ECE346 Microcontrollers

L9 Interrupts

Interrupts

- Interrupts are like receiving a telephone call while you are in a face-to-face meeting:
  - The phone rings (i.e., an interrupt is sent)
  - Tell the person you are meeting with to please wait
  - Answer the phone, provide a response (e.g.: take a message), hang up
  - Resume the meeting, picking up where you left off.
- Efficiency demands that you:
  - Keep the phone call short
  - Don’t allow a second call to interrupt the first call (no call waiting)
  - Satisfy the caller so they do not immediately call you back after hanging up.

Interrupts

- From a computer system perspective, the CPU is always busy “in a meeting” by running your main program: it must fetch the instruction, execute it, and then advance to the next instruction.
- To get the attention of the CPU, a device can **interrupt** the currently executing instruction by sending a logic ‘1’ signal to a dedicated **interrupt request (IRQ)** input pin on the CPU. This may happen when a KEY was just pressed, or when a time delay elapses.
- The CPU usually has a handful of these interrupt request pins, often named IRQ0, IRQ1, …, IRQn. The computer system hardware designer connects each device to a different IRQ pin.
- Roughly speaking, whenever an IRQ pin is set to a 1, the CPU stops the current instruction before it is executed, and then jumps to a special subroutine called an **interrupt service routine** or **ISR**. **Each device gets its own ISR**, and this subroutine responds to events only from that device.

Interrupts

- The **ISR**:
- 1. **Cannot accept** any input parameters, or return any result.
- 2. **Must respond** to the device and handle the event; it can do this by reading the state of the device and various memory locations, then writing new values back to the device or memory locations.
- 3. **Must clear** the source of the interrupt, by telling the device to stop sending it.
- 4. **Must not pause or delay** unnecessarily; it should be quick to exit.
- 5. **Must not be interrupted** by another device.
- Keep in mind that the ISR is only for hardware device interrupts. The only way the ISR should be executed is by having the device raise its IRQ pin to '1'. That is, the main program should **never directly call the ISR as a subroutine**.
- After the ISR completes, it returns to the main program at exactly the instruction that was interrupted, and it again attempts to execute that instruction.

Interrupts Issues

- With interrupts, it is possible to **get stuck in an infinite loop**. If you forget step (3), clearing the source of the interrupt, the device will still be sending a 1 on the IRQ pin when the ISR finishes. If this happens, the CPU will be immediately interrupted after exiting the ISR; it will not get a chance to run the main program at all. In this case, the program gets stuck in an infinite loop that repeatedly runs the ISR.
- While an ISR is running, **it must not be interrupted by another device**. As a result, interrupts are automatically disabled just before the ISR starts, and they are automatically re-enabled just after the ISR finishes. This is also why **an ISR should never wait or delay** unnecessarily. If another device sends an interrupt to the CPU, it must wait for the current ISR to finish executing completely before its own ISR can be executed. Get your business done quickly, and get out!
- Since an interrupt can occur at any point in your main program, the ISR **must not** modify any CPU registers. Otherwise, the main program will behave unpredictably, as register values could change at any time and alter the execution of the program. To assure that registers will not be changed, the interrupt process will save and restore register values from the stack. This process is explained next.

Exceptions & Interrupts

- The A9 processor supports eight types of exceptions, including the reset exception and the **interrupt request** (**IRQ**) exception, as well as a number of exceptions related to error conditions.
- Exception processing uses a table in memory, called the ***vector table***. This table comprises eight words in memory and has one entry for each type of exception.
- When an exception occurs, the A9 processor stops the execution of the program that is currently running and then fetches the instruction stored at the corresponding vector table entry. The vector table usually starts at the address **0x00000000** in memory. The first entry in the table corresponds to the reset vector, and the IRQ vector uses the seventh entry in the table, at the address **0x00000018**.
- The IRQ exception allows I/O peripherals to generate interrupts for the A9 processor. All interrupt signals from the peripherals are connected to a module in the processor called the **generic interrupt controller** (**GIC**).

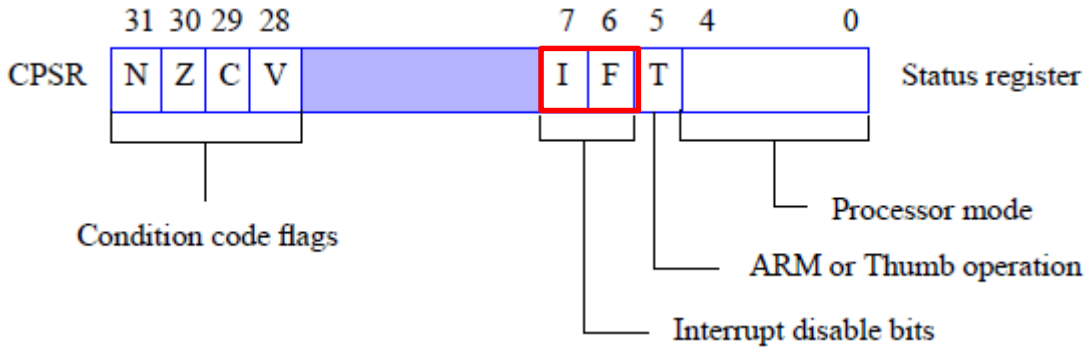
Exceptions & Interrupts

- The GIC allows individual interrupts for each peripheral to be either enabled or disabled. When an enabled interrupt happens, the GIC causes an IRQ exception in the A9 processor.
- Since the same vector table entry is used for all interrupts, the software for the interrupt service routine must determine the source of the interrupt by querying the GIC.
- Each peripheral is identified in the GIC by an interrupt identification (ID) number.

I/O Peripheral	Interrupt ID #
A9 Private Timer	29
HPS GPIO1	197
HPS Timer 0	199
HPS Timer 1	200
HPS Timer 2	201
HPS Timer 3	202
FPGA Interval Timer	72
FPGA Pushbutton KEYs	73
FPGA Second Interval Timer	74
FPGA JTAG	80
FPGA JP1 Expansion	83
FPGA JP7 Expansion	84
FPGA Arduino Expansion	85

Interrupts in ARM Cortex-A9

- There is a **Current Program Status Register**, **CPSR**, in ARM Cortex-A9 processer.



- Interrupt-disable bits, ***I*** and ***F***, where
  - – ***I*** = 1 disables the IRQ interrupts
  - – ***F*** = 1 disables FIQ interrupts

Operating Modes

$CPSR_{4-0}$	Operating Mode
10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

- The ARM processor can operate in a number of different modes, as follows:
  - **User mode** – is the basic mode in which application programs run. This is an unprivileged mode, which has restricted access to system resources.
  - **System mode** – provides full access to system resources. It can be entered only from one of the exception modes listed below.
  - **Supervisor mode** – is entered when a software interrupt is raised by a program executing a *Supervisor Call* instruction, SVC. It is also entered on reset or power-up.
  - **Abort mode** – is entered if a program attempts to access a non-existing memory location.
  - **Undefined mode** – is entered if the processor attempts to execute an unimplemented instruction.
  - **IRQ mode** – is entered in response to a normal interrupt request from an external device.
  - **FIQ mode** – is entered in response to a *fast interrupt* request from an external device. It is used to provide faster service for more urgent requests.
- The current operating mode is indicated in the processor status bits $\boldsymbol{CPSR_{4-0}}$.

Supervisor Mode Move Instructions

- In the Supervisor mode, the special Move instructions, MRS and MSR, can be used to access the processor status registers CPSR and SPSR_svc (Stack Pointer Status register). The instruction

MRS Rd, CPSR

- copies the contents of CPSR into register Rd. Writing into the status registers can be done by affecting one or more fields of the register. The processor status registers have four fields of eight bits, identified by the field specifiers _f, _s, _x and _c, which correspond to PSR_{31-24}, PSR_{23-16}, PSR_{15-8} and PSR_{7-0}, respectively. Thus, the instruction

MSR CPSR_c, Rd

- copies the contents of Rd into CPSR7-0, which affects only the processor mode and interrupt disable bits. All bits can be affected by the instruction

MSR CPSR_cxsf, Rd

- We should note that the field specifiers must be used in the MSR instruction; otherwise, an error will occur at compile time.
- In an exception mode, such as IRQ, it is the banked saved status register that is accessed. Thus,

MRS Rd, SPSR

- copies the contents of SPSR_irq into register Rd.

Software Interrupt

- A software interrupt, which is called a software exception in ARM literature, occurs when an SVC instruction is encountered in a program. This instruction causes the processor to switch into Supervisor mode. The address of the next instruction is saved in the banked register LR_svc and the contents of CPSR are saved in SPSR_svc. Then, the address of entry 8 in the exception vector table is loaded into the Program Counter. A branch instruction at that location leads to to the required exception-service routine.
- Upon completion of the exception-service routine, a return to the interrupted program can be realized with the instruction

```
MOVS  PC, LR
```

- Note that the suffix S in the OP-code mnemonic normally specifies that the Condition Code flags should be set. However, when the destination register is PC, the suffix S causes the saved contents in register SPSR_mode, in this case SPSR_svc, to be loaded into the processor status register CPSR. Since this instruction also loads the saved return address into PC, a return to the interrupted program is completed.
- A common use of the software interrupt is to transfer control to a different program, such as an operating system.

Hardware Interrupt

- Hardware interrupts can be raised by external sources, such as I/O devices, by asserting one of the processor's interrupt-request inputs, IRQ or FIQ. When the processor receives a hardware interrupt request, it enters the corresponding exception mode to service the interrupt. It also saves the contents of PC and CPSR.
- The saved contents of the PC are supposed to be the return address. However, **this is not the case with the ARM Cortex-A9 processor**. This processor prefetches instructions for execution. While the current instruction is being executed, the next instruction is prefetched and its processing is started. This means that the Program Counter points to the instruction after the prefetched one. Namely, the updated contents of PC are the address of the current instruction plus 8.
- Since the interrupt is serviced upon completion of the current instruction, the next prefetched instruction is discarded and it must be executed upon return from the interrupt. Therefore, the address saved in the link register must be decremented by 4 prior to returning to the interrupted program. This can be done by having

SUBS PC, LR, #4

- as the last instruction in the exception-service routine. Note that the suffix S causes a proper return to the interrupted program, as explained above.

IRQ Interrupts

- Upon accepting an IRQ interrupt request, the processor saves the contents of CPSR in the SPSR_irq register, and it saves the contents of PC in the link register LR_irq. It also sets the mode bits in CPSR to denote the IRQ exception mode, and it sets the I bit to 1 to disable further IRQ interrupts. Then, it executes the instruction at location 0x018 of the exception vector table, which has to cause a branch that leads to the IRQ exception-service routine.
- The return from the exception-service routine should be performed with the instruction

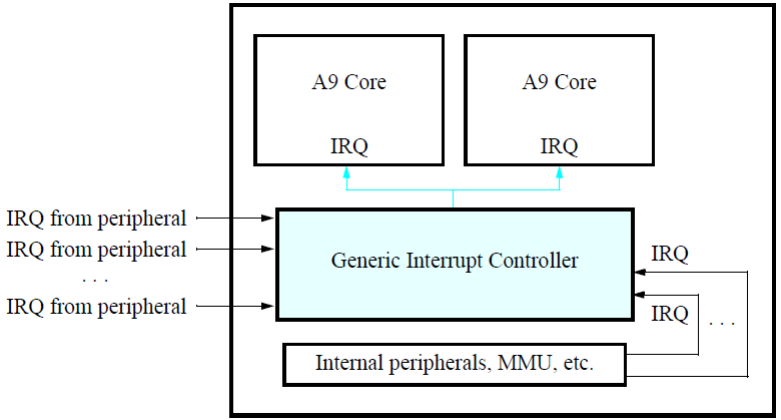
SUBS PC, LR, #4

Nested Interrupts

- When two or more interrupts or exceptions occur at different priority levels, causing the processor to enter different modes of operation, their servicing can proceed immediately because the banked registers in various modes are used to save the critical information about the interrupted program.
- However, if multiple interrupts can occur at the same priority level, typically multiple IRQ requests, then it is necessary to nest the exception-service routines. This includes saving the contents of the banked link register, LR_mode, on the stack before enabling subsequent requests.
- Before returning from the corresponding exception-service routine, the contents of the register must be restored.

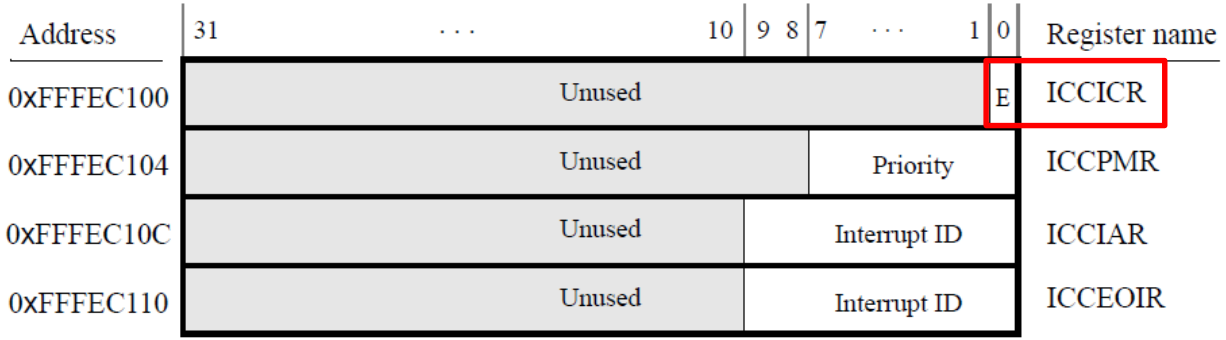
Generic Interrupt Controller(GIC)

- The ARM **generic interrupt controller (GIC)** is a part of the ARM A9 MPCORE processor.
- The GIC is connected to the IRQ interrupt signals of all I/O peripheral devices that are capable of generating interrupts.
- Most of these devices are normally external to the A9 MPCORE, and some are internal peripherals (such as timers).
- The GIC included with the A9 MPCORE processor in the Altera Cyclone V SoC family handles up to 255 sources of interrupts. When a peripheral device sends its IRQ signal to the GIC, then the GIC can forward a corresponding IRQ signal to one or both of the A9 cores.
- Software code that is running on the A9 core can then query the GIC to determine which peripheral device caused the interrupt, and take appropriate action.



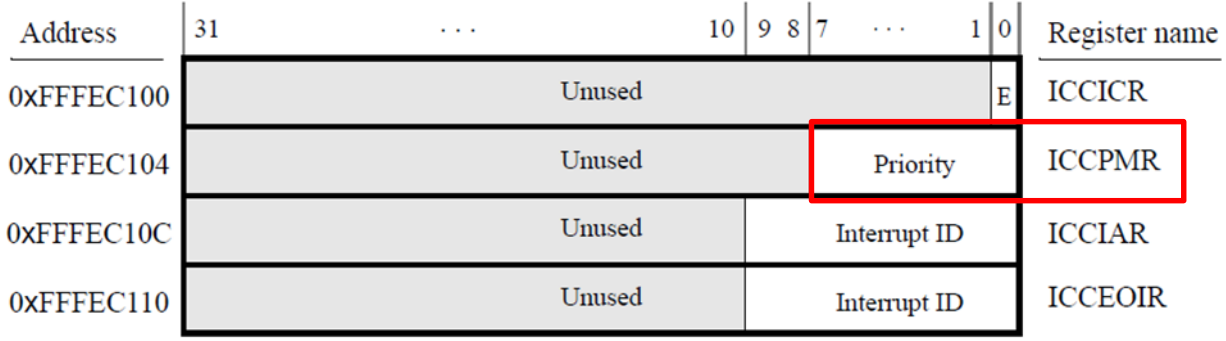
CPU Interface Control Register (ICCICR)

- The CPU Interface in the GIC is used to send IRQ signals to the A9 cores. There is one CPU Interface for each A9 core in the MPCORE.
- The ***CPU Interface Control Register*** **(ICCICR)** is used to enable forwarding of interrupts from the CPU Interface to the corresponding A9 core. Setting bit E = 1 in this register enables the sending of interrupts to the A9 core, and setting E = 0 disables these interrupts.



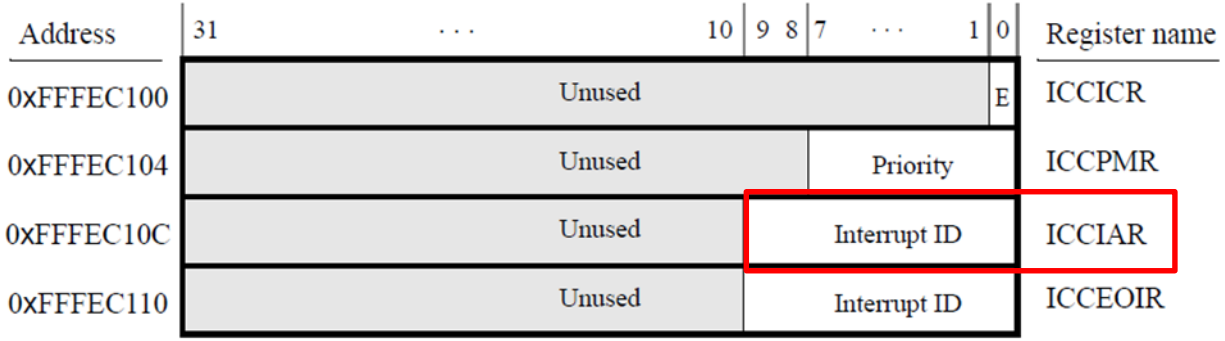
Interrupt Priority Mask Register (ICCPMR)

- The ***Interrupt Priority Mask Register*** **(ICCPMR)** is used to set a threshold for the priority-level of interrupts that will be forwarded by a CPU Interface to an A9 core.
- Only interrupts that have a priority level greater than the Priority field in ICCPMR will be sent to an A9 processor by its CPU Interface.
- Lower priority values represent higher priority, meaning that level 0 is the highest priority and level 255 is the lowest.
- Setting the Priority field in ICCPMR to the value 0 will prevent any interrupts from being generated by the CPU Interface.



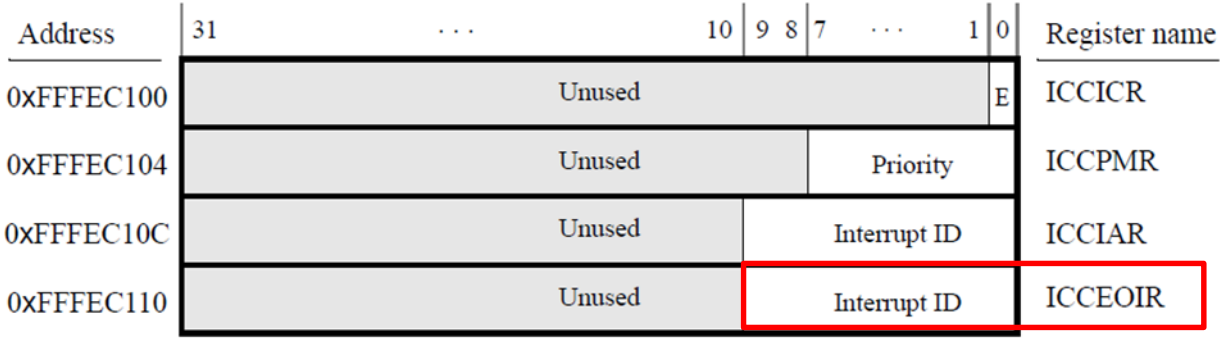
Interrupt Acknowledge Register (ICCIAR)

- The ***Interrupt Acknowledge Register*** **(ICCIAR)** contains the Interrupt ID of the I/O peripheral that has caused an interrupt.
- When an A9 processor receives an IRQ signal from the GIC, software code (i.e., the interrupt handler) running on the processor must read the ICCIAR to determine which I/O peripheral has caused the interrupt.

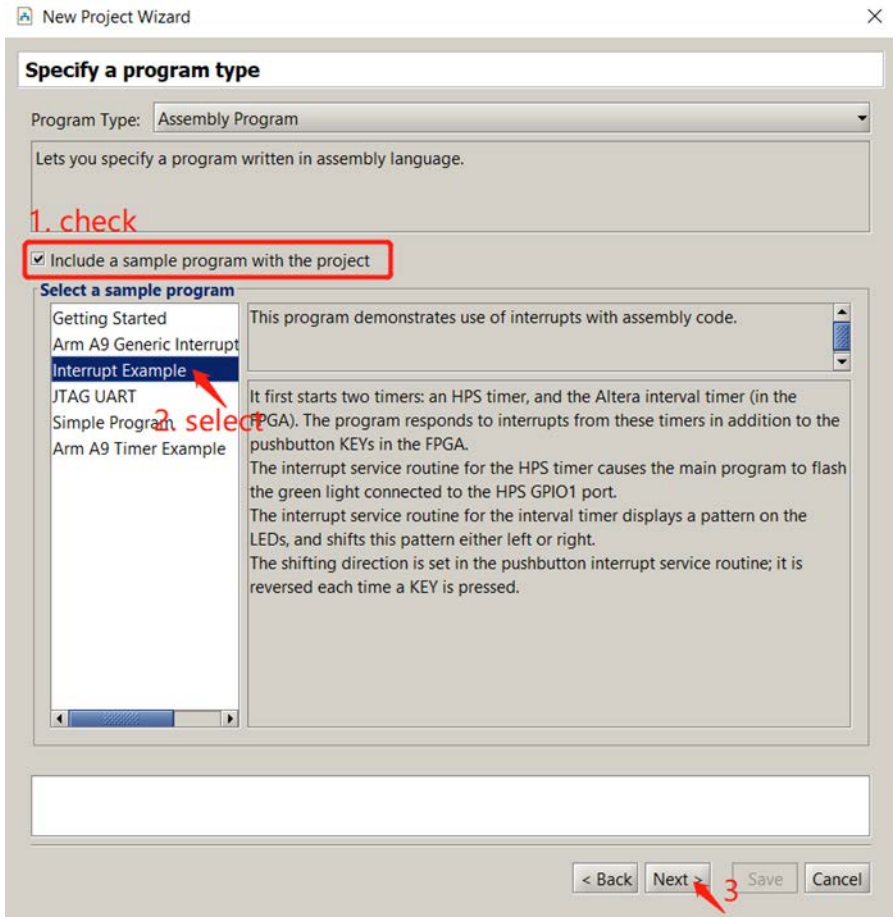


End of Interrupt Register (ICCEOIR)

- After the A9 processor has completed the handling of an IRQ interrupt generated by the GIC, the processor must then clear this interrupt from the CPU Interface.
- This action is accomplished by writing the appropriate Interrupt ID into the ***Interrupt ID*** field at the ***End of Interrupt Register*** **(ICCEOIR)**.
- After writing into the ICCEOIR, the interrupt handler software can then return control to the previously-interrupted main program.



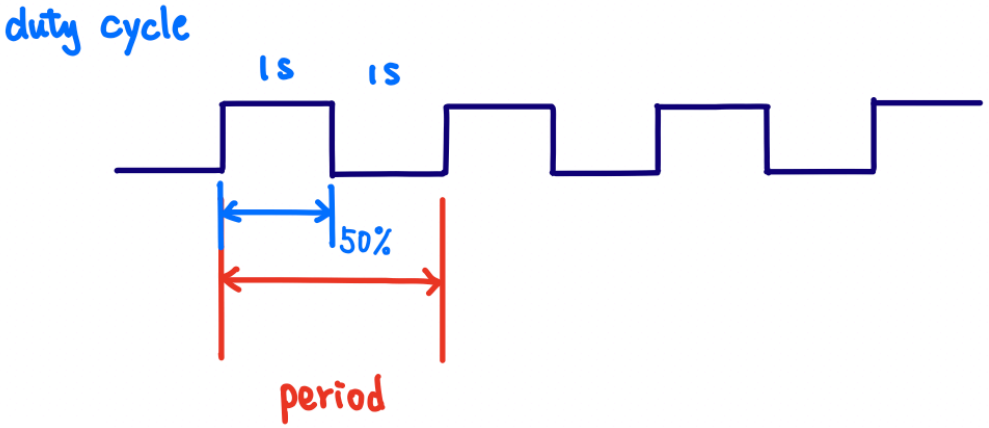
Interrupt Example



- One example code can be found from the Monitor Program.
- All .s files are provided.

Example – Changing duty cycle

- If the LEDs flash with the period of 2 seconds (1 second ON and 1 second OFF), it has the duty cycle of 50%.

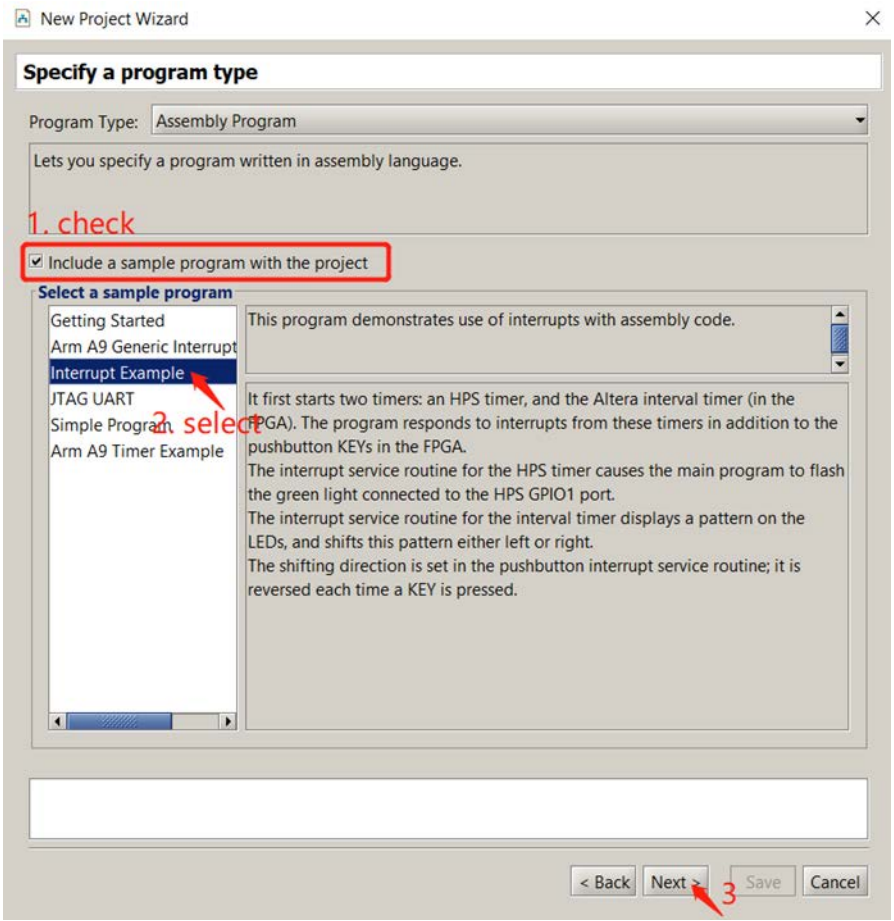


- https://en.wikipedia.org/wiki/Duty_cycle

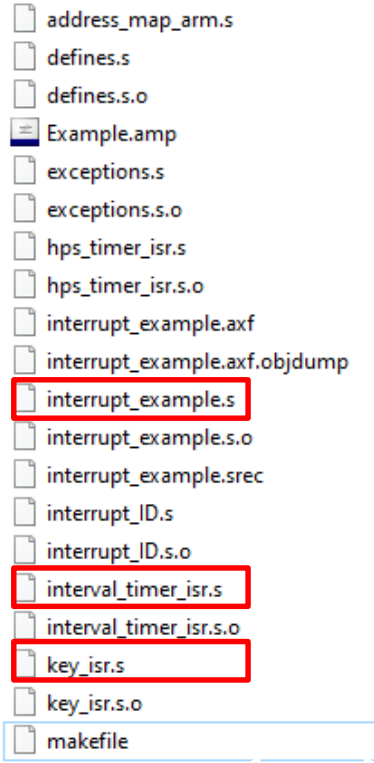
Example – Changing duty cycle

- In this example, the duty cycle will be altered by 10% each time when the pushbutton ***KEY1*** is pressed, and the on-board DIP switch (SW1) determines the duty cycle to be changed increasingly (ON) or decreasingly (OFF).
- The upper bound of duty cycle is 90% and the lower bound is 10%. If we try to further decrease the duty cycle when the duty cycle has already reached the lower bound 10%, then the system should ignore the command and the duty cycle remains 10%, vice versa for the upper bound.

Example – Changing duty cycle



- Implement this example based on the code given in the Monitor Program software (See left figure).
- We just need to modify the **interrupt_example.s**, **interval_timer_isr.s** and **key_isr.s** since we use the interval timer ISR and the KEY ISR in this example.
- More detailed explanation of those source code can be found in the DE10-Nano User Manual.



Modification in the *interrupt_example.s*

```
/* Change to SVC (supervisor) mode with interrupts disabled */
        MOV     R1, #INT_DISABLE | SVC_MODE
        MSR     CPSR_c, R1                  // change to supervisor mode
        LDR     SP, =DDR_END - 3            // set SVC stack to top of DDR3 memory

        BL      CONFIG_GIC                  // configure the ARM generic interrupt
                                            // controller
        //BL      CONFIG_HPS_TIMER            // configure the HPS timer
        BL      CONFIG_INTERVAL_TIMER       // configure the Altera interval timer
        BL      CONFIG_KEYS                 // configure the pushbutton KEYs
```

1. Commenting on the CONFIG_HPS_TIMER line (Line 49) and Line 74 to 85 in the original code since we do not need the HPS Timer here.

```
/* Configure the Altera interval timer to create interrupts at 50-msec intervals */
CONFIG_INTERVAL_TIMER:
        LDR     R0, =TIMER_BASE
/* set the interval timer period for scrolling the LED displays */
        LDR     R1, =50000000               // = 0.5 sec
        STR     R1, [R0, #0x8]              // store the low half word of counter
                                            // start value
        LSR     R1, R1, #16
        STR     R1, [R0, #0xC]              // high half word of counter start value

                                            // start the interval timer, enable its interrupts
        MOV     R1, #0x7                    // START = 1, CONT = 1, ITO = 1
        STR     R1, [R0, #0x4]
        BX      LR

/* Configure the pushbutton KEYS to generate interrupts */
CONFIG_KEYS:
                                            // write to the pushbutton port interrupt mask register
        LDR     R0, =KEY_BASE               // pushbutton key base address
        MOV     R1, #0x3                    // set interrupt mask bits
        STR     R1, [R0, #0x8]              // interrupt mask register is (base + 8)
        BX      LR
```

2. Keep the configuration of Interval Timer and the pushbutton KEYS.

3. Add the values of periods to calculate the ON/OFF period. The total period is 2 seconds, the ON period is initialized to 1 second, and the increase and decrease segments are set to 0.2 seconds.

```
/* Global variables */
.global     TICK
TICK:
.word       0x0                             // used by HPS timer
.global     Total_Period
Total_Period:
.word       200000000                       // period = 2 sec
.global     On_Period
On_Period:
.word       100000000                       // period = 1 sec initially
.global     Segment
Segment:
.word       20000000                        // increment = 0.2 sec
```

Modification in the *interval_timer_isr.s*

```
.include    "address_map_arm.s"
.include    "defines.s"
/* externally defined variables */
.extern     Total_Period
.extern     On_Period
/*****************************************************************************
 * Interval timer interrupt service routine
 *
 * Change the duty cycle on the LED lights. The duty cycle
 * is determined by the external variable KEY_PRESSED.
 *****************************************************************************/
.global     TIMER_ISR
TIMER_ISR:
        PUSH    {R4-R7}
        LDR     R1, =TIMER_BASE // interval timer base address
        MOVS    R0, #0
        STR     R0, [R1]        // clear the interrupt

        LDR     R2, =LED_BASE   // LED base address
        LDR     R3, =0xFFFFFFFF // set up a pointer to the pattern for LED displays
        LDR     R4, =Total_Period
        LDR     R4, [R4]
        LDR     R5, =On_Period
        LDR     R5, [R5]

        LDR     R6, [R2]        // load pattern for LED displays
        CMP     R6, #0
        BEQ     Display         // Change LEDs and R5=On_Period

Off_Period:
        SUB     R5, R4, R5      // Off_Period = Total_Period - On_Period

Display:
        EOR     R6, R3, R6
        STR     R6, [R2]        // store to LEDs

Set_Timer:
        MOV     R0, #0x8
        STR     R0, [R1, #0x4]

        STR     R5, [R1, #0x8] //store the low half word of period start value
        LSR     R5, R5, #16
        STR     R5, [R1, #0xC] //store high half word of period start value
        MOV     R0, #0x7       // START = 1, CONT = 1, ITO = 1 (7 = 111)
        STR     R0, [R1, #0x4] // start the interval timer, enable its interrupts

END_TIMER_ISR:
        POP     {R4-R7}
        BX      LR
.end
```

1. Load period values into registers (R4 for the total period, R5 for the ON period).

2. Calculate the OFF period.

3. Display the values on LEDs (all LEDs are on).

4. Set timer, enable the interval timer's interrupts.

Modification in the *key_isr.s*

```
.include    "address_map_arm.s"
.include    "defines.s"
.extern     Total_Period
.extern     On_Period
.extern     Segment
/****************************************************************************************
 * Pushbutton KEY - Interrupt Service Routine
 ****************************************************************************************/
.global     KEY_ISR
KEY_ISR:
        PUSH    {R4-R7}
        LDR     R0, =KEY_BASE   // base address of pushbutton KEY parallel port
/* KEY[1] is the only key configured for interrupts, so just clear it. */
        LDR     R1, [R0, #0xC]  // read edge capture register
        STR     R1, [R0, #0xC]  // clear the interrupt

        LDR     R1, =SW_BASE    // check the direction determined by the on-board switch
        LDR     R4, =Total_Period
        LDR     R4, [R4]
        LDR     R5, =On_Period
        LDR     R5, [R5]
        LDR     R6, =Segment
        LDR     R6, [R6]

        LDR     R2, [R1]        // increase (>0) or decrease (0)
        CMP     R2, #0
        BEQ     DECR
INCR:
        SUB     R4, R4, R6      // upper bound of duty cycle is 90% (2-0.2=1.8 sec)
        CMP     R5, R4          // if the period reaches the upper bound
        BEQ     END_KEY_ISR     // it will not continue to increase
        ADD     R5, R5, R6      // increase the period by Segment
        LDR     R7, =On_Period
        STR     R5, [R7]        // update the ON period
        B       END_KEY_ISR
DECR:
        CMP     R5, R6          // lower bound of duty cycle is 10% (0.2 sec)
        BEQ     END_KEY_ISR     // it will not continue to decrease.
        SUB     R5, R5, R6      // otherwise, decrease the period by Segment
        LDR     R7, =On_Period
        STR     R5, [R7]        // update the ON period
END_KEY_ISR:
        POP     {R4-R7}
        BX      LR
.end
```

1. Load base address of pushbutton KEY and the on-board switch.

2. Load period values into registers (R4 for the total period, R5 for the ON period and R6 for the Segment).

3. Check if increase (ON) or decrease (OFF) based on the value of switch.

4. Increasing.

5. Decreasing.