Exercise 3: AVR32 UC3

We will take a break from programming in Linux for now, and instead use the AVR32 UC3-A3 xplained card. The AVR32 UC3 is a simple 32-bit microcontroller, that is similar, but not equal to the AVR 8-bit microcontroller some of you probably have experience with. Just as the 8-bit AVR, it can run C-code directly on the microcontroller, without an operating system. The code for AVR32 is, however, different from the code used for 8-bit AVR.

It might sound strange to use a device without an operating system, in a course about real-time operating systems. There are concepts that are more easily demonstrated on the AVR32. This exercise also introduces the *Butterfly Real-Time Tester* (BRTT), which we will use in several of the next exercises. The BRTT is an application running on an AVR Butterfly that test the response time of a real-time system. Documentation of the BRTT can be found on it's learning. Do not attempt to reprogram the Butterfly, as it might give incorrect results.

1. Developing with AVR32

1.1. Running the test application

The exercise assumes that you use the **AVR studio 5** on Windows. It should be possible to do the same in Ubuntu, but it has not been tested, and it is not guaranteed that the appropriate tools are installed as working properly. You can download a template project on it's learning (AVR32 Template.zip). This should be unzipped at a location of your choosing and the solution file opened in AVR Studio 5.

The AVR32 UC3 should be connected to the PC and JTAG as described in the Getting Started guide you can find on it's learning. Additional information about the card can be found in the Hardware User's Guide. When the card is connected, you can click the green play button in AVR Studio to build the test application, program the card and run the application. If there is a message that the JTAG tool must be upgraded, just follow the instructions on the screen.

When the program start running on the card, a LED will start blinking every second. Instead of clicking the green play button to run the program, you can also click the blue play and pause button, to start debugging. You will then be able to step through the code one line at the time, add breakpoints etc.

1.2. Serial Debugging

Some of you might have noticed that there is a `printf()` function in the test application. The AVR32 UC3 card doesn't have a screen, but `printf()` is written to write to a USB-to-serial device, and the text can be shown in a serial console. It is suggested that you use the Termite serial console

Unfortunately the AVR32 UC3 doesn't behave exactly as a normal serial port. Start the Termite program while the AVR32 is running. Click settings, select the last port (on my computer it is COM 3, but can vary), baud rate of 115200, 8 data bits, 1 stop bits, none parity and flow control. Click ok and then on the

button on the top left of the main termite window to connect. It should now print “tick” every second, if not, you can disconnect and connect the usb cable to the AVR32 (restart it).

The problem is that if you now reprogram the AVR32 from AVR studio, the print will stop working. The trick is to reprogram or restart the AVR32 while the Termite program is “waiting for port”. Printing will then work normally when the AVR32 starts running. To make the Termite waiting for port, disconnect and try to connect to the correct serial port while the AVR32 is not powered or while it is being programmed. Apart from the hassle of getting the printing to start properly, it works as expected. Remember that `printf()` should be avoided in real-time code, as it is slow and unpredictable.

1.3. Description of the project

The `main.c` file contain the `main()` function which runs when the application is stated on the AVR32. This is the only file that you should edit in this exercise. The other files are similar to library files that have functions you can use instead of doing low level register programming, as you would have done if programming 8-bit AVR.

The `main.c` file starts with including `asf.h`, which again includes a large number of other header files, and `busy_delay.h`, which implements micro and milli seconds busy wait functions. After the includes, there are several defines, the first one is necessary for USB to serial, while the rest defines the I/O pins of the J3 header that you will use in this exercise.

The `init()` function initialize the LEDs and button on the card, the interrupts, USB to serial and busy wait delays. This function must be run at the start of the `main()` function, and should not be edited.

The `interrupt_J3()` function is a interrupt handler that can be configured to run based on the pins on the J3 header. This will be explained when interrupts are introduced.

The `main()` function start running the `init()` function, which should always be there. After the “start code from here” point, you can change the code as you like.

Of the other files in the project, you only need to be aware of a few of them:

- The files in `asf/avr32/boards/uc3_a3_xplained` folder initialize the card.
- The files in `asf/avr32/drivers/gpio defines` functions for using the general IO pins of the card, including the J3 header which we will use to connect to the Butterfly.
- `busy_delay.c` and `busy_delay.h` is provided as easy to use busy wait delay functions. These are modifications of the original files for delaying found in `asf/avr32/services/delay`.

1.4. Test the AVR32 UC3 Card

You should now try to change the `main.c` file and be comfortable with programming the LEDs on the card and using `printf()` to print to Termite. You can find out what how to configure and use the gpio pins by studying the `gpio.h` header and by looking at how the LEDs and push button is configured in `ini.c` in the boards folder. When you feel like you have the hang of it, continue to the next part.

2. Testing the Reaction Time of AVR32 UC3

2.1. BRTT Reaction Test

The BRTT reaction test measures the time a system uses to react to an external "real world" event. This is an important property of real-time systems. The BRTT can evaluate any system that has simple digital IO pins. In this course we will test different systems and techniques for responding to one or several signals as fast as possible.

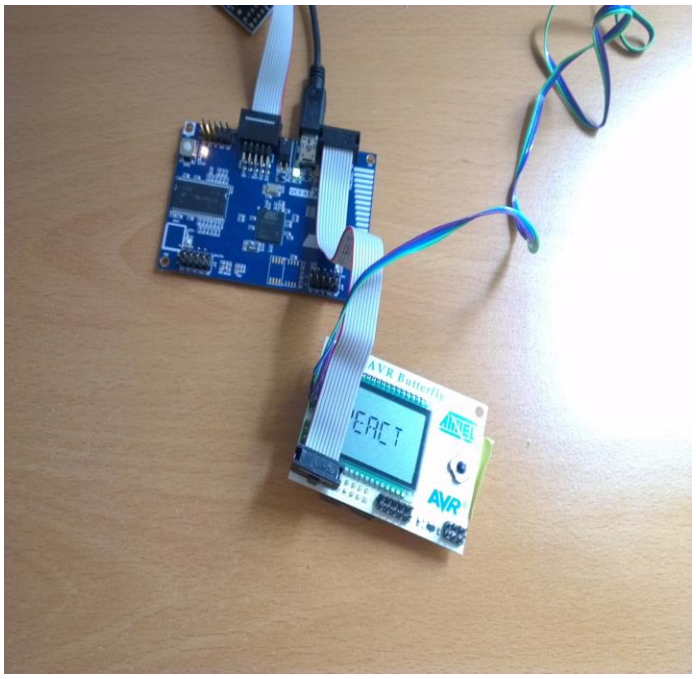
By doing the same test a large number of times, we will get a distribution of response time. For real-time systems, the worst-case response time will be more important than the average. If we consider a real-time system that controls the brakes of a car, an average response time of 1 ms is of no interest if the worst-case response time is so high that is cause accidents.

We will now use the BRTT to test the reaction time of the AVR32 UC3 device. The test consists of the BRTT sending signals that the AVR32 UC3 should notice and respond to as fast as possible. Details on how to use the BRTT can be found on it's learning, in the "Butterfly Real-Time Tester" folder.

2.2. Connecting Butterfly

In the picture below, you can see how the Butterfly is connected to the AVR32 card. This will both power the Butterfly and connect the necessary I/O pins. The Butterfly should also be connected to the PC with the provided cable and a normal serial cable. The BRTT prints the results of the test to the serial line, and you should use Termite to get hold of this data. The serial port is COM1 and the baud rate is 9600.

An Excel/Libreoffice Calc sheet for analyzing the results can be found on it's learning. The results are formatted so they can be pasted directly into the excel/calc sheet. The excel/calc sheet will show the distribution of response times in a histogram. Since the different systems we will test will have different response times, you can choose the time scale. If you choose a scale that is too low, the results that are higher than the scale will be shown in the rightmost column. To see the whole distribution, the rightmost column should ideally be zero.



2.3. Busy Wait

There are several methods we can use to respond to the reaction test. The most simple and the one we will test first is called busy wait. It consists of continuously read an input pin until a signal is discovered. The reason it is called busy wait, is that the program is busy while it is waiting for a signal, preventing the processor from doing anything else that might be useful. Because of this, busy wait is usually avoided.

We will start with a test of type A, meaning that only one signal is sent from the BRTT. The BRTT signals the AVR32 by setting the TEST A pin low. The AVR32 should then as fast as possible set the RESPONSE A pin low. The pins for BRTT is configured and used in the same way as the pins for controlling the LEDs on the card, just using the macros defined in main. The LEDs are set up in board_init(). Try marking this function and pressing alt+g. This should show how this function is implemented. Taking you to */src/asf/avr32/boards/uc3_a3_xplained/init.c*

Both test and response signals are normally high (1) and a low value (0) indicates an active signal.

Assignment A:

Create a program for the AVR32 that is using the busy-wait technique for responding to a test of type A. The following pseudo-code can be used:

```
responseA = 1


loop forever
     # wait for test A signal
     wait until testA = 0


     # send response A signal
     responseA = 0
     delay 5 us // hold the signal for a short while
     responseA = 1
end loop
```

When running a test with 10 subtests, you should get a result similar to this:

```
Reaction test
type  A
number     10


max value  65536
unit  us


test  A
0     7
1     5
2     6
3     7
4     5
```

```
5     6

6     7

7     7

8     5

9     7
```

You can also get a result informing you that there have been an overflow in the test, which will abort the whole test. This means that the counter that measures time in the Butterfly times out before a response signal is received. It is most likely caused by incorrect code or connections. An overflow result will look like this:

```
Reaction test

type  A

number     10


max value  65536

unit  us


test  A

0     overflow
```

2.4. Using Excel/Libreoffice Calc to Analyze the Results

An excel/calc sheet is provided on it's learning to analyze the results from the reaction test. The pasted result should look like this:

	D	E	F	G	H	I
1						
2		Test name:				
3		**Busy-wait**				
4						
5		Reaction test				
6		type	A+B+C			
7		number	1000			
8						
9		max value	65536			
10		unit	us			
11						
12		test	A	B	C	
13		0	949	17	26	
14		1	778	833	843	
15		2	206	272	262	

Be careful to get row and columns correct, or it will not work. The “Paste here…” text should be replaced with “Reaction test”. In the graphs sheet you can see the time distributions of up to six different tests. You can also adjust the time scale, so the data fits well inside the graphs. There is also a compare sheet that is one large diagram comparing up to six tests.

Assignment B:

Extend your application, so it will check all three TEST signals (A, B and C) and set the corresponding RESPONSE signal if a signal is detected. Do a reaction test with at least 1000 subtests of type A+B+C, and store the results. You should paste the results into the excel/calc sheet provided on it’s learning.

The Butterfly will make a noise when running a A+B+C test, because the pins used for the C test shares pins with the speaker on the back of the Butterfly. This does not affect the test.

2.5. Interrupts

As mentioned busy-wait is often avoided since it wastes resources and prevents other code to be executed. To use *interrupts* is often a preferred solution, especially for AVR32 UC3 where interrupts are very easy to set up. If a pin is configured with an interrupt it is monitored without disturbing the rest of the program on the AVR32. When a signal is detected the main process is paused and an interrupt handler function is called.

Half of the job of setting up an interrupt has already been done in the `init()` function and since the `interrupt_J3()` function has been prepared. In addition to this, you must register each of the pins you use as an interrupt, using the `gpio_enable_pin_interrupt()` function.

If you register more than one pin as interrupt, they will all run the same interrupt handler function. You must therefore check which of the pins have an active interrupt flag in the interrupt handler function, with the function `gpio_get_pin_interrupt_flag()`. When you have identified which gpio that has the active flag, it must be cleared using `gpio_clear_pin_interrupt_flag()`.

Remember to have an infinite loop at the end of the main function, otherwise the application will complete and will not see the interrupts. A good idea is to have a while loop that toggles a LED every once in a while, so you can see that the application is still running.

Assignment C:

Create a program that does the same as in assignment B using interrupts instead of busy-wait. Do the same test and analysis as in assignment B.

2.6. Big While

Since interrupt handlers runs at a higher priority, they can block lower priority interrupts from running. It is therefore considered good practice to run for as short a period as possible within the interrupt handler. When responding to a reaction test, this is not really a concern, since we only do a few things in the interrupt handler.

We will however pretend that we have a time consuming calculation to perform when an interrupt happens, then you can use a big while method to make sure that as little time as possible is spent in the interrupt handler. When an interrupt occurs, the interrupt handler should only set a *flag* corresponding to the pin that got the signal. In the main function, we have a while loop that checks for the different flags and run code for responding to the interrupt. Remember to clear the flag after use.

When a normal function and an interrupt should use the same global variable, it should be declared as volatile or it might not work properly.

Assignment D:

Create a program that does the same as in assignment B using interrupts and a big while. Do the same test and analysis as in assignment B.

The results will probably be worse than the two previous methods, as the amount of overhead needed for each signal is increased. The principle is however important and often used.

What will happen if you use a big while method, and the processor is in the middle of a time consuming calculation due to one interrupt, and a more important interrupt happens?

3. Approving the exercise

For this exercise you need to show the following to a student assistant to get the exercise approved.

1. Run a test of type A+B+C against your program from Assignment D. It is not necessary to show the earlier versions of the program, since it is assumed that they were working if the final test works.
2. Show the results from assignments B, C and D in the excel/calc sheet. It is not necessary to show how the test was performed, only the results.
    - Explain the different results.
    - The results from C (interrupts) probably showed different results for the three tests A, B, and C. What is the reason for this?