KTH Information and
Communication Technology

IL2206 EMBEDDED SYSTEMS

Home Laboratory : Input/Output and Interrupts in Nios II Systems

IMPORTANT: *Students are expected to complete the home laboratory at least 10 days before lab 2! However, start working on the laboratory directly after course start. The laboratory requires a lot of work!*

1 Objectives

This laboratory will help you to

- get used to the hardware and software of the laboratory environment
- deepen your understanding of how C- and Assembler functions interact
- deepen your understanding of I/O-handling with polling and interrupt
- access embedded peripherals by means of C-Macros and the HAL[1] System Library
- get used to the Altera documentation

2 Laboratory Tasks

This laboratory shall give the students the necessary skills, which are required to work with the DE2 FPGA board, the Nios II processors, and the related software. This laboratory will be conducted as a home laboratory and each student on his own is repsonsible to acquire the necessary knowledge. The laboratory will be conducted by groups of two students in private, i.e. outside scheduled lab sessions. It will ***not*** be verified by the course assistants but the knowledge acquired will be needed and tested as part of *Lab 2: Introduction to Real-Time Operating Systems*.

In case a student group encounters a technical problem, they are strongly encouraged to interact with other students through the interactive KTH course

[1] Hardware Abstraction Layer

home page. This page will also be monitored by the course assistants, who will take actions in case of larger technical problems.

The solutions for this lab will be published on the course web, so that students can validate their own solutions. Please use the provided solutions with care and only at a late stage, so that you really acquire new own knowledge.

2.1 Setting up the Lab Environment

In order to perform the lab, a virtual machine (VM) with all the needed software is provided. Install the virtual machine on your computer using the instructions on the course homepage, where you also can find basic information on how to use the VM (e.g. a collection of important commands).

You also need to download and extract the source and configuration files for this lab, which are available in a `.zip` file from the course homepage.

2.1.1 Testing Nios 2 IDE

Run the shell script on the virtual machine desktop and open Nios 2 IDE, inputting the command `nios2-ide` .

Start a new "Hello world" project:

- click on Workbench
- New / Project... / Nios II C/C++ Application / Next
- click on "Specify Location". Keep both the default name and location for now.
  Attention: neither name nor location should contain white spaces
- in the "SOPC Builder System PTF File" box load the `.ptf` file provided in the Lab 0 sources folder.
- click on "Finish". A `hello_world.c` file will be generated.
- compile this program and run it in the Instruction Set Simulator (ISS). On the left side of the screen, in the "Nios II C/C++ Projects" window, right-click on your project `hello_world_0` / Run As / Nios II Instruction Set Simulator.
- wait until the program compiles (it may take several minutes). The program should print out on the terminal (visible on the bottom window of the IDE) a few warnings and the message "Hello from Nios II!".

2.1.2 Testing the DE2 board

Exit the IDE, saving your progress. Connect the DE2 board to your computer through the "BLASTER" USB port and turn it on.

Attention: make sure that the switch left of the LCD display is set to RUN.

- in the Nios 2 shell type in `jtagconfig`. You should see the USB-Blaster driver recognized.

- type in `nios2-configure-sof <sof-file>` where `<sof-file>` is the `.sof` file downloaded from the course web page. The board should now be configured with one Nios 2 processor, and you should see the message "Quartus II Programmer was successful."
- run the IDE again with the command `nios2-ide`.
- your `hello_world_0` project should be already opened.
- run this program from the board this time: right-click on project `hello_world_0` / Run As / Nios II Hardware.
- After the program is compiled and downloaded on the board, you should see the message "Hello from Nios II!" on the terminal.

Congratulations! The lab environment is set up correctly on your machine. You should now proceed to the next lab tasks.

2.2 Literature

A lot of information that helps to solve the laboratory tasks can be found in the Altera documentation on Nios II [Altb, Altc, Alta], which is available on the course homepage. The amount of information in these documents is huge. Since one goal of this course is that students shall be able to cope with industrial documentation, students should take some time to scan through the documentation and to learn how to get a good overview, to navigate and to make use of it. Often the given references provide example code, for instance how to program the timer to generate "alarm" and how to program the input keys to generate interrupts. The main focus of the reading should be on the following:

- Introduction to the Altera Nios II Soft Processor [Altd]
  Read this thoroughly, it is a very good introduction.
- Nios II "Processor Architecture" and "Programming Model": Chapters 2 and 3 of [Altb]
- Nios II "Application Binary Interface" and "Instruction Set Reference": Chapters 7 and 8 of [Altb].
  Especially important are parameter passing and size of datatypes.
  The reference (Chapter 8) provides detailed information about all instructions.
- Hardware Abstraction Layer and HAL system library: Chapters 2, 5, 6, 8, and 14 of [Altc].
  Here you can find out which macros to use for reading and writing registers in I/O-circuits: PIO and Timer. You can also find information on how to use HAL-functions to initiate and use interrupts.
- Nios II "PIO Core" and "Interval Timer Core": Chapters 10 and 28 of [Alta] Especially I/O-register organization and use in PIO and Timer.

Details of the Cyclone II FPGA board that is used in this laboratory is described in the DE2 Development and Education Board User Manual ([DE2]).

You should also have a look on the project templates that are available when you create a new application in the Nios II IDE.

Valuable information can also be found in the lecture notes that can be found on the course homepage.

2.2 completed ◯

2.3 C and Assembler

In this lab you will develop a program that will display a time value with different output devices and read from input devices for manipulation. You will start with simply printing out the value into a terminal window.

Create a new project called `lab0` in the Nios-II IDE. The configuration file (.ptf) is available in a zipped archive from the course homepage. Add also the source files from the archive (.c and .s) to your new project.

You will find the following source files:

puttime.c contains the C-function `void puttime (int* timeloc)` that reads the time value stored at the memory address given by the pointer parameter `timeloc` and prints it to a terminal window. The printout is in the format `mm:ss` (minutes:seconds).

tick.c contains the C-function `void tick (int* timeloc)` that increments the time value stored at memory address given by the pointer parameter `timeloc` by one.

delay_asm.s contains a Nios-II assembly subroutine which delays program execution by the number of milliseconds given by a parameter.

hexasc_asm.s contains a stub for a Nios-II assembler subroutine which you need to implement, as described below. This subroutine is called from the C-code within the function `puttime`.

lab0.c contains the main loop of the program.

Take a look at the file delay_asm.s. The subroutine `delay` has one input parameter in r4 that determines how many milliseconds the subroutine will wait before returning to its caller. Draw a flow diagram that visualizes how the subroutine `delay` works.

In order to achieve a 1 millisecond delay in the inner loop, you need to adjust the parameter `delaycount`, which is defined at the top of the file and currently set to 0. To calculate the delay-time, use the fact that the processor runs at 50 MHz, and assume that it usually can execute one instruction per cycle. Also assume that the simulator has a slowdown factor of 300 (to simulate one second of Nios II runtime will take 5 minutes of real time). This will give a reasonable starting-value for the delay routine parameter. Also calculate the parameter for running it on the board, as you will need it during the lab session.

NOTE: *For the following part of this section (Section 2.3), it is not mandatory that the students are able to write the assembler code on their own. Instead*

they can view the solution and gain an understanding how C and Assembler interact using the Nios II Application Binary Interface [Altc].

Draw a flow-chart for a subroutine to convert a 4-bit hexadecimal value to the corresponding 7-bit ASCII-code. See the full specification for `hexasc` below.

***Examples*:**

binary 0010 (hexadecimal digit 2) is converted to 011 0010 (ASCII-code for '2')

binary 1011 (hexadecimal digit B) is converted to 100 0010 (ASCII-code for 'B')

Implement the subroutine starting from the skeleton provided within the file `hexasc_asm.s`.

Specification

Name: The subroutine must be called `hexasc`.

Input parameters: Only one, in register r4. The 4 least significant bits in register r4 specify a number, from 0 through 15. The values of all other bits in the input must be ignored.

Return value: Only one, returned in register r2. The 7 least significant bits in register r2 must be an ASCII code as described below. All other bits in the output must be zero.

Required action: Input values 0 through 9 must be converted to the ASCII codes for the digits '0' through '9', respectively. Input values 10 through 15 must be converted to the ASCII codes for the letters 'A' through 'F', respectively.

Side effects: The values in registers r2 through r15 may be changed. All other registers must have unchanged values when the subroutine returns.

You MUST follow the specification, which is based on the application binary interface of the Nios II [Altb].

Helpful hints

Use registers r8 through r15 for any temporary values within your subroutine.

You can find the ASCII chart on many websites, e.g., on Wikipedia.

Edit the main program such that the time value is incremented and printed out once every second. Use the Nios II Instruction-Set Simulator to run your program. Using the simulator, tune your `delay` subroutine to an accuracy better than 10% (no more than 6 seconds of error in one minute of simulation). Verify that the time is printed once per second. 2.3 completed ◯

2.4 Parallel I/O

2.4.1 PIO macros

In order to access the hardware, such as I/O-ports on the DE2-board, by software, Altera provides hardware abstraction layer (HAL) library functions (macros) that "hide" the memory-mapped low-level interface to the device. Information about these macros can be found in *Software Developers Handbook* ([Altc]), Chapter 7: "Developing Device Drivers for the Hardware Abstraction Layer"

In order to write to a parallel I/O-port use the macro `IOWR_ALTERA_AVALON_PIO_DATA`.

Example:
The command `IOWR_ALTERA_AVALON_PIO_DATA (DE2_PIO_REDLED18_BASE, 0x3ffff)` writes 1s to all 18 red leds and thus turns on all `LEDR`s.

In order to read from a parallel I/O-port use the macro `IORD_ALTERA_AVALON_PIO_DATA`.

Example:
The command `IORD_ALTERA_AVALON_PIO_DATA(DE2_PIO_KEYS4_BASE)` returns state of the buttons as an integer bit pattern, a 0 means that the corresponding button is pressed.

Symbolic names of I/O-ports and I/O-registers can be found in the `system.h` file. This file is generated when the system-library is built and compiled. You can find the `system.h` file in your projects.

It is situated in the system library in folder *Debug* → *system_description* → *system.h*.

You have to include some files in your program if you are using macros and/or symbolic names of IO-ports, for instance

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
```

Symbolic names used in our CPU-configuration are

- `DE2_PIO_REDLED18_BASE` for the eighteen `LEDR`s. `LEDR17` is MSB[2] (bit 17) and `LEDR0` LSB[3] (bit 0).
- `DE2_PIO_GREENLED9_BASE` for the nine `LEDG`s. `LEDG8` is MSB (bit 8) and `LEDG0` LSB (bit 0).
- `DE2_PIO_KEYS4_BASE` for the four buttons. `KEY3` is MSB (bit 3) and `KEY0` LSB (bit 0).
- `DE2_PIO_HEX_LOW28_BASE` for the seven segment displays.

The seven-segment display has four digits. The leftmost digit is controlled by the seven most significant bits (`HEX3` = bits 27 through 21); the left-middle digit is controlled by the next seven bits (`HEX2` = bits 20 through 14); then follows the right-middle digit (`HEX1` = bits 13 through 7), and the rightmost digit by the least significant bits (`HEX0` = bits 6 through 0).

[2] most significant bit
[3] least significant bit

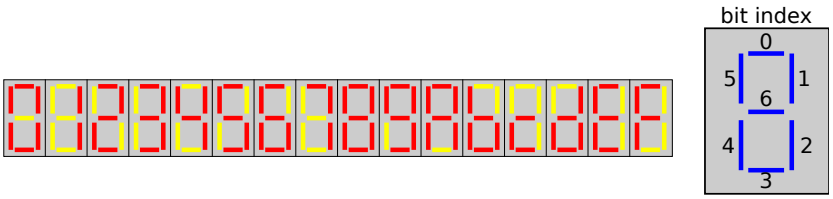
Update the main loop in your program and include a macro that will cause the time value to be written in binary form on the 16 red LEDs `LEDR15` through `LEDR0`. Use the natural BCD[4] encoding with four groups of four bits each. 2.4.1 completed ◯

2.4.2 The bcd2sevenSeg function

Write a C function bcd2sevenSeg with the following prototype:

```
int bcd2seven(int inval)
```

The purpose of the function is to convert 4-bit binary code to 7-bit "seven-segment-code". The four least significant bits of the input value are to be converted into a 7-bit value, such that this value can be sent directly to a seven-segment display to produce the appropriate digit. Recommended digit patterns are shown in the figure below. To light up a segment of a seven-segment display, the corresponding bit should be zero (0). A one (1) will shut the segment off.



2.4.2 completed ◯

2.4.3 The puthex function

Write a C function puthex with the following prototype:

```
void puthex (int inval)
```

The purpose of the function is to display the current time on the seven-segment digits `HEX3` through `HEX0`. The puthex function should call `bcd2sevenSeg`. Add a call to the `puthex` function in your C-coded main loop.

The call to puthex function will be tested on the lab occasion, not in the simulator. 2.4.3 completed ◯

2.5 Polling

Recall which macro to use when you want to read values from the buttons `KEY3` to `KEY0`.

Create a new project, called `lab0_IO`, and copy all files from the previous project to this new project.

Write a C function `pollkey` with the following prototype:

```
void pollkey()
```

The function shall poll the keys and affect the behaviour of the time presentation. Each time you push a button the behavior must change.

As a simplification you may assume that a button is always released before another (or the same) button is pressed and that two or more buttons are never pressed at the same time. (It is out of the scope of this lab to take care of all possibilities but you are of course allowed to do it).

[4] BCD stands for *binary coded decimal*: four bits are used for one digit, e.g., 87 is represented as 1000 0111

KEY0 Each time you push the button `KEY0` the time shall *start* counting.

KEY1 Each time you push the button `KEY1` the time shall *stop* counting.

KEY2 Each time you push the button `KEY2` the time value presented shall be incremented by one second.

KEY3 Each time you push the button `KEY3` the time value shall be reset to 00:00.

Draw a flow chart to describe the behavior of the `pollkey` function.
Add a call to a `pollkey` function in your C-coded main loop.

Helpful hints

Write the C-code of the function `pollkey` in the same file as the `timeloc` variable and the main program. This makes it possible for the `pollkey` function to reference the `timeloc` variable.
You can use a variable `RUN` to indicate if the counting is ON or OFF and use this variable to decide if `tick` is going to be called or not.

2.5 completed ◯

2.6 Response Time

If you run the program above you will find that the response time when you push a button might be disturbingly long. Probably also some button pushes are lost because the `pollkey` function is called only once per second.

Modify your program so that the `pollkey` function is called once per millisecond. Make sure that the time variable (`timeloc`) still will be incremented only once per second.

2.6 completed ◯

2.7 Interrupt using HAL-functions

Refresh your memory concerning how to use interrupts in software and hardware, especially in the Nios II CPU. Find out which HAL-functions to use to initiate and register interrupts in Nios II.

Information on HAL-functions can be found in:
Software Developers Handbook ([Altc]), chapter 8: “Exception Handling”.

In the previous program most of the CPU-time is spent polling the keys to detect changes. Most of the (CPU-) time no changes occur and valuable CPU-time is wasted. It might be smarter to let the CPU work with something useful instead of waiting for buttons to change.

The buttons have hardware support for interrupts. Information on this can be found in the *Embedded Peripherals IP User Guide* ([Alta]), chapter 9: “PIO Core”.

Create a new project called `lab0_int` and copy all files from the previous project to the new project.

Modify the program in your new project to use interrupts from the keys instead of polling. This means that you must write an interrupt service routine for the button interrupt (name the function `key_InterruptHandler`), and that you must initialize the system so that interrupts from the button PIO will cause the `key_InterruptHandler` function to be executed. You must also

enable interrupts from the button PIO by writing appropriate value(s) to the appropriate register(s) in the button PIO using the appropriate macro(s).

The behavior caused by the buttons shall be changed compared to the previous exercise:

KEY0 The button `KEY0` shall toggle between *start* and *stop* counting the time.

KEY1 Each time you push the button `KEY1` the time value shall be incremented by one.

KEY2 Each time you push the button `KEY2` the time value shall be set to 00:00.

KEY3 Each time you push the button `KEY3` the time value shall be set to 59:57.

2.7 completed ◯

2.8 Using a Hardware Timer

Most processors offer possibilities to use hardware timer circuits. As the timer circuits are clocked by a crystal oscillator the accuracy is in the order of 1 per million or even better. In the Nios-II processor several timers are available. Timers play a very important role in real-time systems, where functions have to be computed within a given time frame. The timer also off-loads the processor by running in parallel to the processor and keeping track of the time.

Update your program to use a timer instead of using a programmed delay. Information on timers with the Nios-II can be found in the *Software Developers Handbook* ([Altc]), Section "Using Timer Devices" of Chapter 6: "Developing Programs Using the Hardware Abstraction Layer".

Create a new project called `lab0_timer` and copy all files from the previous project to the new project.

Modify the new project to use HAL-functions to order "alarm" once per second to print the current time value on terminal and hex-display. Keep the functionality of the buttons from the previous exercise.

You must write the alarm-handler function and use the HAL-function to initiate the system to use it.

2.8 completed ◯

2.9 Introducing Valuable Foreground Work

In the end of this lab manual, you will find the code for the function `nextPrime`. Its prototype is:

```
int nextPrime(int inval)
```

This function returns the smallest prime number greater than the parameter `inval`. Calling `nextPrime` with a value of 17 will for example return the value 19, since this is the next prime. Calling it with the value 26 will return the value 29, and so on.

Update your program from the previous task so that prime numbers are calculated by the main program loop and printed on the console window at the "same time" as time is shown on the HEX-displays (and LEDRs). Make sure that the behavior can still be manipulated by the buttons, as specified above.

The function `nextPrime` will work on the DE2-board at the lab session. To be able to run the nextPrime function in the NiosII Instruction Set Simulator you need to:

1. right-click on the project folder,
2. choose *System Library Properties*,
3. check the box *Unimplemented instruction handler*,
4. click *Apply*

If you forget to do this, the simulator may give an error message like: "*Break instruction called without debugger attached*".

2.9 completed ◯

2.10 Questions

Study the configuration used for the DE2-board, which you find in the `system.h` file. Answer the following questions.

1. Which Nios-processor is used in this configuration? Give the number of stages in the processor pipeline.
2. The configuration contains a number of different memories. Order these memories after typical access times and give the size of each memory.
3. Which I/O devices and timer devices are used in the preperation tasks of this lab? Give their address spaces and IRQ-levels.

2.10 completed ◯

References

[Alta] Altera. *Embedded Peripherals IP User Guide*. Version UG-01085-10.0.0.

[Altb] Altera. *Nios II Processor Reference Handbook*. Version NII5V1-10.0.

[Altc] Altera. *Nios II Software Developers Handbook*. Version NII5V2-10.0.

[Altd] *Introduction to the Altera Nios II Soft Processor*.

[DE2] Altera. *DE2 Development and Education Board User Manual*. Version 1.4.

A nextPrime C-code

In the main loop, call nextPrime as follows:

```
 next = nextPrime (next);    /* Produce a new prime. */
 printf("\nNext Prime is %d",next);
```

The `nextPrime` function itself:

```
/*
  * NextPrime
  *
  * Return the first prime number larger than the integer
  * given as a parameter. The integer must be positive.
  */
#define PRIME_FALSE      0      /* Constant to help readability. */
#define PRIME_TRUE       1      /* Constant to help readability. */
int nextPrime( int inval )
{
     int perhapsprime;          /* Holds a tentative prime while we check it.
*/
     int testfactor;            /* Holds various factors for which we test
                                   perhapsprime. */
     int found;                 /* Flag, false until we find a prime. */
     if (inval < 3 )            /* Initial sanity check of parameter. */
     {
       if(inval <= 0) return(1); /* Return 1 for zero or negative input. */
       if(inval == 1) return(2); /* Easy special case. */
       if(inval == 2) return(3); /* Easy special case. */
     }
     else
     {
       /* Testing an even number for primeness is pointless, since
         * all even numbers are divisible by 2. Therefore, we make sure
         * that perhapsprime is larger than the parameter, and odd. */
       perhapsprime = ( inval + 1 ) | 1 ;
     }
     /* While prime not found, loop. */
     for( found = PRIME_FALSE; found != PRIME_TRUE; perhapsprime += 2 )
     {
       /* Check factors from 3 up to perhapsprime/2. */
       for( testfactor = 3;
             testfactor <= (perhapsprime >> 1);
             testfactor += 1 )
       {
          found = PRIME_TRUE;       /* Assume we will find a prime. */
          if( (perhapsprime % testfactor) == 0 )
          {                    /* If testfactor divides perhapsprime... */
            found = PRIME_FALSE;    /* ...then, perhapsprime was non-prime. */
            goto check_next_prime; /* Break the inner loop,
                                      go test a new perhapsprime. */
```

```
            }
          }
          check_next_prime:; /* This label is used to break the inner loop. */
          if( found == PRIME_TRUE ) /* If the loop ended normally,
                                       we found a prime. */
          {
             return( perhapsprime ); /* Return the prime we found. */
          }
        }
        return( perhapsprime );        /* When the loop ends,
                                          perhapsprime is a real prime. */
}
```