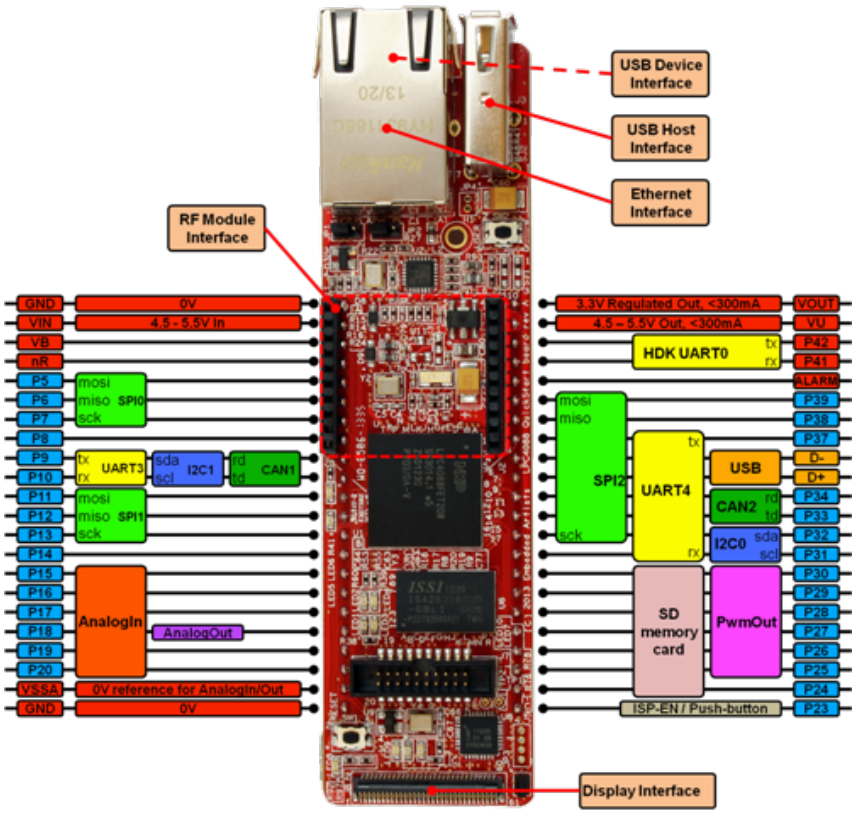
Operating systems and concurrency B02

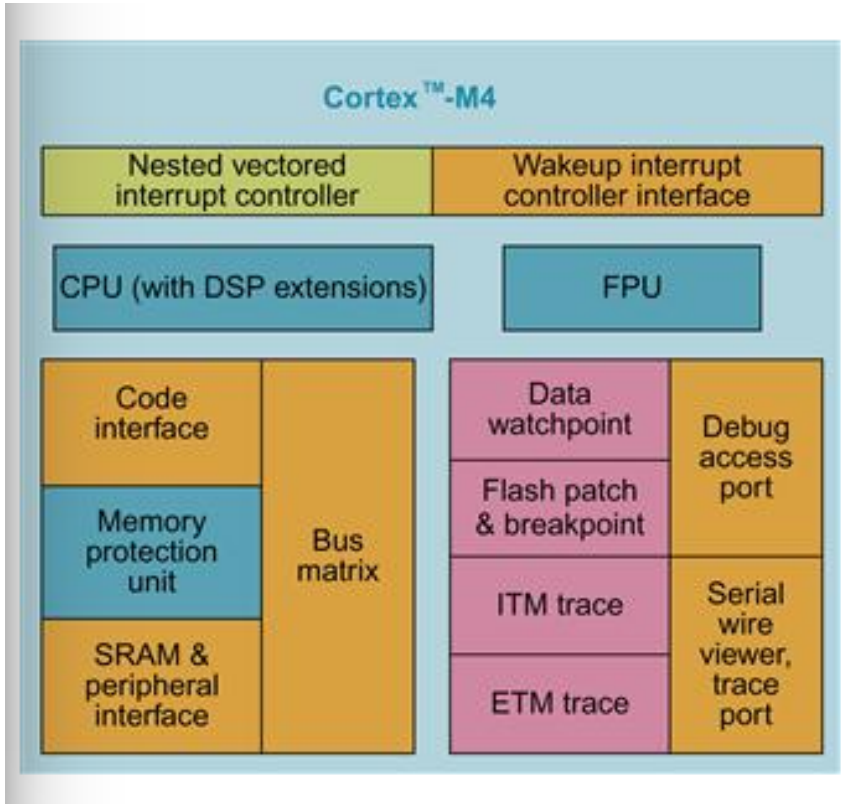
David Kendall

Northumbria University

Architecture of the ARM Cortex M4

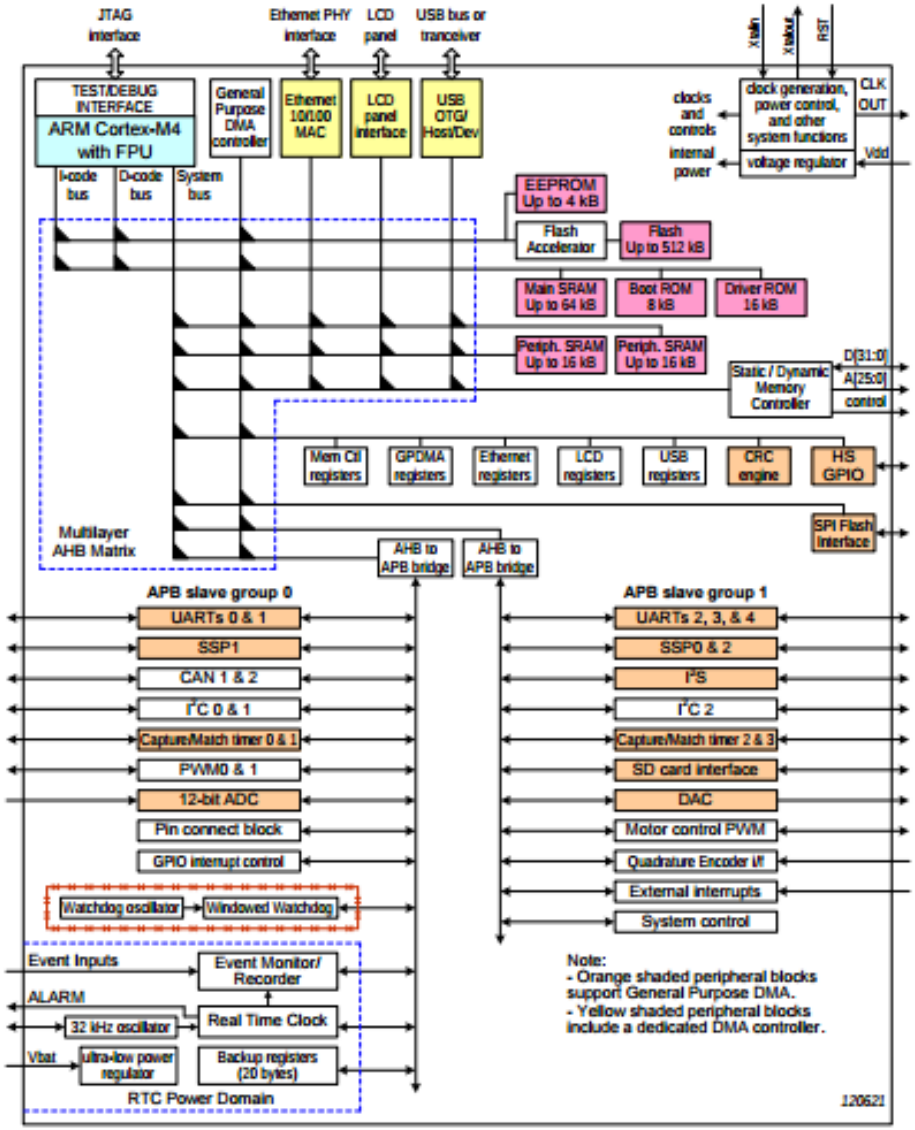
An OS acts as a *hardware abstraction layer*.
Let's look at the structure of some real hardware from which we can abstract.





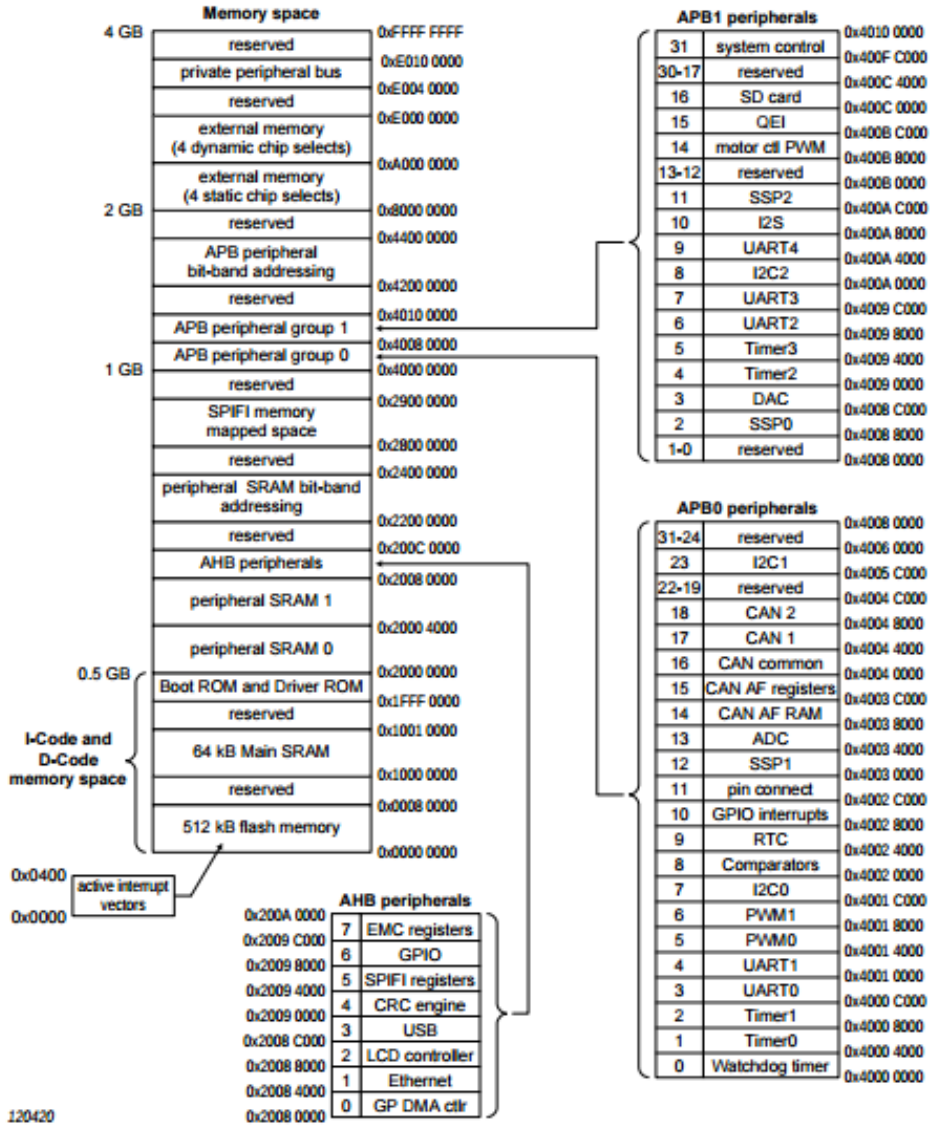
Martin, T. *The Designer's Guide to the Cortex-M Processor Family: A Tutorial Approach*, Newnes, 2013

LPC4088 Block Diagram



NXP, UM10562 LPC408x/407x User manual, NXP Semiconductors, 2014 (p.13)

LPC4088 Memory Map



NXP, UM10562 LPC408x/407x User manual, NXP Semiconductors, 2014 (p.16)

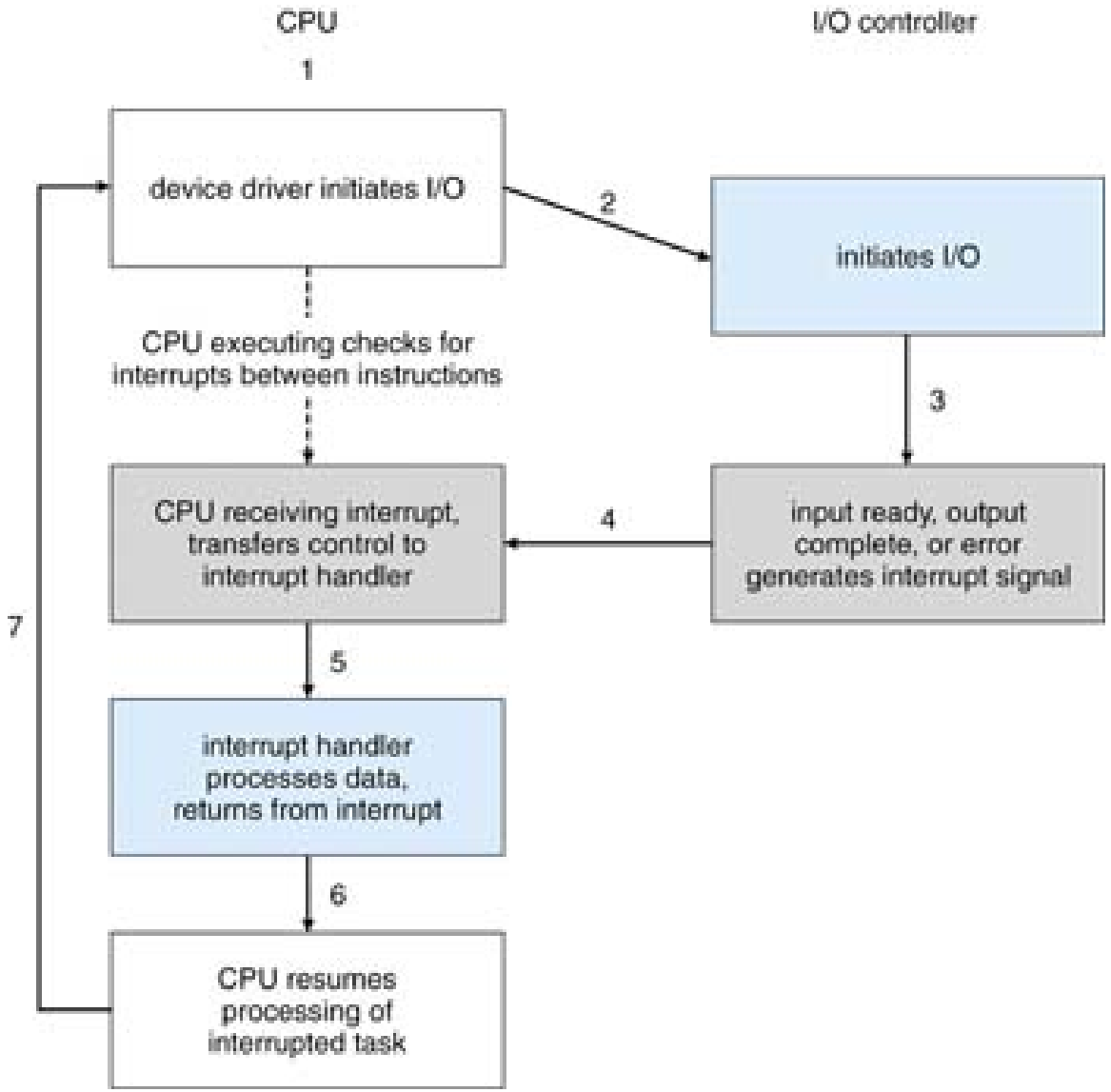
Device handling - Polling

- Typical I/O operation
  1. CPU repeatedly tests status register to see if device is busy
  2. When not busy, CPU writes a command into the command register
  3. CPU sets the command-ready bit
  4. When device see command-ready bit is set, it reads the command from the command register and sets the busy bit in the status register
  5. When device completes I/O operation, it sets a bit in the status register to indicate the command has been completed
  6. CPU repeatedly tests status register, waiting for command to be completed
- Problem: *busy-waiting* at steps 1 and 6

Device handling - Interrupt-driven

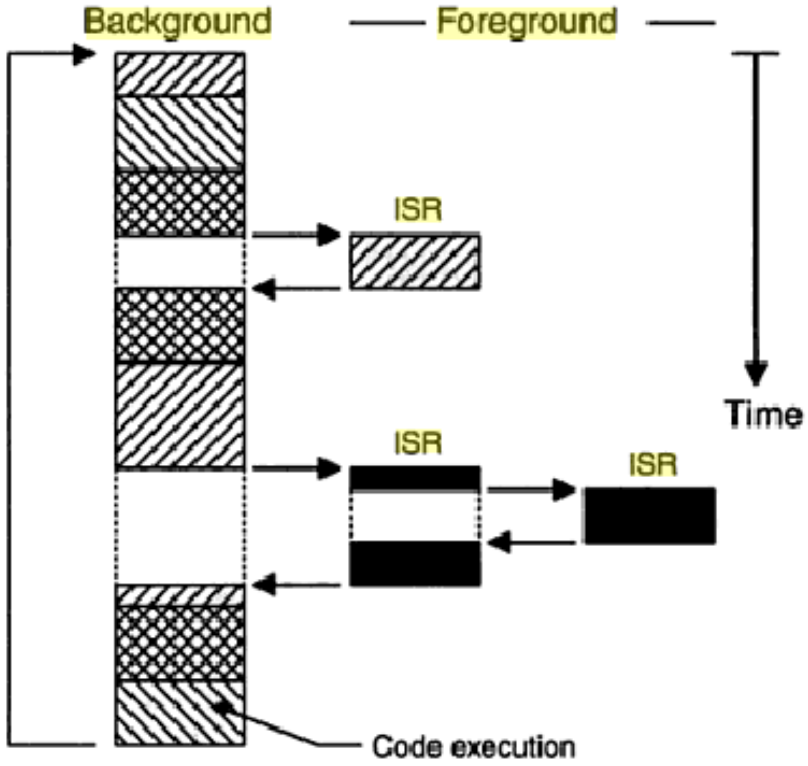
- Busy-waiting can be an inefficient use of the CPU
- CPU could be doing other useful computation instead of waiting
- E.g.
  - Assume: 10 ms for a disk I/O operation to complete; CPU clock speed of 120 MHz; average instruction requires 1 clock cycle – (rough estimates)
  - How many instructions could the CPU execute instead of waiting for the disk I/O?
- So, instead of waiting, CPU performs other useful work and allows the device to *interrupt* it, when the I/O operation has been completed

Interrupt-driven I/O cycle



Simple interrupt-driven program structure

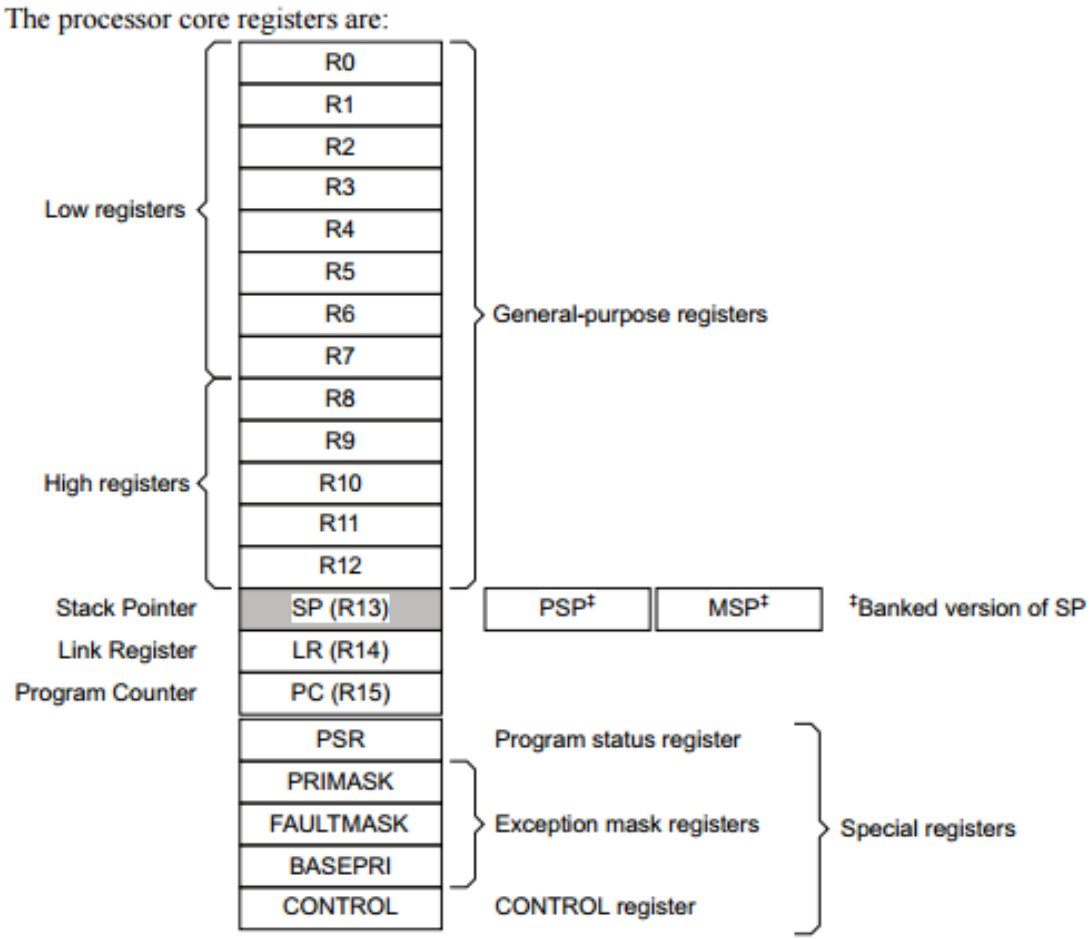
- Foreground / Background
- *Background*: Main (super) loop calls functions for computation
- *Foreground*: Interrupt service routines (ISRs) handle asynchronous events (interrupts)



Context switch

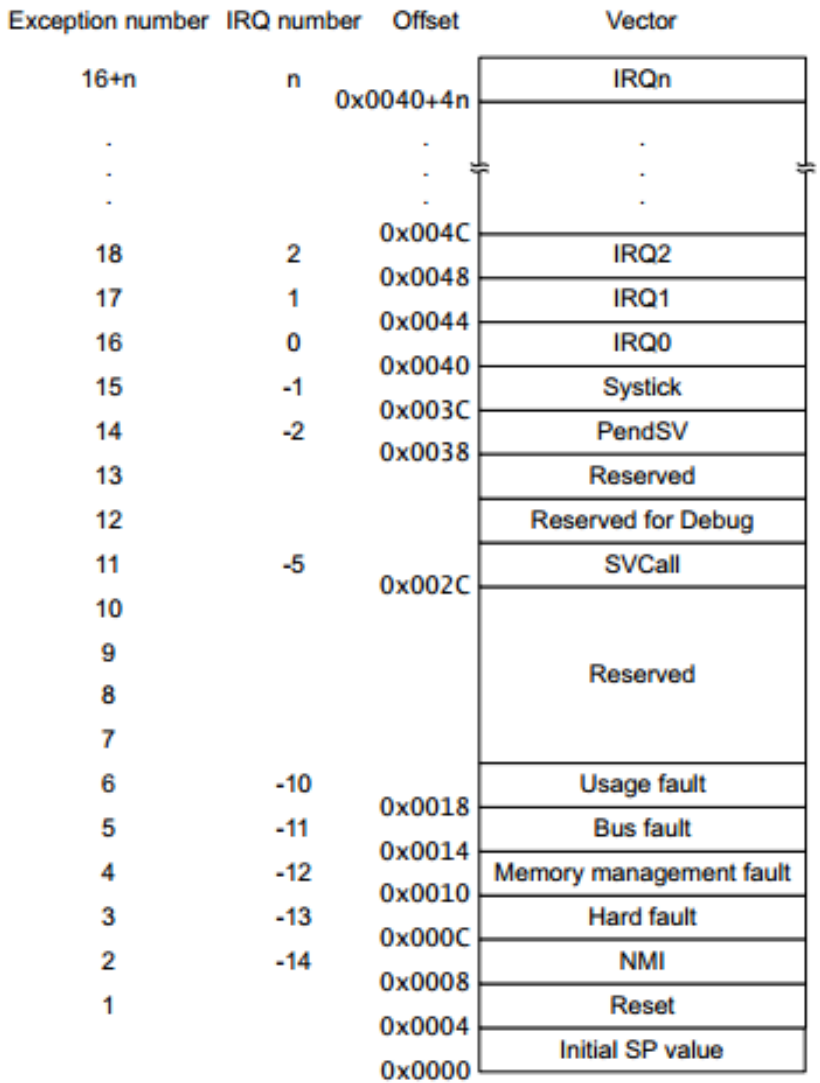
- Notice that the state (*context*) of the background task must be restored on returning from servicing an interrupt
  - so that it can carry on its work, after the interrupt has been serviced, *as though it had not been interrupted*
- If the context is to be restored, it must first be saved
- What is the context of the background task?
  - . . . *the complete set of user-mode registers*

ARM Cortex M3 Core Registers



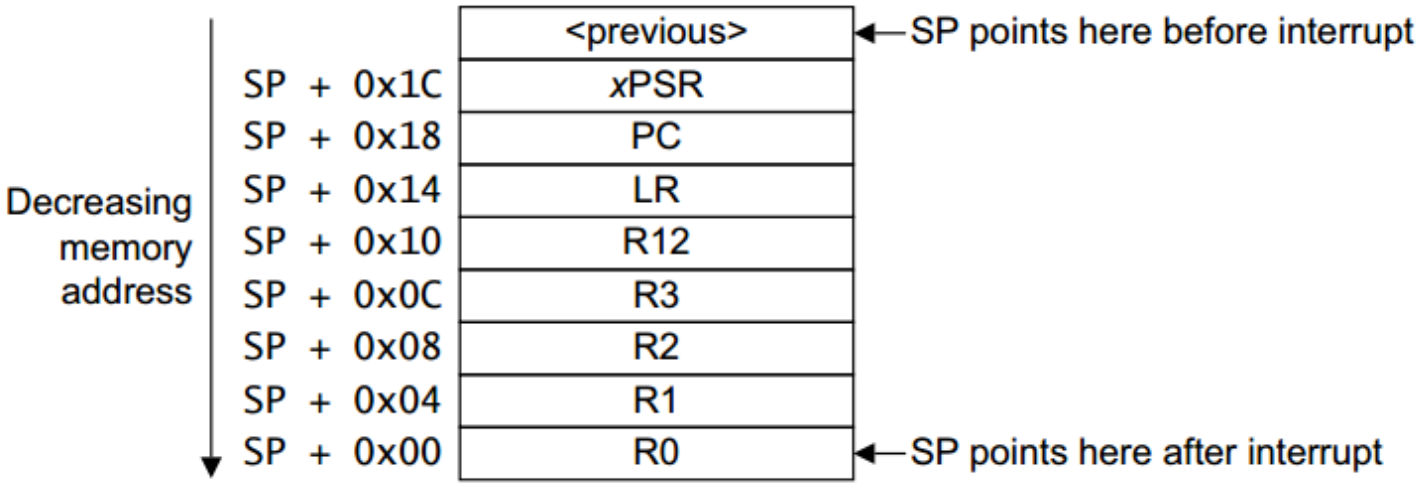
ARM, Cortex-M3 Devices Generic User Guide, ARM 2010 (p.2-3)

ARM Cortex M3 Vector Table



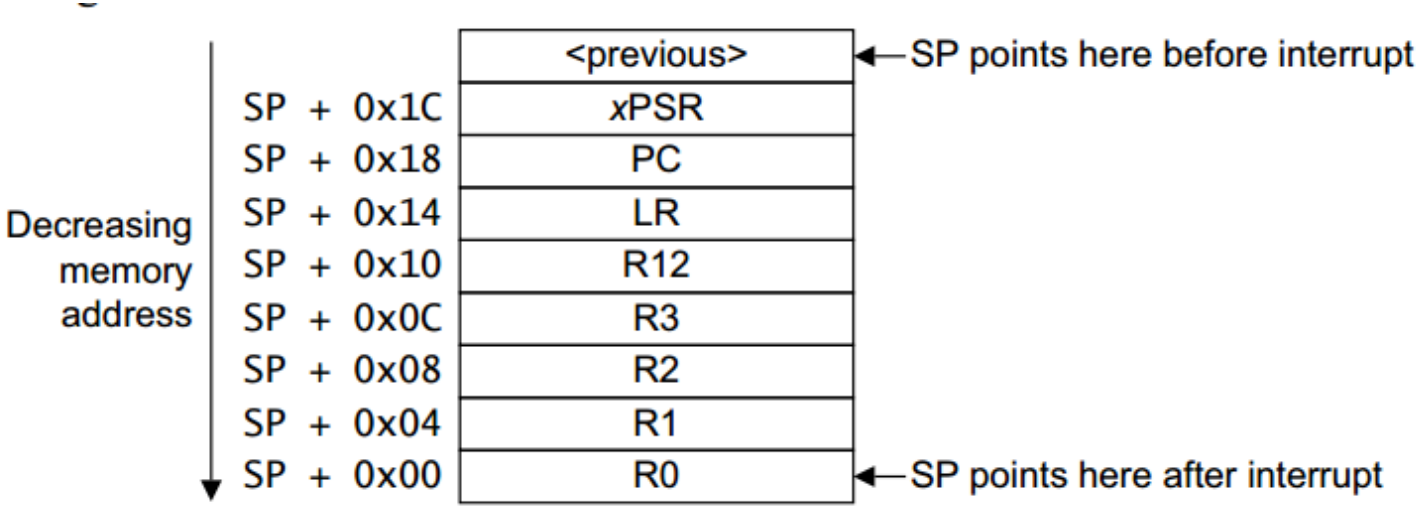
ARM, Cortex-M3 Devices Generic User Guide, ARM 2010 (p.2-24)

Interrupt entry



- Microcontroller peripheral raises interrupt; NVIC causes ISR vector to be fetched from vector table and put into R15 (PC); at same time, CPU pushes key registers onto the stack and stores special ‘return code’ in link register (R14/LR)
- If ISR needs to save more context, it must do so itself

Interrupt exit



- ISR returns just like a normal function call, except special ‘return’ code in LR causes processor to restore stack frame automatically and resume normal processing
- Any extra context that was saved on entry must be restored before exit
- Often necessary to clear the interrupt status flags in the peripheral before returning from ISR

Acknowledgements and Reading

Acknowledgements

- **[SGG09]** Silberschatz, A., Galvin, P. and Gagne, G., Operating systems concepts, John Wiley, 8th edition, 2009
- **[SSW04]** Sloss, A., Symes, D. and Wright, C., ARM System Developer's Guide, Morgan Kaufmann, 2004
- **[MAR13]** Martin, T., The Designer's Guide to the Cortex-M Processor Family: A Tutorial Approach, Newnes, 2013

Reading

- SGG09, 13.1 – 13.3
- MAR13, 1.1 – 1.5