Embedded systems engineering
Distributed real-time systems

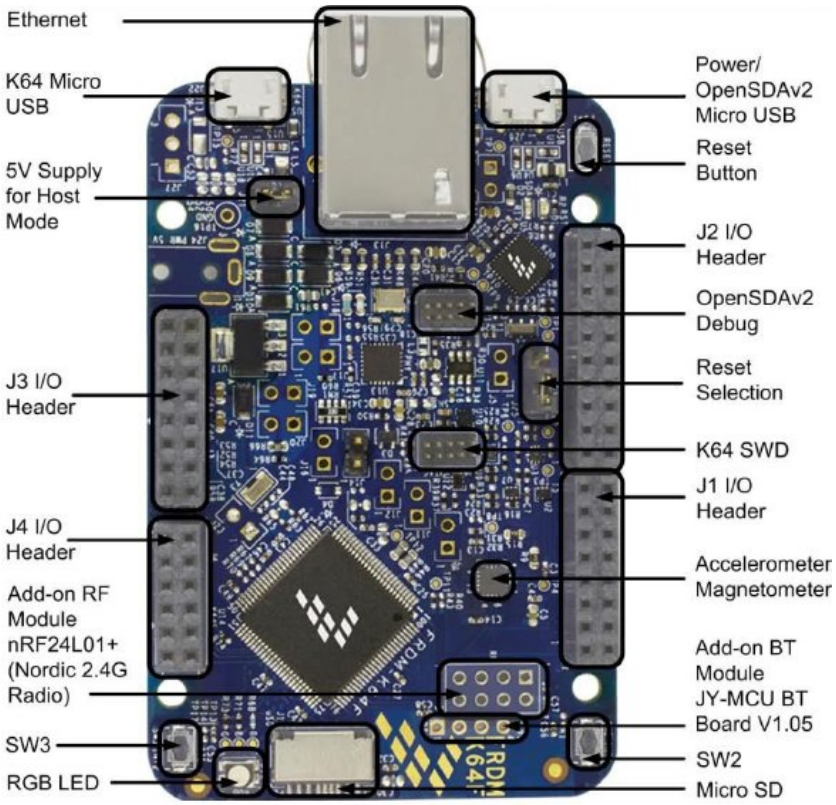
David Kendall

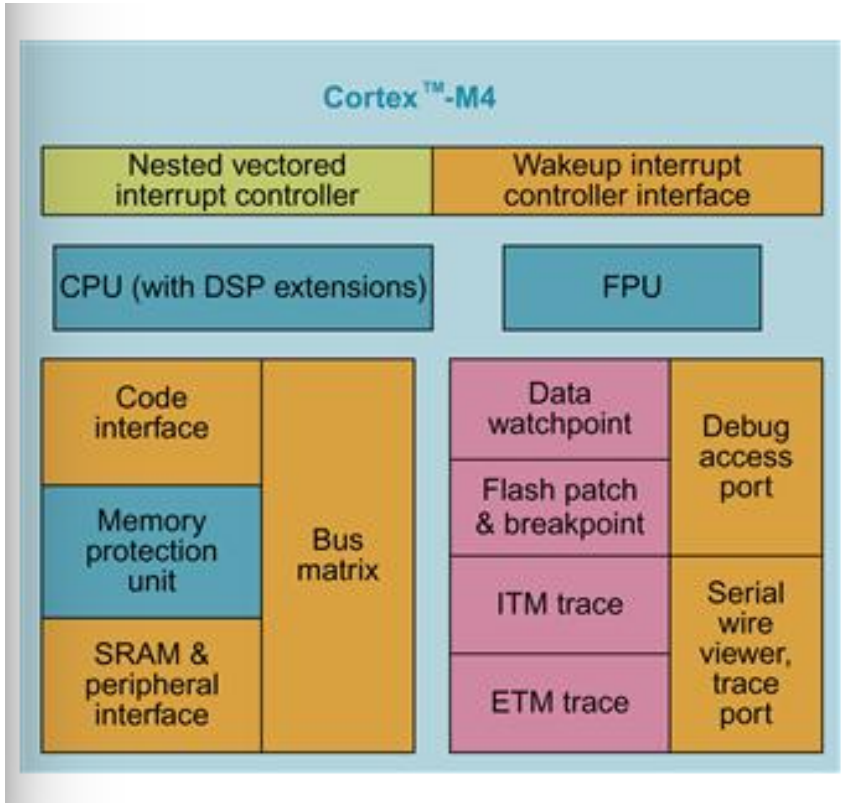
Introduction

- Time-triggered systems and event-triggered systems rely on interrupt handling
- Time-triggered – only one source of interrupt; a timer
- Event-triggered – potentially many sources of interrupt
- Build understanding of interrupt handling by looking in detail at:
  - Installing and executing an interrupt handler (ISR)
  - Configuring a timer as an interrupt source
- Microcontroller – FRDM-K64F (ARM Cortex M4)

Architecture of the ARM Cortex M4

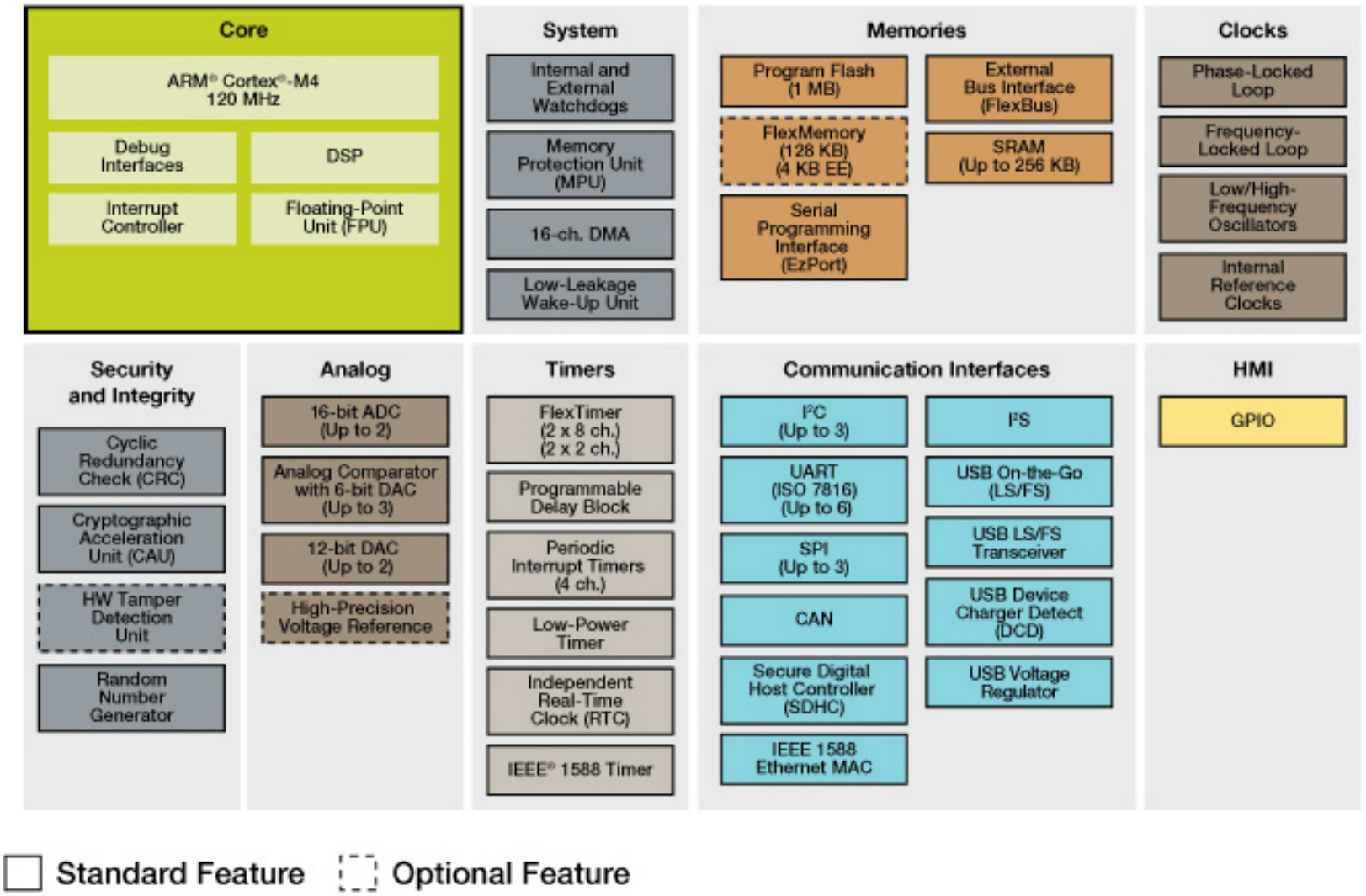
Modern microcontrollers - like the FRDM-K64F - have many sources of interrupt and provide hardware support for identifying the source of a particular interrupt.





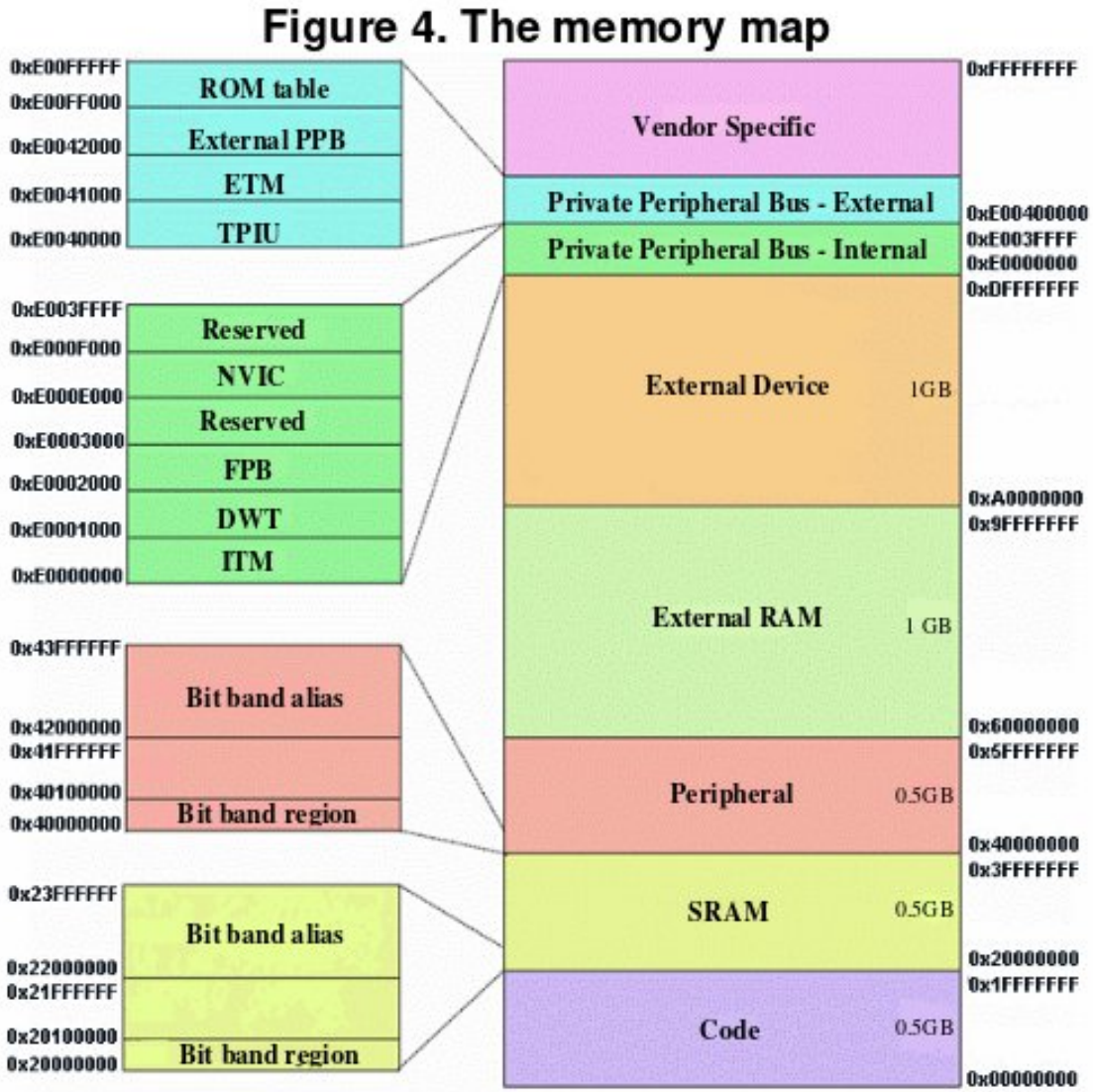
Martin, T. *The Designer's Guide to the Cortex-M Processor Family: A Tutorial Approach*, Newnes, 2013

K64F Block Diagram



Kinetis K63/K64 MCU Family Block Diagram, NXP

Memory Map



An Introduction to the ARM Cortex M3 Processor, ARM, 2006

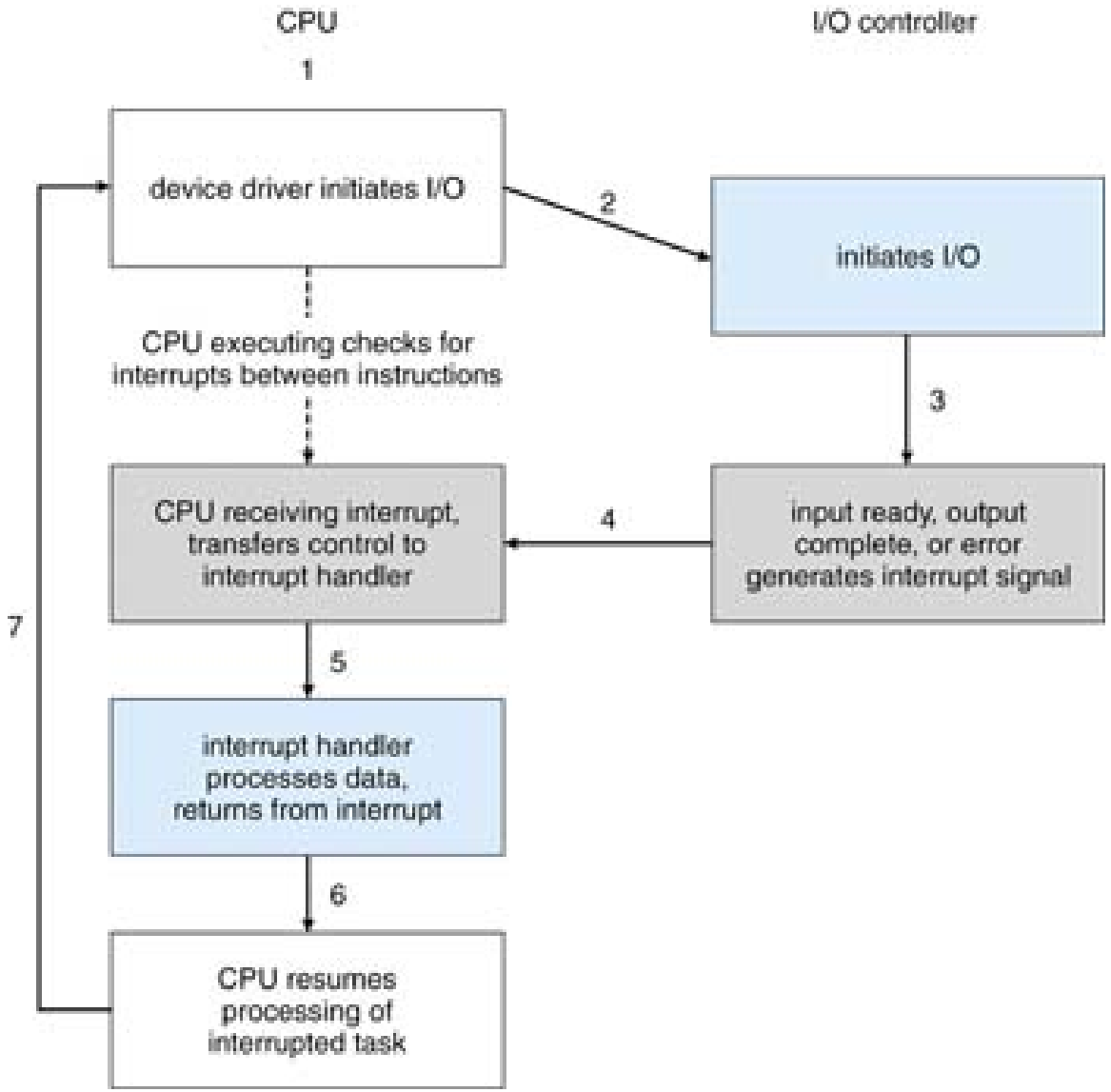
Device handling - Polling

- 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

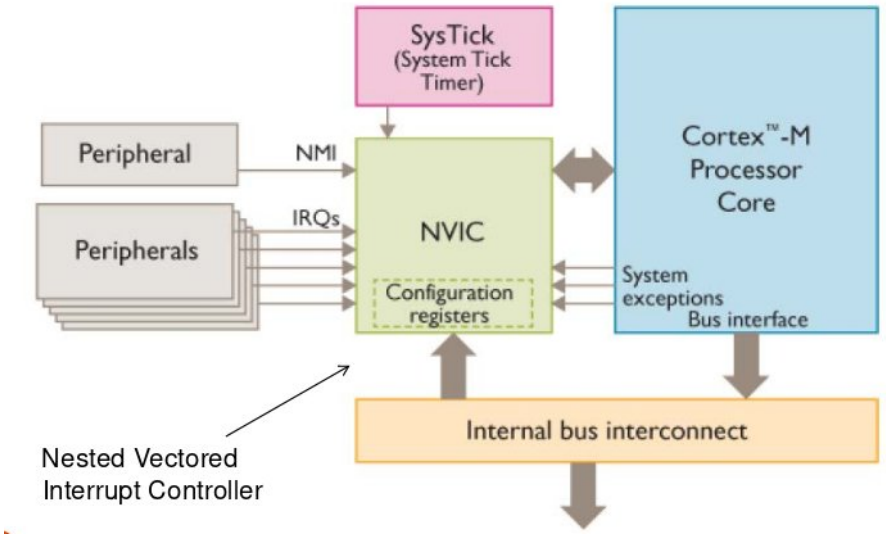
- Busy-waiting can be an inefficient use of the CPU
- CPU could be doing other useful computation instead of waiting
- E.g.
  - Assume: 10 ms for a disk I/O operation to complete; CPU clock speed of 120 MHz; average instruction requires 1 clock cycle – (rough estimates)
  - How many instructions could 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

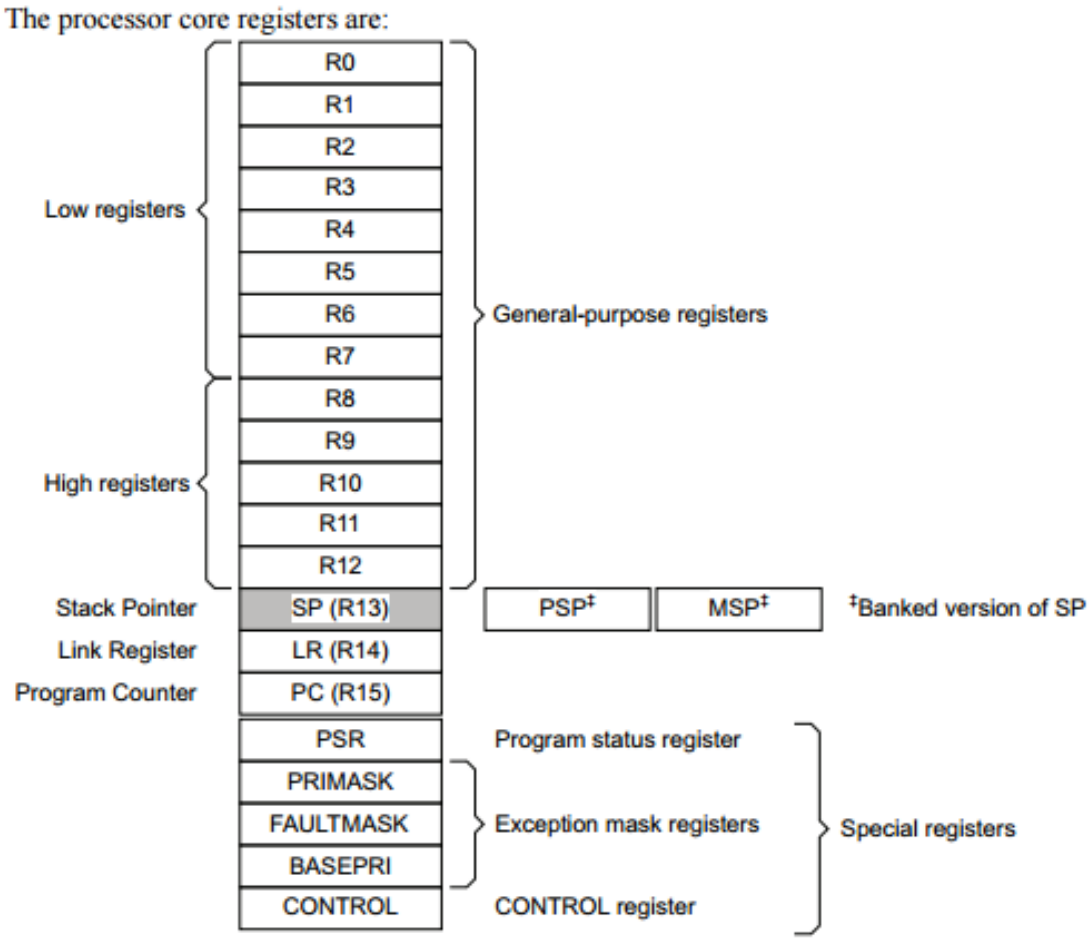
- Foreground / Background
- *Background*: Main (super) loop calls functions for computation
- *Foreground*: Interrupt service routines (ISRs) handle asynchronous events (interrupts)



Context switch

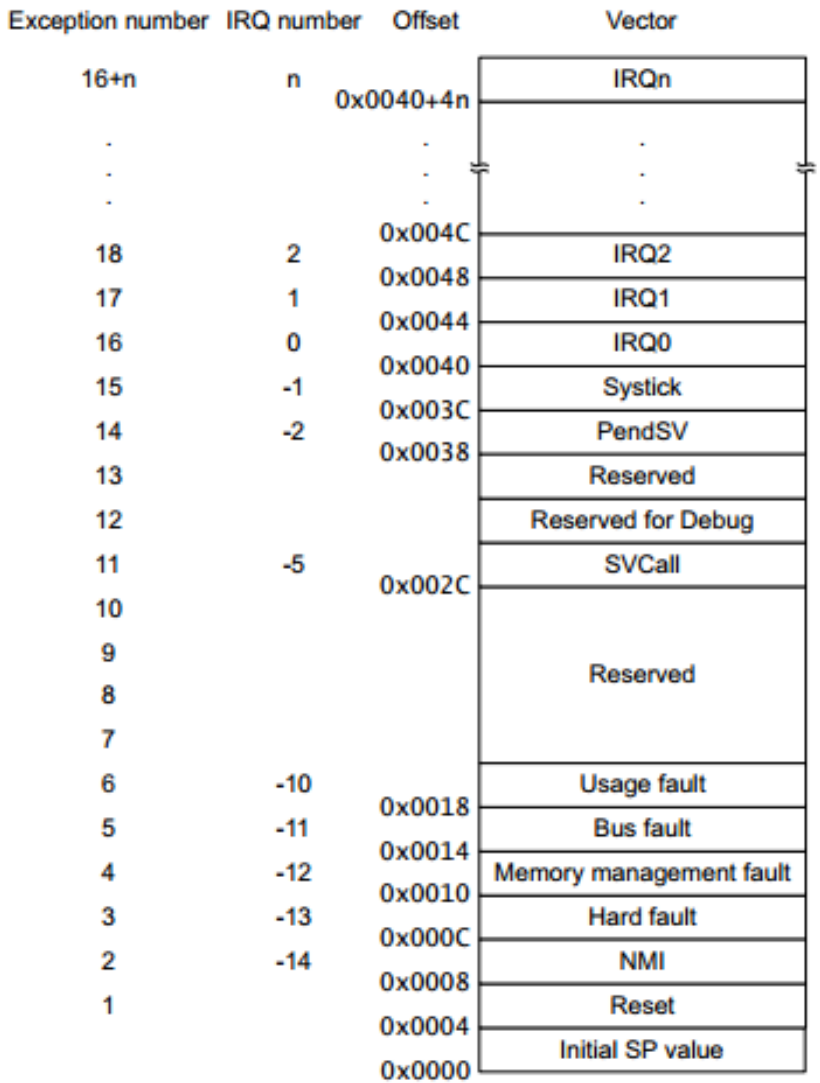
- Notice that the state (*context*) of the background task must be restored on returning from servicing an interrupt
  - so that it can carry on its work, after the interrupt has been serviced, *as though it had not been interrupted*
- If the context is to be restored, it must first be saved
- What is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

ARM Cortex M3 Core Registers



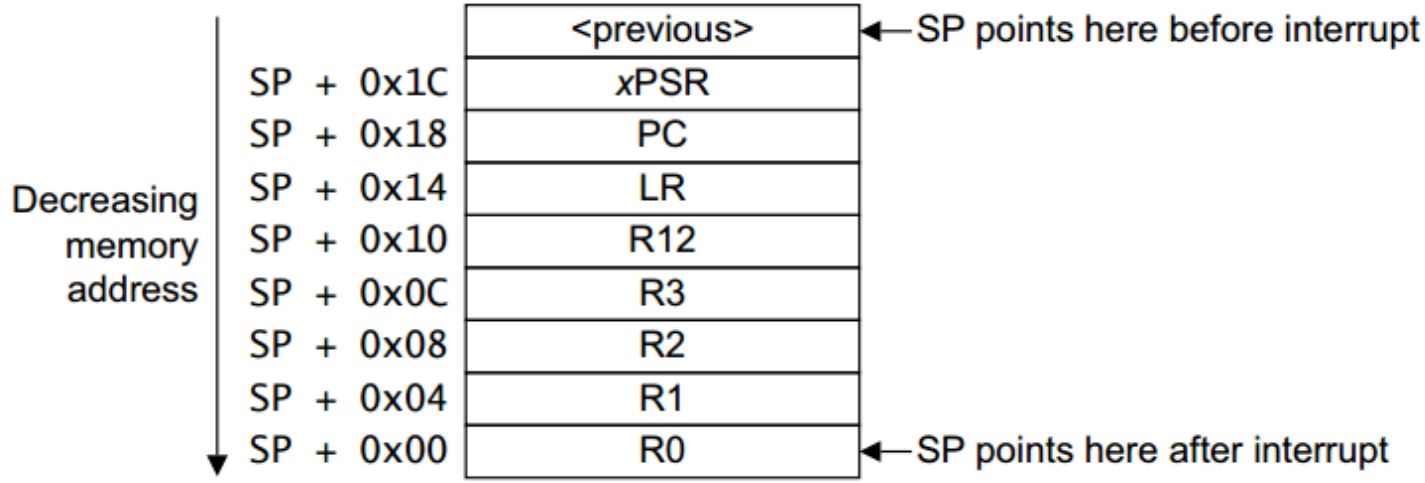
ARM, Cortex-M3 Devices Generic User Guide, ARM 2010 (p.2-3)

ARM Cortex M3 Vector Table



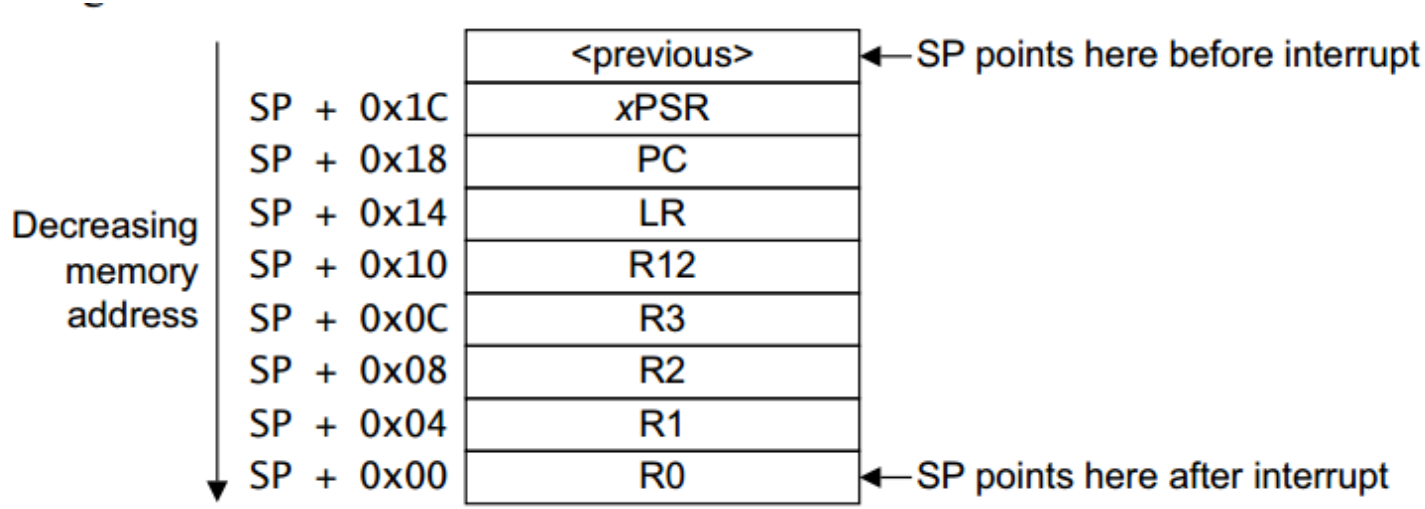
ARM, Cortex-M3 Devices Generic User Guide, ARM 2010 (p.2-24)

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

K64F Vector Table – details

```
    .section .isr_vector, "a"
    .align 2
    .globl __isr_vector
__isr_vector:
    .long   __StackTop          /* Top of Stack */
    .long   Reset_Handler       /* Reset Handler */
    .long   NMI_Handler         /* NMI Handler*/
    .long   HardFault_Handler   /* Hard Fault Handler*/
    .long   MemManage_Handler   /* MPU Fault Handler*/
    .long   BusFault_Handler    /* Bus Fault Handler*/
    .long   UsageFault_Handler  /* Usage Fault Handler*/
    .long   0                   /* Reserved*/
    .long   0                   /* Reserved*/
    .long   0                   /* Reserved*/
    .long   0                   /* Reserved*/
    .long   SVC_Handler         /* SVCall Handler*/
    .long   DebugMon_Handler    /* Debug Monitor Handler*/
    .long   0                   /* Reserved*/
    .long   PendSV_Handler      /* PendSV Handler*/
    .long   SysTick_Handler     /* SysTick Handler*/

                                /* External Interrupts*/
    .long   DMA0_IRQHandler     /* DMA Channel 0 Transfer Complete*/
    .long   DMA1_IRQHandler     /* DMA Channel 1 Transfer Complete*/
    ...
    .long   PIT0_IRQHandler     /* PIT timer channel 0 interrupt*/
    .long   PIT1_IRQHandler     /* PIT timer channel 1 interrupt*/
    .long   PIT2_IRQHandler     /* PIT timer channel 2 interrupt*/
    .long   PIT3_IRQHandler     /* PIT timer channel 3 interrupt*/
    ...
```

In the GCC_ARM tools, this code appears in the file `startup_MK64F12.S`

K64F Vector Table - notes

- The startup file gives default entries for all elements in the vector table
- For the external interrupt handlers, like `PITO_IRQHandler`, the code associated with the handler is just a simple empty loop (i.e. it does nothing except loop back to itself)
- The address of this handler is stored in its slot in the vector table and exported to the rest of the program as a .weak symbol - this means that it can be overwritten by our own handler, using the same name.

Installing our own interrupt handler

- So, to install our own handler for any interrupt, we look in the file `startup_MK64F12.S` at the vector table to find the name of the handler function, e.g. `PIT0_IRQHandler`. We then write our own C function with the same name, e.g.

```
void PIT0_IRQHandler() {
...
}
```

  - *Note:* Interrupt handlers must be compiled with C linkage, not C++ linkage, otherwise they will be ignored.
- The interrupt must also be enabled in the NVIC. We can use a predefined CMSIS function to do that ...

```
NVIC_EnableIRQ(PIT0_IRQn);
```

Now that we know how to install an interrupt handler, let's see how to get one of the peripheral devices to generate an interrupt for us to handle ...

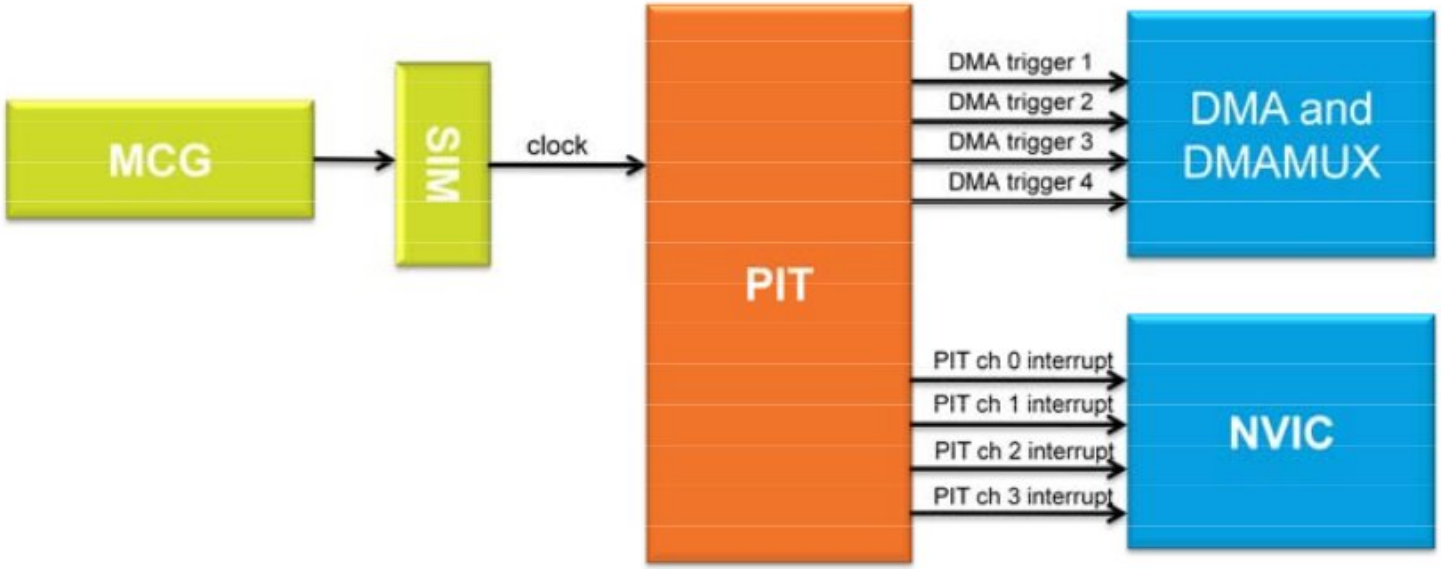
Periodic Interrupt Timer (PIT)

The PIT module is an array of timers that can be used to raise interrupts and trigger DMA channels.

Main features:

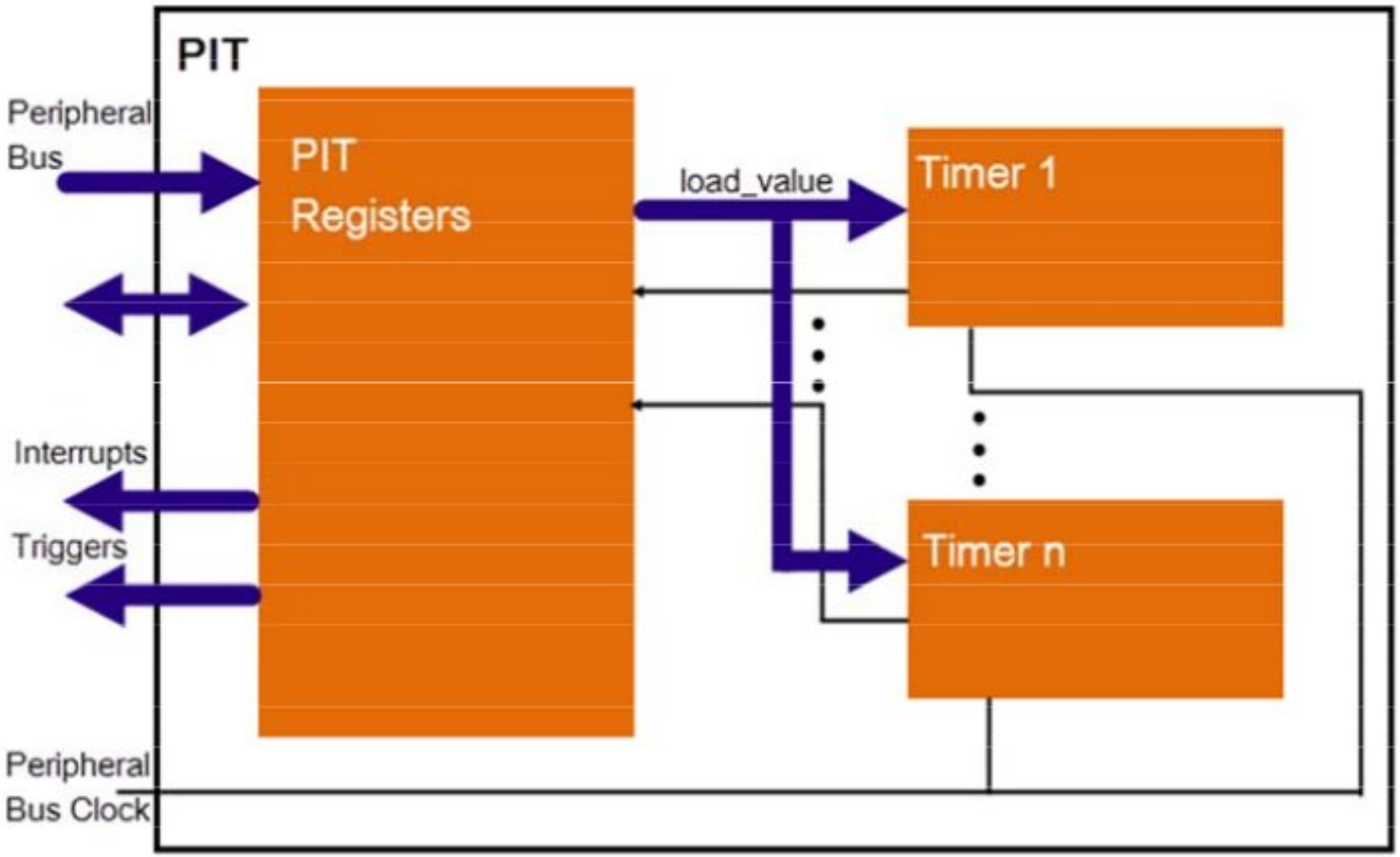
- Ability of timers to generate interrupts
- Ability of timers to generate DMA trigger pulses
- Maskable interrupts
- Independent timeout periods for each timer

PIT Relationship to other modules



- **MCG** - Multipurpose Clock Generator
- **SIM** - System Integration Module
- **PIT** - Periodic Interrupt Timer
- **DMA** - Direct Memory Access
- **NVIC** - Nested Vectored Interrupt Controller

PIT Block Diagram



PIT memory map

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
4003_7000	PIT Module Control Register (PIT_MCR)	32	R/W	0000_0006h	41.3.1/1085
4003_7100	Timer Load Value Register (PIT_LDVAL0)	32	R/W	0000_0000h	41.3.2/1087
4003_7104	Current Timer Value Register (PIT_CVAL0)	32	R	0000_0000h	41.3.3/1087
4003_7108	Timer Control Register (PIT_TCTRL0)	32	R/W	0000_0000h	41.3.4/1088
4003_710C	Timer Flag Register (PIT_TFLG0)	32	R/W	0000_0000h	41.3.5/1089
4003_7110	Timer Load Value Register (PIT_LDVAL1)	32	R/W	0000_0000h	41.3.2/1087
4003_7114	Current Timer Value Register (PIT_CVAL1)	32	R	0000_0000h	41.3.3/1087
4003_7118	Timer Control Register (PIT_TCTRL1)	32	R/W	0000_0000h	41.3.4/1088
4003_711C	Timer Flag Register (PIT_TFLG1)	32	R/W	0000_0000h	41.3.5/1089
4003_7120	Timer Load Value Register (PIT_LDVAL2)	32	R/W	0000_0000h	41.3.2/1087
4003_7124	Current Timer Value Register (PIT_CVAL2)	32	R	0000_0000h	41.3.3/1087
4003_7128	Timer Control Register (PIT_TCTRL2)	32	R/W	0000_0000h	41.3.4/1088
4003_712C	Timer Flag Register (PIT_TFLG2)	32	R/W	0000_0000h	41.3.5/1089
4003_7130	Timer Load Value Register (PIT_LDVAL3)	32	R/W	0000_0000h	41.3.2/1087
4003_7134	Current Timer Value Register (PIT_CVAL3)	32	R	0000_0000h	41.3.3/1087
4003_7138	Timer Control Register (PIT_TCTRL3)	32	R/W	0000_0000h	41.3.4/1088
4003_713C	Timer Flag Register (PIT_TFLG3)	32	R/W	0000_0000h	41.3.5/1089

PIT control

To program a PIT channel to generate an interrupt every p seconds, given a bus clock with a frequency of f Hz

- The clock gate to the PIT must be enabled in the System Clock Gating Control Register (`SCGCR6`)
- The clock to the standard PIT timers must be enabled in the PIT Module Control Register (`PIT_MCR`)
- The Timer Load Value Register (`PIT_LDVALn`) must be loaded with a value, $v = pf - 1$
- The Timer Interrupt Enable (`TIE`) bit must be set in the Timer Control Register (`PIT_TCTRLn`)
- The timer must be started by setting the Timer Enable (`TEN`) bit in the Timer Control Register (`PIT_TCTRLn`)
- The PIT interrupt must be enabled in the NVIC
- Every time the PIT interrupt is raised, it must be cleared by writing 1 to the Timer Interrupt Flag (`TIF`) in the Timer Flag Register (`PIT_TFLGn`)

PIT behaviour

When the PIT has been programmed, . . .

- The timer start value is given by the value in the Timer Load Value Register
- The timer value is reduced by 1 on every bus clock tick
- When the value of the timer becomes 0, the timer interrupt is raised
- When the interrupt has been raised, the value of the timer is reset to the value in the Timer Load Value Register and the cycle begins again
- Writing a new value to the Timer Load Value Register does not restart the timer; instead the value will be loaded when the timer expires

PIT example program

Assume that we want to toggle the blue LED every 0.5 seconds and that the frequency of the bus clock is 60 MHz ...

```
#include "MK64F12.h"  /* Include the CMSIS header file */
...                   /* Some code omitted             */
void PIT_init(void) {
    SIM_SCGC6 |= (1u << 23);
    PIT_MCR_REG(PIT) = 0u;
    PIT_LDVAL_REG(PIT, 0) = 29999999;
    PIT_TCTRL_REG(PIT, 0) |= PIT_TCTRL_TIE_MASK;
    PIT_TCTRL_REG(PIT, 0) |= PIT_TCTRL_TEN_MASK;
    NVIC_EnableIRQ(PIT0_IRQn);
}

void PIT0_IRQHandler(void) {
    blue_toggle();
    PIT_TFLG_REG(PIT,0) |= PIT_TFLG_TIF_MASK;
}
```

Acknowledgements

- Trevor Martin, The Designer's Guide to the Cortex-M Processor Family: A Tutorial Approach, Newnes, 2013
- K64 Sub-Family Reference Manual, Freescale Semiconductor Inc., 2014