F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 2 — 2018/19

[0] No proprietary software has been used in producing these slides

Outline

1. Lecture 1: Introduction to Systems Programming
2. Lecture 2: Systems Programming with the Raspberry Pi
3. Lecture 3: Memory Hierarchy
   - Memory Hierarchy
     - Principles of Caches
4. Lecture 4: Programming external devices
   - Basics of device-level programming
5. Lecture 5: Exceptional Control Flow
6. Lecture 6: Computer Architecture
   - Processor Architectures Overview
     - Pipelining
7. Lecture 7: Code Security: Buffer Overflow Attacks
8. **Lecture 8: Interrupt Handling**
9. Lecture 9: Miscellaneous Topics
10. Lecture 10: Revision

Lecture 7.
Interrupt Handling

What are interrupts and why do we need them?

- In order to deal with internal or external events, **abrupt** changes in control flow are needed.
- Such abrupt changes are also called **exceptional control flow (ECF)**.
- The system needs to take special action in these cases (call interrupt handlers, use non-local jumps)

[0] Lecture based on Bryant and O'Hallaron, Ch 8

Revision: Interrupts on different levels

An abrupt change to the control flow is called **exceptional control flow (ECF)**.
ECF occurs at different levels:

- **hardware level:** e.g. arithmetic overflow events detected by the hardware trigger abrupt control transfers to **exception handlers**
- **operating system:** e.g. the kernel transfers control from one user process to another via **context switches**.
- **application level:** a process can send a signal to another process that abruptly transfers control to a signal handler in the recipient.

We covered the application level in a previous class, today we will focus on the OS and hardware level.

Timers with assembler-level system calls

We have previously used C library functions to implement timers.
We will now use the ARM assembler `SWI` command that we know, to trigger a system call to *sigaction*, *getitmer* or *setitimer*.
The corresponding codes are[1]:

- *sigaction*: **67**
- *setitimer*: **104**
- *getitmer*: **105**

The arguments to these functions need to be in registers: **R0**, **R1**, **R2**, etc

[1] See Smith, Appendix B "Raspbian System Calls"

Reminder: Interface to C library functions

getitimer, setitimer - get or set value of an interval timer

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval *
   new_value,
             struct itimerval *old_value);
```

setitimer sets up an interval timer that issues a signal in an interval specified by the *new_value* argument, with this structure:

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
  };
  struct timeval {
    time_t      tv_sec;         /* seconds */
    suseconds_t tv_usec;        /* microseconds */
  };
```

Reminder: Setting-up a timer in C

Signals (or software interrupts) can be programmed on C level by associating a C function with a signal sent by the kernel.
sigaction - examine and change a signal action

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

The sigaction structure is defined as something like:

```
   struct sigaction {
              void     (*sa_handler)(int);
              void     (*sa_sigaction)(int, siginfo_t *,
                  void *);
              sigset_t   sa_mask;
              int        sa_flags;
              void     (*sa_restorer)(void);
                                  }
```

NB: the `sa_sigaction` field defines the action to be performed when the signal with the id in `signum` is sent.

[1]See `man sigaction`

Timers with assembler-level system calls

We will now use the ARM assembler `SWI` command that we know, to trigger a system call to *sigaction*, *getitmer* or *setitimer*.
The corresponding codes are[2]:

- *sigaction*: **67**
- *setitimer*: **104**
- *getitmer*: **105**

The arguments to these functions need to be in registers: **R0**, **R1**, **R2**, etc

[2]See Smith, Appendix B “Raspbian System Calls”

Example: Timers with assembler-level system calls

We need the following headers:

```
#include <signal.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/time.h>

// sytem call codes
#define SETITIMER 104
#define GETITIMER 105
#define SIGACTION 67

// in micro-sec
#define DELAY 250000
```

[2] Sample source itimer21.c

Our own getitimer function

```
static inline int getitimer_asm(int which, struct
    itimerval *curr_value){
  int res;
  asm(/* inline assembler version of performing a system
      call to GETITIMER */
       "\tB␣_bonzo105\n"
       "_bonzo105:␣NOP\n"
       "\tMOV␣R0,␣%[which]\n"
       "\tLDR␣R1,␣%[buffer]\n"
       "\tMOV␣R7,␣%[getitimer]\n"
       "\tSWI␣0\n"
       "\tMOV␣%[result],␣R0\n"
       : [result] "=r" (res)
       : [buffer] "m" (curr_value)
         , [which] "r" (ITIMER_REAL)
         , [getitimer] "r" (GETITIMER)
       : "r0", "r1", "r7", "cc");
}
```

Our own `setitimer` function

```
static inline int setitimer_asm(int which, const struct
    itimerval *new_value, struct itimerval *old_value) {
  int res;
  asm(/* system call to SETITIMER */
      "\tB␣_bonzo104\n"
      "_bonzo104:␣NOP\n"
      "\tMOV␣R0,␣%[which]\n"
      "\tLDR␣R1,␣%[buffer1]\n"
      "\tLDR␣R2,␣%[buffer2]\n"
      "\tMOV␣R7,␣%[setitimer]\n"
      "\tSWI␣0\n"
      "\tMOV␣%[result],␣R0\n"
      : [result] "=r" (res)
      : [buffer1] "m" (new_value)
        , [buffer2] "m" (old_value)
        , [which] "r" (ITIMER_REAL)
        , [setitimer] "r" (SETITIMER)
      : "r0", "r1", "r2", "r7", "cc");
}
```

Our own `sigaction` function

```
int sigaction_asm(int signum, const struct sigaction *act
   , struct sigaction *oldact){
  int res;
  asm(/* performing a syscall to SIGACTION */
      "\tB␣_bonzo67\n"
      "_bonzo67:␣NOP\n"
      "\tMOV␣R0,␣%[signum]\n"
      "\tLDR␣R1,␣%[buffer1]\n"
      "\tLDR␣R2,␣%[buffer2]\n"
      "\tMOV␣R7,␣%[sigaction]\n"
      "\tSWI␣0\n"
      "\tMOV␣%[result],␣R0\n"
      : [result] "=r" (res)
      : [buffer1] "m" (act)
        , [buffer2] "m" (oldact)
        , [signum] "r" (signum)
        , [sigaction] "r" (SIGACTION)
      : "r0", "r1", "r2", "r7", "cc");
}
```

Example: Timers with assembler-level system calls

The main function is as before, using our own functions:

```
int main () {
 struct sigaction sa;
 struct itimerval timer;

 /* Install timer_handler as the signal handler for
    SIGALRM. */
 memset (&sa, 0, sizeof (sa));
 sa.sa_handler = &timer_handler;

 sigaction_asm (SIGALRM, &sa, NULL);
```

Example: Timers with assembler-level system calls

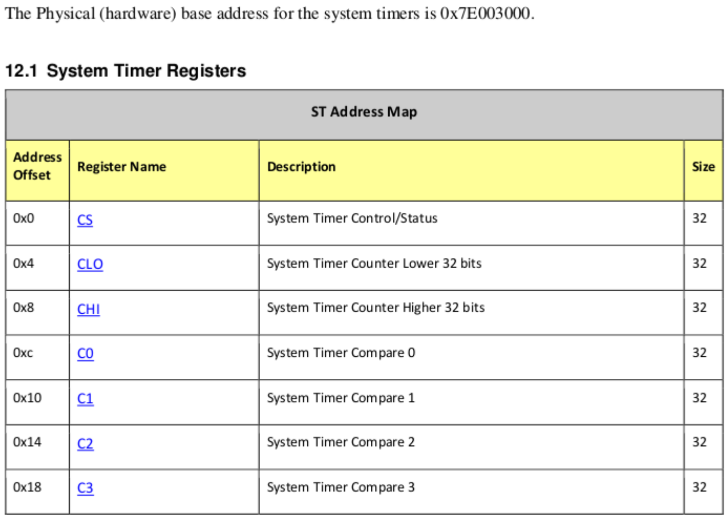
```
 /* Configure the timer to expire after 250 msec... */
 timer.it_value.tv_sec = 0;
 timer.it_value.tv_usec = DELAY;
 /* ... and every 250 msec after that. */
 timer.it_interval.tv_sec = 0;
 timer.it_interval.tv_usec = DELAY;
 /* Start a virtual timer. It counts down whenever this
    process is executing. */
 setitimer_asm (ITIMER_REAL, &timer, NULL);

 /* Busy loop, but accepting signals */
 while (1) {} ;
}
```

[2] Sample source in itimer21.c

Timers by probing the RPi on-chip timer

- The RPi 2 has an on-chip timer that ticks at a rate of 250 MHz
- This can be used for getting precise timing information
- (in our case) to implement a timer directly.
- As before, we need to know the base address of the timer device
- and the register assignment for this device.
- We find both in the **BCM Peripherals Manual, Chapter 12, Table 12.1**

GPIO Register Assignment



The Physical (hardware) base address for the system timers is 0x7E003000.

12.1 System Timer Registers

ST Address Map			
Address Offset	**Register Name**	**Description**	**Size**
0x0	CS	System Timer Control/Status	32
0x4	CLO	System Timer Counter Lower 32 bits	32
0x8	CHI	System Timer Counter Higher 32 bits	32
0xc	C0	System Timer Compare 0	32
0x10	C1	System Timer Compare 1	32
0x14	C2	System Timer Compare 2	32
0x18	C3	System Timer Compare 3	32

[2] See BCM Peripherals Manual, Chapter 12, Table 12.1

Example code

```
#define TIMEOUT 3000000
...
static volatile unsigned int timerbase ;
static volatile uint32_t *timer ;
...
timerbase = (unsigned int)0x3F003000 ;
// memory mapping
timer = (int32_t *)mmap(0, BLOCK_SIZE, PROT_READ|
   PROT_WRITE, MAP_SHARED, fd, timerbase) ;
if ((int32_t)timer == (int32_t)MAP_FAILED)
  return failure (FALSE, "wiringPiSetup:␣mmap␣(TIMER)␣
      failed:␣%s\n", strerror (errno)) ;
else
  fprintf(stderr, "NB:␣timer␣=␣%x␣for␣timerbase␣%x\n",
      timer, timerbase);
```

As usual we memory-map the device memory into the accessible address space.

Example code

```
{ volatile uint32_t ts = *(timer+1); // word offset
  volatile uint32_t curr;

  while( ( (curr=*(timer+1)) - ts ) < TIMEOUT )  {  /*
      nothing */ }
}
```

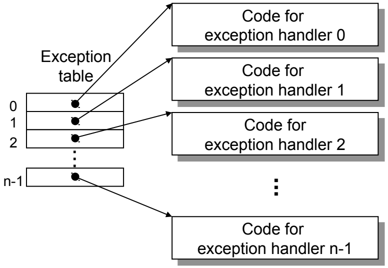
To wait for `TIMEOUT` micro-seconds, the core code just has to read from location `timer+1` to get and check the timer value.

[2]Sample source in itimer31.c; see also this discussion on the BakingPi pages

Summary

- In order to implement a time-out functionality, several mechanisms can be used:
  - C library calls (on top of Raspbian)
  - assembler-level system calls (to the kernel running inside Raspbian)
  - directly probing the on-chip timer available on the RPi2
- We have seen sample code for each of the 3 mechanisms.
- Also, on embedded systems time-critical code is often needed, so access to a precise on-chip timer is important for many kinds of applications.

Interrupt requests in Assembler


Exception table
0
1
2
n-1
Code for exception handler 0
Code for exception handler 1
Code for exception handler 2
Code for exception handler n-1

The central data structure for handling (hardware) interrupts is the **interrupt vector table** (or more generally exception table).

Interrupt handlers in C + Assembler

We will now go through the steps of handling hardware interrupts directly in assembler.
To implement interrupt handlers directly on the RPi2 we need to:

1. Build vector tables of interrupt handlers
2. Load vector tables
3. Set registers to enable specific interrupts
4. Set registers to globally enable interrupts

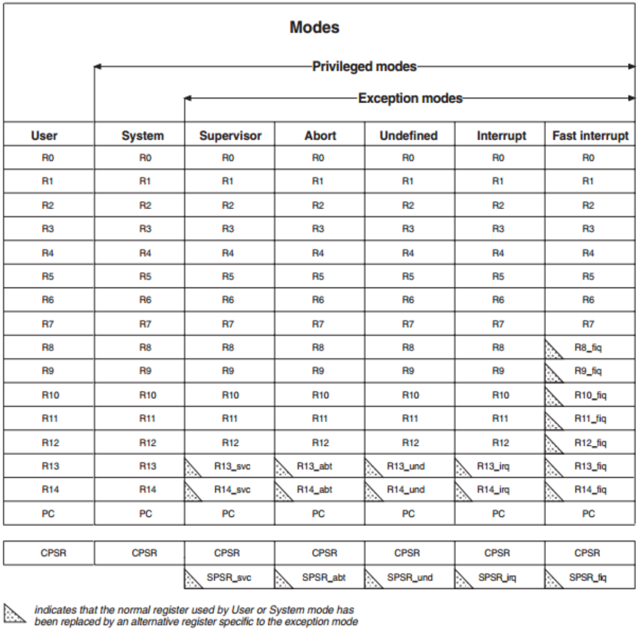
[2] Valvers: Bare Metal Programming in C (Pt4)

Building an Interrupt Vector Table

The relevant information for the Cortex A7 processor, used on the RPi2, can be found in the ARMv7 reference manual in Section B1.8.1 (Table B1-3).

Exception Type	Mode	VE	Normal Address
Reset	Supervisor		0x00
Undefined Instruction	Undefined		0x04
Software Interrupt (SWI)	Supervisor		0x08
Prefetch Abort	Abort		0x0C
Data Abort	Abort		0x10
IRQ (Interrupt)	IRQ	0	0x18
IRQ (Interrupt)	IRQ	1	undef
FIQ (Fast Interrupt)	FIQ	0	0x1C
FIQ (Fast Interrupt)	FIQ	1	undef

NB: each entry is 4 bytes; just enough to code a branch operation to the actual code **NB:** when an exception occurs the processor changes mode to the **exception-specific mode**



Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	◺ R8_fiq
R9	R9	R9	R9	R9	R9	◺ R9_fiq
R10	R10	R10	R10	R10	R10	◺ R10_fiq
R11	R11	R11	R11	R11	R11	◺ R11_fiq
R12	R12	R12	R12	R12	R12	◺ R12_fiq
R13	R13	◺ R13_svc	◺ R13_abt	◺ R13_und	◺ R13_irq	◺ R13_fiq
R14	R14	◺ R14_svc	◺ R14_abt	◺ R14_und	◺ R14_irq	◺ R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		◺ SPSR_svc	◺ SPSR_abt	◺ SPSR_und	◺ SPSR_irq	◺ SPSR_fiq

◺ *indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode*

Figure A2-1 Register organization

Coding Interrupt Handlers

An interrupt handler is a block of C (or assembler) code, that is called on an interrupt. The interrupt vector table links the interrupt number with the code.
We need to inform the compiler that a function should be used as an interrupt handler like this:

```
void f () __attribute__ ((interrupt ("IRQ")));
```

Other permissible values for this parameter are: IRQ, FIQ, SWI, ABORT and UNDEF.

Example interrupt handler

A very basic “undefined instruction” handler looks like this:

```
/**
    @brief The undefined instruction interrupt handler

    If an undefined instruction is encountered, the CPU will
        start
    executing this function. Just trap here as a debug
        solution.
*/
void __attribute__((interrupt("UNDEF")))
    undefined_instruction_vector(void)
{
    while( 1 )
    {
        /* Do Nothing! */
    }
}
```

Constructing Vector Tables

“Our vector table:”

```
_start:
    ldr pc, _reset_h
    ldr pc, _undefined_instruction_vector_h
    ldr pc, _software_interrupt_vector_h
    ldr pc, _prefetch_abort_vector_h
    ldr pc, _data_abort_vector_h
    ldr pc, _unused_handler_h
    ldr pc, _interrupt_vector_h
    ldr pc, _fast_interrupt_vector_h

_reset_h:                              .word   _reset_
_undefined_instruction_vector_h:       .word
    undefined_instruction_vector
_software_interrupt_vector_h:          .word
    software_interrupt_vector
_prefetch_abort_vector_h:              .word
    prefetch_abort_vector
_data_abort_vector_h:                  .word   data_abort_vector
_unused_handler_h:                     .word   _reset_
_interrupt_vector_h:                   .word   interrupt_vector
```

Constructing Vector Tables

```
_reset_:

    mov     r0, #0x8000
    mov     r1, #0x0000
    ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
    stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
    ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
    stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
```

NB: using tools such as `gdb` and `objdump` we know that “our” vector table is at address `0x00008000`; in supervisor mode we can write to any address, so the code above moves our vector table to the start of the memory, where it should be

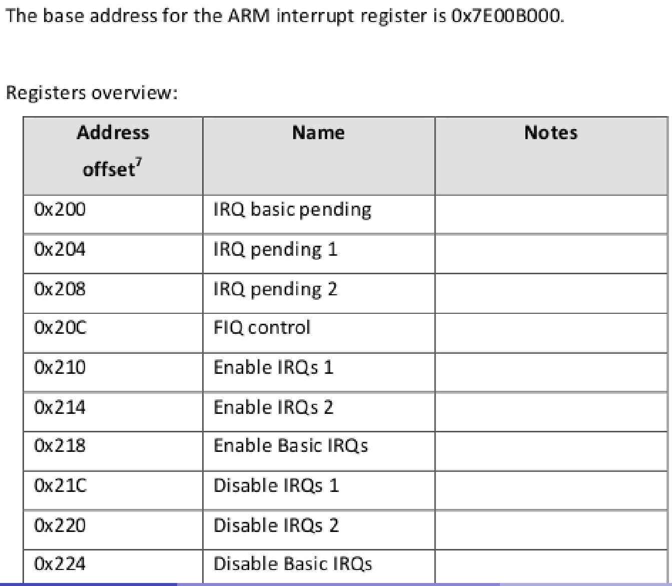
The Interrupt Controller

We need to enable interrupts by

- enabling the kind of interrupt we are interested in;
- globally enabling interrupts

The global switch ensures that disabling interrupts can be done in just one instruction.
But we still want more detailed control over different kinds of interrupts to treat them differently.

Interrupt Control Registers



The base address for the ARM interrupt register is 0x7E00B000.

Registers overview:

Address offset[7]	Name	Notes
0x200	IRQ basic pending	
0x204	IRQ pending 1	
0x208	IRQ pending 2	
0x20C	FIQ control	
0x210	Enable IRQs 1	
0x214	Enable IRQs 2	
0x218	Enable Basic IRQs	
0x21C	Disable IRQs 1	
0x220	Disable IRQs 2	
0x224	Disable Basic IRQs	

Interrupt Control Registers

We now define a structure for the **interrupt controller registers**, matching the table on the previous slide

```
/** @brief See Section 7.5 of the BCM2835 ARM Peripherals docu
    */
#define RPI_INTERRUPT_CONTROLLER_BASE   ( PERIPHERAL_BASE + 0
    xB200 )

/** @brief The interrupt controller memory mapped register set
    */
typedef struct {
    volatile uint32_t IRQ_basic_pending;
    volatile uint32_t IRQ_pending_1;
    volatile uint32_t IRQ_pending_2;
    volatile uint32_t FIQ_control;
    volatile uint32_t Enable_IRQs_1;
    volatile uint32_t Enable_IRQs_2;
    volatile uint32_t Enable_Basic_IRQs;
    volatile uint32_t Disable_IRQs_1;
    volatile uint32_t Disable_IRQs_2;
    volatile uint32_t Disable_Basic_IRQs;
```

Auxiliary functions

Functions to get the base address of the peripherals:

```
/** @brief The BCM2835 Interupt controller peripheral at its
    base address */
static rpi_irq_controller_t* rpiIRQController =
        (rpi_irq_controller_t*)RPI_INTERRUPT_CONTROLLER_BASE;


/**
    @brief Return the IRQ Controller register set
*/
rpi_irq_controller_t* RPI_GetIrqController( void )
{
    return rpiIRQController;
}
```

The ARM Timer Peripheral

The ARM timer is in the basic interrupt set. To enable interrupts from the ARM Timer peripheral we set the relevant bit in the **Basic Interrupt enable register**:

```
/** @brief Bits in the Enable_Basic_IRQs register to enable
    various interrupts.
    See the BCM2835 ARM Peripherals manual, section 7.5 */
#define RPI_BASIC_ARM_TIMER_IRQ          (1 << 0)
#define RPI_BASIC_ARM_MAILBOX_IRQ        (1 << 1)
#define RPI_BASIC_ARM_DOORBELL_0_IRQ     (1 << 2)
#define RPI_BASIC_ARM_DOORBELL_1_IRQ     (1 << 3)
#define RPI_BASIC_GPU_0_HALTED_IRQ       (1 << 4)
#define RPI_BASIC_GPU_1_HALTED_IRQ       (1 << 5)
#define RPI_BASIC_ACCESS_ERROR_1_IRQ     (1 << 6)
#define RPI_BASIC_ACCESS_ERROR_0_IRQ     (1 << 7)
```

and in our main C code to enable the ARM Timer IRQ:

```
/* Enable the timer interrupt IRQ */
RPI_GetIrqController()->Enable_Basic_IRQs =
    RPI_BASIC_ARM_TIMER_IRQ;
```

Before using the ARM Timer it also needs to be enabled.
Again, we map the ARM Timer peripherals register set to a C struct to give us access to the registers:

```
/** @brief See Section 14 of thed BCM2835 Peripherals PDF */
#define RPI_ARMTIMER_BASE          ( PERIPHERAL_BASE + 0xB400 )

/** @brief 0 : 16-bit counters - 1 : 23-bit counter */
#define RPI_ARMTIMER_CTRL_23BIT             ( 1 << 1 )

#define RPI_ARMTIMER_CTRL_PRESCALE_1        ( 0 << 2 )
#define RPI_ARMTIMER_CTRL_PRESCALE_16       ( 1 << 2 )
#define RPI_ARMTIMER_CTRL_PRESCALE_256      ( 2 << 2 )

/** @brief 0 : Timer interrupt disabled - 1 : Timer interrupt
    enabled */
#define RPI_ARMTIMER_CTRL_INT_ENABLE        ( 1 << 5 )
#define RPI_ARMTIMER_CTRL_INT_DISABLE       ( 0 << 5 )

/** @brief 0 : Timer disabled - 1 : Timer enabled */
#define RPI_ARMTIMER_CTRL_ENABLE            ( 1 << 7 )
#define RPI_ARMTIMER_CTRL_DISABLE           ( 0 << 7 )
...
```

Accessing the ARM Timer Register

This code gets the current value of the ARM Timer:

```
static rpi_arm_timer_t* rpiArmTimer = (rpi_arm_timer_t*)
    RPI_ARMTIMER_BASE;

rpi_arm_timer_t* RPI_GetArmTimer(void)
{
    return rpiArmTimer;
}
```

ARM Timer setup

Then, we can setup the ARM Timer peripheral from the main C code with something like:

```
/* Setup the system timer interrupt */
/* Timer frequency = Clk/256 * 0x400 */
RPI_GetArmTimer()->Load = 0x400;

/* Setup the ARM Timer */
RPI_GetArmTimer()->Control =
        RPI_ARMTIMER_CTRL_23BIT |
        RPI_ARMTIMER_CTRL_ENABLE |
        RPI_ARMTIMER_CTRL_INT_ENABLE |
        RPI_ARMTIMER_CTRL_PRESCALE_256;
```

Globally enable interrupts

We have now configured the ARM Timer and the Interrupt controller. We still need to globally globally enable interrupts, which needs some assembler code.

```
_enable_interrupts:
    mrs     r0, cpsr       @ move status to reg
    bic     r0, r0, #0x80 @ modify status
    msr     cpsr_c, r0     @ move reg to status

    mov     pc, lr
```

An LED control interrupt handler

Our **interrupt handler** should control an LED, as usual.
Note that we need to clear the interrupt pending bit in the handler, to avoid immediately re-issuing an interrupt.

```
/* @brief The IRQ Interrupt handler: blinking LED */
void __attribute__((interrupt("IRQ"))) interrupt_vector(void)
{
    static int lit = 0;

    /* Clear the ARM Timer interrupt */
    RPI_GetArmTimer()->IRQClear = 1;

    /* Flip the LED */
    if( lit ) {
        LED_OFF();
        lit = 0;
    } else {
        LED_ON();
        lit = 1;
    }
```

Kernel function

On a bare-metal system, the following wrapper code is needed to start the system:

```
/** Main function - we'll never return from here */
void kernel_main( unsigned int r0, unsigned int r1, unsigned int atags )
{
    /* Write 1 to the LED init nibble in the Function Select GPIO
       peripheral register to enable LED pin as an output */
    RPI_GetGpio()->LED_GPFSEL |= LED_GPFBIT;

    /* Enable the timer interrupt IRQ */
    RPI_GetIrqController()->Enable_Basic_IRQs = RPI_BASIC_ARM_TIMER_IRQ;

    /* Setup the system timer interrupt */
    /* Timer frequency = Clk/256 * 0x400 */
    RPI_GetArmTimer()->Load = 0x400;

    /* Setup the ARM Timer */
    RPI_GetArmTimer()->Control =
            RPI_ARMTIMER_CTRL_23BIT |
            RPI_ARMTIMER_CTRL_ENABLE |
            RPI_ARMTIMER_CTRL_INT_ENABLE |
            RPI_ARMTIMER_CTRL_PRESCALE_256;

    /* Enable interrupts! */
    _enable_interrupts();

    /* Never exit as there is no OS to exit to! */
    while(1)
```

Summary

- **Interrupts trigger an exceptional control flow**, to deal with special situations.
- Interrupts can occur at several levels:
  - hardware level, e.g. to report hardware faults
  - OS level, e.g. to switch control between processes
  - application level, e.g. to send signals within or between processes
- The **concept** is the same on all levels: execute a short sequence of code, to deal with the special situation.
- Depending on the source of the interrupt, execution will continue with the same, the next instruction or will be aborted.
- The **mechanisms** how to implement this behaviour are different: in software on application level, in hardware with jumps to entries in the interrupt vector table on hardware level

[2] Complete bare-metal application: Valvers: Bare Metal Programming in C (Pt4)