

Namn:
Laborationen godkänd:

Computer Organization 6 hp



LUNDS TEKNISKA HÖGSKOLA
Lunds universitet

Interrupt handling

Purpose

The purpose of this lab assignment is to give an introduction to interrupts, i.e. asynchronous events caused by external devices to which the processor may need to respond. One should learn how to write (1) initialization procedures for the devices that can cause interrupts, (2) initialization procedures for the processor such that it can respond to interrupts caused by different external devices and (3) interrupt handlers, i.e. different routines that should be executed as a response to the interrupts that have been caused by any of the different external devices.

Interrupts

An *interrupt* refers to an external event that needs immediate attention from the processor. An interrupt signals the processor, indicating the need of attention, and requires interruption of the current code the processor is executing. As a response, the processor suspends its current activities, saves its state and executes a particular function to service the event that has caused the interruption. Such function is often called an *interrupt handler* or an *interrupt service routine*. Once the processor has responded to the interrupt, i.e. after the processor has executed the interrupt handler, the processor resumes its previously saved state and resumes the execution of the same program it was executing before the interrupt occurred.

The interrupts are often caused by external devices that communicate with the processor (Interrupt-driven I/O). Whenever these devices require the processor to execute a particular task, they generate interrupts and wait until the processor has acknowledged that the task has been performed. To be able to receive and respond to interrupts a processor is equipped with an interrupt port. Through the interrupt port the processor can receive the interrupt request signals and can respond to these requests through the interrupt acknowledge signals. An illustration is presented in **Figure 1**.

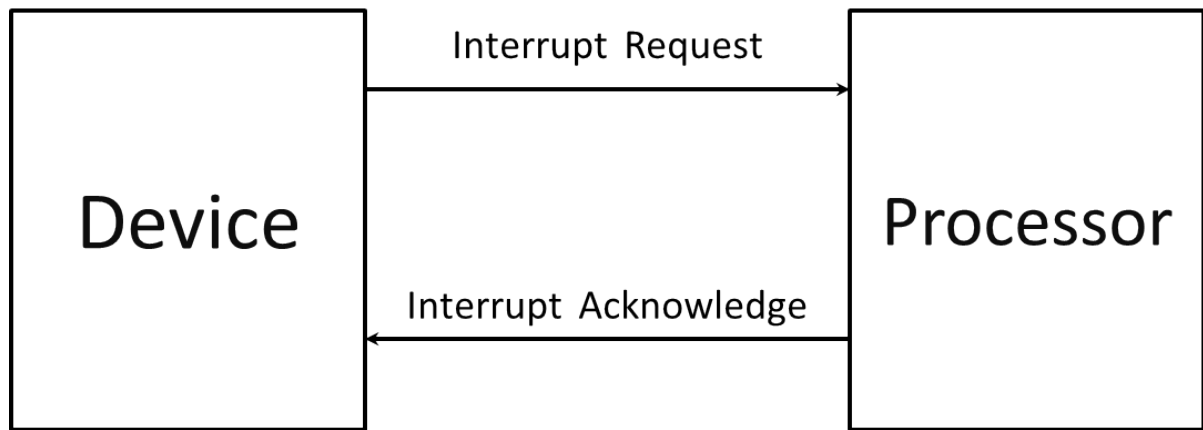


Figure 1. Interrupt request and acknowledge

The interrupt port of the MicroBlaze processor contains one interrupt request line and therefore, can only receive a single interrupt signal. However, several devices in the hardware platform can generate interrupts. To manage multiple interrupts, the processor is connected to an interrupt controller, i.e. a programmable I/O device that can receive multiple interrupts and can be programmed such that it can generate a single interrupt request output that is connected to the interrupt request input of the MicroBlaze processor.

Interrupt controller

In the hardware platform, the interrupt controller is used to merge the interrupts that can be generated by the two timer I/O devices. The interrupt controller has a set of registers, mapped to known memory addresses. These registers are used for storing interrupt vector addresses, checking the status of the interrupt request lines, enabling and acknowledging interrupts. An illustration of the interrupt system, i.e. how the timers, the interrupt controller and the MicroBlaze processor are connected, is presented in **Figure 2**.

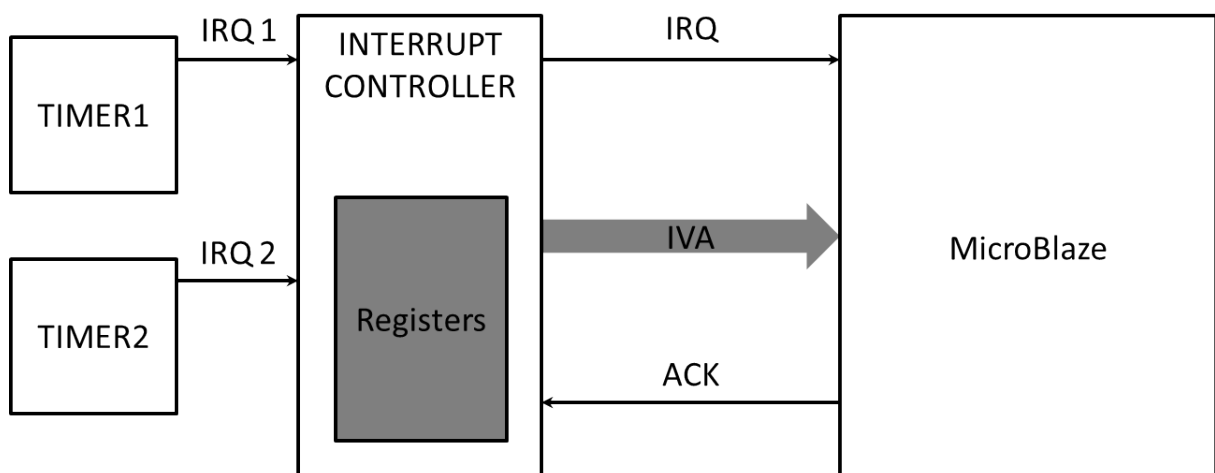


Figure 2. Interrupt system

The interrupt controller contains a 32-bit wide Master Enable Register (MER). The least significant bit, i.e. the bit at bit-position zero, of this register acts as a Global Interrupt Enable (GIE) and it is responsible for enabling/disabling the IRQ output of the interrupt controller that is connected to the MicroBlaze processor. If the GIE bit is set to '1', then the interrupt controller is capable of asserting the IRQ output and therefore, it can interrupt the program that the processor is executing. Otherwise, the interrupt controller cannot assert the IRQ output. The bit at bit-position one of the MER, acts as a Hardware Interrupt Enable (HIE) and it enables the interrupt controller to detect (capture) the interrupt events from the various devices connected through the input interrupt request lines. By default, the HIE bit is set to zero, and therefore, the interrupt controller does not detect any interrupt requests. Once the HIE bit is set to one, the value of this bit cannot be modified.

Provided that the HIE bit in the MER is set, the interrupt controller can capture the interrupts coming from the input interrupt request lines, e.g. IRQ1 and IRQ2, and stores the status of the interrupt request lines in an Interrupt Status Register (ISR). The interrupt request lines are mapped to particular bit-positions in the ISR. When an interrupt request line is active (asserted), i.e. either drives a logic '1' or a low-to-high transition has been detected, '1' is written to the corresponding bit position in the ISR.

As the interrupt controller is a programmable (configurable) device, it is possible to configure the controller such that it only signals interrupts to the MicroBlaze processor when some of the devices have generated an interrupt. This is done by writing to the Interrupt Enable Register (IER). Writing a '1' at a particular bit position in the IER enables the corresponding bit of the ISR to cause assertion of the IRQ signal that is connected to the MicroBlaze processor (see **Figure 2**), provided that the GIE bit in the MER is set to one. To disable interrupts from a particular device, i.e. often called interrupt masking, '0' has to be written at a particular bit-position in the IER to disable the corresponding bit of the ISR to cause assertion of the IRQ signal. For example, if an interrupt request line of a device is mapped to bit-position zero of the ISR, then interrupt requests from this device can reach the MicroBlaze processor only if '1' is written to bit-position zero in the IER, i.e. the controller is configured to enable interrupts generated by this device, and the GIE bit has been set. However, if '0' is written to bit-position zero in the IER, then the interrupt requests from this device do not reach the MicroBlaze processor, i.e. the interrupt controller masks (disables) the interrupts coming from this device. Note that when interrupts are disabled, i.e. when all bits in the IER are set to '0', it does not stop the devices to generate interrupts and modify the contents of the ISR. Thus, even if the controller is configured to disable the interrupts coming from a particular device, whenever that device generates an interrupt it will set the corresponding bit in the ISR register, provided that the HIE bit in the MER is set.

Provided that the GIE bit in the MER is set, if the interrupt controller enables interrupts from a given device, then each time the device generates an interrupt, i.e. asserts the interrupt request line, the interrupt controller asserts the IRQ line connected to the MicroBlaze processor, and provides the Interrupt Vector Address (IVA), i.e. the memory location where the interrupt service routine is stored, to the processor (see **Figure 2**). For each device connected to the interrupt controller, there is one Interrupt Vector Address Register (IVAR) that keeps the memory address of the interrupt service routine that should be executed by the processor in order to service the interrupt request coming from a particular device. When configuring the interrupt controller, the address of each interrupt service routine should be written in the corresponding IVAR register.

Finally, to acknowledge (clear) an interrupt, the Interrupt Acknowledge Register (IAR) is used. Writing a '1' at a particular bit position in the IAR, clears (set to zero) the corresponding bit-position in the ISR.

Timer I/O

The timer I/O is a configurable device that can generate periodic interrupts. This device contains three 32-bit wide registers, receives one input clock signal and provides one output signal that serves as an interrupt request. The timer I/O is illustrated in **Figure 3**.

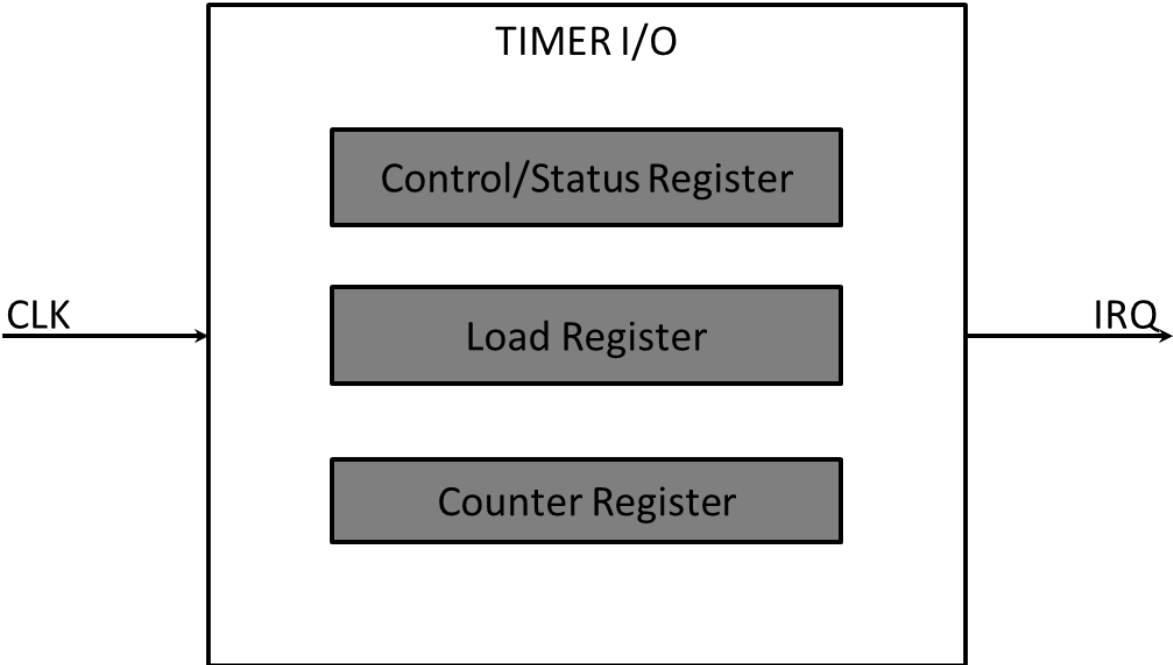


Figure 3. Timer I/O

The intended operation of the timer I/O is to generate periodic interrupts. In such mode, the Counter Register (CR) keeps the current count value. When the counter is running, each clock cycle the count value is updated (incremented or decremented based on a particular bit in the Control/Status Register (CSR)). When the counter overflows, i.e. when the state of the CR changes from “0xFFFFFFFF” to “0x00000000” or from “0x00000000” to “0xFFFFFFFF”, an interrupt is generated. To provide periodic interrupts, whenever an interrupt is generated the counter is automatically reloaded with the value stored in the Load Register (LR) and proceeds with counting. This is done by setting a particular bit in the CSR.

To control the operation of the timer I/O the CSR register is used. Next, we discuss the different bits in the CSR. The least significant bit is at bit-position zero.

The bit at bit-position one in the CSR, i.e. the Up/Down Count (UDC) bit, defines whether the counter counts up or down. When the UDC bit is set to ‘0’ the counter counts up, otherwise the counter counts down.

The bit at bit-position four in the CSR, i.e. the Auto Reload/Hold (ARH) bit, defines what happens when the counter overflows. If the ARH bit is set to ‘1’, then the CR is reloaded with the value provided in the LR each time when the CR overflows, otherwise the counter holds at the termination value (does not run).

The bit at bit-position five in the CSR, i.e. the Load (L) bit, is responsible for loading the value of the LR in the CR. When the L bit is set to ‘1’, the value of the CR is loaded with the value available in the LR. Setting this bit to ‘1’ prevents the counter from running.

The bit at bit-position six in the CSR, i.e. the Enable Interrupt (EI) bit, enables the assertion of the interrupt signal. When the EI bit is set to ‘1’, then the each time the CR overflows, the IRQ signal is asserted. Otherwise, the IRQ signal is not asserted.

The bit at bit-position seven in the CSR, i.e. the Enable (E) bit, is responsible for running/halting the counter. When the E bit is set to ‘1’, the counter runs, otherwise the counter halts. When setting the E bit to ‘1’, the L bit needs to be set to ‘0’.

The bit at bit-position eight in the CSR, i.e. the Interrupt Flag (IF) bit, indicates if the condition for an interrupt has occurred. If the EI bit has previously been set, and an interrupt condition has occurred (the counter has overflowed), then the status of this bit is set to ‘1’ and it remains ‘1’ while driving the IRQ output signal. To acknowledge that the interrupt has been serviced, the processor needs to write ‘1’ at the IF bit in the CSR of the timer I/O. By doing so, the IF bit is cleared.

The clock signal provided to the timer I/O runs at a frequency 100MHz.

MicroBlaze Interrupts

Similar to the interrupt controller and the timer I/O, the MicroBlaze processor can enable/disable interrupts. This is done by accessing the Machine Status Register (**rmsr**) which is one of the special purpose registers. The bit at bit-position one, i.e. the Interrupt Enable (IE) bit, is responsible for enabling/disabling interrupts. When the IE bit is set to '1', the processor can respond to interrupts. To set this bit, one needs to use the specific instructions **MFS** (move from special register) and **MTS** (move to special register).

When the IE bit is enabled, at an interrupt event, the Program Counter (**rpc**) register is copied to **r14**, and the address of the particular interrupt handler (interrupt service routine) is loaded into the **rpc** register. Thus, the processor proceeds by executing the interrupt service routine.

Interrupt Service Routine (Interrupt Handler)

An interrupt service routine follows few important steps. The first step is to save the state of all registers. This is done by updating the stack pointer (allocating space on the stack) and pushing the contents of all the registers on the stack. The second step is specific to what the handler should do. The final step, is to restore the registers from the stack, update the stack pointer (deallocate space from the stack) and return from the interrupt service routine. Returning from the interrupt service routine is done by using the **RTID** (return from interrupt) instruction, where the return address is provided through register **r14**.

Assignment 1.

The purpose of this assignment is to provide an example application that also deals with interrupts. The goal is to create an application where the bottom-most horizontal segment, i.e. we refer to it as *blinker*, in one of the seven-segment displays blinks periodically twice in a second. Hence, within one second, in the first quarter of the second the blinker is on, in the second quarter the blinker is off, in the third quarter it is on again and finally in the last quarter the blinker is off again. Pressing the up, down, left or right push buttons changes the index of the seven-segment display where the *blinker* appears. The action of pressing one of these four pushbuttons depends on the “*application state*”. The “*application state*” is defined by how many times the middle pushbutton has been pressed. The following states are available:

- *State 0* : The middle button has not been pressed at all since the application was started, or it has been pressed four times since the last time the application was running in *State 0*. In this state, if the left pushbutton is pressed, the *blinker* is moved to the seven-segment display one position to the left, unless the current position of the *blinker* is the left-most seven-segment display. If the right pushbutton is pressed, the *blinker* is moved to the seven-segment display one position to the right, unless the current position of the *blinker* is the right-most seven-segment display. No action is taken if the up or the down pushbuttons are pressed in this state. If the middle pushbutton is pressed, the *application state* changes to the following state, i.e. *State 1*.
- *State 1* : The middle button has been pressed once since the application was started, or it has been pressed four times since the last time the application was running in *State 1*. In this state, if the left pushbutton is pressed, the *blinker* is moved to the seven-segment display one position to the right, unless the current position of the *blinker* is the right-most seven-segment display. If the right pushbutton is pressed, the *blinker* is moved to the seven-segment display one position to the left, unless the current position of the *blinker* is the left-most seven-segment display. No action is taken if the up or the down pushbuttons are pressed in this state. If the middle pushbutton is pressed, the *application state* changes to the following state, i.e. *State 2*.
- *State 2* : The middle button has been pressed twice since the application was started, or it has been pressed four times since the last time the application was running in *State 2*. In this state, if the up pushbutton is pressed, the *blinker* is moved to the seven-segment display one position to the left, unless the current position of the *blinker* is the left-most seven-segment display. If the down pushbutton is pressed, the *blinker* is moved to the seven-segment display one position to the right, unless the current position of the *blinker* is the right-most seven-segment display. No action is taken if the left or the right pushbuttons are pressed in this state. If the middle pushbutton is pressed, the *application state* changes to the following state, i.e. *State 3*.
- *State 3* : The middle button has been pressed three times since the application was started, or it has been pressed four times since the last time the application was running in *State 3*. In this state, if the up pushbutton is pressed, the *blinker* is moved to the seven-segment display one position to the right, unless the current position of the *blinker* is the right-most seven-segment display. If the down pushbutton is pressed, the *blinker* is moved to the seven-segment display one position to the left, unless the current position of the *blinker* is the left-most seven-segment display. No action is taken if the left or the right pushbuttons are pressed in this state. If the

middle pushbutton is pressed, the *application state* changes to the following state, i.e. *State 0*.

Initially when the application starts running, the *application state* is *State 0*, and the *blinker* appears on the right-most seven-segment displays. As an indicator to the current *application state*, we shall use the LEDs.

The state transition diagram is presented in **Figure 4**.

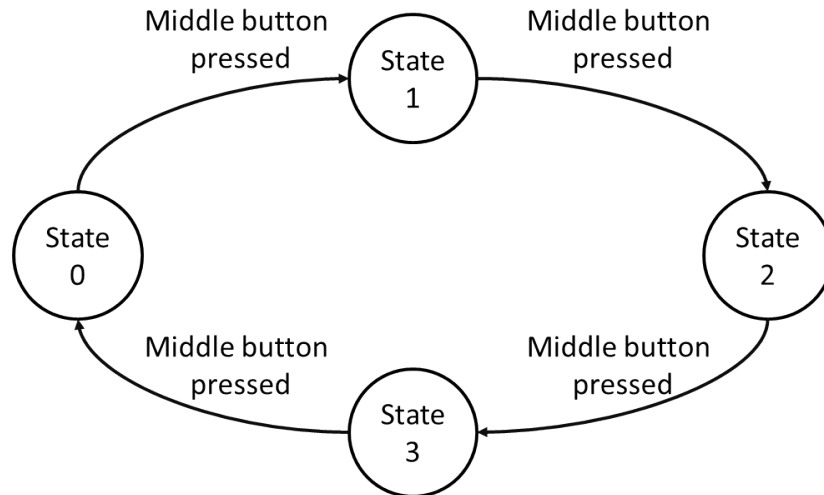


Figure 4. State transition diagram for the Blinker application

To avoid the problem of bouncing, the pushbuttons need to be read once every 20ms. To change the state of the *blinker*, i.e. on or off, an event that appears with a frequency four times in one second, i.e. once every 250ms, is required.

To have these timing events satisfied, we shall use the timers to generate periodic interrupts with the given rates. Therefore, the first step is to initialize the timers. This is done by configuring the CSR and the LR register in both timer devices. These registers are mapped to known memory addresses. To access these registers we shall define pointers, i.e. `TIMER1_CTRL` and `TIMER1_LOAD` that point to the CSR and the LR in `TIMER1`; and `TIMER2_CTRL` and `TIMER2_LOAD` that point to the CSR and the LR in `TIMER 2`.

In the project, create a new header file “timers.h”. Copy the following lines in the header file “timers.h” between the “#define” and “#endif” directives.

```
#define TIMER1_CTRL (unsigned int *) 0x41C10000  
#define TIMER1_LOAD (unsigned int *) 0x41C10004  
#define TIMER2_CTRL (unsigned int *) 0x41C00000  
#define TIMER2_LOAD (unsigned int *) 0x41C00004
```

To initialize the timers, we shall define an initialization function. This function will be used to program the timers to generate interrupts with the given rates. We declare this function in the “timers.h” header file.

In the “timers.h” header file, add the following line

```
void initTimers();
```

The definition of “**initTimers**” function will be provided in a separate source file. Create a new source file “timer.c”, and copy the following code in this source file.

```
#include "timers.h"  
void initTimers()  
    *TIMER1_LOAD=1999999;  
    *TIMER1_CTRL=(1<<8)|(1<<6)|(1<<5)|(1<<4)|(1<<1);  
    *TIMER2_LOAD=0xFFFFFFFF-25000000;  
    *TIMER2_CTRL=(1<<8)|(1<<6)|(1<<5)|(1<<4);  
}
```

Note that the “**initTimers**” function, does not enable the timers, i.e. the counters are not yet running. By setting a particular value in the LR and setting the UDC bit in the CSR of each timer, we can specify the rate at which interrupts will be generated by the particular timer. Note that both timers operate at 100Mhz.

Study the code of the “**initTimers**” function.

How long is the interval between two consecutive interrupts that can be generated by TIMER1? _____

How long is the interval between two consecutive interrupts that can be generated by TIMER2? _____

What is the longest achievable interval between two consecutive interrupts that can be generated by these timer devices? _____

What is the shortest achievable interval between two consecutive interrupts that can be generated by these timer devices? _____

Suggest how to setup the timers such that the interval between two consecutive interrupts is set to the longest achievable interval? _____

Suggest how to setup the timers such that the interval between two consecutive interrupts is set to the shortest achievable interval? _____

Once the timers are properly initialized, in order to generate interrupts, the timers need to be enabled, i.e. the counters within the timers should start ticking. For that reason, next, we define the function “**enableTimers**”, and declare it in “timers.h”:

```
void enableTimers();
```

The definition of the function “**enableTimers**” should be provided in the source file “timers.c”. Copy the following code in the source file “timers.c”:

```
void enableTimers(){
    *TIMER1_CTRL=(*TIMER1_CTRL|(1<<7))&(~(1<<5));
    *TIMER2_CTRL=(*TIMER2_CTRL|(1<<7))&(~(1<<5));
}
```

Once the timers are enabled, they will start generating interrupts at the given rate. Each of the timers requires its own interrupt service routine. As explained earlier, the interrupt service routines follow some steps that differ from a regular subroutine (function). In particular, the two major differences are as follows. First, an interrupt service routine must store the state of all registers on the stack, while for a regular subroutine only the non-volatile registers should be stored on the stack. Second, an interrupt service routine ends with an **RTID** instruction, while a subroutine ends with an **RTSD** instruction.

To avoid writing the interrupt service routine in assembly, the MicroBlaze C compiler provides the attribute **fast_interrupt**. Using this attribute, a regular function is compiled as an interrupt handler, hence, all registers are stored on the stack and the **RTID** instruction is used instead of **RTSD**.

We declare the interrupt service routines for each of the timers in the “timers.h” header file. Add the following two lines in “timers.h”

```
void timer1InterruptHandler() __attribute__((fast_interrupt));
void timer2InterruptHandler() __attribute__((fast_interrupt));
```

The definition of these two functions will be discussed later.

The next step is to initialize the **INTERRUPT CONTROLLER** device. To do so, we need access to some of its registers. These registers are mapped to known memory addresses. We shall define pointers to access these registers.

In the current project, create a new header file “interrupt_controller.h”. Add the following lines in the “interrupt_controller.h” header file between the “**#define**” and “**#endif**” directives.

```
#define IER    (unsigned int*) 0x41200008
#define IAR    (unsigned int*) 0x4120000C
#define MER    (unsigned int*) 0x4120001C
#define IVAR1  (unsigned int*) 0x41200104
#define IVAR2  (unsigned int*) 0x41200108
```

To configure this device we define an initialization function, i.e. “**initController**”. This function should enable the interrupts coming from the timers, enable the interrupt request output from the **INTERRUPR CONTROLLER** device, and specify the interrupt vector address for the two interrupt handlers. The interrupt request of **TIMER1** is mapped to bit-position one of the **ISR** in the interrupt

controller, while the interrupt request of TIMER2 is mapped to bit-position two of the ISR in the interrupt controller. The IVAR1 pointer should point to the memory address where the interrupt handler for TIMER1 is stored, while the IVAR2 pointer should point to the memory address where the interrupt handler for TIMER2 is stored.

We declare this function in the “interrupt_controller.h” header file. In the header file add the following line:

```
void initController();
```

The definition of the “initController” function will be given in a separate source file. Create a new source file “interrupt_controller.c”, and add the following code:

```
#include "interrupt_controller.h"
#include "timers.h"
void initController(){
    *IER|=0b110;
    *IVAR1=(unsigned int) timer1InterruptHandler;
    *IVAR2=(unsigned int) timer2InterruptHandler;
    *MER|=0b11;
}
```

Next, we discuss the interrupt handlers for the two timers. The definitions of the handlers will be provided in the “timers.c” source file.

The interrupt handler for TIMER1 reads the state of the pushbuttons and stores it in a global variable that can be accessed by the main program. Furthermore, the handler needs to clear the interrupt flag in TIMER1, and acknowledge the interrupt by writing to the IAR of the INTERRUPT CONTROLLER.

The interrupt handler for TIMER2 keeps track of whether the *blinker* should be turned on or off. This is achieved by constantly inverting the bits in a given global variable, defined in the main program. Furthermore, the interrupt handler needs to clear the interrupt flag in TIMER2, and acknowledge the interrupt by writing to the IAR of the INTERRUPT CONTROLLER.

The definitions of the handlers are provided with the code below. Copy these definitions in the “timers.c” source file:

```
void timer1InterruptHandler(){
    currentButtonsState=*BUTTONS_DATA;
    *TIMER1_CTRL|=(1<<8);
    *IAR=0b10;
}
void timer2InterruptHandler() {
    blinkerOnOff=~blinkerOnOff;
    *TIMER2_CTRL|=(1<<8);
    *IAR=0b100;
}
```

Few justifications should be performed. First, since “**timer1InterruptHandler**” requires access to the pointer “**BUTTONS_DATA**” defined in “**buttons.h**”, in the source file where the handler is defined, i.e. “**timers.c**”, we need to include the header file where the pointer “**BUTTONS_DATA**” is defined. Add the following line as a first line in the “**timers.c**” source file:

```
#include "buttons.h"
```

Second, since both interrupt handlers require access to the pointer “**IAR**” defined in “**interrupt_controller.h**”, this header file needs to be included in the “**timer.c**” source file. Add the following line at the beginning of the “**timers.c**” source file:

```
#include "interrupt_controller.h"
```

Third, since both interrupt handlers require access to global variables that will be defined in the main program but not in the same source file where the handlers are defined, it is important to declare these global variables as external. To declare these variables as external, in the header file “**timers.h**” add the following lines:

```
extern unsigned int currentButtonsState;  
extern unsigned int blinkerOnOff;
```

The reason for doing the three justifications described previously is because each source file is compiled separately. Without the justifications, the compiler will complain that it cannot find definitions for some of the variables used in the functions when compiling the “**timer.c**” source file.

To enable the interrupts for the MicroBlaze processor we need access to the special purpose register **rmsr**. Furthermore, the access to this register is enabled by specific machine instructions. Therefore, we are forced to use assembly programming when defining the routine “**enableMicroBlazeInterrupts**” that enables the interrupts in the MicroBlaze processor.

We declare the function in a header file. Create a new header file “**microBlaze.h**” and add the following line between the “**#define**” and “**#endif**” directives:

```
extern void enableMicroBlazeInterrupts();
```

The definition of the “**enableMicroBlazeInterrupts**” routine will be provided in assembly language and will be stored in a separate source file. Create a new source file “**microBlaze.s**”, and add the following code:

```

.global enableMicroBlazeInterrupts
.text
.ent enableMicroBlazeInterrupts
enableMicroBlazeInterrupts:
    mfs        r11, rmsr
    ori        r11, r11, 2
    mts        rmsr, r11
    rtsd      r15, 8
    nop
.end enableMicroBlazeInterrupts

```

Finally, the code for the main program is provided in the file “[Blinker.c](#)” available on the course web-page. Copy the contents of the “[Blinker.c](#)” file into the “main.c” file in your project, and study the code.

Build the project and run the application. When running the application, you observe that sometimes the blinker leaves trails, i.e. the bottom-most horizontal segment of some seven-segment displays is turned-on.

Modify the code such that the problem of trailing is avoided. Write down the modification that you have suggested, build the project and run the application to verify that the applied modifications resolve the problem of trailing.

Can an interrupt request interrupt the processor while it is executing an interrupt handler of another interrupt request?


Ensure that the application is terminated. In the C/C++ environment, insert breakpoints to the following lines:

*LED_DATA = 1 << application_state; (in the “main.c” file)

currentButtonsState=*BUTTONS_DATA; (in the “timer.c” file)

blinkerOnOff=~blinkerOnOff; (in the “timer.c” file)

To insert a breakpoint at a given line, put the cursor on the given line and press CTRL+SHIFT+B on the keyboard.

Run the application. Observe that status of the **rmsr** register each time the program is suspended. Once you have checked the status of the **rmsr** register, resume the program, i.e. press the Resume  button. What is the conclusion? _____

Assignment 2.

The purpose of this assignment is to develop an alarm-clock application. The alarm-clock application has 4 different states: RUN, SET_TIME, SET_ALARM and ALARM. The time is presented on the seven-segment displays in the following order: the two left-most seven-segment displays represent the hours, the next two displays to the right represent the minutes, the next two represent the seconds, finally the last two displays represent the hundreds of a second. The alarm-clock operates as follows. Initially, the alarm-clock is in the RUN state and at every 0.01s displays the updated time. If the current time matches with the specified alarm time, and if the switch that enables the alarm is on, i.e. the left-most switch on the board, then an alarm is triggered, and the state changes to ALARM. Once the state has switched from RUN to ALARM, the application can remain in the ALARM state for at most 5 seconds, i.e. the alarm indication will be visible for at most 5 seconds once the alarm goes off. The alarm indication is done by toggling the 16 LEDs available on the board. During an alarm indication, the LEDs toggle from turned-on to turned-off four times in one second. The alarm-clock should include a snooze function. If a user presses the MIDDLE pushbutton, while the alarm-clock is in the ALARM state, the alarm time is updated to “current_time +10s” and the state changes to RUN. However, if the MIDDLE button is not pressed, after being in the ALARM state for 5 seconds, the application moves back to the RUN state. If a user presses the UP button, while the application is in the RUN state, then the state changes from RUN to SET_TIME. When the state has changed to SET_TIME state, a *blinker* indicates the current time unit (hour, minute, second, hundreds of a second) that can be modified by pressing the UP and DOWN button. As each of the time units are represented with two digits, the *blinker* should be visible on the two seven-segment displays representing the selected time unit. To switch between the time units, the LEFT and RIGHT buttons are used. The *blinker* should toggle with the same frequency as the *blinker* in Assignment1. Finally, when the correct time is set, pressing the MIDDLE button changes the state from SET_TIME to RUN. If a user presses the DOWN button, while the application is in the RUN state, then the state changes from RUN to SET_ALARM. When the state has changed to SET_ALARM state, all LEDs are turned-on, and a *blinker* indicates the current time unit (hour, minute, second, hundreds of a second) that can be modified by pressing the UP and DOWN button. To switch between the time units, the LEFT and RIGHT buttons are used. The *blinker* should toggle with the same frequency as the *blinker* in Assignment1. Finally, when the correct alarm-time is set, pressing the MIDDLE button changes the state from SET_ALARM to RUN. The state transition diagram is presented in **Figure 5**.

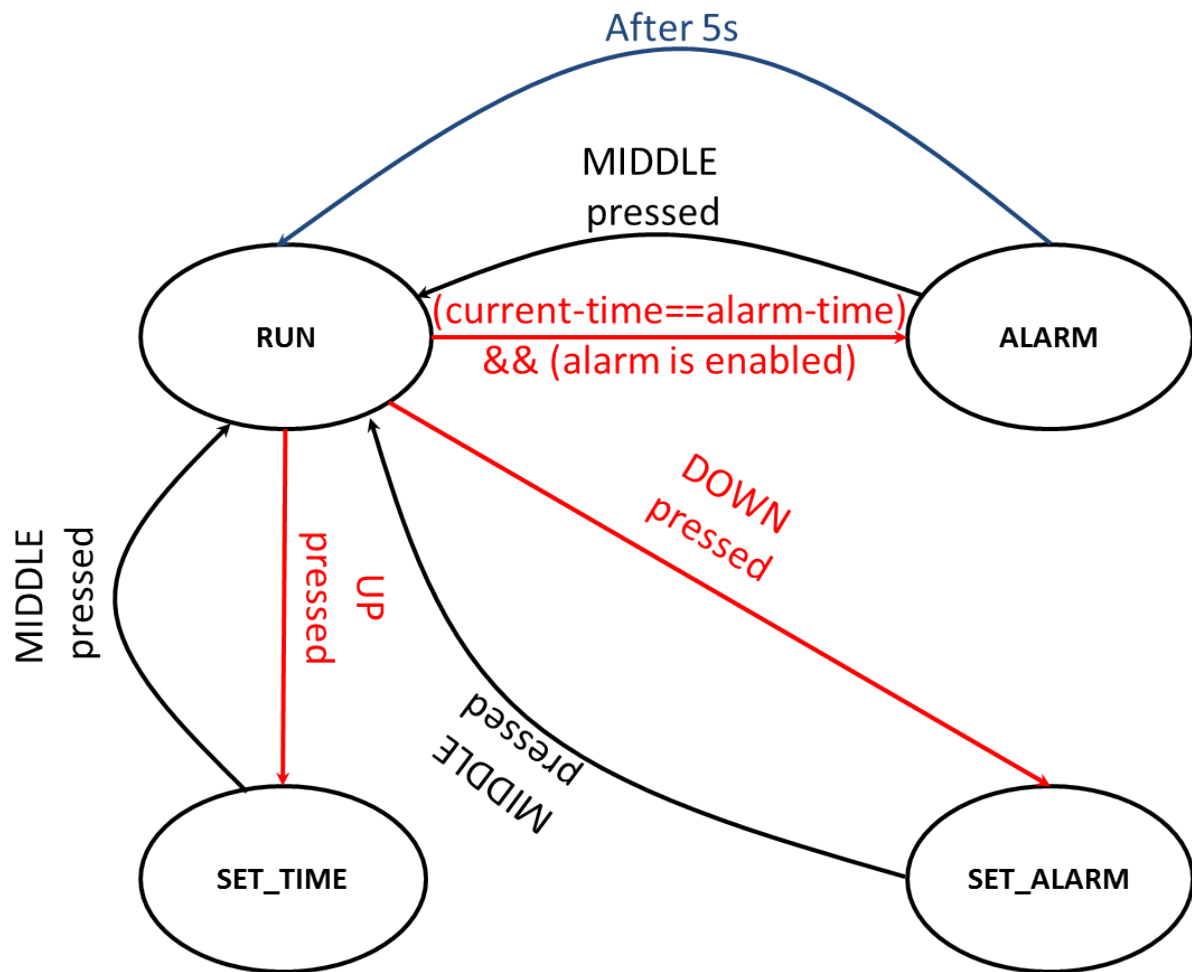


Figure 5. Alarm-clock State Transition Diagram

TIMER1 should generate interrupts every 10ms. The interrupts coming from this timer should update the time, but also indicate when to read the buttons such that bouncing is avoided. Buttons should be read every 20ms. Furthermore, when the alarm-clock is in either SET_TIME or SET_ALARM state, the interrupt events from this timer should be used to update the blinker every 250ms.

TIMER2 will not be used to generate periodic interrupts. Instead, this timer should be used such that it triggers an interrupt condition 5s after it has been enabled.

A skeleton for the application is available on the course web-page.

In your project, create a new header file with the name “alarm_clock.h”. Copy the contents of the file “[header.h](#)” and insert it into “alarm_clock.h”. Copy the code from the file “[alarm_clock.c](#)” and insert it into the “main.c” file in your project.