Realtidsprogrammering
Laborationsanvisningar

Mathias Broxvall, April 2014

General Instructions

As part of the laborations in the course realtime programming you will develop programs for PC/104 based target systems. You will develop programs for them both using non-realtime operating system (an ordinary Linux distribution), as well as using a realtime operating system (VxWorks).

For the development under Linux you will log on to the the target systems (using ssh) and perform the development onboard. For the development on VxWorks you will use the development environment *Windriver Workbench* with which you will crosscompile your applications from Windows and remotely upload, run and debug the programs on the target system.

Program code

To solve the exercises you will need to use or build upon some preexisting code. This code and all the other files you need for the laborations are available on the web page:
`http://aass.oru.se/~mbl/realtime`.

For the laborations where you develop directly onboard the target system you can either edit the files under windows and upload them for compilation, or edit them directly on the target system.

Do not leave any of your files left on the target system when you leave the labroom and expect that all the files can be deleted after the laborations. *It is your own responsibility to make sure that you always retain an up to date copy of your files on eg. your windows account*.

Preparations

Before each lab you are expected to have done these preparations. *Failure to do so may result in you not getting assistance during the lab!*

1. To have read through the exercises and any other relevant material carefully. Also download and read through any of the files relevant for the lab. If there is anything in the problem formulation that is unclear you should be ready to ask the lab assistant about it in the begining of the lab.

2. Thought about how the exercise can be solved. This can be done by for instance sketching the structure of your program, listing the functions you need to implement and reading up on which VxWorks functions you can use.

3. Thought about what can go wrong in your program and how you can detect such bugs. Also think about how you can test your program, is it sufficient to just test run it once? What effect does non-determinism have on your program?

Examination

The examination of each individual lab consists of **two obligatory parts**:

- A written lab report (PDF) that details the program and your solution. Describe the general method for the solution and give a paragraph detailing what each of your function does. For each stated *question* in the lab give a 1-2 paragraph answer/discussion in the report. The report should also contain the code for each laboration as a separate zip file in electronic form.

  You only need to send in the report and the code in electronic form to the assistant using blackboard. Before you send in the report, make sure that the code is commented well and correctly indented/formatted. Code that does not satisfy the demands are returned uncorrected and counts as one failed examination chance.

- A demonstration and oral examination where the lab assistant can ask you questions about your solutions. This is typically done during the scheduled lab session or upon agreement with the lab assistant.

In general you may work individually or in pairs of two. However, note that the examination is individual and each student must *have worked on* and be able to *answer questions* on every part of the code. Thus splitting the exercises between you in the lab group is not allowed. Furthermore, you are not allowed to copy code, reports, design documents or otherwise cooperate with other laboration groups except for asking basic questions regarding the functioning of the hardware or built in VxWorks functions.

To pass the laboration part of the course you are required to hand in and pass each laboration in time. This requires:

1. Each exercise that is required to be handed in should be so *at lastest* the day printed at the exercise. Laborations are typically corrected within one week and returned at the next scheduled laboration. The result of the correction can be either *pass* (godkänd), *revisions* (komplettering) or *fail* (underkänd).

2. In order for a laboration to be corrected it must be a reasonable attempt to solve the exercise, be handed in time and have *well commented code*. Otherwise it will automatically be graded *revisions*.

3. If you receive grade *revisions* you have one more chance to turn in a revised version of your lab *within two weeks*. The correction of a revised lab is typically done within one week and will give either pass or fail. **Observe that you cannot revise a lab more than once. If you miss the first chance to hand in the lab you have no more chances for revisions**.

Note also that you are expected to work outside of the assisted and scheduled hours in order to finish the labs in time and that you are expected to be able to debug your code independently.

Examiner: Dr. Mathias Broxvall, mathias.broxvall@oru.se

Lab 1

In this lab you will first write a simple program for Linux (non realtime version) that controls some simple hardware and investigate the potential problems that can be encountered when using a non-realtime operating system.

Next, you will continue by getting familiar with the development environment Windriver Workbench, with it's simulator and debugger. You will test to use these to write a *realtime* version of the programs and to see what effect the *priorities* of processes have for these programs.

In order to accomplish this you will need to solve some simpler C programming exercises and to use the debugger. Note that it is *highly recommended* for you to spend some extra time on getting familiar with the debugger since this will save you considerable amounts of time in the later exercises.

In order to not require all to use the same target systems (Linux or VxWorks) at the same time the Linux and VxWorks exercises can be done independently. Some of you can start with exercises A - B (Linux) while some of you start with C - E (simulated) followed by F (VxWorks).

Preparations

In addition to the general preparations described in the introduction to the lab course you should also make sure to bring some form of general C programming book (eg. *Vägen till C*, Bilting and Skansholm). You should also read the text "Windriver Workbench och VxWorks" from the course webpage. Also, download the files `clocks.c`, `delayLib.c`, `lab-1d.c` and `diodkort.c` from the course web page and read through them.

Note that we throughout this lab compendium will use the abbreviation "us" for microseconds (10^{-6} seconds) respectively "ms" for milliseconds (10^{-3} seconds).

A. Hello World

This is a very simple exercises to understand how to develop programs on the target systems running Linux. **This exercises does not need to be demonstrated or reported**.

Before you begin, make sure that you have exclusive access to one of the target systems that run Linux. Note which IP address that your target system runs. This is 192.168.210.X, where X is different for the different systems and is written on the CF card or IDE-flash card that is attached to the target systems. Ask the lab assistant if unclear.

Start by opening "PuTTY $>$ SSH" under Windows and connect to your target system. Use the username and password that you got at the lecture, or ask the lab assistant.

You should now be logged on to the target system and have a normal Linux command prompt. Use `ls` to see which files you have in the current folder, `cd foldername` to go down a folder and `cd ..` to go up a folder. To create new folders use `mkdir foldername`. Other useful commands are `cp fileA fileB` and `mv fileA fileB` to copy respectively move fileA to fileB. To remove files use `rm`. To find more information and commands, use google.

If you are not fimilar with how to operate a linux command line make sure to use `man some-name` to look up information a given command, and use `apropos -s 1 some-search`

to search for commands that do something. For example `apropos -s 1 copy` gives the following output:

```
mbl@sleipner:/media/summer/teaching/realtime/content/www/labs$ apropos -s 1 co
cp (1)               - copy files and directories
cpio (1)             - copy files to and from archives
dd (1)               - convert and copy a file
debconf-copydb (1)   - copy a debconf database
...
```

Continue by creating a file "hello.c" under Windows, you can use Wordpad or similar windows program to create it. It should contain the following:

```
#include <stdio.h>
int main() { printf("Hello World\n"); return 0; }
```

Next, start "PuTTY > SFTP" under windows, you will need to find browse your way to the folder that contains "PuTTY" on C: to find this program. Connect to the target system by typing "open <IP>". This is an encrypted FTP connection through which you can upload files. Use the commands `lcd` to change local folder (Windows folder), `rcd` to change remove folder (on the target system) and `put <filename>` to upload a file from Windows to the target system. (`get` does the opposite).

Next, you should compile and test run your program. Use `cd` and `ls` in the PuTTY window to locate your new `hello.c` file. Compile this file by typing `gcc hello.c -o hello`. This will create the program `hello`, which you can run by typing `./hello`.

Tip: you can open files on the target system using either `emacs` or `nano` if you want to edit them on the target system. If so, remember to copy them back to the windows system before you leave the lab!

B. Pulse modulated diodes

In this exercise you will use the *diodcard* to manipulate a few light emitting diods in order to examine the potential problems with timing issues.

Start by downloading the file "diodkort.c" from the course web page, upload it to the target system and compile it. This is a simple program that write an I/O port on the target system to turn on respectively turn of the LED's. After compiling the program you should run `iopermit ./diodkort`, this will run the program with the elevated privileges required to have access to the hardware[1].

One you got this working you can start with the real exercise:

1. Start by modifying the program so that it runs in a cycle turning on all the LED's on the first port (0x180), waits 10.000 microseconds, turns of all the LED's and waits another 10.000 microseconds. This should lead to the LED's appearing to shine with only 50% of their full strenght. See 1 (left) for an example of how this would appear on an oscilloscope.

[1] This is not a standard linux program, on other systems you can use `sudo` instead

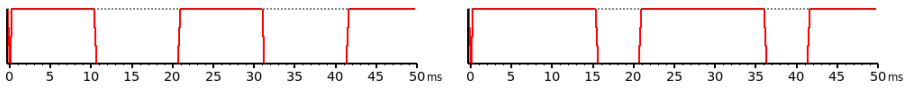


Figure 1: Example of 50% light strength (left) and 75% light strength (right) when the lamp is beeing regulated every 20ms (50Hz).

2. Change your code so that the intensity of the lamp can be regulated every 20.000us and then get any strength in the interval 0% - 100%. Eg. if it is supposed to shine at 75% you should have it turned on 15.000us and turned of 5.000us. Test run your program with a few settings of light intensity.

   Also add a counter that is supposed to be incremented every 20ms (ie. after each light cycle). Use this counter to decide the light intensity as `inten=(counter/2)%100`. This gives a value of the light strength in the interval 0 - 99. This should make the diods appear to go gradually from zero intensity to full intensity and then start over at zero again every two seconds.

3. Use the function `inb` to read the value from the switches on the diodcard (see the section "Using the I/O card" below, especially regarding how to enable reading from the card). Count how many 1's you have among these 8 bits and let this regulate the light intensity to the second port (eg. if four switches are on it should shine at 50%). Observe that you thus have to regulate *both* the first and the second at the same time, with 20ms control loops.

4. Download the program `slow.c` from the course webpage and upload it to the target system. This is a program that will make some computations and use approximatly 10-20% of the CPU. Start anoher PuTTY session to the target system, compile this program and run it *at the same time* as your diodcard program is running.

   Observe what is happening and describe in the lab report **what is happening and why**. Discuss what affect the internal scheduling of linux have on your two programs and why this effect happens.

   Also test to use the command `nice ./slow`. This will make your program to run at a lower priority. Again, explain in the lab report *what is happening and why*.

C. Getting started with Windriver VxWorks and Workbench

In this and the next two exercises you will test using WindRiver workbench in order to develop VxWorks program. To do this you need to read the compendium "Windriver Workbench och VxWorks - en liten manual för att komma igång". **This exercises does not need to be demonstrated or reported**.

You wil familiarise yourself with the development environment which lets you write, compile and test running programs on the target system all from the comfort of a graphical development environment. First you should execute a few simple programs in a simulated target system

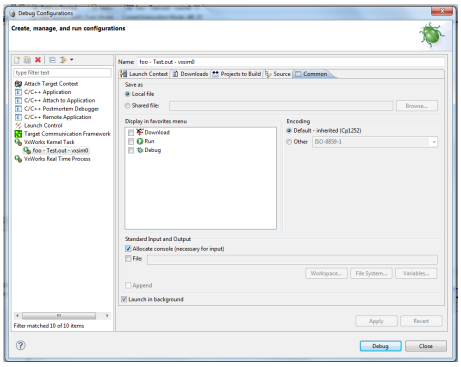


Figure 2: Dont forget to enable “Allocate console” for your debug tasks!

running in the simulator VxSim. After this you will test to use a smualted target system environment så you will not need to have acccess to to any target system during this and the next exercise.

Note that when you develop programs for VxWorks you should *not* name any C-routines `main` – it can lead to problems since VxWorks have a share namespace for all uploaded code.

1. In the workbench, start a new “VxWorks Downloadable Kernel Module” project that you call eg. `lab-1c.c` and create a new `helloWorld.c` file. Include `stdio.h` and create a function `helloWorld` that uses `printf` to write a message to the screen. Compile your project.

2. Create a new simulation environment using the menu ‘Target > New Connection”. Select “VxWorks 6.x > Wind River VxWorks 6.x Simulator Connection” and press *finish*. You should now get a new shell window on the simulated server. You can also right-click on it and select “Target Tools > Target Console” to get more shell windows.

3. Right click on the result of your compiled file (“lab-1c > Binaries > lab-1c.out”) and select *download* to upload (!) the code to the simulator. Test running your function writing `helloWorld()` in your shell window.

D. Using the debugger

For simple programs as the ones above it is usually sufficient to use only print statements to find any bugs. However with more complex programs this quickly becomes infeasible. Therefore you will need to learn to use the debugger that is included with te Windriver system.

Create a new project (lab-1d) from the same workspace as before and copy the file `lab-1d.c` from http://aass.oru.se/~mbl/realtime.

This is a simple program that calculates the length of a contour that is fed to a (fictional) milling device. It consists of two functions, one that asks the user for the coordinates describing the contour and one that remembers the last coordinate and calculates the length that is to be sawed.

1. Compile the program and upload it to the simulated target sytem. Test run the program by typing `doInput()` in your shell-window. Abort the program by pressing Ctrl-C.

2. Start the debugger, go to "Kernel Task > entry point" and give `doInput`[2] as the new task that is to be started and debugged. Also enter the tab "Common" and mark "Allocate Console". Finally, press "Debug" to start a new process and debug it. You can step through the program using the "Step over" and "Step info" button. Note that the inputs/outputs of the program is visible in Eclipse's "Console" tab.

3. What happens when you try to *step over* the first "gets(str)" expression? Give $-1.0, 0.0$ as the first coordinate and inspect the different local variable in `doInputs` when you step through it.

4. *Step into* the `cutContour` rutine and inspect how the program flow goes. Set a *break-point* and *continue* the program until this you reach this routine a second time. Give $1.0, 0.5$ as the second coordinate and keep stepping until you see what value `deltaX` is assigned as a value when the first length is calculated.

5. Use the browser to inspect the static variable in the `cutContour` routine. What happens when the program flow returns to `doInputs` and then returns to `cutContour`. Can you inspect these variables when the program flow is outside this routine?

E. SysClkRate

In this exercise you will examine how VxWorks system clock and *ticks* works.

1. Start by downlaoding the file `clocks.c` from the course webpage. Right click on your project and select *propertis*, go to *build propertis* and *build path*. Press *add* to add `-I$(WIND_BASE)/target/usr/h` as a new search path for include files. Create, compile and upload a new project containing this file.

   This project now have a function `timenow` that reports what the time is with a high precision, a function `delay` to delay the process a given relative time and a function `clockTest` that attempts to run a loopat exactly 7Hz and print what the time is.

   Start by running `clockTest` in the simultor. Can you run the function in *exactly* 7Hz? (Ie: subtract the values from consecuctive printouts and see if they are 1/7s from each other). Examine what tick frequency that VxWorks hve by usin the `sysClkRateGet` function and explain why the function cannot run in 7Hz exactly.

2. Test changing the VxWorks tick frequency by calling *sysClkRateSet*. What are a sutiable value to use for this system? What would be a suitable value if you additionally also had another function that attempted to run at 50Hz in parallel with the first? What is the disadvantage of having a *too low* or a *too high* value on the tick rate? Explain in the report what happens on each tick and what happens when the process sleeps (calls *nanosleep*) and when the operatingsystem wakes it up again.

[2] note: no parenthesis

F. Using VxWorks

In this exercise you shall develop a simpler program for VxWorks to control the IO cards, same as in exercise A above, and compare the result with how vanilla Linux and all other non-realtime operating systems behaves.

For this you will need access to a target system with VxWorks on it as well as a diod card that is connected to the target system. Note which IP address the system has. This is 192.168.210.X, where X is different for the different systems and is written on the compact-flash or IDE-flash card that is attached to the target system. Ask the lab assistant when unsure.

Read the instructions in "Windriver Workbench och VxWorks - en liten manual för att komma igång" to see how you can connect to the VxWorks target system using this IP address. Ie. you shall **not** use the sam programs as when connecting to a Linux system.

Note: you are not allowed to change the cards connected to the systems yourself since you risk short-circuiting and destroying the target systems otherwise! Always ask the lab assistant for help if you need to do this.

1. Start by connecting to you target system. Open a WinShell window from Workbench and examine the different values you can set for the tick frequency. Do you get the same results as in the simulated environment you used in exercise E? Examine what is the highest sysClkRate that the target system supports. Compute how many micro seconds this corresponds to and explain what effect this have if you have tasks that have run at very exact timepoints.

2. Read first "Using the I/O card" in the next section and test manipulating the diod card directly from you WinShell session. To do this, first run `sysOutByte(0x184,1)` to turn on the outputs of the card. After thsi you can call `sysOutByte(address,value)` where `address` is 0x180 plus the pot number (ie. 0x180, 0x181 or 0x182). As `value` you should give a byte where each 1 in the binary representaion corresponds to a lit LED and zero's correspond to unlit LED's. Use `sysInByte(0x182)` to read the values from the switches (if they have been enabled on the card, and if you have previously written 1's to that port).

3. Download the file "delayLib.c" and "delayLib.h" from the coursewebpage. Start a new "VxWorks Downloadable Kernel Module" project and import these two files. Right click on your project and select *properties*, go to *build properties* and *build path*. Press *add* to append `-I$(WIND_BASE)/target/usr/h` as a new path for include files.

   These files now give you a function `delayUsec` to make a so called *busy wait* that delays the program a specific number of microseconds.

   Describe in the report why a busy wait is neccessary if you want to delay the program with higher resolution than what is given by the VxWorks ticks.

   Next, write a function "diodcard" that cyclically turns on all the LED's on th first port, waits 10.000 microseconds, turns of all the LED's and waits another 10.000 microseconds. This should give the appearance of the LED's shining with 50% strength. See Figure 1

(left) for an example of how this would look on an oscilloscope. Testrun this program directly from WinShell.

4. Write a function "start" that starts this process and runs the function "diodcard" as a process in the backgrond. Also create a function "stop" that ends this function. Use *global variables* to keep track of the processes identity number.

   Use the command "i" in your WinShell to see which process are currently running. Test what happens with a few dirrent values of the priorities for your new process, primarily what happens if it gets a higher priority (ie. a lower value) than some of the build in process already running on the target system.

5. Change your code so that the intensity of the lamp can be regulated every 20.000us and then get any strength in the interval 0% - 100%. Ie. if it is supposed to shine at 75% you should have it turned on 15.000us and turned of 5.000us. Test run your program with a few settings of light intensity.

   Also add a counter that is supposed to be incremented every 20ms (ie. after each light cycle). Use this counter to decide the light intensity as `inten=(counter/10)%100`. This gives a value of the light strength in the interval 0 - 99. This should make the diods appear to go gradually from zero intensity to full intensity and then start over at zero again every two seconds.

6. Use the function `inb` to read the value from the switches on the diodcard (see the section "Using the I/O card" below, especially regarding how to enable reading from the card). Count how many 1's you have among these 8 bits and let this regulate the light intensity to port (eg. if four switches are on it should shine at 50%). Observe that you thus have to regulate *both* port 0 and port 1 at the same time, with 20ms control loops.

7. Download the file `slow-task.c` from the course webpage and add it to your project. This is a function that performs some computations and consumes ca. 10-20% of the CPU. Change your start functon to also start this function in a *separate process* parellel to your diodcard process.

   Observe what is happening with the LED's and describe *what happens and why it happens* in your labreport. Test to change the priority of this proces to either (a) the same as your diodcard process, (b) higher priority or (c) lower priority. For each of these cases, describe what happens and why it happens. Do you see a different depending on which process is started *first* in your start function?

8. Activate time sharing mode with the function "kernelTimeSlice(int ticks)" with the values 1 or 10 as argument. Test what happens when the two processes have the same or differnt priorities. Note how the blink frequency is change and the printouts from the computation process. Document what you ae doing and why this happens in each case.

X. Examination

To pass the lab you must hand in a written lab report at **latest** Sunday 11/5 2014. Furthermore you must demonstrate the lab to the assistant and to be able to *individually* answer any questions regarding any part of the code.

The following parts must be included in the report.

- The sourcecode for each exercise. The code shall be properly formatted, commented and with a header specifying which lab it is, your names and your study program.
- A description of how you have solved each exercise and what **conclusions** that you can draw from the lab. What have you learned by each exercise?
- Demonstate exercises B3-B4 and F6-F8 for the lab assistant.
- Answer all questions in the exercises.

Remember that the written report must be readable also for someone that have not performed the lab. Proper language and a good report writing style is required for the report to pass. Do not copy text directly from these instruction, but describe your tasks and solutions with your own words.

Using the I/O card

The different I/O ports on the PC-104 ISA style expansion card can be accessed as direct memory ports by reading or writing indidual bytes to the memory addresses 0x180 - 0x184. However, due to the special nature of this memory area you need to use specific operating system instructions to do this. Under Linux this can be done using the `inb(int address)` and `outb(char data,int address)` commands, respectively. For VxWorks you can use `sysInByte(int address)` and `sysOutByte(int address,char data)`. Note the difference in order under Linux and VxWorks.

Under Linux, you the `inb` and `outb` commands require escalated privileges, either by running as root or by using the `iopermit` command in front of the program to be run.

To send out data from the I/O card you must first activate it's outputs. This is done by writing 0x01 to the port 0x184. After this you can write data to port 0 - 2 at memory addresses 0x180 - 0x182, respectively. This is done by writing a byte to these memory addresses whic causes the corresponding bits that are ones to be held high (+Vcc) with a high impedance and the ones with a zero to be held low (Gnd). Since the cathode of the LED's are connected to ground they should thus shine iff the corresponding for each LED is a one. (Note that the LED's connected to port 2 may not shine unless the switch on the card is in the right position).

To read data from a port (only port 2 is of interest in this exercise), you must first hold the outputs high (ie. write 0xFF to address 0x182) which you need to do only once. After this you can read from the port and will get zeros on the bits that are *forced low* by the diodcard. Observe that the diodcard have two small switches that determine if the big switches should affect port 2 or not (when this switch is of, port 2 only works as extra LED's).

Usefull VxWorks functions

The following functions will be usefull for you in this laboration. Include the file "vxWorks.h" to use them.

```
int taskSpawn (char *name, int priority, int options, int stackSize,
FUNCPTR entryPt, int arg1, ...  int arg10)
```

Starts a new task (process) with a given name, a given priority, some options, a given stack size and the function (entryPnt) that should be called at the start of the task. The options that exists are not of any interest for this lab (use 0 here). The arguments arg1 ... arg10 are parameters that will be passed to the entryPnt function, and can be of any datatypes or even omitted. The priority of the task should be in the range 0 - 222, where 0 is highest prioerty and 255 lowest priority.

The function returns a task-ID (a non-zero integer), or 0 if there is an error. An example on how to start a process:

```
jc = taskSpawn("tjc", 100, 0, 1000, johnCounter)
```

The variable `jc` should of course have been declared as an integer and `johnCounter` as a function.

Hint: Normally priority values over 50 should be used to avoid blocking the systems own processes. However for some time sensitive task with very high demands on resolution may require lower priorities, if so make sure that the tasks do not run for very long time periods. Also ensure that your task gets enough stack space.

```
STATUS taskDelete (int tid)
```

Terminates a task with the given ID. STATUS can either be the value OK (0) or ERROR (-1) as defined in the headerfile.

```
STATUS taskDelay (int ticks)
```

Makes the current task "sleep" atleast the given amount of time, measured in clockticks. If ticks is `NO_WAIT`, no actual sleep will be done but the command will give any other tasks the chance to run.

```
int sysClkRateGet()
```

Returns the number of clockticks that correponds to a second.

```
int sysClkRateSet()
```

Sets the number of clockticks that should be generated each second.

```
STATUS taskPrioritySet (int TID, int newPriority)
```

Sets the priority for a given task where TID is a task-ID.

```
STATUS taskPriorityGet (int TID, int *pPriority)
```

Gives the priority for a given task.

`void sysOutByte(int port,char data)`

Writes a byte to the given memory address (port). Requires `sysLib.h` to be included.

`char sysInByte(int port)`

Reads a byte from the given memory address (port). Requires `sysLib.h` to be included.

`WIND_TCB *taskTcb(int tid)`

Returns the "task control block" for a given task.

`STATUS kernelTimeSlice(int ticks)`

Sets how many clockticks that each segment of time sharing should last in time sharing mode. If ticks is 0, timesharing is disabled.

`struct timespec{long tv_sec, tv_nsec;}`

Structure to represent time with a second (tv_sec) and nano second (tv_nsec) precision. Must include `sys/time.h` to use.

`struct timeval{long tv_sec, tv_usec;}`

Structure to represent time with a second (tv_sec) and micro second (tv_usec) precision. Must include `sys/time.h` to use.

`clock_gettime(CLOCK_REALTIME,struct timespec *tv)`

Gives the current timepoint relative to the EPOCH, you can consider this an arbitrary point of reference and only use relative values.

`nanosleep(struct timespec *tv)`

Delays the process **atleast** as long as given in the timespec structure.

Hints:

- Have a good C-programming handbook readily available during the lab!
- Take the effort to learn to work with the debugger. It will help you significantly in debugging your later labs and is a skill you are expected to have when working "for real".

Lab 2

In this lab you will once again work with a few LED's, however with much harder timing constraints. By shining the LED's into a rotating mirror and carefully timing light pulses through them you will create the illusion of a small 16x8 pixel *colour* screen. In addition to relying on much shorter timing intervals you will get a new problem since the diods have to be synchronized with the mirror, which can be done by relying on a small IR detector that is triggered by the rotating mirror.

A. Creating the illusion of a screen

Start by reading the subsection "Hardware" below.

1. Begin by attempting to turn on and off the different LED's by writing the correct bitpattern to the different ports. Describe in the lab report how you can make the first diod shine red, the second green, third blue and the fourth yellow while the rest of the diods are off.

2. Write a program that takes an array of $8 * 3$ integers (int) with values in the range 0 - 255 and which use Pulse Width Modulation (PWM) to give the appearance of each diod having a different intensity of red, green, blue corresponding to the integer values (ie, 255 corresponds to always on, 0 to always off and values in between to suitable proportions of on/off).

   What is a suitable period-time in order to make the diods not appear to shimmer? Try some different values and give your subjective opinions in the report.

   Write a function that reads groups of four integers from the command line and sets the diod given by the first number to the light intensities given by the second integer. The following session shall thus make the first five diods go from white to black, and the last three diods from green to blue. Demonstrate this program for the lab assistant.

```
0 255 255 255
1 192 192 192
2 128 128 128
3 64 64 64
4 0 0 0
5 0 255 0
6 0 128 128
7 0 0 255
```

3. Next, modify your thread so that it waits until it notices a *pulse* from the reflex detector. After this it should wait 1ms, turn on all LED's during 2ms and then turn them off again.

   Observe what happens when you let the diods be reflected in the mirror, if the diods are aligned in the direction of the rotation axiss you shall see a square reflected in the mirror. When the pulse detection is working correctly this square should appear to be stationary

on the same area of the rotating mirrro. (Note that the sensitivity of the reflex detector is very light sensitive and you may need to try different positions and alignments of the hardware).

Do not continue with the next exercise until you have this working!

4. Start by creating a global variable `int pixels[16][8][3]`. This variable shall be used to represent a screen consisting of 16 columns with 8 rows of Red-Green-Blue values.

   Modify your program from the previous exercise so that it instead of sending a 2ms long pulse where all the LED's always are on it should instead split the pulse into 16 different parts (columns) of 128μs each. In each such column, read the corresponding column from the `pixels` variable and let each of the $8 * 3$ diods corresponding to the rows be turned on only if the it's colour intensity is non-zero.

   Test writing different values in the pixels variable. Check so that you can get different colours (red, green, blue, yellow) on the differnt rows and columns.

   Do not continue with the next exercise until you have this working!

5. Modify your program above so that the diods are turned on/off in different pulse lengths in proportion to the requested light intensity of the corresponding pixel. This will require very tight timing constraints on the computer in order to be fast enough.

   You do not need to have precision of 256 colour intensities, but it is sufficient with atleast 4 or 8 different light intensity strengths. Eg. a LED should be all off if it has a value between 0-63, be on for 32μs if it has a value or 64 - 127, etc.

   Hint: If you a value in the range 0 -255 down by 6 bits, the remaining value will be on the range 0 - 3 while if you shift it down by 5 bits, the remaining value will be in the range 0 - 7.

6. Download the file `lab3.c` from the course webpage. It contains a function that will set the colour values to the pixel variable and a thread that shall be run twice per function. Use this code to create an interesting animation on the screen[3].

   What is the resulting animation?

   Hints: Since you cannot get interrupts from these I/O cards you must actively poll for the pulse from the detector. But since we know that it will take a few milliseconds before the next pulse comes you can make a short taskDelay before polling the next time. Consider how *priorities* will affect the working of your different threads.

X. Redovisning

To pass the lab you must hand in a written lab report at **latest** Sunday 11/5 2014. Furthermore you must demonstrate the lab to the assistant and to be able to *individually* answer any questions regarding any part of the code.

The following parts must be included in the report.

[3] This graphics would have been more impressive in the early 1980's

- The sourcecode for each exercise. The code shall be properly formatted, commented and with a header specifying which lab it is, your names and your study program.
- A description of how you have solved each exercise and what **conclusions** that you can draw from the lab. What have you learned by each exercise?
- Demonstate exercises A2 and A6 for the lab assistant.
- Answer all questions in the exercises.

Remember that the written report must be readable also for someone that have not performed the lab. Proper language and a good report writing style is required for the report to pass. Do not copy text directly from these instruction, but describe your tasks and solutions with your own words.

Hardware

In this exercise you will use a diodcard consisting of 8 sets of red green and blue diods. Since each corresponding red, green and blue diod is placed in the same casing they can be used to give the appearance of approximately[4] any colour that can be seen by the human eye.

The cathode of the eight red diods are connected to port 0x180, the cathode of the green diods to 0x181 and the blue to 0x182. The anode of all of the diods are fed constantly by 5V through a resistance. This means that the diods will shine when you send a zero for each corresponding bit on the I/O card, and will be turned off when you send a one.

In addition to the diods there are a detector that will detect when the mirror passes a certain point. This detector is connected to the first bit of port 0x183 and will drive it low when the mirror triggers the sensor[5]. Note that to read this sensor you need to first set the bit high, and then read and test the first bit.

Finally, there is a motor with a mirror mounted on it. This motor cannot be controlled from the hardware but is run at a more or less constant speed, depending on the surrounding air and other conditions. This means that it's exact rotation speed will be *non-deterministic*.

[4]Due to the light sensitivity of the eye some hues of eg. brown cannot be represented by RGB, and cannot be reproduced on a computer screen or CMYK printers

[5]Or due to light conditions, bad angles etc.

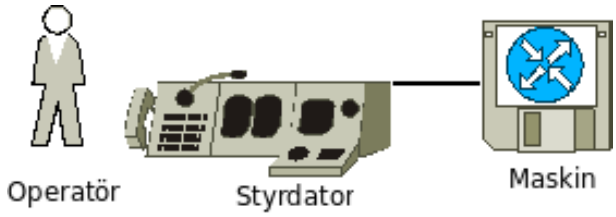
Lab 3

This lab is intended as a larger project where you will have use of what you have learned in earlier labs. For this you will use the alternative hardware described in the hardware section below.

There are two motors on this card that moves two circular discs with holes in them. These discs represent two tools in larger machine, that due to space constraints have restrictions on the positions they can occupy at the same time. If you move them incorrectly they will make a bad noise (which for a real machine it might represent significant damages and repair costs!).

To this machine is also connected a small numeric keyboard which the operator can use to give command sequences and a warning lamp that will pulsate in specific patterns so that the operator will know which state the machine is in, eg. idle, performing a job, in calibration mode or in error mode. Finally, there is a fan that you will regulate using *pulse width modulation* to control the strength with which it operates.

Your task is to *design* and *implement* the software for a the computer controlling this machine.



Specification

The control computer shall control a machine consisting of two tool, a numerical operator keyboard, a fan and a warning lamp. The two tools must never operate at the same time.

0. Make a design document that describes the different processes, semaphores, global variables and functions that you will use in this lab. Do this *before* you start implementing it.

Commands to the machine is given on the numerical keyboard and have an effect which is dependent on which *mode* the machine is in. The machine should have the following modes: configuration mode, run mode, and error mode.

Configuration mode

When the machine is started the tools can be placed in any position at all. Therefore the proper machine configuration must be given by the operator manually moving the tools by hand or by giving them small motor steps until they are in a *safe configuration*, and to avoid accidents the power to the motors must be guaranteed to be off. The machine must operate as follows during this mode:

1. The power to the motors must be off.
2. The warning lamp must signal calibration mode. This is done by PWM controlling the lamp going smoothly from 0% intensity to 100% intensity over 2s and then starting over abruptly from 0% again. See the hardware description.
3. The following 3 operator commands can be given:
   - A Let motor 1 advance one half-step. Turning off power immediately after.
   - B Let motor 2 advance one half-step. Turning off power immediately after.
   - C End configuration mode and go to *run-mode*.

After finished configuration mode the machine should enter *run-mode*.

Run mode

The machine is ready to perform jobs. During the run mode the power to the motors should be on at all time, to prevent them from moving expect during the execution of jobs. The system wait for commands from the operator specifying jobs to be performed.

5. Incomming commands are processed in parallel for the two engines.
6. When a machine is running, the other machine should always be still. Under no circumstance are the machines allowed to run at the same time, and there must always be a safety margin of 1s from one machine stopping before the next can start. This margin should *not* be applied when starting a second job on the same motor, or when starting from both motors idle.
7. When the machines are operating, the fan should be working at 100%, otherwise the fan should be working at 50%.
8. When the machines are working, the lamp should blink with 3 on/off flashes every second.
9. When the machines are not, the lamp should smoothly go from zero intensity to full intensity, and then smoothly back to zero. Each such cycle should take 4s.
10. Since there is no sensor on the engine positions, their location must be kept track of by counting the steps that are performed. There are 96 half-steps on one full rotation of the engines.
11. Motor 1 should always have highest priority and if it receives a job while the other motor is operating it should pause the other motor at the next *safe* position. There are four safe positions on each full rotation of a tool.
12. Both motors have a setting for the speed with which they should operate and in which direction all rotations should occur. The engines can either operate in high speed with 60 steps per second or low speed with only 10 steps per second. After calibration mode the engines should default to low speed clockwise rotations.

13. The following is a list of the command the operator should be able to perform on the numerical keyboard.

    - "1": Rotate tool 1 one full rotation.
    - "2": Rotate tool 1 one half rotation.
    - "3": Change direction of rotations for tool 1.
    - "A": Change the speed of rotations for tool 1.
    - "4": Rotate tool 2 one full rotation.
    - "5": Rotate tool 2 one half rotation.
    - "6": Change direction of rotations for tool 2.
    - "B": Change the speed of rotations for tool 2.
    - "0": Stop the motors at the next safe position. Release the power to the engines, turn of the lamp and fan and shutdown all processes.
    - "F": Emergency stop. Immediately stop all the stepping of the engines, turn off the power to the engines, the fan, the lamp and enter error mode (see below).

Error mode

This mode occurs whenever the emergency stop is pressed.

14. The motors should stop rotating immediately.
15. The job queues for the engines are emptied.
16. The warning lamp signals error mode by turning on for 1s and then off for 1s, repeatedly.
17. The system waits to be reset.
18. The following commands can be given:

    - "D": Restart, the system enters calibration mode.

Design specification

You should use *at least* the following processes:

- one process for each tool
- one process for reading keyboard inputs

Note that you may need to use additional processes also to handle the PWM of the fan and lamps.

You can use any combination of message queues, semaphores, global variables, monitors etc. to solve the exercise as long as you can guarantee that it executes correctly 100% of the time.

Write your program in different modules (.c & .h files) for the different processes and document the interface between them.

Hints

- Start by for instance making a process that rotates the first engine at constant speed, or one that controls the lamp or fan. Test so that they are working correctly and that you can change their speed or pattern.
- Do not use "too many" message queues or semaphores. Doing so is just as wrong as using too few!
- Test the different modules separately before attempting to integrate them.
- Do not forget to use the debugger to localize bugs.

Examination

To pass the lab you should demonstrate the working implementation of your program at latest on the last lab occasion, Furthermore you should hand in the written report, the design document and the code at latest 11/5 2014.

Hardware

For this lab you will use a card containing of the control circuits for two step engines, a lamp, a fan, a numerical keyboard as well as a few analog inputs and outputs.

The first digital port (0x180) is connected to the numerical keyboard which is a simple matrix keyboard. Use can use the function `readKeyboard` provided in `machine-utils.c` to read which (if any) of the keys are pressed.

The second digital port (0x181) is used to control the two step motors. By sending different signals to the 8 bits of this port you can turn on/off the power to the motors and to move them one half-step clockwise or counter clockwise. The effects of the different bits are described below, and the corresponding hex numbers have also been defined in `machine-utils.h`.

mnemonic	hex	Effect
M1_STEP	0x08	Steps motor 1 on negative edge ($1 \rightarrow 0$).
M1_HFM	0x04	When low, takes half steps (96 halfsteps/revolution) on motor 1.
M1_DIR	0x02	Direction of motor 1.
M1_INHIB	0x01	When high, turns of the power on motor
M2_STEP	0x10	Steps motor 2 on negative edge ($1 \rightarrow 0$).
M2_HFM	0x20	When low, takes half steps (96 halfsteps/revolution) on motor 2.
M2_DIR	0x40	Direction of motor 2.
M2_INHIB	0x80	When high, turns of the power on motor 2

Observe that it takes a few milliseconds for the engine to move after sending a command to it. So to make them move continously you need to make a small delay between the pulses triggering the movement. To exemplify, if you want to move engine 1 counter clockwise while keeping the power to engine 2 off use the following:

```
for(;;) {
  sysOutByte(0x181,M2_INHIB|M1_STEP|M1_DIR);
  taskDelay(1);
  sysOutByte(0x181,M2_INHIB|M1_DIR);
  taskDelay(1);
}
```

The third digital port (**0x182**) is connected to the fan (bit 6, 0x40) and the lamp (bit 7, 0x80) on the card.

Hint. When controlling the lamp intensity, the PWM switching need to be fast enough to be invisible to the eye. When controlling the PWM switching of the fan you need to do so *at least* at 50hz to make it smooth, where it can be either on/off fully during each cycle or on for half a cycle and off for half a cycle. Think about how you can control this without taking up all the CPU time for just controlling the fan!

Note. On some of the card the cables for lamp and fan have been exchanged. This can be seen on the socket directly and will have to be until someone with a suitably small screwdriver bothers to fix it.

Some useful functions

Semaphore

To use semaphores you need to include "semLib.h". A semaphore can be declared by:
`SEM_ID semname;`
There are a number of functions needed to create different types of semaphores.

`SEM_ID semBCreate(int options, SEM_B_STATE initialState)`

Creates a binary semaphore. Available options are SEM_Q_PRIORITY (0x1) or SEM_Q_FIFO (0x0). The former specifies that the processes are queued after priority while the later in first-in-first-out order.

Binary semaphores can be initialized either as SEM_FULL (1) or SEM_EMPTY (0). Returns a semaphore-ID or NULL in case of error.

Example - creates a binary semaphore that is initially empty with a FIFO queue for waiting processes.

```
mySem = semBCreate(SEM_Q_FIFO, SEM_FULL);
```

`SEM_ID semMCreate(int options)`

Creates a mutex semaphore that deals with priority inversion. Same options as above, but always start empty.

`SEM_ID semCCreate(int options, int initialCount)`

Creates a general semaphore that can access any natural numbers as initial value. Same options

as above.

```
STATUS semDelete(SEM_ID)
```

Terminates a semaphore with the given ID, and deallocates memory (except for the ID) used by the semaphore. Returns OK or ERROR.

The functions to give and take semaphores are always the same, regardless of type.

```
STATUS semGive(SEM_ID semId)
```

Releases a semaphore. Returnerar OK, eller ERROR om det angivna semafor-ID't är ogiltigt.

```
STATUS semTake(SEM_ID semId, int timeout)
```

Takes a semaphore. A time-out can be specified in numbers of ticks or as WAIT_FOREVER (-1) or NO_WAIT (0). Returns OK, or ERROR if the given ID is incorrect or the timeout is reached.

Message queues

To use message queues you need to include "msgQLib.h". A message queue is declared as:
`MSG_Q_ID msgQname;`

```
MSG_Q_ID msgQCreate (int maxMsgs, int maxMsgLength, int options)
```

Creates a message queue that can contain at most `maxMsgs` numbers of messages of the given maximum length. Relevant options are MSG_Q_FIFO(0x00) or MSG_Q_PRIORITY (0x01).

```
STATUS msgQDelete(MSG_Q_ID msgQId)
```

Terminates a message queue and de-allocates all used memory. Returns OK or ERROR.

```
STATUS msgQSend(MSG_Q_ID msgQId, char *buffer, UINT nBytes,int timeout,
int priority)
```

Sends a message stored in bufer with the given length in bytes. If the message queue is full the sending process will wait until the given timeout or return ERROR. Timeout is specified in number of ticks or as WAIT_FOREVER (-1) or NO_WAIT (0). The priority parameter specifies if the message is *urgent* or normal priority by giving it as MSG_PRI_NORMAL(0) (adding the message in the end of the queue) or MSG_PRI_URGENT (1) (adding it in the begining of the queue). Returns OK or ERROR.

```
int msgQReceive(MSG_Q_ID msgQId, char *buffer, UINT maxNBytes,int
timeout)
```

Receives a message from the given queue. The message is copied into the given buffer *up to* the given length maxNBytes. If the queue is empty the receiving process waits for the timeout, or WAIT_FOREVER (-1) or NO_WAIT (0). Returns number of bytes that has been received or ERROR if the timeout is reached.

```
int msgQNumMsgs(MSG_Q_ID msgQId)
```

Returns the number of messages in the queue.

Hints

The functions to send/receive messages sees a message as an arbitrary binary blob. If you want to send other types of messages than strings this can be done by casting them into strings. It can be good to create a *struct* and or a *union* for each queue specifying the kind of messages and their data that can be sent on that queue. For example:

```
#define MESSAGE_INTS 0
#define MESSAGE_COORDINATES 1
struct InputMessage {
  int messageKind;
  union {
    int ints[10];
    double coord[3];
  } data;
};
```

In the above example, if you want to send a message containing ints you fill in a structure `foo` with `foo.messageKind=MESSAGE_INTS` and the data array with the values `foo.data.ints[i] = ...`. The receiver can then easily see which kind of data it is that he has received and treat it properly.

Use the `sizeof` operator to specify the size of the message when you send and/or read the structure from a message queue.