Programming Embedded Systems

Lecture 13

Overview of memory management

Monday Feb 27, 2012

Philipp Rümmer
Uppsala University
Philipp.Ruemmer@it.uu.se

Lecture outline

- Memory architecture of micro-controllers, in particular ARM CORTEX M3
- Memory management in embedded software
- Scoped memory in RTSJ

Memory architecture of micro-controllers

(in particular considering ARM CORTEX M3)

Architecture

- MCUs often use a Harvard approach
  - Separate memory + buses for code (ROM) and data (RAM)
  - Code is executed “in place” (not copied to RAM before execution)
- Some amount of RAM/ROM is integrated in MCU
  - Typically: much ROM, little RAM
- Can be extended using external memory if needed

Read-only memory (ROM)

- Used to store code + static data (e.g., lookup tables)
- Unrestricted reading, separate procedure for writing
- Most common (today): flash memory
- Other kinds: PROM, UV-EPROM, EEPROM
- STM32F10x has 16KiB – 1MiB of ROM

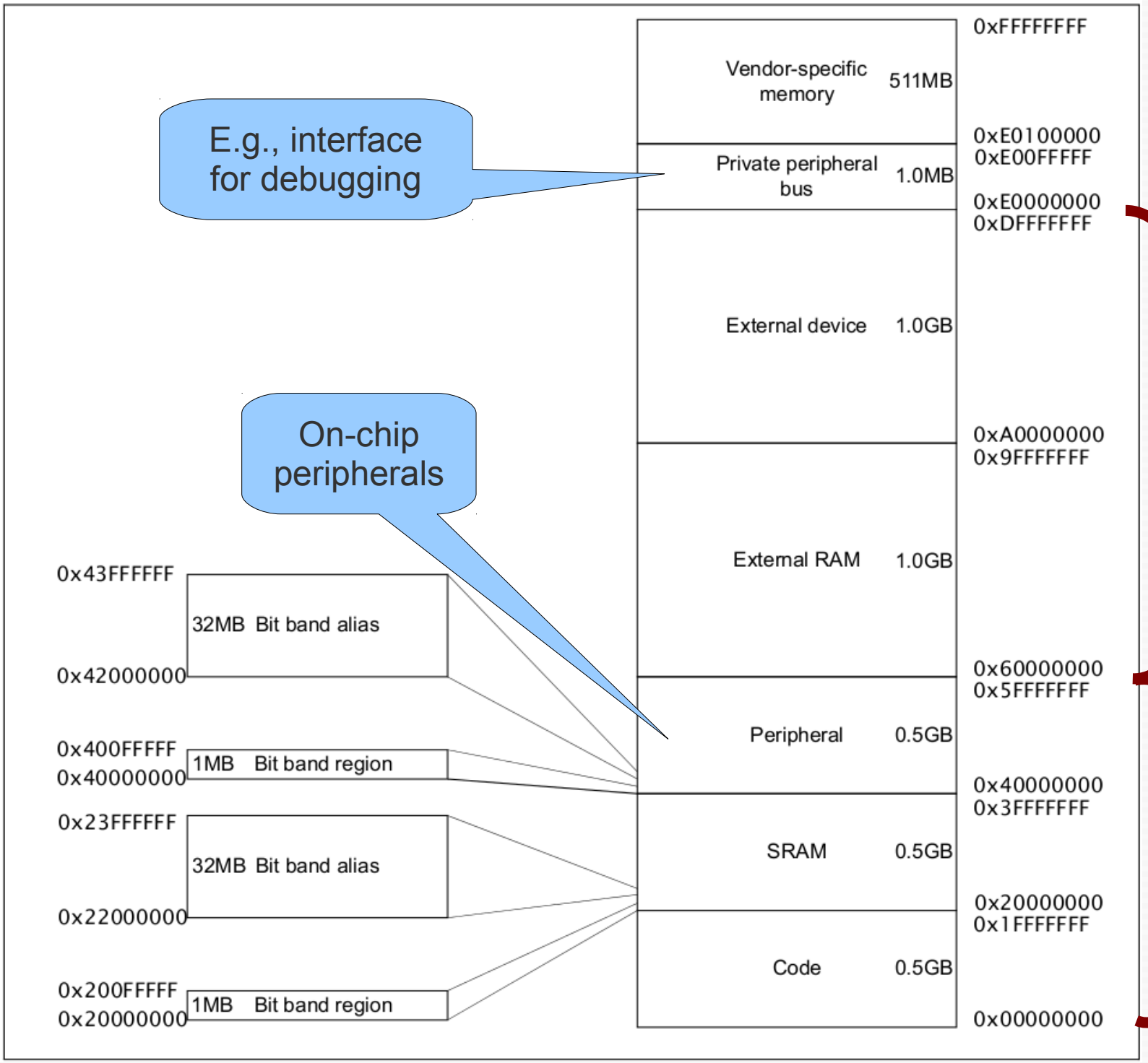
Random-access memory (RAM)

- Used for data (stack, heap)
  - In principle also for code, but uncommon
- Two main kinds: SRAM, DRAM
  → Both very common
- Often accessed through a hierarchy of caches
  - Not so common for internal RAM in MCUs; difficult WCET analysis
- STM32F10x has 4KiB – 96KiB of internal SRAM, no cache (but low latency)

CORTEX M3 memory map

- RAM + ROM + special-purpose regions are uniformly mapped into a 32bit address space
- Details depend of CORTEX M3 version

STM32F10x memory map



Bit band interface

- Motivation: accessing individual bits in memory is cumbersome
  - Multiple instructions to set/reset bit, not atomic
  - But very often necessary: access control bits, locks, etc.

Bit band interface (2)

- **Bit band alias:**
  32MiB region B that allows to access individual bits of a 1MiB region A
    - Every bit in A corresponds to a word (4 byte) in B
    - `0` ↔ `0x00000000`
    - `1` ↔ `0xFFFFFFFF`

Managing memory

Ways to manage memory

- At compile-time
- At startup-time
- Dynamically during runtime

Management at compile time

- Compiler/linker creates segments for different kinds of code/data
  - code (read-only, .text)
  - read-only data
  - read-write data
  - zero-initialised read-write data

put into ROM

- Further segments possible, as needed
- Code/data is directly flashed to MCU; RAM is initialised during start-up

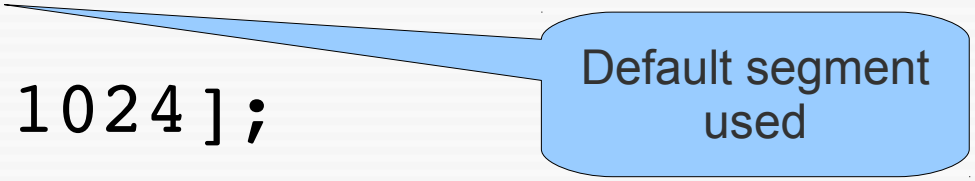
Management at compile time

- Mapping of data to segments can be automatic or explicit (compiler-specific)
    - `#pragma` to specify segment
- Layout of segments in memory can be chosen

In MDK-ARM

- Zero-initialised read-write data (→ .bss, RAM):

```
char rwData[1024];
```



- Read-write data (→ .data, RAM):

```
char rwData[3] = {1, 2, 3};
```

- Read-only data (→ .constdata, ROM):

```
const char roData[3] = {1, 2, 3};
```

In MDK-ARM (example)

```
int x1 = 5;                          // in .data (default)
int y1[100];                         // in .bss (default)
int const z1[3] = {1,2,3};           // in .constdata (default)
#pragma arm section rwdata = "foo", rodata = "bar"
int x2 = 5;                          // in foo (data part of region)
int y2[100];                         // in .bss
int const z2[3] = {1,2,3};           // in bar
char *s2 = "abc";                 // s2 in foo, "abc" in .conststring
```

In embedded systems

- *Most common way of memory management:*
  Allocate all required memory at compile-time (in read-write or read-only segments, as needed)
- Little risk of runtime errors due to memory management
  (e.g., it is checked at compile-time whether enough memory is present)

Dynamic memory management

- Manual management: `malloc`, `free`
- More flexible, might be necessary in some cases
    - E.g., to implement OS functions: allocate stack for tasks, task control blocks, queues, semaphores, etc.

Dynamic memory management (2)

- Can cause various problems in embedded systems
  - Possibly insufficient memory at runtime
  - Fragmentation
  - Implementation of `malloc`, `free` can be of substantial size
  - `malloc`, `free` thread-safe?
- Compromise: no `free`, `malloc` is only used during startup

Dynamic memory management (3)

- FreeRTOS comes with 3 allocators (programmer can choose)
  - Heap_1: only `malloc` is implemented, memory is never freed
  - Heap_2: also provides `free`
  - Heap_3: in addition, thread-safe implementation
- Size of managed heap is specified in FreeRTOSConfig.h:

```
#define configTOTAL_HEAP_SIZE		( ( size_t ) ( 17 * 1024 ) )
```

More high-level memory management?

Memory in Real-time Java

Real-time garbage collectors

- GCs for real-time systems are hard to get right
    - Interruptions by GC have to be scheduled correctly
    - GC must keep up with application
    - Overhead in memory consumption due to delayed deallocation
- Too large, complicated, not trustworthy
  → Mostly for soft real-time systems

Kinds of memory in RTSJ

- Heap memory
  - "Real-time" garbage-collected, otherwise same as in normal Java
- Immortal memory
  - Memory that is never reclaimed
- Scoped memory
  - Region-based memory management
  - Allocation using `new`, deallocation by deleting regions (reference-counting)
- Physical memory

Basic idea of scoped memory

- Organise memory in different **regions**
  e.g.,
  one region for long-living objects of a thread,
  one region for temporary objects of a method, ...
- Allocate memory as usual, but within regions (using `malloc`, `new`, etc.)
- Free memory by explicitly deleting whole regions

Memory Areas

- Areas are in RTSJ represented using class `MemoryArea`
- NB: each `MemoryArea` object is related to two memory regions:
  - The region in which the `MemoryArea` object **itself is allocated**
  - The region **represented** by the `MemoryArea` object
- Important MemoryArea methods: `enter, newInstance, newArray`

Scoped memory

- Every `RealTimeThread` runs in a **memory allocation context**
  - Refers to some `MemoryArea`
  - Chosen explicitly by programmer
  - Nested contexts possible → stack
- While in allocation context A, objects created with `new` are stored in A

Scoped memory (2)

- Contents of a scoped `MemoryArea` are cleared when the last thread exits the context
  (“reference counting”)
- `MemoryArea` can then be reused

Examples

- Simple use of memory scopes
- Nesting

Observations

- Scope nesting has to be acyclic
- References between scopes are restricted

Allowed references

Stored In	Reference to Heap	Reference to Immortal	Reference to Scoped	null
Heap	Permit	Permit	Forbid	Permit
Immortal	Permit	Permit	Forbid	Permit
Scoped	Permit	Permit	Permit, if the reference is from the same scope, or an outer scope	Permit
Local Variable	Permit	Permit	Permit	Permit

(Taken from “The Real-Time Specification for Java”, Version 1.0.2)

Typical use of scopes

- Identify self-contained computations (or parts)
- Estimate memory consumption of each computation
- Allocate sufficient amount of memory, using `LT/VTMemory`
- Execute computation within `enter`-block
- **NB: memory areas can be reused!**

Further reading

- Peter C. Dibble, “Real-Time Java Platform Programming”