Microcontroller programming

Fuszenecker Róbert
2011.

Embedded systems

An embedded system is a

- computer system
- designed to perform one or few dedicated functions
- often with real-time computing constraints.

Definition: from Wikipedia.

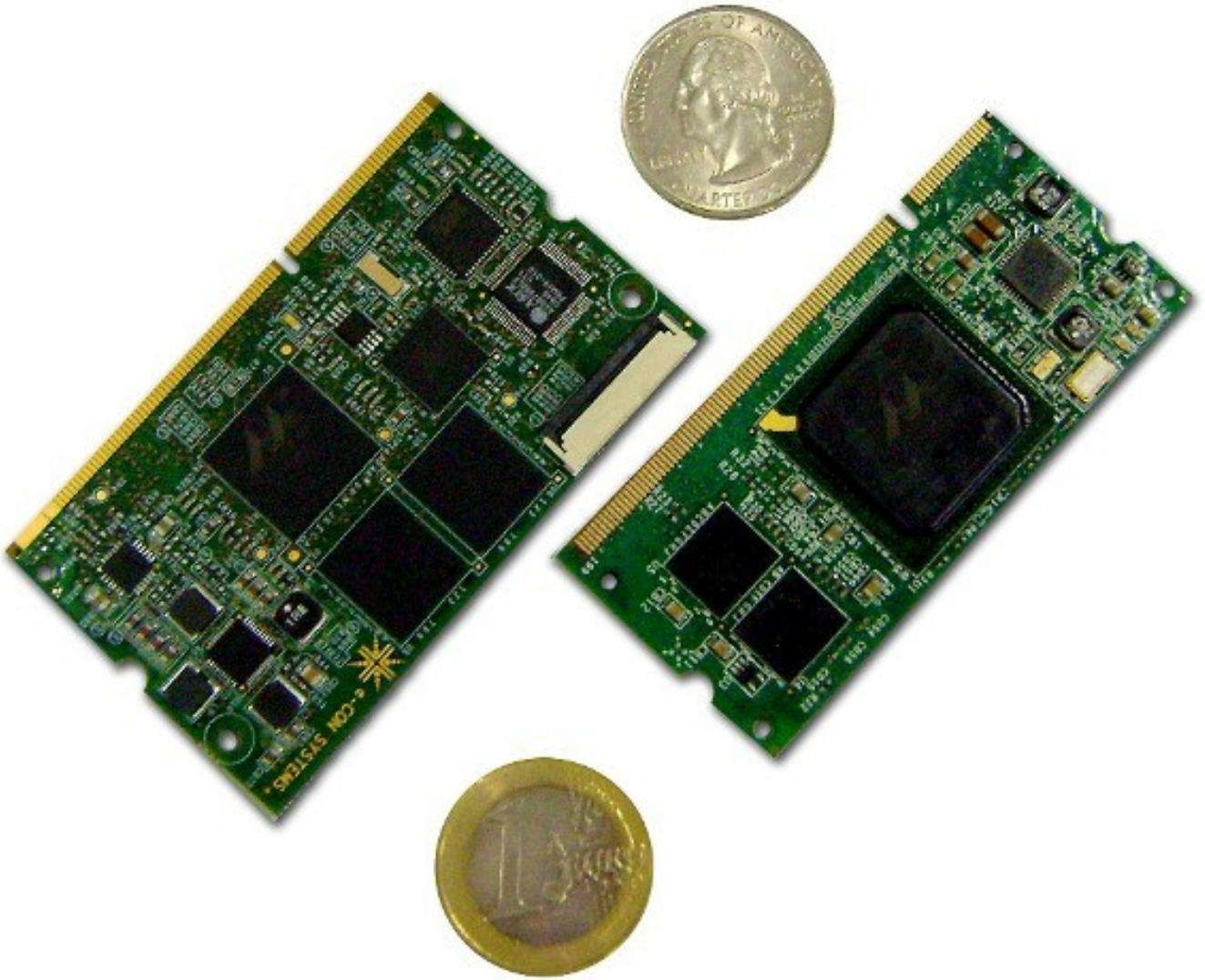
Embedded systems

Typical types of embedded systems:

- Industrial computer,
- System on module, computer on module,
- Microcontroller,
- FPGA.

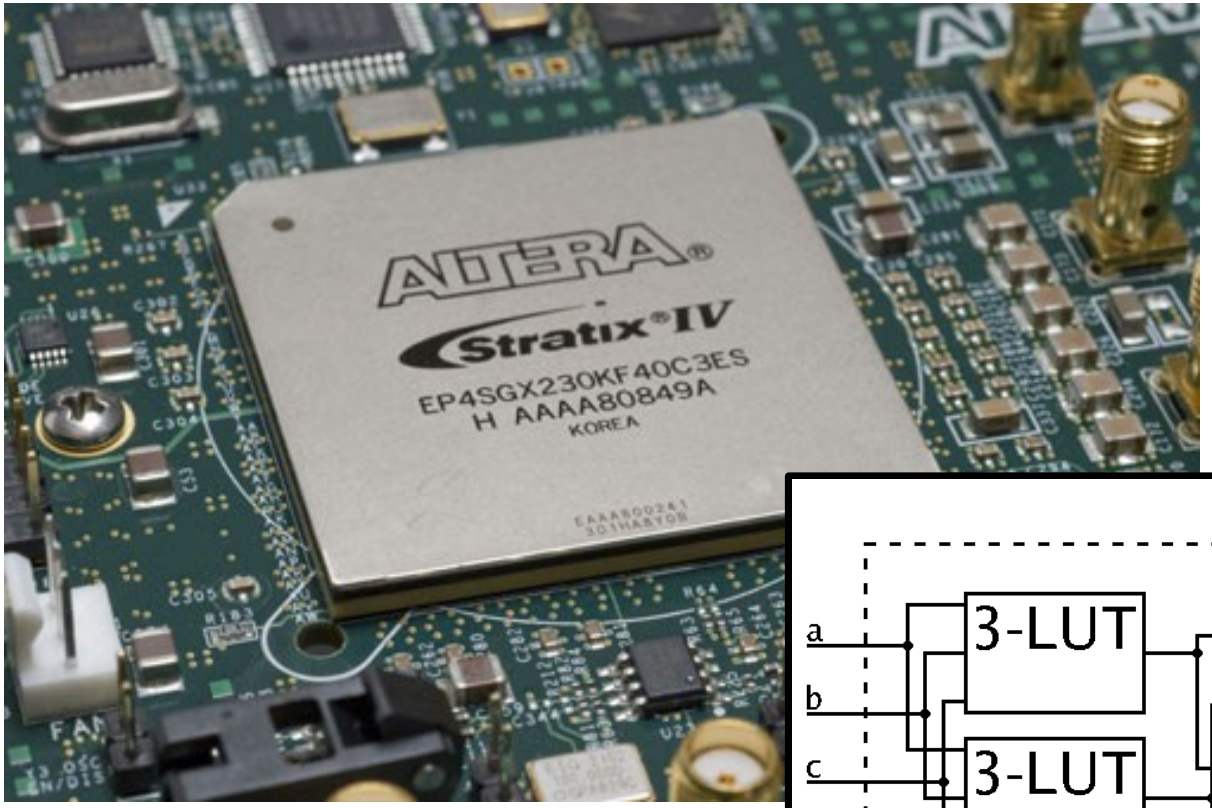
Industrial computers

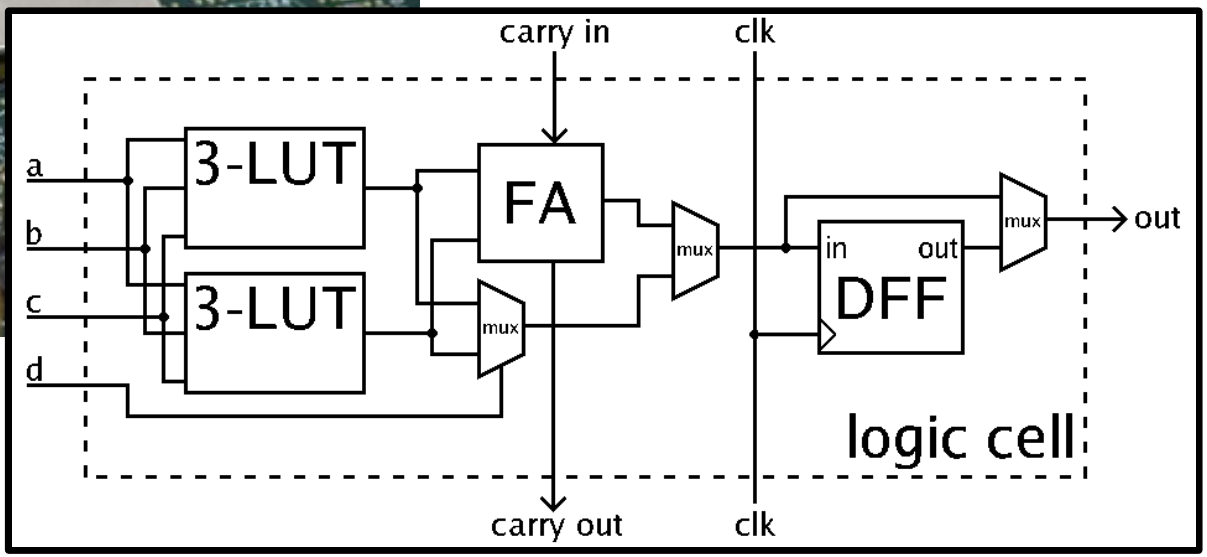
System on modules



Microcontrollers

FPGAs





FPGA: Field Programmable Gate Array

Embedded operating systems

- Industrial computers and system-on-modules are based on 32-bit processors (with MMU) and have enough resources to run operating systems, i.e. Linux or Windows CE.
- Microcontrollers sometimes run real-time operating systems: FreeRTOS, QNX Neutrino, etc.
- FPGAs generally don't run anything,
  in certain cases they realize soft microprocessors that may run Linux: MicroBlaze, NIOS 2.

ARM cores

ARM Holdings don't produce any silicon chips, they design only the core of the processors.

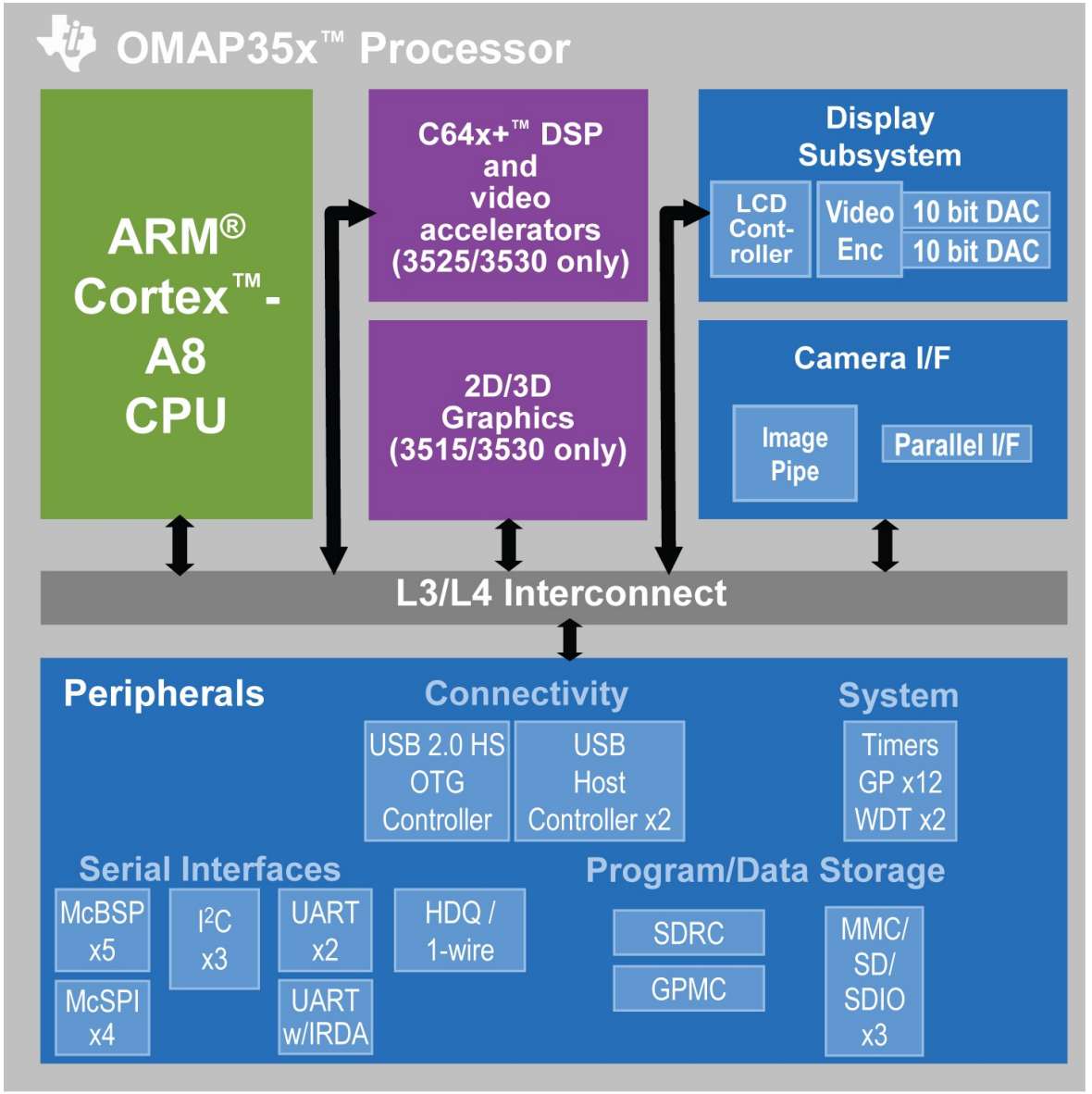
ARM: Advanced RISC Machine.

ARM cores

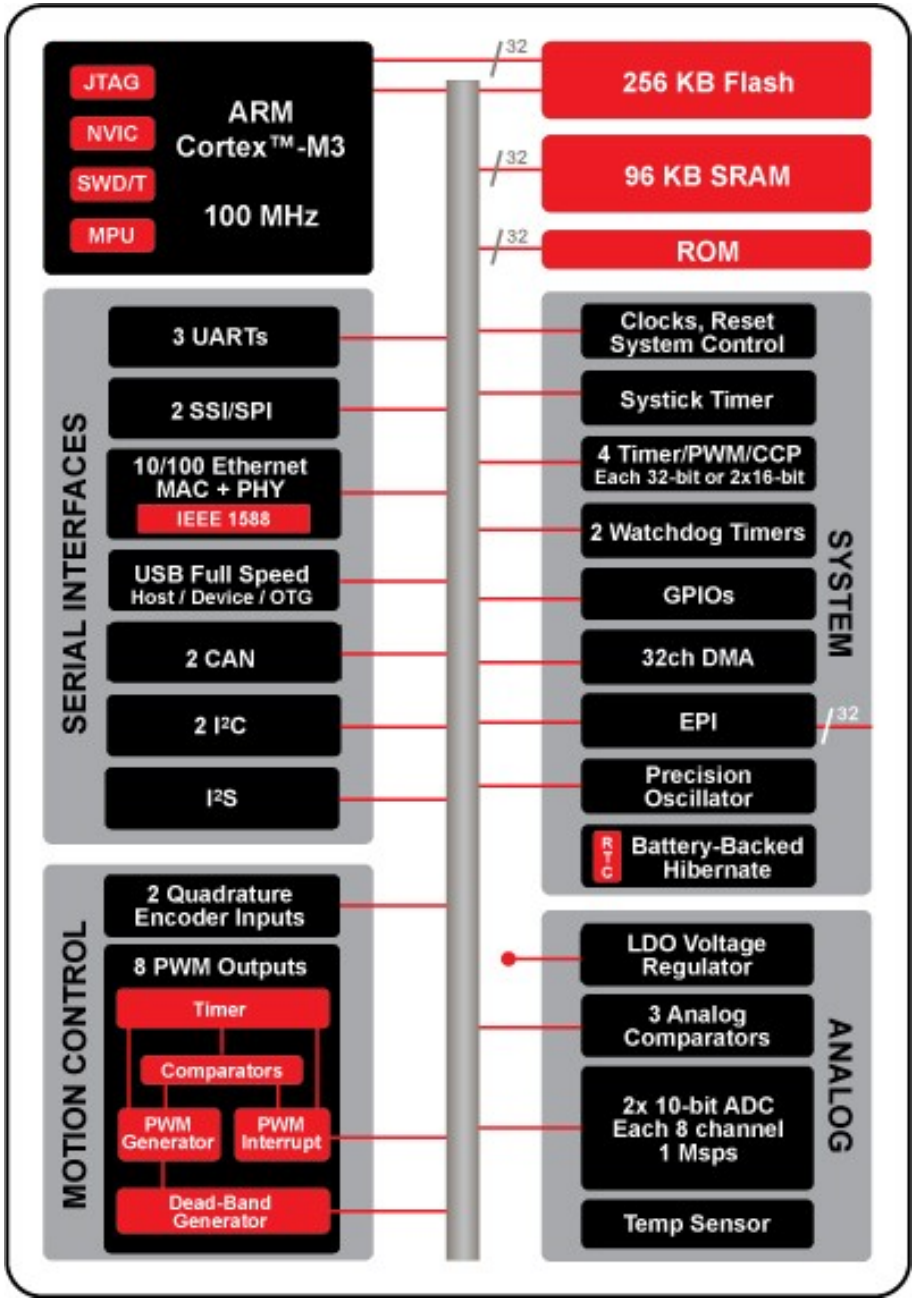
There are 3 kinds of ARM cores:

- Application processors (can run Linux).
  - Cortex-A8 (single core), Cortex-A9 (dual or quad core),
  - Cortex-A15 (quad core, servers),
- Microcontroller cores.
  - Cortex-M3, Cortex-M1 (low power), Cortex-M0 (FPGA),
- Obsolete cores (not recommended for new designs),
  - ARM7, ARM9, ARM11.

ARM cores (Cortex-A8)



ARM cores (Cortex-M3)



Processor modes (Cortex-A)

- Abort Mode,
- Fast Interrupt Mode (FIQ),
- Interrupt Request Mode (IRQ),
- Supervisor Mode (kernel),
- System Mode (similar to User Mode),
- Undefined Mode,
- User Mode (user space applications),
- Monitor Mode (debug).

Processor modes (Cortex-M)

- Thread mode (to execute application software),
- Handler (interrupt and exception handlers),
- Supervisor mode (after RESET).

Generally, “thread mode” is not used in embedded systems.

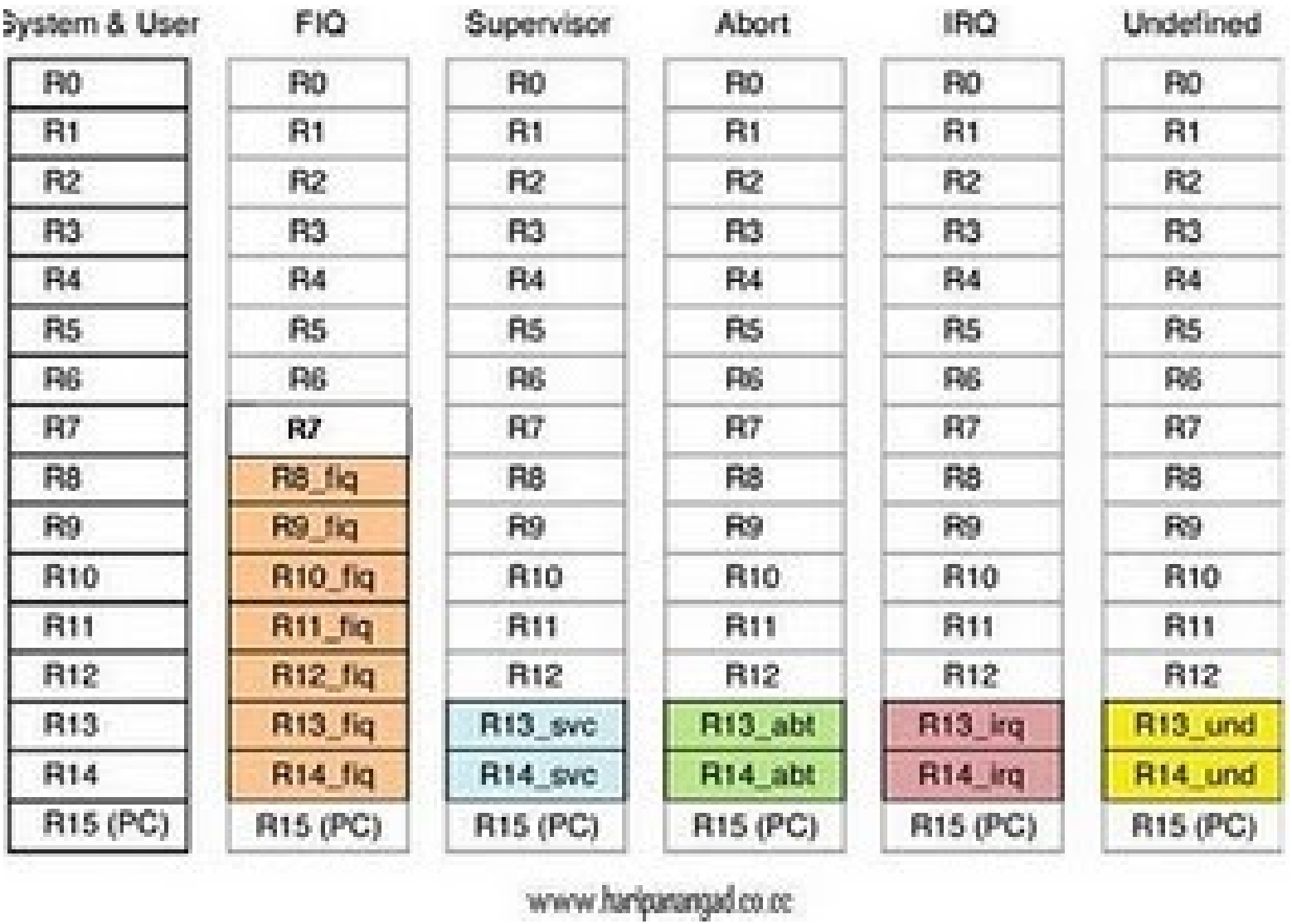
Register set

ARM processors have 16 × 32 bit registers, many of them has special functions:

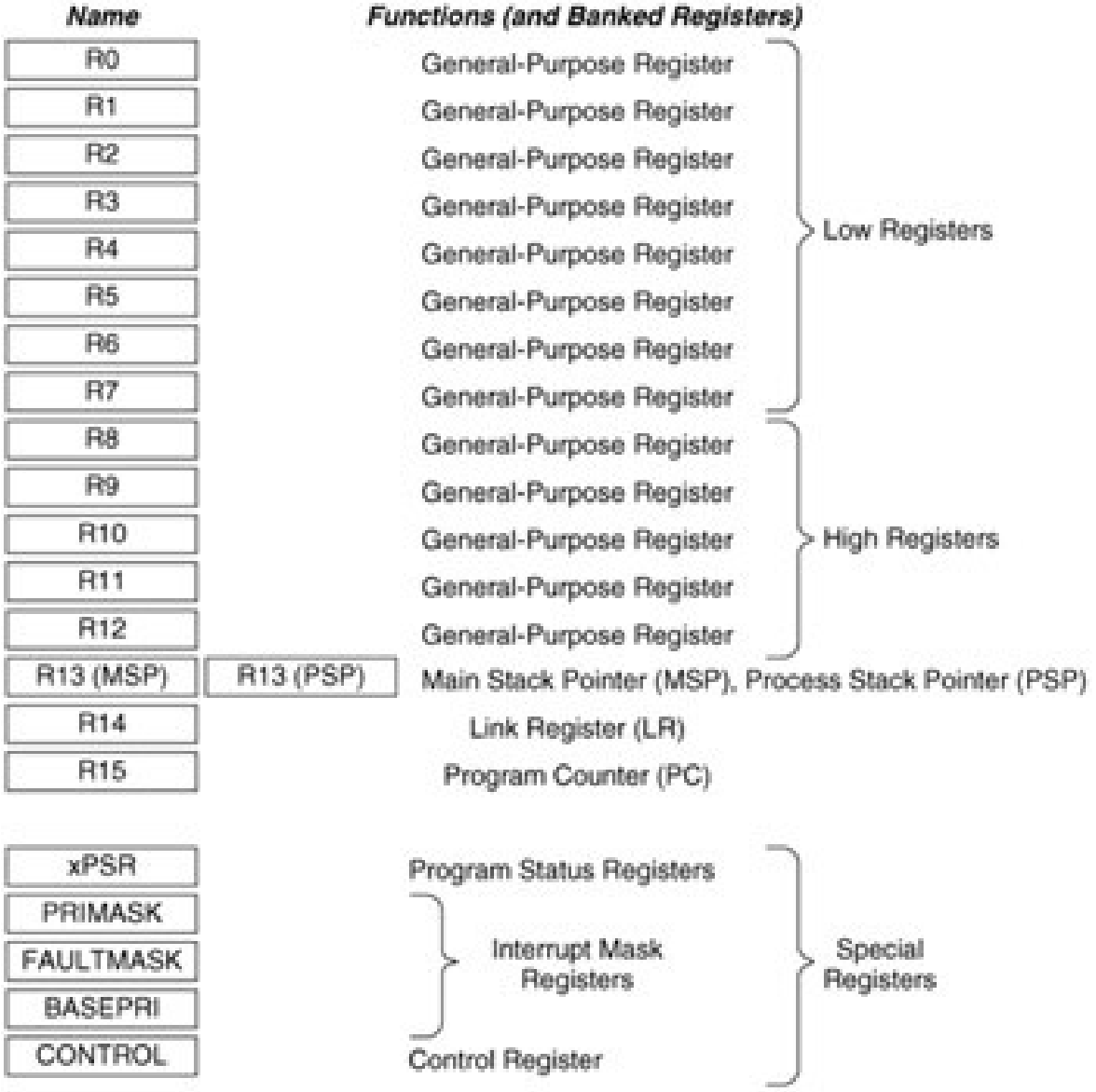
- R15: instruction pointer,
- R14: link register,
- R13: stack pointer,
- R11: frame pointer (optional).

The other ones are general purpose registers.

Register set (Cortex-A)



Register set (Cortex-M)



Instruction set

- **ARM Cortex-A** cores support both ARM (32 bit) and THUMB (16 bit) instructions (as well as many floating point, DSP and SIMD instructions).
- **ARM Cortex-M** cores support THUMB2 (16 or 32 bit) instructions only.

THUMB2 instructions are the 16 or 32 bit equivalent of the 32-bit ARM instructions. They are used to save memory in resource-critical applications (µC).

Instruction set

- ARM instruction example (always encoded as a 32-bit word):

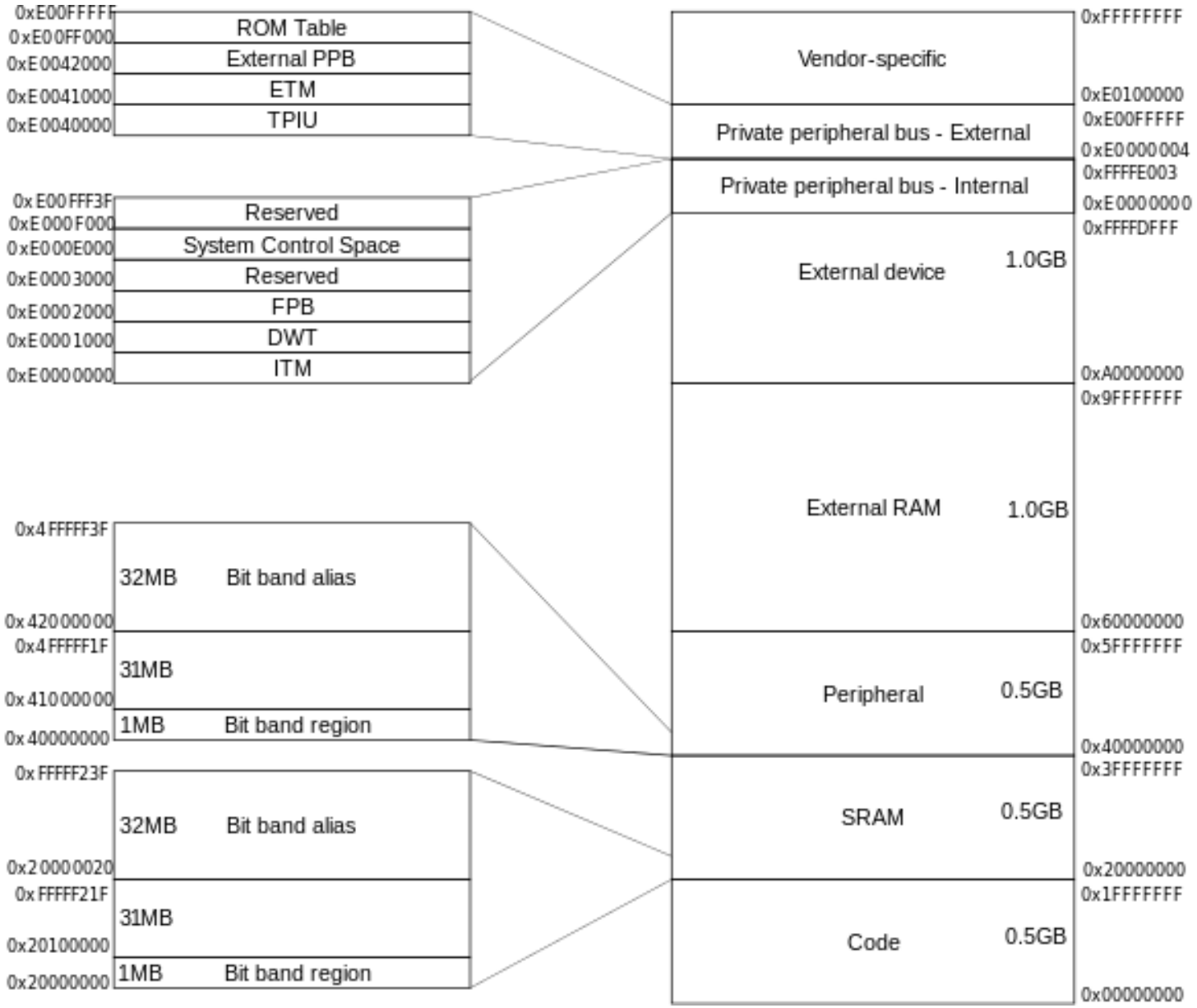
e08631a7 add r3, r6, r7, lsr #3

- THUMB2 instruction examples (16 or 32 bit, depends on the instruction itself):

683a ldr r2, [r7, #10]

fbc3 1204 smlal r1, r2, r3, r4

Memory map (Cortex-M)



Interrupt vectors (Cortex-A)

Physical Address	Vector name
0x0000'0000	**Reset**
0x0000'0004	Undefined instruction
0x0000'0008	Software interrupt
0x0000'000C	Prefetch abort
0x0000'0010	Data abort
0x0000'0014	(not used)
0x0000'0018	IRQ
0x0000'001C	FIQ

Interrupt vectors (Cortex-M)

Physical Address	Vector name
0x0000'0000	**Initial SP**
0x0000'0004	**Reset Handler**
0x0000'0008	NMI Handler
0x0000'000C	Hard Fault Handler
0x0000'0010	Memory Management Handler
0x0000'0014	Bus Fault Handler
0x0000'0018	Usage Fault Handler
...	...
0x0000'0040	Hardware Interrupt 0
0x0000'0044	Hardware Interrupt 1
...	...
0x000'003FC	Hardware Interrupt 243

Booting the processor (Cortex-A)

- After the RESET event the **Cortex-A*** processor starts to execute the **instruction** that can be found at address 0x0000'0000.
- This is typically an unconditional jump to the ROM code that loads the “real” boot loader.

```
00000000    ldr pc, [pc, #ROM_CODE]
```

- The boot loader should be able to load and start the Linux kernel or anything else.

Booting the processor (Cortex-M)

- The 0th element of the vector table is the initial value of the stack pointer: usually the last addressable cell of the internal RAM.
- The 1st item is the RESET vector, this is an **ADDRESS** that points to the “main” function.

Booting the processor (Cortex-M)

```
__attribute__ ((section(".interrupt_vector")))

void (*const Vectors[])(void) = {
    (void *)0x20003FFC  // The initial stack pointer
    Reset_Handler,      // The reset handler
    ...                 // Other interrupt handlers
};


extern int main(void);

void Reset_Handler(void) {
    While(1) {
        main();
    }
}
```

The main() function

- Since the stack pointer is set up by the hardware, the processor core is ready to start the “real” task.
- Typically the main() function
  - sets up the hardware peripherals (by using the Firmware library),
  - the interrupt controller,
  - optionally the Memory Protection Unit,
  - and waits for interrupts (low power mode).

Memory mapped I/O

- Several “memory cells” are directly connected to hardware peripherals.
- Hardware registers can be accessed by (memory) read and write instructions, similarly to RAM operations.
- Hardware registers can be protected by the Memory Management Unit (permission control) or the Memory Protection Unit (MPU).

Accessing HW registers

```
#define ADC_BASE_ADDRESS        0x4000A000
#define ADC_START_CONVERSION    0x00000004

typedef struct {
    uint32_t  adc_status;
    uint32_t  adc_value;
    uint32_t  adc_error;
} adc_hw_t;

adc_hw_t *adc0 = (adc_hw_t *)ADC_BASE_ADDRESS;
adc0->adc_status |= ADC_START_CONVERSION;
return_value = adc0->adc_value;
```

Firmware library

- The firmware library is provided by the chip manufacturer (hardware-dependent).
- Contains high level interface to hardware registers:
  - Names instead of addresses (#define),
  - Functions to manipulate the HW registers.
- Sometimes a digital signal processing library is included, so that the programmer doesn't need to work with assembly instructions.

Firmware library example

```
int main()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPadConfigSet(GPIO_PORTA_BASE, GPIO_PIN_3,
        GPIO_STRENGTH_8MA, GPIO_PIN_TYPE_OD);

    GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_3);
```

Firmware library example

```
    SysTickPeriodSet(SYS_TICK_PERIOD);
    SysTickIntRegister(systick_handler);
    SysTickIntEnable();
    SysTickEnable();
    IntMasterEnable();

    for (c = 0; ; c++) {
        counter += c;
        asm("wfi");
    }
}
```

Firmware library example

```
volatile unsigned int pin_data = 0;
volatile int counter;

void systick_handler()
{
    pin_data = GPIO_PIN_3 - pin_data;
    GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_3, pin_data);
    counter++;
}
```

With “volatile”

```
for (c = 0;; c++) {
  counter += c;
   104: 681a        ldr r2, [r3, #0]
   106: 18a2        adds  r2, r4, r2
   108: 601a        str r2, [r3, #0]
  asm("wfi");
   10a: bf30        wfi
   10c: f104 0401   add.w r4, r4, #1
   110: e7f8        b.n 104 <main+0x64>
```

R2: counter

R3: &counter

R4: c

Without “volatile”

```
for (c = 0;; c++) {
  counter += c;
  asm("wfi");
    fc: bf30          wfi
    fe: e7fd          b.n fc <main+0x5c>
```

Compiling the sources

- Commercial compilers:
  - ARM RVDS,
  - Keil,
  - CrossWorks,
  - etc.
- Open source compilers:
  - gcc,
  - clang/llvm.

Compiling the sources (gcc)

With cross-compiled gcc:

- Compiling the source code:

```
gcc -march=arm -c -o crt.o crt.c
```

- Linking the objects:

```
ld -T cm3.ld -o fw crt.o \
      main.o fwlib.a
```

- Converting to the appropriate format

```
objcopy -O binary fw fw.bin
```

Compiling the sources (LLVM)

With LLVM (not cross-compiled):

- Compiling the C source code:

```
clang -emit-llvm -c main.c
```

- Translating the LLVM object:

```
llc -march=thumb -mcpu=cortex-m3 \
        -o main.s main.ll
```

- Compiling the assembly source:

```
as -o main.o main.s
```

Compiling the sources (LLVM)

- Linking the object files and the driver library (the same as the gcc way):

```
ld -T cm3.ld -o fw crt.o \
      main.o fwlib.a
```

- Generating binary output file (FLASH image):

```
objcopy -O binary fw fw.bin
```

Behind the scenes: linking

- There are sections in the object files, i.e.:
  - Code (**.code**; the instructions to be run)
  - Data (**.data, .bss**; global and static local variables)
  - Read-only data (**.rodata**; const values, initial values of global variables)
  - Debug information (**.debug_info**, **.debug_line**, ...)
  - Interrupt vectors section (**.interrupt_vector** – **defined by us**:

```
__attribute__((section(".interrupt_vector")))
```

Behind the scenes: linking

- Sections in executable files:
  - Sections of the object files should be merged,
  - Sections should be reordered according to the linker script,
  - Function calls, variable access should be resolved,
- Linker scripts
  - Defines the sections (and their order) of the executable files.

Linker scripts

```
MEMORY
{
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x40000
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x10000
}
```

Linker scripts

```
SECTIONS
{
    .text :
    {
       *(.stack_pointer)
       *(.interrupt_vector)
       _text = .;
       *(.text .text.*)
       *(.rodata .rodata*)
       _etext = .;
    } > FLASH


    .bss : AT (_etext)
    {
       _data = .;
       *(.data .data.* .bss .bss.*)
       _edata = . ;
    } > SRAM
```

After linking

```
cortex-m3:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00002674  00000000  00000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 vtable        0000011c  20000000  00002674  00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000008  2000011c  00002790  0001011c  2**2
                  ALLOC, LOAD, DATA
```

Debug

- We can use a simulator to debug our code:
  - gdb's built-in simulator (emulates a processor),
  - qemu (emulates a complete device),
- Or we can use the target hardware to debug the firmware; we need
  - a program to download the FLASH content (firmware),
  - a gdb-proxy (which communicates with the hardware),
  - the gdb (GNU's debugger; ARM version),
  - optionally an IDE.

Debugger operations

- Step-by-step C/C++ code analysis
- Breakpoints on instructions
- Watchpoints on variables
- Step-by-step assembly code analysis
- Memory and register dump

Programming languages

Can we use programming languages other than C?

- Industrial computers and system-on-modules boot Linux or Windows CE, thus the answer is: yes.
- C++(0x) code for microcontrollers is not rare, but C# (.NET micro framework) is not really used.
- If the FPGA realizes a soft processor, any languages can be used.

Difficulties with C++

- No memory management
  [`operator new()` and `operator delete()` can be implemented],
- Exception handling is problematic,
- Virtual function calls are difficult to implement,
- GNU's libstdc++ is not enough resource-efficient,
  - But we can use CLANG's libstdcxx instead of GNU's libstdc++.

Questions?

Thank you for your attention!

References

- Cortex™-A8 Technical Reference Manual (Revision: r3p2)
- Cortex™-M3 Technical Reference Manual (Revision r2p1)
- ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition
- ARMv7-M Architecture Reference Manual