Interrupt handling and Timers

KF6010 – Distributed Real Time Systems

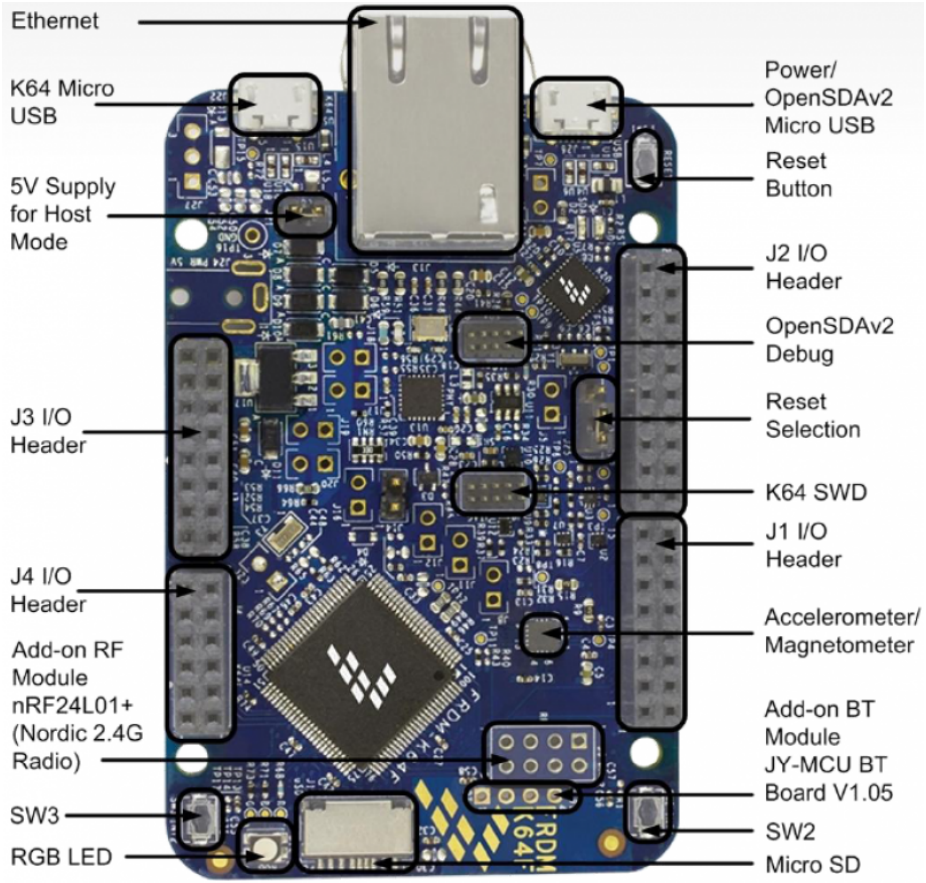
Dr Alun Moon
alun.moon@northumbria.ac.uk

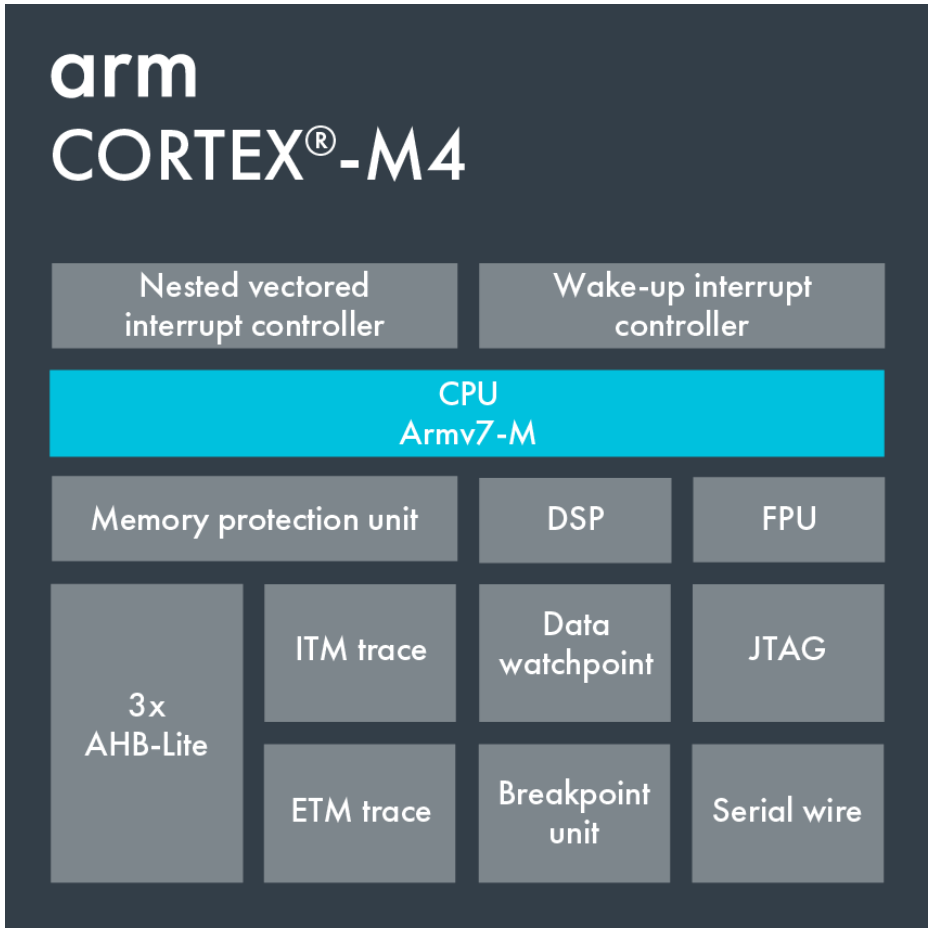
Lecture 2

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 interrupt handlers (ISR)
  - configuring a timer as an interrupt source
- We are using the FRDM-K64F microcontroller, based on the ARM Cortex M4 architecture
- Modern microcontroller systems have many sources of interrupt and provide hardware support for these.

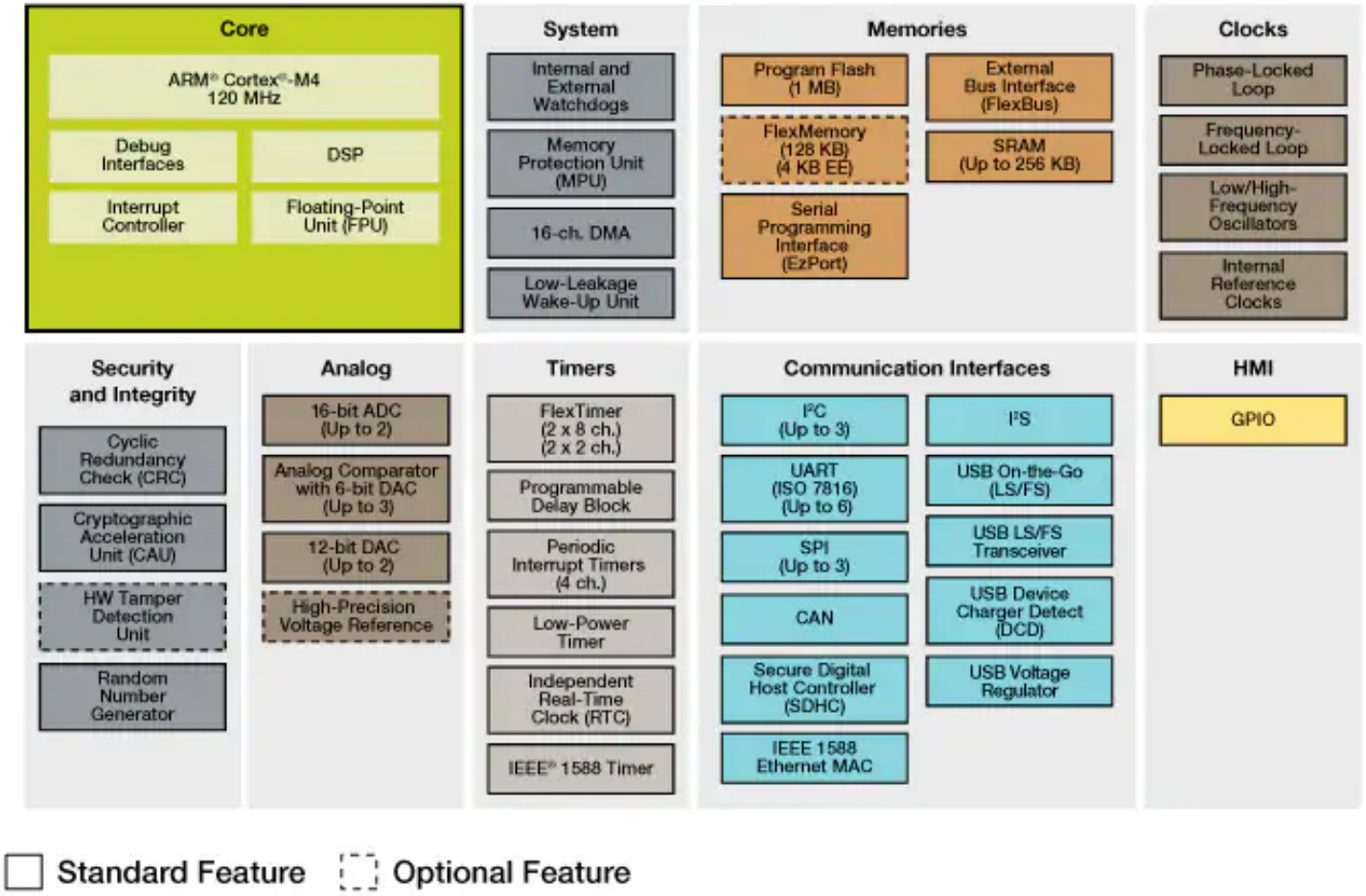
K64F





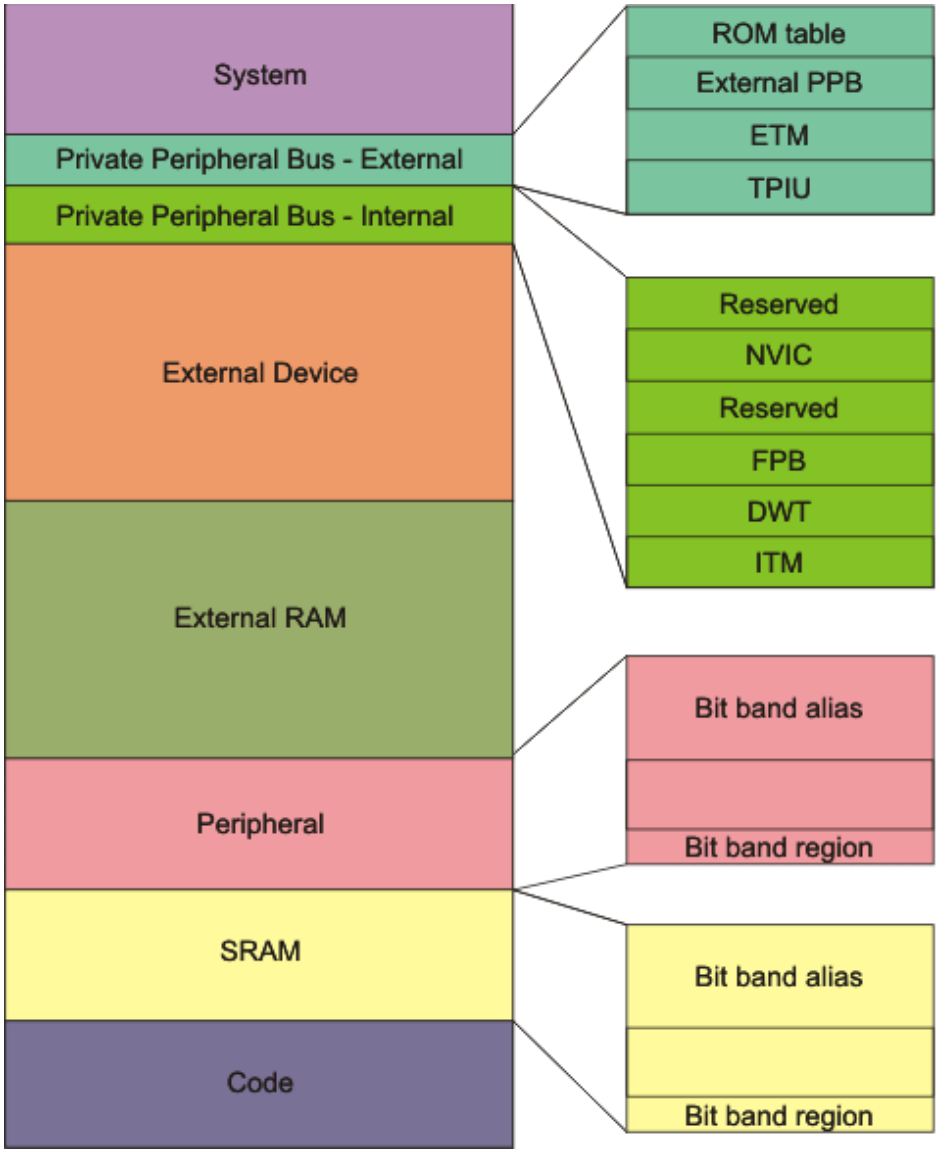
Freedom Development Platform for Kinetis K64, K63, and K24 MCUs
Cortex-M4

K64F Block diagram



K64_120: Kinetis® K64-120 MHz, 256 KB SRAM Microcontrollers (MCUs) based on Arm® Cortex®-M4 Core

Memory Map



ARM Cortex-M3 and Cortex-M4 Memory Organization

- Devices are memory mapped
- Simpler and more-flexible IO architecture

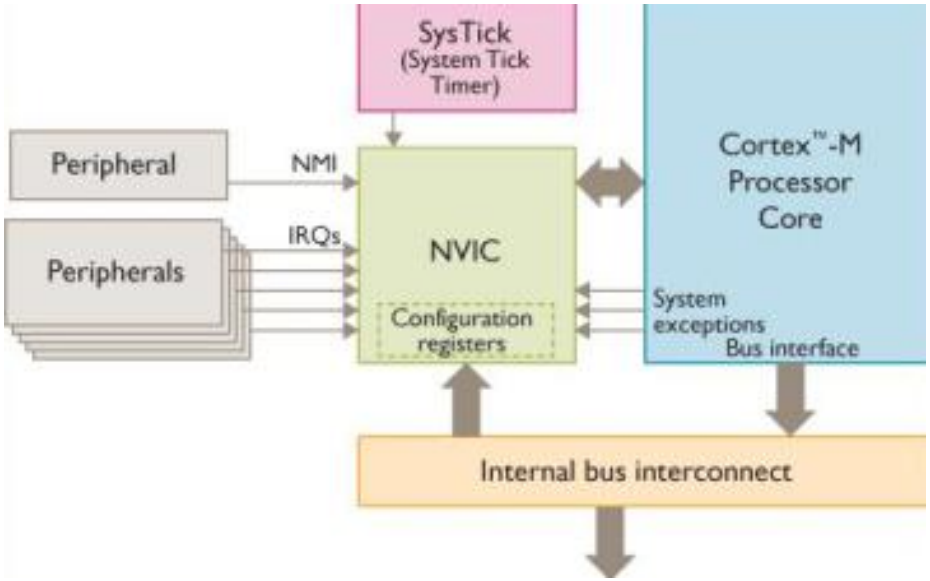
Device Polling

- One way to handle devices is by polling
  1. CPU repeatedly tests status register to see if device is busy
  2. When not busy, CPU writes a command into the control register
  3. CPU sets the control-ready bit
  4. When the device sees the control-ready bit set:
     - reads the command from the control register
     - sets the busy bit in the status register
  5. When the device completes I/O operation, it sets a bit in the status register
  6. CPU repeatedly tests status register
- Problem busy-waiting in steps 1 and 6

Problems

- Busy waiting can be inefficient use of the CPU
- CPU could be doing other useful computation instead of waiting
- For example:
  - Assume:
    - 10 ms for a disk I/O operation to complete.
    - CPU clock speed of 120 MHz
    - average instruction requires 1 clock cycle
  - 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 *interrupt* it, when the I/O operation has been completed.

Interrupt



- Cortex M4 supports many interrupts
- Interrupt Vector Table maps interrupt sources to ISRs

Simple Interrupt driven program structure

- Partition task into Foreground and Background tasks.

  Background Main super loop calls functions for computation

```
int main()
{
    for(;;) {
        /* computation tasks */
    }
}
```

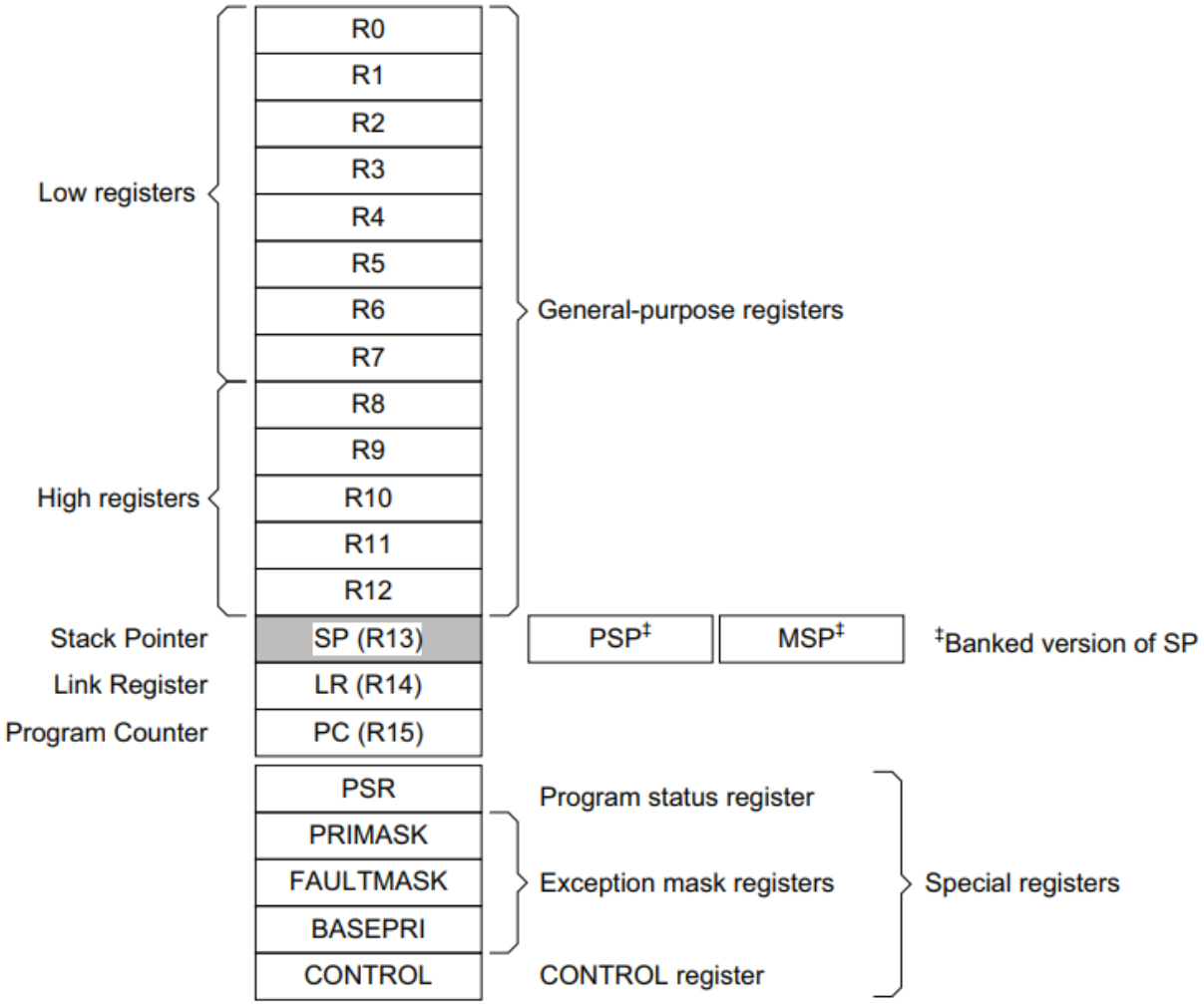
  Foreground Interrupt Service Routines (ISR) handle asynchronous events

Recall Fetch-Execute cycle

When an interrupt occurs:

- Program execution is transferred to the ISR
- On completion of the ISR program flow is resumed
- On return to the main part, execution must continue *as if no interrupt had occured*
- We need to save and restore execution context.
- `main()` is the background task, as this is the one suspended to handle foreground events (ISR) tasks

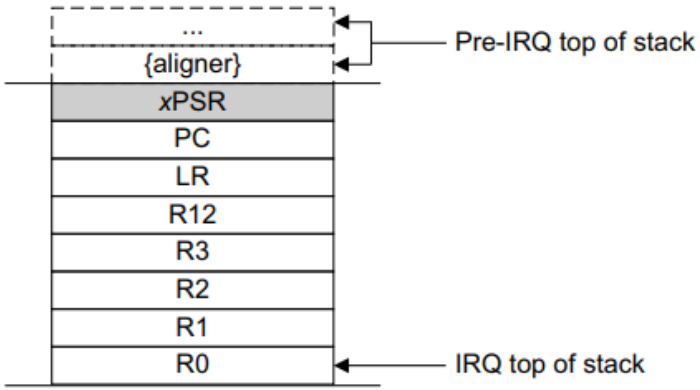
ARM Cortex M4 Core Registers



- 15 32bit registers
- 5 special purpose registers

Stack-frame

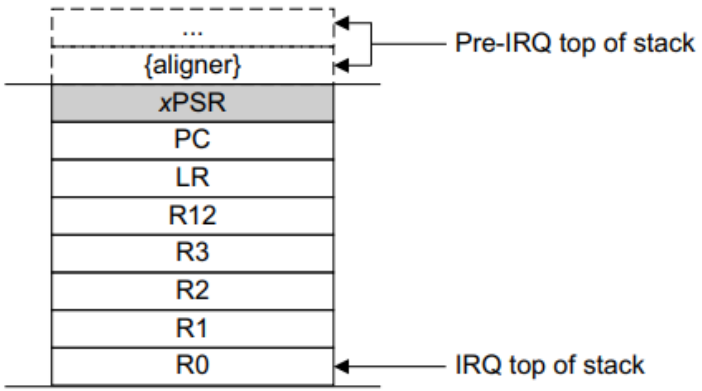
Interrupt entry



- Interrupt automatically saves 8 registers, pushing them to the stack
- NVIC fetches ISR and writes into PC/R15
- Return address is written to LR (Link-Register/R14)
- Pushes R0–R3 to stack
- if ISR needs to use more registers it must handle this itself

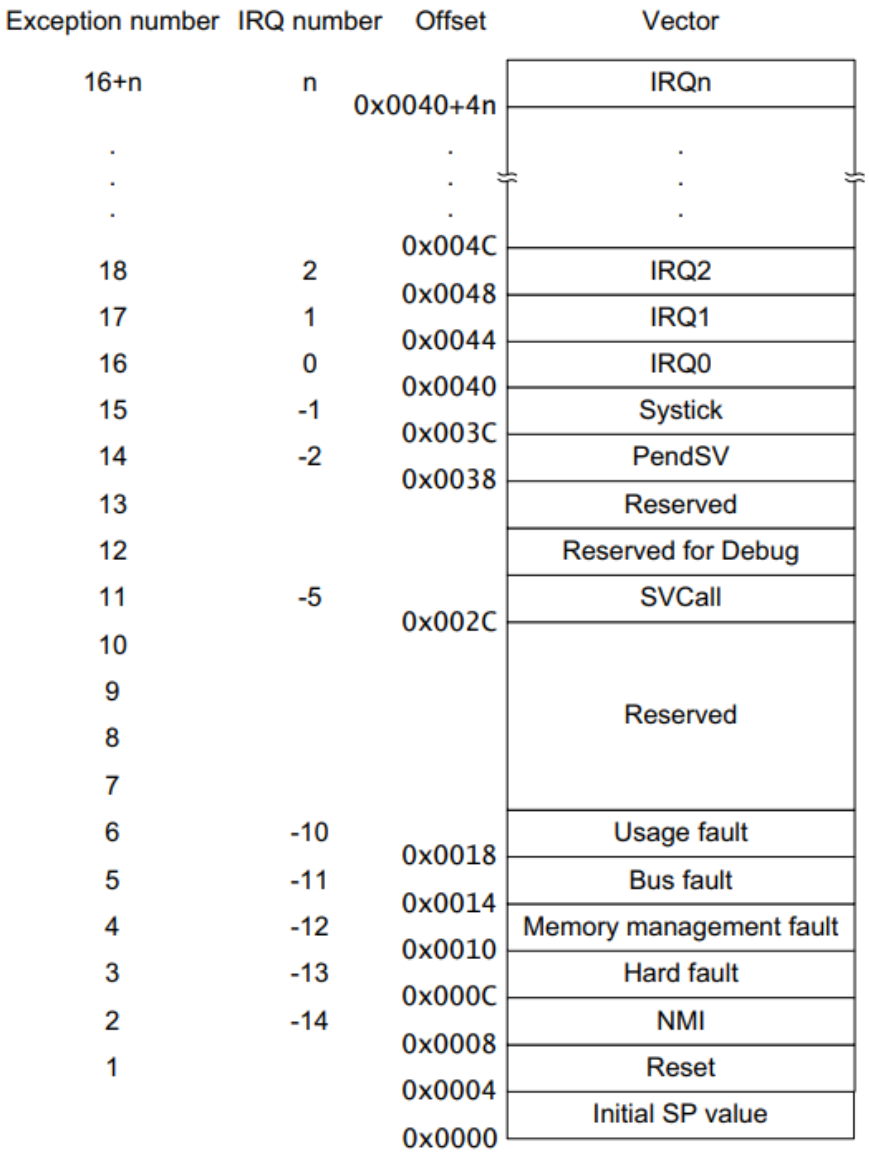
Stack-frame

Interrupt exit



- ISR returns like a normal function,
  - except special 'return' code in LR caused processor to restore stack frame automatically
- Any extra registers pushed , must be restored before exit
- it is often necessary to clear the interrupt status flags in the peripheral before returning from the ISR

Interrupt Vector Table



- The NVIC controller uses the vector table to determine which ISR to call
- The vector table is just a list of addresses to call as ISRs
- The vector used is determined by the source of the interrupt — read the manual!

startup code

```
__isr_vector:
    .long   __StackTop
    .long   Reset_Handler
    .long   NMI_Handler
    .long   HardFault_Handler
    .long   MemManage_Handler
    .long   BusFault_Handler
    .long   UsageFault_Handler
    .long   0
    .long   0
    .long   0
    .long   0
    .long   SVC_Handler
    .long   DebugMon_Handler
    .long   0
```

PIT Interrupt vector

```
.long   PIT0_IRQHandler
.long   PIT1_IRQHandler
.long   PIT2_IRQHandler
.long   PIT3_IRQHandler
```

- Looking down the file we find the name of the interrupt at line 113
- This is an assembler label
- We want to write a C function with this name

Installing an Interrupt Handler

```
void PIT0_IRQHandler () {

}
int main() {

    NVIC_EnableIRQ(PIT0_IRQn);

}
```

- C function has same name as label in vector table
- Note: Interrupt handlers *must* be compiled with C linkage and not C++ linkage
- The interrupt *must* be enabled in the NVIC. We can use a CMSIS function from the device header

Installing an Interrupt Handler

```
void PIT0_IRQHandler () {

}
int main() {

    NVIC_EnableIRQ(PIT0_IRQn);

}
```

- C function has same name as label in vector table
- Note: Interrupt handlers *must* be compiled with C linkage and not C++ linkage
- The interrupt *must* be enabled in the NVIC. We can use a CMSIS function from the device header

As long as your filenames have `.c` and `.h` extensions, the gcc compiler will automatically use C conventions. When we move to C++ you will have to prefix the ISR with `extern "C"`

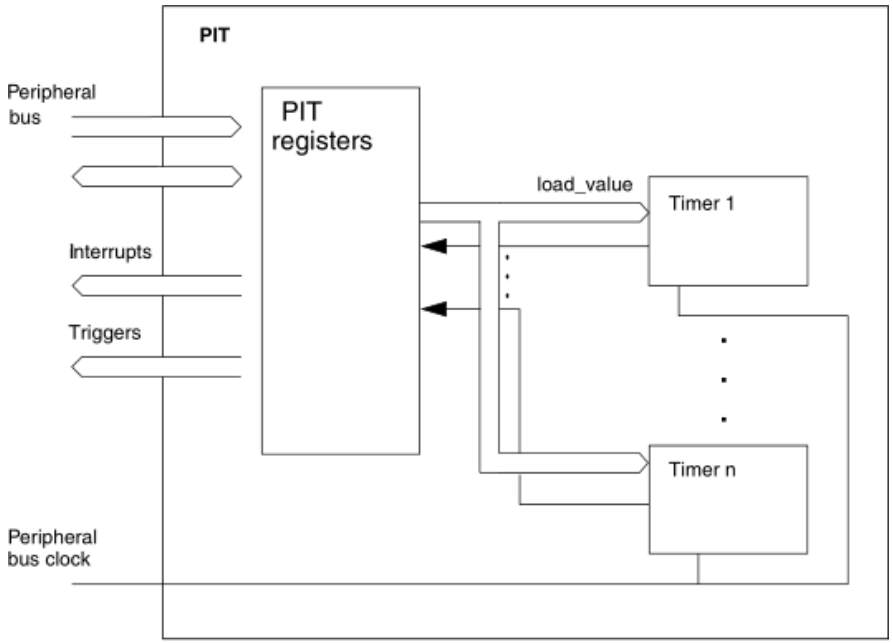
```
extern "C" PIT0_IRQHandler () {
}
```

Periodic Interrupt Timer (PIT)

The PIT module is an array of timers that can be used to raise interrupts at a set periodic rate.
Main features:

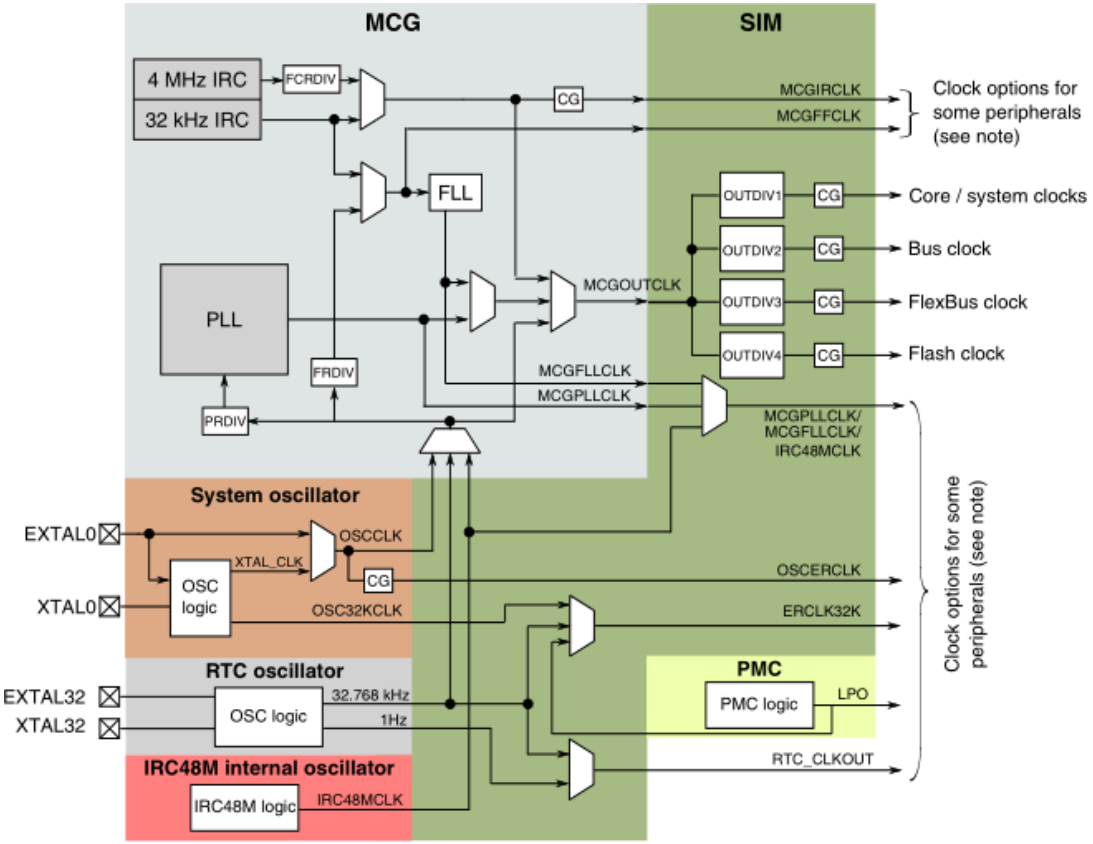
- Timers generate PIT interrupts
- Timers generate DMA trigger pulses
- Mask-able interrupts
- Independent timeout periods for each timer

PIT configuration



- PIT driven by the *Peripheral Bus Clock*
- In the FRDM K64F this runs at 60 MHz

Clock configuration



MCG Multi-purpose Clock Generator

SIM System Integration Module

Memory Map

Address	Register name		function
4003 7000	PIT Module Control	PIT_MCR	Module control – effects all timers
4003 7100	Timer Load Value	PIT_LDVAL0	PIT0 reset value – determines timer period
4003 7104	Current Timer Value	PIT_CVAL0	
4003 7108	Timer Control	PIT_TCTRL0	Controls behaviour of PIT0
4003 710C	Timer Flags	PIT_TFLG0	Timer-0 status flags
4003 7110	Timer Load Value	PIT_LDVAL1	

PIT control from C

To program a PIT channel to provide an interrupt, several things have to happen:

1. the clock gate to the PIT must be enabled in the System Clock Gating control register (`SIM_SCGC6` K64F-Ref, 12.2.13)
2. the clock to the PIT timers must be enabled in the PIT Module Control Register (`PIT_MCR` K64F-Ref, 41.3.1
3. The Timer Load Value (`PIT_MCR`) **?**, 41.3.2) must be loaded with the correct value
4. The Timer Interrupt Enable (`TIE`) bit must be set in the Timer Control register (`PIT_TCTRL0` K64F-Ref, 41.3.4)
5. The timer must be started by setting the Timer Enable (TE) bit in the control register
6. The PIT interrupt must be enabled in the NVIC

The PIT Interrupt handler

Every time the PIT Interrupt is raised, it must be cleared by writing a 1 to the Timer Interrupt Flag in the Timer Flag register (`PIT_TFLG0` K64F-Ref, 41.3.5)

PIT Behaviour

- The timer start value is loaded from the Load Value Register
- For each tick of the bus, the timer counter is decremented by 1
- When the value of the timer reaches 0, the timer interrupt is raised
- When the interrupt has been raised, the timer resets and loads the start value from the Load Value Register

Timer Load Value

calculating the value for the LDVAL register

- If LDVAL is 0 – we get an interrupt every tick
- If LDVAL is 1 – we get an interrupt every other (2) ticks
- The Load Value is $n - 1$ ticks, for an interrupt every n ticks
- The bus runs at 60 MHz or 60×10^6 ticks per second

For an interrupt every t seconds

$$L = \left\lfloor t \times 60 \times 10^6 \right\rfloor - 1$$

For an interrupt every q micro-second (µs)

$$L = \lfloor q \times 60 \rfloor - 1$$

Timer Load Value

We want to toggle the red LED every 0.5 seconds

Timer Load Value

We want to toggle the red LED every 0.5 seconds

$$L = t \times 60 \times 10^6 - 1 = 0.5 \times 60 \times 10^6 - 1 = 30 \times 10^6 - 1 = 29\,999\,999$$

Timer Load Value

We want to toggle the red LED every 0.5 seconds

$$L = t \times 60 \times 10^6 - 1 = 0.5 \times 60 \times 10^6 - 1 = 30 \times 10^6 - 1 = 29\,999\,999$$

We want to flash the blue LED every half-second

Flash the LED, is turn on and turn off every half-second, toggle every quarter second.

Timer Load Value

We want to toggle the red LED every 0.5 seconds

$$L = t \times 60 \times 10^6 - 1 = 0.5 \times 60 \times 10^6 - 1 = 30 \times 10^6 - 1 = 29\,999\,999$$

We want to flash the blue LED every half-second

Flash the LED, is turn on and turn off every half-second, toggle every quarter second.

$$L = t \times 60 \times 10^6 - 1 = 0.25 \times 60 \times 10^6 - 1 = 15 \times 10^6 - 1 = 14\,999\,999$$

Timer Load Value

We want to toggle the red LED every 0.5 seconds

$$L = t \times 60 \times 10^6 - 1 = 0.5 \times 60 \times 10^6 - 1 = 30 \times 10^6 - 1 = 29\,999\,999$$

We want to flash the blue LED every half-second

Flash the LED, is turn on and turn off every half-second, toggle every quarter second.

$$L = t \times 60 \times 10^6 - 1 = 0.25 \times 60 \times 10^6 - 1 = 15 \times 10^6 - 1 = 14\,999\,999$$

We want to flash the green LED 10 times a second

10 times a second is 10 Hz, period is 0.1 s, toggle every 0.05 s (50×10^3 µs)

Timer Load Value

We want to toggle the red LED every 0.5 seconds

$$L = t \times 60 \times 10^6 - 1 = 0.5 \times 60 \times 10^6 - 1 = 30 \times 10^6 - 1 = 29\,999\,999$$

We want to flash the blue LED every half-second

Flash the LED, is turn on and turn off every half-second, toggle every quarter second.

$$L = t \times 60 \times 10^6 - 1 = 0.25 \times 60 \times 10^6 - 1 = 15 \times 10^6 - 1 = 14\,999\,999$$

We want to flash the green LED 10 times a second

10 times a second is 10 Hz, period is 0.1 s, toggle every 0.05 s (50×10^3 µs)

$$L = q \times 60 - 1 = 50 \times 10^3 \times 60 - 1 = 3000 \times 10^3 - 1 = 2\,999\,999$$

C code

```c
#include "MK64F12.h"

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;
    NVIC_EnableIRQ(PIT0_IRQn);
    PIT_TCTRL_REG(PIT, 0) |= PIT_TCTRL_TEN_MASK;
}

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

Bibliography

M4-UG. *Cortex-M4 Devices Generic User Guide*. ARM Limited, 2010. http://hesabu.net/kf6010/DUI0553A_cortex_m4_dgug.pdf.

K64F-Ref. *K64 Sub-Family Reference Manual*. Freescale Semiconductor, Inc., rev. 2 edition, January 2014. http://hesabu.net/kf6010/K64P144M120SF5RM.pdf.