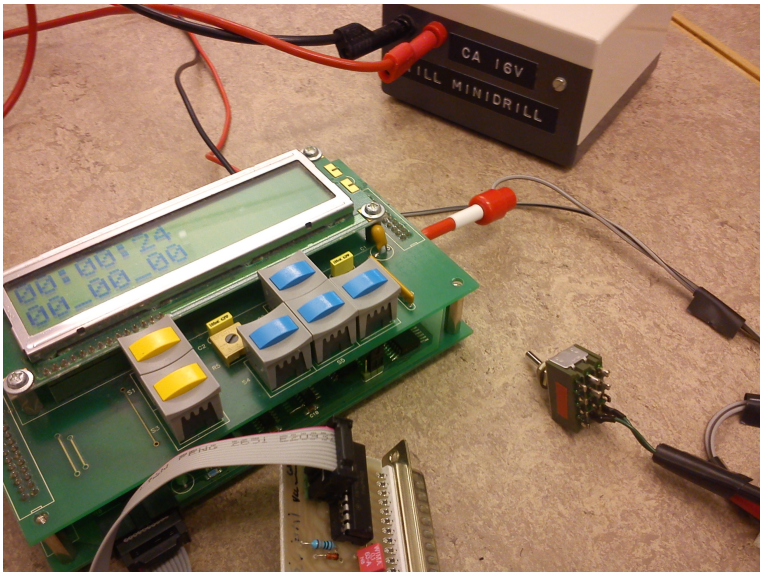
1 Lab 1 Overview



Read the entire document before starting!

In this lab you are going to implement an alarmclock to run on an AVR board equipped with LCD display and six buttons. You will develop the software in three steps:

- Implementing the alarmclock in Java using semaphores and executing on a Java simulator
- Compile the Java code to C and execute on a PC simulator
- Cross-compile to the AVR board and execute on the actual hardware

Java is the choice language in the course, but really small embedded systems may not have the resources for executing Java directly. This is the case here. However, almost all small system are capable of executing C code, so therefore we translate the code to C through a tool and compile it, first towards PC and linux for testing, and then towards the real hardware.

2 Objective

After this lab you will have implemented the real-time software for a small embedded system. You should know the following terms:

- Sporadic and periodic threads
- Semaphore

and be able to

- Use semaphores for signaling and mutual exclusion
- Share data between threads through a passive object protected by a semaphore
- Design small realtime systems containing both event-driven and periodic activities

3 Preparation (before the lab)

It is generally a good idea to start programming the exercise before the lab occasion. *Before* you start programming you need to have a design of the system. This is what you prepare at the exercise session before the lab. The design should answer the following questions:

- What event-driven and what periodic activities are there in the system?
- What semaphores are there? Which are used for signalling and which for mutual exclusion?
- What information is shared between activities? Where is that information stored? How is it protected against data corruption?
- For sample operations (such as setting alarm time, shutting off the alarm, ...), how do the activities cooperate and exchange information?

You need to get a good grasp of the handout code before starting to program. You should obtain a copy to your home folder (assuming you are using a linux shell, you can also use some sftp software like filezilla):

```
scp -r username@login.student.lth.se:/usr/local/cs/rtp/lab/alarmclock .
```

or if you are on a school computer:

```
cp -r /usr/local/cs/rtp/lab/alarmclock .
```

Do not forget the end point (.). You should get to know the hardware interface classes called ClockInput and ClockOutput, available in the done folder of the handout code and also described in the material for exercise 2.

4 Java AlarmClock

4.1 Getting started

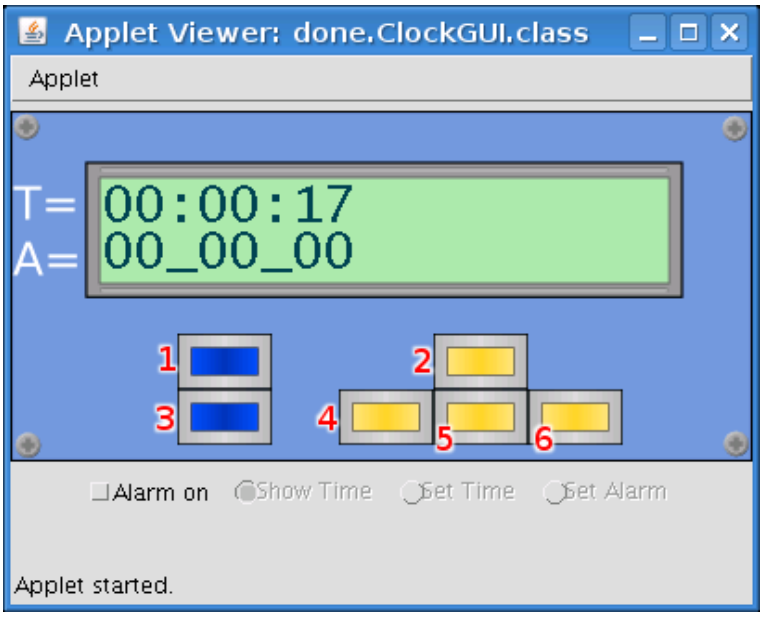
The instruction assumes you are sitting at a linux school computer with a shell (terminal/console). It is preferable if you have created a home folder for the course, i.e. `mkdir eda040; cd eda040`.

- Your first task is to copy the handout code to your home folder.
  `cp -r /usr/local/cs/rtp/lab/alarmclock .`
  Do not forget the end point (.).

- Next you should try to compile and run the handout code as is.
  ```
  initcs
  cd alarmclock
  javac done/*.java
  javac todo/*.java
  appletviewer webpage.html
  ```

- Also try to translate the Java code to C and compile and run on the linux computer.
  `/usr/local/cs/rtp/lab/build_clock.sh`
  Answer `Y` and press return to the question `Compile for host machine?`.

4.2 Using the simulator



First, you must click somewhere on the image of the clock to give the applet keyboard focus (light blue area). Clicking the check box or another window will steal keyboard focus again.

- Hold *Shift* to set clock time (button 1).
- Hold *Ctrl* to set alarm time (button 3).
- Hold *Shift+Ctrl* to toggle alarm on or off.
- Use direction keys for buttons 2-6.

The tow lines on the LCD display should be fairly obvious. The first line displays the current clock time. The second line displays the alarm time. When the alarm is set the separators in the alarm time turn into colons, otherwise they remain as underscores. When the alarm is beeping the separators will flash with the beeps of the alarm.

Below the buttons is a small status field which displays the alarm status with a check box and the current input mode with three radio buttons. The radio buttons may not be manipulated directly, but the check box can be used to modify the alarm status (on/off).

When either of the modes *Set Time* or *Set Alarm* is active the user may increase the digit under the cursor (indicated by an underscore) by pressing the *up* button (2), decrease the digit under the cursor with the *down* button (5), shift cursor left with the *left* button (4) and shift cursor right with the *right* button (6).

The C simulator works in a similar fashion.

4.3 Programming

You will implement your realtime system in the todo folder by modifying the present AlarmClock class and adding further classes as your design requires. To ensure that your solution is possible to translate to C you should also try to translate your code from time to time. Read below for a bit of advice on making your Java code compatible with the translation tool.

5 C on linux-PC

5.1 Translation script

Translating from Java to C and compiling can be a cumbersome process. Therefore a build script has been prepared for you to hide most of the grizzly details. To build for the PC you need to stand where the `todo` and `done` folders are located and type:

```
/usr/local/cs/rtp/lab/build_clock.sh
```

Answer Y to build for the host (linux) system. If all goes well, after some time you will get a statement asking you to press return to run the resulting compiled program. If something goes wrong you will be informed that the build failed and you can look at the `build.err` files located in the `build_lnx` and `build_lnx/build` folders to try to determine what went wrong.

5.2 Some advice

The Java to C translator cannot cope with the entire Java language and framework. Also, from an embedded systems point of view it is best to

import as little as possible to minimize memory requirements. Avoid * when importing, just import the classes that are needed.

For the lab a very small framework subset is necessary:

- Thread
- System.getCurrentTimeMillis()
- LJRT semaphores
  - se.lth.cs.realtime.semaphore.Semaphore
  - se.lth.cs.realtime.semaphore.MutexSem
  - se.lth.cs.realtime.semaphore.CountingSem

Some advice on what to avoid:

- Generics do not work.
- Try not to use more than two threads.
- Avoid parseInt, etc. for converting time values. Use integer arithmetic instead.
- Avoid recursive calls. Each thread has a very limited stack size.
- System.out.* is not implemented. Comment out your debug output before translating to C.

5.3 A comment on LJRT

The curious reader might wonder why we use an *inhouse* package to provide synchronization primitives. The explanation is that the LJRT package (Lund Java RealTime) was specifically developed to support Java to C translation to small embedded systems. What is happening is that during translation the Java primitives are essentially replaced by their C counterparts. The translation tool we use was developed as part of a doctoral realtime thesis presented a few years back. The tool is now hosted on `https://launchpad.net/ljrt`.

6 C on AlarmClock hardware

Building for the AVR hardware is very simple. You use the same build-script as when building for the linux PC, but you answer `N` to the question on building for the host system.

You need to make sure of a few things:

- You need to build on the computer to which the AVR hardware is connected since your program is uploaded and flashed using a parallel port interface. Should you forget the build script will complain with a message similar to:

```
/usr/local/cs/rtp/lab/build_clock.sh
Compile for host machine?
Answer Y or N (default Y):
N
compiling to run on target (AVR)...
build finished, press RETURN to upload program to AVR

avrdude: can't open device "/dev/parport0": No such file or directory
avrdude: failed to open parallel port "/dev/parport0"
```

- Before pressing RETURN to upload the program the reset switch should point towards the red.

Always tell the lab supervisor when you want to run on the AVR hardware.

7 Getting approved

To pass you need to show your solution to the lab supervisor. You should be able to explain what you have done and motivate why you have done it. A working Java solution free of realtime problems is compulsory, common problems are data corruption and excessive use of CPU. In case you run into problems with the Java to C translation it is up to the lab supervisor to decide if and when you have done enough to pass.

8 Miscellaneous

- *At home* You should be able to complete the Java part of the exercise on your own computer. You need to download the code using a sftp client such as filezilla. However, for translation to C and running on the AVR platform you need the school computers.
- *Eclipse* If you create a project in Eclipse you need to remember to add the `csrtsem.jar` file to your project (look in project settings).
- *Semaphores* You are (in this lab) only allowed to use take and give operations on the semaphores, tryTake is forbidden.