
- ***"An Embedded Software Primer" by David E. Simon***
  - Chapter 5: Survey of Software Architecture
  - Chapter 6.1: Tasks & Task State
  - Chapter 6.2: Tasks & Data