Memory Architecture

Klaus-Peter Zauner

COMP2215: Computer Systems II
Harvard architecture
Separate memory for instructions and data.

Program bus:
- Flash 128 KB

Data bus:
- Registers (32×8 bit)
- SRAM 8 KB
- EEPROM 4 KB
Harvard architecture

Separate memory for instructions and data.

Program bus:
- Flash 128 KB

Data bus:
- Registers (32×8 bit)
- SRAM 8 KB
- EEPROM 4 KB
Von Neumann- vs. Harvard-architecture

Von Neumann:

- Memory
- Processor

No separation between data and program: CPU decides what to execute.

Harvard:

- Program Memory
- Processor
- Data Memory

Less bus congestion.
## Types of Memory on AVR microcontrollers

<table>
<thead>
<tr>
<th>Bus</th>
<th>Technology</th>
<th>Size</th>
<th>Persistent</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Flash</td>
<td>128 KB</td>
<td>yes</td>
<td>slow</td>
</tr>
<tr>
<td>Data</td>
<td>EEPROM</td>
<td>4 KB</td>
<td>yes</td>
<td>slow</td>
</tr>
<tr>
<td>Data</td>
<td>SRAM</td>
<td>8 KB</td>
<td>no</td>
<td>fast</td>
</tr>
</tbody>
</table>

EEPROM: Electrically Erasable Programmable Read Only Memory

SRAM: Static Random Access Memory
### Types of Memory on AVR microcontrollers

<table>
<thead>
<tr>
<th>Bus</th>
<th>Technology</th>
<th>Size</th>
<th>Persistent</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Flash</td>
<td>128 KB</td>
<td>yes</td>
<td>slow</td>
</tr>
<tr>
<td>Data</td>
<td>EEPROM</td>
<td>4 KB</td>
<td>yes</td>
<td>slow</td>
</tr>
<tr>
<td>Data</td>
<td>SRAM</td>
<td>8 KB</td>
<td>no</td>
<td>fast</td>
</tr>
</tbody>
</table>

Slow writes not only impact performance, but need to be considered also from a reliability perspective (e.g., power failure)

EEPROM: Electrically Erasable Programmable Read Only Memory

SRAM: Static Random Access Memory
Memory Technology I

SRAM

- Very fast
- Volatile
- Cell level random read/write
- Unlimited read/write
Memory Technology II

Flash

- Non-volatile
- Fast Cell-level Read
- Block write
- Write requires erase
- Limited number of writes \( \approx 100k? \)
Memory Technology III

EEPROM

- Non-volatile
- Cell-level Read
- Cell-level Write
- Limited number of writes $\approx 100k$?
Addressing

Flash, RAM and EEPROM memory, all have address spaces staring at $0000.
RAM Layout

low addresses

.data

.bss

.heap ↓

high addresses

.stack ↑
Heap/Stack

RAM not taken up by the global variables and the local static variables is shared dynamically between the heap and the stack.

Heap

- Used to serve explicit runtime memory requests
- Library functions may use this (io.h)

Stack

- Required for function calls
- Starts at the top of the RAM, grows down
RAM layout

- No memory management unit (hardware)
- Stack and Heap can collide
- Overwriting variables unlikely
Heap/Stack Collision

- Stack growth
  - recursive function calls
  - large local variables
  - many local variables
- Heap growth
  - beware of heap fragmentation
  - watch out for libraries
Heap/Stack Collision

- Stack growth
  - recursive function calls
  - large local variables
  - many local variables

- Heap growth
  - beware of heap fragmentation
  - watch out for libraries
Heap/Stack Collision

- Stack growth
  - recursive function calls
  - large local variables (e.g., image array)
  - many local variables

- Heap growth
  - beware of heap fragmentation
  - watch out for libraries
Heap/Stack Collision

- Stack growth
  - recursive function calls
  - large local variables
  - many local variables

- Heap growth
  - beware of heap fragmentation
  - watch out for libraries

(e.g., image array)
Heap/Stack Collision

- **Stack growth**
  - recursive function calls
  - large local variables
  - many local variables

- **Heap growth**
  - beware of heap fragmentation
  - watch out for libraries

(e.g., image array)
Heap/Stack Collision

- **Stack growth**
  - recursive function calls
  - large local variables
  - many local variables

- **Heap growth**
  - beware of heap fragmentation
  - watch out for libraries (e.g., image array)
Heap/Stack Collision

- Stack growth
  - recursive function calls
  - large local variables
  - many local variables

- Heap growth
  - beware of heap fragmentation
  - watch out for libraries
  (e.g., image array)
Handling Memory in C

1. `malloc(SIZE)`
   Request memory from the heap

2. Check whether the memory has been granted
   - `malloc()` returns pointer to assigned memory
   - `NULL` pointer $\Rightarrow$ out of memory

3. `free(POINTER)`
   Return memory to the heap
   - It’s ok to `free(NULL)`
Handling Memory: Speed

- Calls to `malloc()` are relatively slow
- Memory allocation is critical for performance
- No checks are made → would just waste cycles most of the time
Handling Memory: don’t be sloppy...

▶ Always check whether memory has been granted before using it
▶ Free the memory you have been assigned as early as possible
▶ Only call free on addresses you are in charge of
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free on addresses you are in charge of
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free on addresses you are in charge of
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free only addresses you are in charge of

Not following these rules leads to:

- Immediate Crashes
  - Best case!
- Randomly occurring crashes
  - Hard to debug
- Memory Leaks
  - Only noticeable after long runtime
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free only addresses you are in charge of

Not following these rules leads to:

- **Immediate Crashes**
  - Best case!
- Randomly occurring crashes
  - Hard to debug
- Memory Leaks
  - Only noticeable after long runtime
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free only addresses you are in charge of

Not following these rules leads to:

- **Immediate Crashes**
  - Best case!
  - Randomly occurring crashes
  - Hard to debug
- **Memory Leaks**
  - Only noticeable after long runtime
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free only addresses you are in charge of

Not following these rules leads to:

- **Immediate Crashes**
  - Best case!
- **Randomly occurring crashes**
  - Hard to debug
- **Memory Leaks**
  - Only noticeable after long runtime
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free only addresses you are in charge of

Not following these rules leads to:

- **Immediate Crashes**
  - Best case!
- **Randomly occurring crashes**
  - Hard to debug
- **Memory Leaks**
  - Only noticeable after long runtime
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free only addresses you are in charge of

Not following these rules leads to:

- Immediate Crashes
  - Best case!
- Randomly occurring crashes
  - Hard to debug
- Memory Leaks
  - Only noticeable after long runtime
Handling Memory: don’t be sloppy...

- Always check whether memory has been granted before using it
- Free the memory you have been assigned as early as possible
- Only call free only addresses you are in charge of

Not following these rules leads to:

- **Immediate Crashes**
  - Best case!
- **Randomly occurring crashes**
  - Hard to debug
- **Memory Leaks**
  - Only noticeable after long runtime
How is malloc implemented?

- Record size of allocated blocks
  - 2 Bytes pre-pended
- Maintain **free list**
  - linked with addresses in free block
- `malloc()`
  - Find best match in freelist that can be used
- `free()`
  - Merge adjacent blocks