How to begin:

1. Start x2go client in your client computer. It internally uses SSH, so please make sure in advance that you can connect to unix.ncl.ac.uk with SSH.
2. Use x2go to connect to the coursework server unix.ncl.ac.uk; configure x2go to use your campus login, the private key of SSH and XFCE desktop.
3. Copy the example design into your home directory by typing: cp -a /tmp/EEE8087/examples ~/
4. Change your current directory to the examples directory: cd ~/examples
5. Configure the shell environment by typing the following: PATH=/tmp/EEE8087/bin:$PATH
   A. Do this one time for each new shell instance.
6. Compile the example by typing: make
7. Start the debugger connected to the board by typing: kmdr ?0?, where "?0?" is a 3-digit code of the board. The codes are in the format: workstation (1-6), 0, the board connected to the workstation (0-3). Refer to the spreadsheet where the codes are allocated to each student.

   A. In case somebody else is using your board in the allocated to you time slot - find out who by typing "ps aux I grep remserial???" (replace ??? as above).

8. Try the supplied examples and create a couple of your own.
9. Observe the LEDs flashing on the board by connecting to the video camera. There are one camera per workstation. They are accessed by first mapping the camera port from unix.ncl.ac.uk into your PC, and then opening the corresponding port with the browser.

   A. Video ports: unix.ncl.ac.uk, port 10?80 (the digit "?" in the middle is the workstation number 1-6).

   B. Mapping with ssh under linux: ssh -L 10?80:localhost:10?80 loginname@unix.ncl.ac.uk -N
   C. Mapping with PuTTY under Windows: set the tunnels option as above
   D. Open the video in the client machine: http://localhost:10?80/index.html. (Links to an external site.) replace "?" as above.
   E. An option is to open the camera with a browser directly in unix.ncl.ac.uk in x2go window, but this will slow down the server for everybody else. For testing it's fine, but then please map the ports and run the browser in your local workstation.
10. Implement the scheduler in C upon reading the downloaded files, coursework specification and the lecture notes.

    A. Create a "skeleton" of the programme: each block is a naked function, include calls to the other functions and SWI.
    B. Add the timer control
    C. Debug the code by using KMD
    D. Add the task queue and the dispatcher code, debug.
    E. Add code for the tasks, debug.
    F. Add code for fault injection (one selected task goes into a loop on demand) and for protection from overruns

I. Simple protection - IRQ is enabled during the tasks, so the overrunning task gets interrupted
II. More complex protection - the task queue is reordered to move the bad task to the end of it
III. The most complex case - once the overrun is removed, the affected task continue from the brightness appropriate for the moment (the state recalculated to the time of the fault removal)

Note: one can get a distinction mark (70+) without implementing 9.f.iii, and a merit mark (60+) without 9.f.ii, subject to the quality of the report.

Coursework plan

1. Introduction talk. Please ignore the part of the talk about the cloud machine and setup X server at home - these are replaced by accessing unix.ncl.ac.uk server with x2go client installed in the home or lab machine. Try cross-compiling the examples, try KMD debugger -- this will be the main task to comple in the lab on Monday.
2. Experiments with interrupts, timer, LEDs, HW/SW interrupts irq/swi, skeleton of the Timeline scheduler (empty function declarations, adding calls from one block to the other). 24 boards available - for the allocation of boards see this spreadsheet: EEE8087-2021 equipment access.pdf
3. Timeline scheduler implementation - talk
4. PWM task implementation - talk
5. Protection from overruns - talk
6. Finishing

To our previous discussion... This is how to use IRQ, see diagram

.

At the beginning of main() unmask and enable IRQ by the following commands:

*INT_ENABLE_P I= INT_MASK_TIMER;

IRQ_ENABLE;

The IRQ is triggered by a condition when *TIMER_COMPARE_P has the same value as *TIMER_P.

Inside the IRQ function one needs to do the following:

Reset the IRQ signal

*INT_RAW_P=0;

Then program the *TIMER_COMPARE_P to the next interupt.

If at any point the CPU mode is changed, e.g. from IRQ to SVC, then interrupts become disabled, and IRQ_ENABLE may be needed again.

Microcontroller specification:

AT91SAM9261, formerly Atmel, now Microchip

http://www.microchip.com/wwwproducts/en/AT91SAM9261 (Links to an external site.)

ARM CPU architecture (we use ARM7, A32 instruction set):

http://infocenter.arm.com/help/index.jsp (Links to an external site.)

Accessing the Cameras

In order to access the cameras, one needs to set up an SSH tunnel to unix.ncl.ac.uk and import the ports 10180 10280 10380 10480 10580 10680 into the local machine (localhost). Then point your browser (tested with Firefox) to
http://localhost:10180 (Links to an external site.)
http://localhost:10280 (Links to an external site.) (Links to an external site.)
http://localhost:10380 (Links to an external site.) (Links to an external site.)
http://localhost:10480 (Links to an external site.) (Links to an external site.)
http://localhost:10580 (Links to an external site.) (Links to an external site.)
http://localhost:10680 (Links to an external site.)
I use the following command line under linux:

```
ssh -L 10180:localhost:10180 -L 10280:localhost:10280 -L 10380:localhost:10380 -L 10480:localhost:10480 -L 10580:localhost:10580 -L 10680:localhost:10680 nab29@unix.ncl.ac.uk -N
```

The system works as following:
RPi-s in the lab have cameras and run a small web server each at port 8000. I've added a script to them, which starts ssh end exports the port 8000 to unix.ncl.ac.uk as port 10180 etc. (see above). Then, you login to unix.ncl.ac.uk from the outside and import these ports into your home machines. Thus, by accessing http://localhost:10180 (Links to an external site.) and others, you are effectively connecting to the RPi.

Summary of the discussion of the TL scheduler implementation

Start with a "skeleton" of the program, which is a set of functions, each representing a block in the block diagram. All functions are "naked", because the block diagram is an FSM, i.e. we never return from the blocks or functions. Calling without return leaves the return address in the stack, which will eventually use all the memory. With naked functions the return address is not stored. In your code please add a comment to each function indicating which block it represents.

The pre-defined function names which you must use are main(), irq() and swi(), which are blocks Initialize, Update and SWI respectively.

Transitions between the blocks are implemented by calling the corresponding functions by name, except for calling irq() and swi(). The function irq() is called automatically by the action of the hardware timer. The function swi() is called from task() by the command asm("swi 0"), as shown in howto.txt and ex-swi.c .

The timer is initialized in main() and programmed to execute a time interval in both main() and irq(). From main() it makes sense to go to Sleep block rather than directly Update in order to avoid complications associated with arriving to the IRQ handler by different methods, which was discussed.

Update block. It serves the interrupt first: clears RAW_INT register, re-programs the timer by putting in TIMER_COMPARE register the value of TIMER register plus the delay expressed in the timer increment periods (1ms), switches to SVC mode, enables the interrupts again. The second step is to update the task queue by putting the task numbers into the queue array in the order of their execution and by resetting the queue index to point at the beginning of the array, which is index 0. Finally, transition to Dispatcher by calling its function.

Dispatcher. It is a very short and quick function. It is called before starting any task and after each task. When the queue is empty, the dispatcher calls sleep_state() instead of task(). Dispatcher determines whether the queue is empty or not by checking the queue index. If it points to outside the array of the queue, then the queue is empty and a call to sleep_state() is executed. If not empty, then the task id is read from the queue array, put in a global variable (so task() could receive it for acting as a particular task number), then the index is incremented (preparing for the next run of Dispatcher), the mode is switched to USR, and task() is called.

Tasks. There is a single function for all tasks. It acts differently depending on the number provided by Dispatcher and stored in a global variable. As it is executed in USR mode, task() can't access LEDs directly. So, it has to compute the LED values as integers and make them available to SWI. The last instruction in task() is to execute a software interrupt asm("swi 0").

SWI. It is needed to escape from USR mode. As any interrupt it disables interrupts, so don't forget to enable them again. SWI is also a good place to copy the computed by task() values into the LED register. It ends with calling Dispatcher according to the block diagram of the scheduler.

SLEEP. The code is provided, just a simple loop forever. The only escape from this loop is the timer interrupt, so make sure the interrupts are enabled.

Task queue. We use a simple queue, which is an integer array length 6 for storing a sequence of task numbers, plus an integer variable pointing at the current task - queue index. The queue index variable is initialized to 0 in Update, and then incremented by Dispatcher every time. When it starts pointing to the outside the array, the queue is empty. Dispatcher will read the task number from the array according to the queue index in order to supply it to task().