Fachhochschule
Münster University of
Applied Sciences

Lab for Semiconductor Devices and Bussystems

Exercise Embedded Systems WS2012/13
Tutor: Dipl. Informatiker Gregor Rebel

Exercise 2
Memory Management

This course describes different types of dynamic and static memory allocations in embedded systems. The knowledge of available memory types and their organisation is viable to write software that makes best use of the limited memory resources in microcontrollers.

Modern 32 bit microcontrollers provide an address space of 4GB. This sounds as if plenty of memory would be available for the application. Compraed to this huge address space, only very few amount of accessible memory is available. Especially random accessible memory is limited because of two reasons. First reason is that RAM consumes energy to be operated. The second reason is that RAM is also very expensive. Modern desktop computers and laptops easily provide gigabytes of RAM. But a microcontroller that is intended for mass production and shall cost far less than 10,- € including its RAM has a much tighter cost structure.

In a microcontroller, typically different types of memories are available

- **ROM**
  A preprogrammed memory that can store data and program code and is read only during operation. ROM keeps its contents during a power loss.

- **FLASH**
  This memory can be preprogrammed like ROM. Most microcontrollers allow to reprogram it during runtime at a block level (a block of data must be reprogrammed at once).
  FLASH keeps its contents during a power loss.

- **EEPROM**
  Typically can only store data. This memory can be reprogrammed hundrets or thousands times in small blocks. Reprogramming is much slower than write accesses to RAM. EEPROM keeps its contents during a power loss.

- **RAM**
  Can store data and program code on some architectures. RAM can be read and written at byte level with low access times. RAM will loose its contents during a power loss.

The STM32 microcontrollers used in this exercise provide FLASH and RAM type memories. An EEPROM type memory can be emulated by use of FLASH.

Example memory layout of an STM32F107VCT6 microcontroller:

0x0800 0000 – 0x0803 FFFF	FLASH	256kB
0x2000 0000 – 0x2000 FFFF	RAM	64kB

The linker script

At the end of the compilation process, the linker has to turn symbolic names of all global and static variables into real memory addresses. In order to do so, it has to be provided with start addresses, sizes and permissions of all memory areas available on target microcontroller. This information is provided by a so called "linker script". This linker script is added via option -T inside extensions.active/makefile to variable LDFLAGS. This variable is later passed to gcc during link stage. The used linker script used by The ToolChain for STM32 architecture is configs/stm32.ld
→ <ProjectFolder>/configs/stm32.ld

This script defines memory sections like text, data, bss and isr_vector.
Each section defines

- **Data alignment**
  Some architectures allow only memory accesses on adresses that are a multiple of 2 or 4. For these memory spaces, the compiler has to insert extra instructions for individual byte acesses.
- **names, order and start of subsections**
  A dot (.) marks the current memory address. This address increases while memory is allocated inside a section or subsection. Therefore Increasing start address of all later sections and subsections in same memory space.
- **dedicated memory space**
  At the end of each section, the dedicated memory space is noted after a ">" symbol.
  All available memory spaces have to be defined before. The ToolChain dynamically creates the file configs/memory.ld during each compilation process. This file defines all memory spaces available on the current target microcontroller as defined by activated makefiles.
  → <ProjectFolder>/configs/memory.ld

Each memory space defines

- **Name**
- **Access permissions**
  These are similar to Linux file permissions
  r memory space is readable
  w memory space is writable
  x memory space can execute program code
- **Start (Origin)**
- **Size (Length)**

Pointer Facts in C

Fact 1: Whenever A is datatype of some kind, &A gives the address of the memory where the value of A is stored.
Fact 2: Whenever a variable is declared as datatype* A, A points to memory where datatype is stored.
Fact 3: An alternative dereferenciation to * is ->

Example 1

```
u8_t Data = 1;            // 1 byte value
u8_t* DataPtr = &Data;    // 2 byte value on 16-bit architecture
                          // 4 byte value on 32-bit architecture


printf("Data=%i\n", Data);
printf("[DataPtr]=%i\n", *DataPtr);
```

Application view on memory

The GNU Compiler Collection (GCC) divides object files into at least three basic sections.
→ http://gcc.gnu.org/onlinedocs/gccint/Sections.html

- **text**
  Holds instructions and read-only data.

```
const char* Hello = "Hello World";
// "Hello World" written directly to FLASH
```

- **data**
  Holds initialized writable data like global and static variables.

```
u8_t Index = 3;
// 3 written to memory location only once at program start

void test(void) {
    static s16_t Value = -1234;
    // -1234 written to memory on first call of test()
}
```

- **bss**
  Holds uninitialized writable data like e.g. stack memories.

```
void foo(void) {
    char Character = 42;
    // 42 written to memory location on every call of foo()
}
```

Whenever the linker finds a const defined variable, its initial value is directly placed into resulting binary file. During programming the target device, the initial value is written somewhere in FLASH memory. When the constant (e.g. Hello) is accessed, the read access ocurs directly from FLASH memory.
The initialization code for global variables is automatically created by the compiler and put into a single block that is executed at program start before main() is called. The initialization code for local static variables can be placed inside the function or in the global block of global initializers.

Memory stack

Each processor provides at least one special register called the stack pointer (SP). This pointer can be loaded with a memory address. It also provides two special operations push and pop. A push operation writes a given date into memory at address where SP currently points to and automatically increases SP. A pop operation reads data from memory at address where SP currently points to and automatically decreases SP. This simple technique is used to store all local variables and return adresses inside functions. When program flow enters a function, the initial value of each local variable is pushed onto stack. From now on, each variable is accessed as a memory access relative to the current SP value. A 32-bit read access to (SP-4) means to read the last defined local 32 bit variable.

Example 2

```
void foo1(void) {
     u8_t A = 1;
    u16_t B = 2;

    while (1); // Break here!
}

void foo2(void) {
    static u32_t C = 3;
    u8_t D = 1;

    foo1();
}
```

Excercise 2A

1. Implement the above Example 2 in a new project.
2. Call foo2() from your taskMain().
3. Use the ./_/debug.sh script to place a breakpoint in line marked "// Break here!"
4. Continue program execution until breakpoint
5. Display memory addresses of
   A = _____
   B = _____
   C = _____
   D = _____
6. What is stored in RAM between &D and &A?
7. Why is &C located somewhere totally different?

Constants in C

Defining constant data moves data to FLASH or ROM. And therefore frees expensive RAM.
In the early dawn of C, only preprocessor defines were used for constant data. These are still widely used by developers in the embedded area but they incur several problems.

The ANSI C99 standard defines different more elaborate types of constant data.

- **defines**
  Before compilation of C-source takes place, a so called preprocessor parses the source code and parses special preprocessor directives. Preprocessor directives are commonly used in applications targeted at embedded systems. But they have some important drawbacks:
  - Preprocessor directives can produce obscure error messages.
  - Preprocessor directives are not type safe

  Every preprocessor directive starts with a hash symbol (#):
  - **#define NAME REPLACE**
    Every occurance of NAME is replaced by REPLACE.

  - **#define MACRO(ARG1,ARG2, ...) REPLACE ARG1 ARG2**
    Every occurance of MACRO() is replaced by REPLACE.
    ARGn are passed as arguments.

  - **#if (CONDITION)**
    CODE1
    #else
    CODE2
    #endif

    Puts CODE1 or CODE2 into compilation depending on logical value of CONDITION.
    Either CODE1 or CODE2 will not be part of the compiled code. CONDITION is evaluated at compile time.

- **const Data**
  Every datatype of basic or structed type can be defined as being const. This will place its data into FLASH or ROM.

```
const u8_t SerialNo = 123;
printf("SerialNo=%i\n", SerialNo);
```

- **const arguments**
  Unlike preprocessor defines, the compiler can check the use of this data. When a pointer to a const datatype is passed to a function, the corresponding argument must be declared as const. On the other hand, if a pointer argument is declared as const, the compiler ensures that the data being pointed to will not be changed by this function or any function that is called from it.

```
bool checkSerialNo(const u8_t* SerialNo) {
    if (*SerialNo == 123) return TRUE;
    return FALSE;
}
```

- **enums**
  Computers are not that good in interpreting string, but they can deal very good with numbers. In the world of dynamic languages like PERL or JAVA, strings are often passed to functions to switch their operation modes. Strings require lots of memory space and are slow to interpret. In the early days of microcontrollers, preprocessor directives were often used to replace plain numbers by memorizable names. The ANSI C standard provides an elegant datatype for this called an enumeration. An enumeration is a set of names arranged in an order and grouped under a name. By default, the value of each name is increased by 1 for each name in order of their declaration. It is possible to set a new value for every single name. Enums are always constant and cannot be changed during runtime.
  A big advantage of enums is that the compiler can check them for correct type and value.

Example 4

```
typedef enum {
    mul,       // == 0
    add,       // == 1
    power=10,  // == 10
    sqr,       // == 11
    exp=13     // == 13
} Mode_e;

void foo(Mode_e Choice) {
    switch (Choice) {
        case mul: ... break;
        case add: ... break;
        default: break;
    }
}

foo(LF);
```

Exercise 2B

1. Write function **multiply(u8_t* A, u8_t* B, u8_t* R)** that
   1. Calculates product of *A and *B
   2. Stores result in *R

2. Write function **add(u8_t* A, u8_t* B, u8_t* R)** that
   1. works like multiply, but calculates sum instead of product

3. Write function **u8_t calc(u8_t CA, u8_t CB, Mode_e M)** that
   1. decides via switch(M)-statement wether it calls multiply(A,B,R) or add(A,B,R)
   2. uses static variable Temp to store result of multiply()/ add()
   3. returns calculated value

4. Place several calc()-calls in taskMain() to calculate the following:
   1. Value1 = 3 + 4
   2. Value2 = 4 * 5

- Make use of enum for M
- Declare as many variables as const as possible
- Declare as many function arguments as const as possible

Pointers of different type

A CPU can only process data in its native, architecture dependent size. An 8-bit CPU can only process 8-bit (byte) values. A 32-bit CPU can process 8-/ 16- and 32-bit values as all of them fit into 32 Bits. Real world applications Require larger datasets to be processed. Therefore arrays and structs have been invented to be able to define them in your C source. But how are they handled by the CPU?

In fact, if we define an array of Data, the compiler will create instructions that initialize a memory area of exact size for the data plus a pointer to the start of this area.

Example 5

```
u8_t A[4] = { 0,1,2,3 };
// equals to

u8_t* A = 0x123456;  // 0x123456 = some unused address in RAM
A[0] = 0;            // initialization code
A[1] = 1;
A[2] = 2;
A[3] = 3;
```

Structured Memory

A struct allows to define a structure for a memory area. Structs may contain other structs or pointers to other structs. A struct can be typedefined to a new name. This allows to use it without the explicit keyword struct. For most architectures, the compiler will add extra pad bytes to fullfill the alignment setting of dedicated memory section. This can be disabled by use of special compiler pragma __attribute__((__packed__)).

Example 6

```
struct Calculation_s {
    u8_t A; u8_t B; u8_t R; Mode_e M;
    struct Calculation_s* Next;  // points to another place of same struct
} __attribute__((__packed__)); // sizeof(struct Calculation_s) == 3

typedef Calculation_s Calculation_t;

// struct definition and typedef can be integrated into single statement
typedef struct Calculation_s {
    u8_t A; u8_t B; u8_t R; Mode_e M;
    struct Calculation_s* Next;  // points to another place of same struct
} __attribute__((__packed__)) Calculation_t;

// definition and initialization
Calculation_t Calc = { 1,2,3, NULL };
printf("A=%i, B=%i\n", Calc.A, Calc.B);

// Now Next points back to same structure
Calc.Next = &Calc;

// pointer to a struct
Calculation_t* Calc2 = &Calc1;
printf("A=%i, B=%i\n", Calc2->A, Calc2->B);
```

Interpreting memory differently

Embedded systems often communicate with other devices via different communication busses like USART, SPI, I2C, CAN, LIN and others. The native way of interpreting received binary messages is to copy them into a memory space called buffer and to direct a pointer of a certain type to this buffer. In reality things are more complex and a message can have one of many structures. A common used scheme is to provide a type field in the first byte of a message that identifies the struct to apply on it. The problem is: How big must a buffer be to store all supported message types? ANSI C knows how to declare a datatype that big that it can store any of a given list of datatypes. Such a declaration is called a union.

Example 7

```
typedef struct { u8_t Value; }
__attribute__((__packed__)) Value8_t;  // sizeof(Value8_t) == 1

typedef struct { u16_t Value; }
__attribute__((__packed__)) Value16_t; // sizeof(Value16_t) == 2

typedef struct { u32_t Value; }
__attribute__((__packed__)) Value32_t; // sizeof(Value16_t) == 4

typedef union  {
     Value8_t Value8;  Value16_t Value16;  Value32_t Value32;
} Value_u;                                 // sizeof(Value_u) == 4

typedef enum { TypeByte, TypeShort, TypeLong } ValueType_e;

typedef struct {
     ValueType_e ValueType;  Value_u Value;
} Message_t;

void foo(Message_t* Message) {
    u32_t Value = 0;
    switch (Message->ValueType) {
        case TypeByte:  Value = Message->Value.Value8;  break;
        case TypeShort: Value = Message->Value.Value16; break;
        case TypeLong:  Value = Message->Value.Value32; break;
        default: break;
}}
```

Exercise 2C:

- Extend your source from Exercise 2B
- Place A, B, R, M into one struct called Calculation_t
- Add field Next to Calculation_t that is a pointer to another Calculation_t struct
- Functions multiply(), add() and calc() stay unchanged
- create three instances of Calculation_t and create a linked list via their Next fields
  Calc1.Next = &Calc2; Calc2.Next = &Calc3; Calc3.Next = NULL;
- create calcList(Calculation_t* Calc1) to perform calculations on a given linked list of Calculation_t by calling calc() for each entry
- Let calcList() operate on Calc1, ..., Calc3

Dynamic Memories

The memory management techniques seen so far are all natively supported by C language. As C is a static language, these memories have to be declared at compile time and cannot be changed during runtime.
For many applications, a more felxible management is required. For desktop applications, the use of malloc()/ new() and free()/ delete() is common. Is it possible to use them for microcontrollers? How big is their overhead?

Example 8

```
// global array used for dynamic allocations
#define HEAP_SIZE 1000
u8_t Heap[HEAP_SIZE];
u8_t* NextFree = Heap;

void* allocMem(u8_t Size) {
    u8_t* NewMem = NULL;

    if (NextFree + Size < Heap + HEAP_SIZE) {
        NewMem = NextFree;
        NextFree += Size;
    }
    return NewMem;
}

typedef enum { Mode_Add, Mode_Multiply } CalcMode_e;

typedef struct Message_s {
     u8_t A;
     u8_t B;
     CalcMode_e M;
} Calculation_t;

void taskMain(void) {
    Calculation_t* Calc1 = (Calculation_t*) allocMem( sizeof(Calculation_t) );
    Calc1->A = 2;
    Calc1->B = 4;
    Calc1->M = Mode_Add;
}
```

Example 8 shows a simple implementation of a dynamic memory management that can only allocate memory but not free it. The shown implementation just requires two global pointers Heap and NextFree plus the size reserved for the Heap array. One big disadvantage of a dynamic memory management is that we don't exactly know the real usage of Heap[]. We could of course inspect the NextFree pointer during runtime. The problem is, that the memory usage can vary if the dynamic allocations depend on inputs from outside of the microcontroller.

Reusable Memories

The memory management shown in Example 8 is a very basic one. It does not allow to free an allocated memory block as it is common for desktop applications. The problem is: Freeing a memory block means to produce a hole of unused memory inside Heap[]. Start address and size of this hole have to be stored in an extra data structure. This structure gets bigger on every free(). A later allocMem() call would have to search the list of empty memory blocks. This makes

allocMem() much slower. If a block of required size is not found, then a larger block could be reused. The extra space then is wasted. Over time, the amount of available memory can reduce until Heap[] is exhausted and no more memory can be allocated. This effect is called memory fragmentation. Memory fragmentation can force an embedded system to stop working. - Can you think of a car that has to stop for a system reset after some hours of driving?
One solution to memory fragmentation is reusable, dynamic memory. Extra data is added to each memory block that stores size, usage and other data of the block. This extra data allows functions to operate on different sized memory blocks.

Example 9

```
typedef struct memory_s {
    u8_t MaxSize;           // maximum allowed bytes to be stored in Buffer[]
    u8_t Size;              // amount of bytes currently stored in Buffer[]
    struct memory_s* Next; // memory blocks can form a linked list
    u8_t* Buffer;           // points to start of memory block storing Calc
} memory_t;

memory_t* allocMem(u8_t Size) {
    memory_t* NewMem = NULL;
    Size += sizeof(memory_t); // take into account size of extra header

    if (NextFree + Size < Heap + HEAP_SIZE) {
        NewMem = (memory_t*) NextFree;

        // init header
        NewMem->MaxSize = Size
        NewMem->Size = 0;
        NewMem->Next = NULL;
        NewMem->Buffer =  NextFree + sizeof(memory_t);

        NextFree += Size;
    }
    return NewMem;
}
void main(void) {
    memory_t* Calculations1 = allocMem( 2 * sizeof(Calculation_t) );

    // using Buffer for several calculations
    Calculation_t* Calculation = (Calculation_t*) Calculations1->Buffer;
    Calculation->A = 6;
    Calculation->B = 7;
    Calculation->M = Mode_Add;
    Calculation++;
```

Exercise 2D:

- Extend your source with dynamic memory management
- create void calcMem(memory_t* Calculations) to operate on a linked list of memory_t blocks
- calcMem() shall call calcList() for every first Calculation_t entry in a memory block
- create three memory_t instances Calculations1, .. Calculations3 with 2,3 and 4 calculations each
- create a linked list of Calculations1, .. Calculations3
- let calcMem() run on this list and perform all nine calculations
- use gdb to display the content of the second Calculation_t entry in third memory_t by using only the pointer to the first memory_t

Intended Memory Layout