LAB6, State-Machines

Goal

Design an embedded system using state-machines.

Given files

You can find the files you need in `~TDDI11/lab/skel/lab6`.
Copy the files to `~/TDDI11/lab6`. Your modifications goes to `main.c`.

System overview

In this lab your program will work in an *imaginary* embedded system environment similar to the RIMS system described in the Vahid & Givargis book.

The system has one 8-bit memory mapped input A. Simply reading from "variable" A get values from the input. A motion detector is connected to bits 0-6 of this input. A high value on the input indicate violent motion, a low value small trembles. Zero means no motion at all. The last bit, bit 7, is connected to a simple push-button. The bit is high while the button is held down.

The system has one 8-bit memory mapped output B. Simply writing to "variable" B put values on the output. Each bit of the output is connected to a red LED light. Thus each LED can be controlled individually. The lights are placed in a row to produce, for example, the backlight of a bicycle.

I/O on individual bits of A and B must be done by bit manipulation. A0, A1, B2, etc. do *not* exist.

System emulation

Two files provide system emulation, `rims.h` and `system.o`. The file `example.c` provide an example implementation. You will write your implementation in `main.c`. (If you have the RI-Tools that come with the book you can use `RIBS` to build state-machines and convert to C. To make the `RIBS`-generated code compliant you must only specify the return type of functions and add a return statement in main.) The emulation provide the following:

```
void TimerSet(int msec);
```
Set how many `msec` that elapse between `TimerISR()` calls (in milliseconds).

```
void TimerOn();
```
Turn timer on so that `TimerISR()` is really called as specified by `TimerSet()`.

```
void TimerISR();
```
For you to implement.

Study the example program before you start (try to hold the left mouse button down). The push-button in the emulated system is mapped to the left mouse button. The motion detector get values from moving the mouse, and the emulation shall terminate by clicking the right mouse button (if it does not exit it is usually stuck in some infinite loop).

Note: `rims.h` will instruct the preprocessor to do some automatic transformations of your code in order to compile. If you have strange problems, look at the output from the preprocessor to see the same code the compiler sees (command: `cpp main.c | less`).

Assignment

The goal of this assignment is to design the function of an advanced bicycle backlight. The functionality shall be as follow:

The system shall try to detect when the bicycle is in use (“on the road”). The light shall always be off unless it is on the road. Since only the motion detector is available it is impossible to determine if the bicycle is on the road with 100% accuracy, so we try to guess.

When motion is detected on the sensor it can be because someone just bumped the bicycle. Then it is not likely to get more readings immediately. It can also be because someone mounted the bicycle. Then it is likely we get more readings immediately. And it can be because someone is riding the bicycle on the road. Again likely to get more readings. Thus, if we get several consecutive readings we can consider the bicycle to be on the road.

Since the motion sensor is sensitive it may be triggered just because “natural vibrations”, like from a nearby road. Thus we want to put a limit where we say that the detected motion was not high enough to consider. Any motion with a value below 3 is considered as merely noise, and ignored.

For detected motion at 3 or above we will guess, according to above reasoning, that the bicycle is on the road when two consecutive readings detect motion. Once we have detected that the bicycle is on the road we require that it shall be immobile for 10 seconds before considering the bicycle parked.

Unless the bicycle is on the road the light shall always be off.

The push button is used to control the mode of operation. Your light (all LED’s simultaneously) shall support at least the following three modes:

1) In mode 1 the light is always off.
2) In mode 2 the light is blinking.
3) In mode 3 the light is constantly on.

Pushing the button cycles through the modes in the following fashion. The light starts in mode 1. Pushing the button once select mode 2. Pressing again select mode 3, and pushing again reverts back to mode 1. If you hold the button pressed you will cycle through the modes. You may ask of the user that he must press the button firmly (hold it at least 300ms) to switch mode. You may add more modes for more fancy operation, for example letting the light “grow” from the center diodes (00011000 => 00111100 => 01111110 => 11111111).

Implement state-machines to model the system. *You must at least split the system in one part that detect motion (on road or not), and one part that control the light-mode.* Draw the state machines on paper first and get feedback from your assistant, then implement them in C-code in the file `main.c`. Just type `gmake` to compile. In this lab you have to start the compiled program (`./main`) yourself, manually.

Deliverables: A clear and accurate drawing of the resulting state-machine graphs shall be handed in along with a printout of your C-implementation.

Debug: If you have problems to see why your solution does not work you can use regular `printf` to print debug-messages to the terminal. Using `printf` is of course not possible on a real system such as the described, but very convenient during development and emulation.