Programing for Embedded Systems
Final Assignment, Part 1
Implementing a Petrinet for a Two-elevator Concurrent System

Due: December 29th, at 11:59pm

Submission method: Post to CyberUniversity, as per the TA's instructions (kh953609@gmail.com).

What to submit: Two files attached to the email. One file is the modified measure.c file, the other is the modified measure.ini file. Each file must have you student ID # in a comment on the first line of the file. The email title should be "P4ES Final Assignment, Part 1".

Project description:

In class, we have seen how petrinets can be used to expand DFAs to concurrent systems. We have also seen how a DFA can be implemented directly into C, using a while loop and a switch statement. We have not, however, similarly implemented a petrinet. One issue of doing such an implementation is that the transition choices can be nondeterministic, unlike a DFA.

In a previous posting, I provided PowerPoint slides that give the petrinet for one elevator in a three-floor building. (I did not show the 1st and 2nd floors, but they are similar to the 3rd floor: 1st floor uses ▲ instead of ▼, and 2nd floor uses both.)

In this assignment, you will implement that petrinet. And once you get it working, you will add a second elevator. Their behavior will be similar to the behavior of the elevators in the F building. To remind you how elevators and concurrent elevators work, see Figures 1 & 2, below.

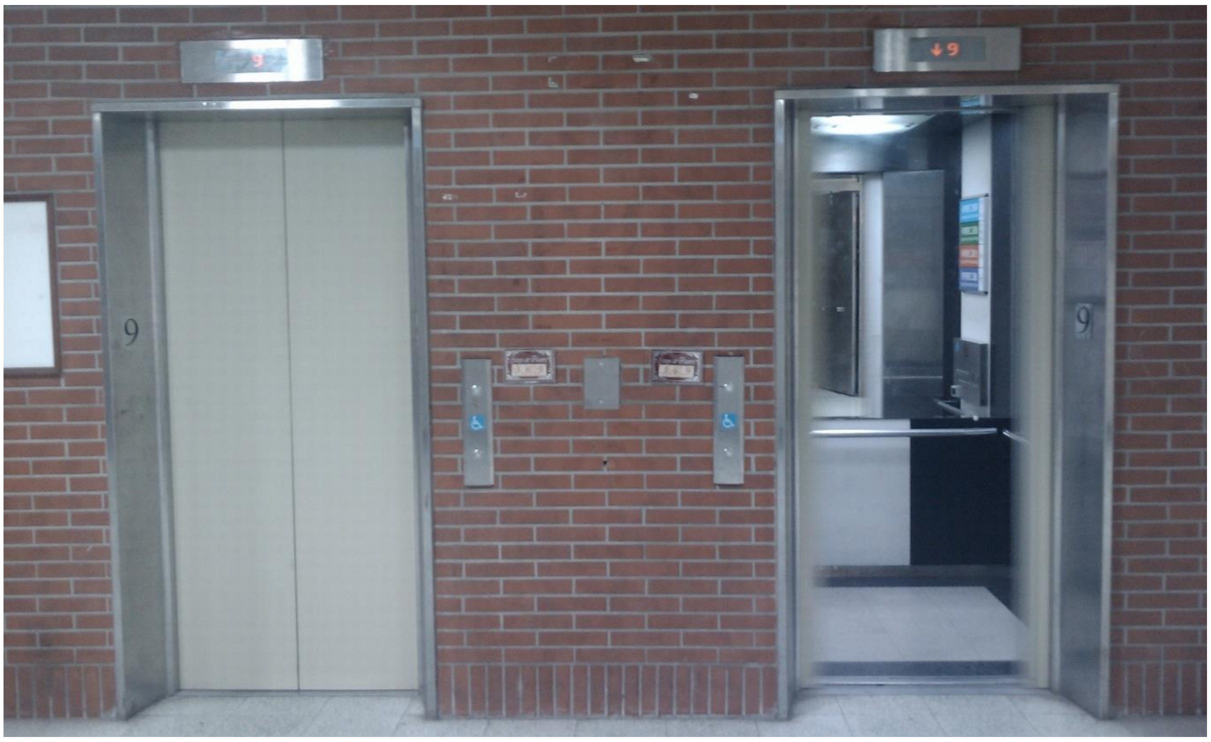


Fig 1. The F building's elevators, as viewed from the 9th floor. From your experience with this elevator, you know that the two ▼ behave the same whichever one you press, and so do the two ▲. Moreover, since the F building only has 9 floors, the ▲ button is useless. In other words, although four buttons are visible, we really only have to talk about one button. Notice also that the right door has opened. We know, from experience, that either door may open in response to the pressing of the ▼.

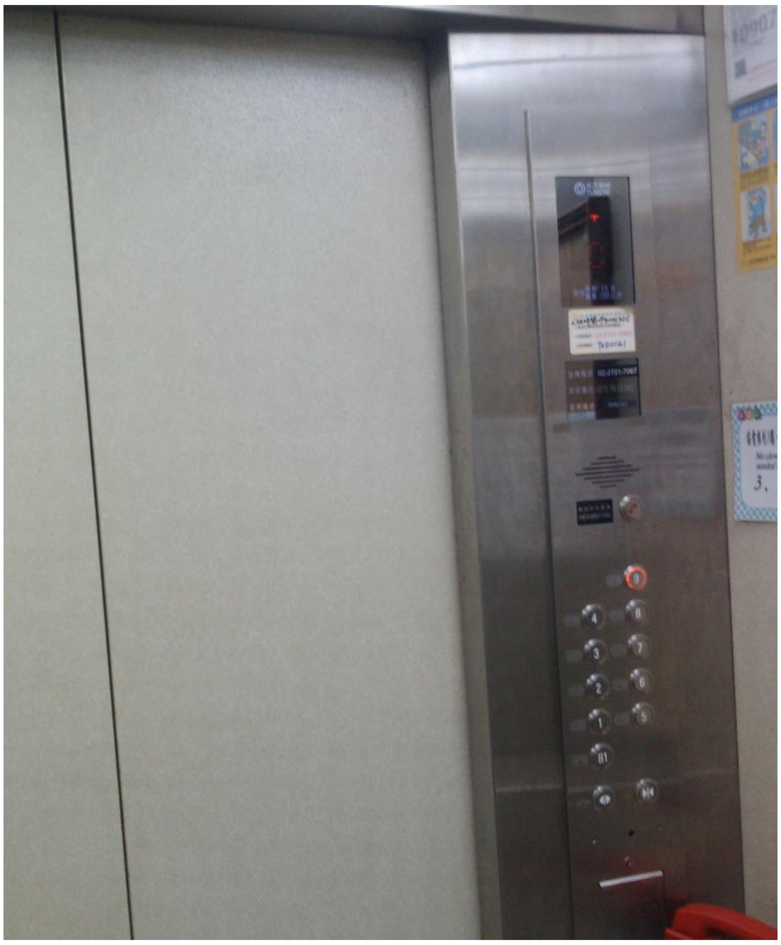


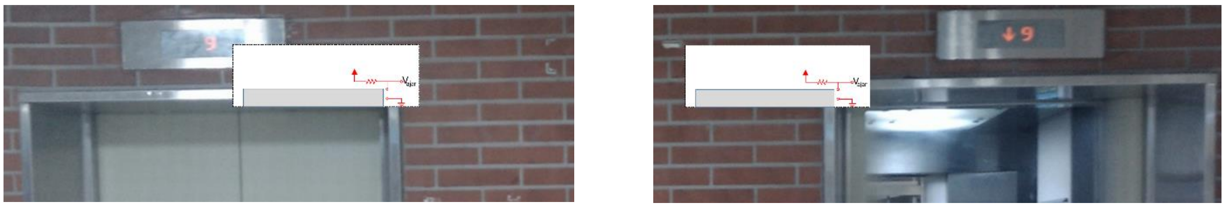
Fig 2. The inside view of the elevator. Notice that each floor number gets a button. Notice also that there are buttons for opening (◀|▶) and closing (▶|◀). In many elevators (including the one in this homework) the closing button doesn't do anything, so we'll ignore it. (This button does work in the F building elevator, however.) Notice also that the button's light stays on until the door opens on the correct floor. (In the F building elevator it blinks; our elevator will just turn it off when the door starts opening.)

The buttons light up when pressed, and remain lit up until they are serviced. This behavior is well-suited to implementation by petrinet, because the tokens indicating that the buttons were pressed can also be used to drive the lights. The physical design of the system is that each button drives a pin on port 0, and each light is driven by the corresponding pin on port 1. The button inputs are noisy and need to be debounced for 3ms. Table 1 gives their pin positions.

Besides these buttons, the elevators also have sensors telling them their status. Table 1 gives these sensors' pin positions as well. One sensor tells the elevator when its door is either fully open or fully closed; in other words: $\overline{\text{ajar}}$ (the word "ajar" refers to a door that is partially open). The sensor is noisy and needs to be debounced. If you are wondering how such a sensor could be made, Figure 3 shows one possible implementation. The second sensor indicates whether the elevator is between floors (so, looking at the petrinet PowerPoint slide, the sensor will provide a high signal when the elevator is at floor 1.5 or 2.5. This signal is not noisy. (In a real elevator, there would be additional signals for being overweight or having the door blocked – but we won't do those things.)

L⬌	P0.8	R⬌	P0.12	1▲	P0.16	$\overline{L_{ajar}}$	P0.20
L❶	P0.9	R❶	P0.13	2▲	P0.17	$L_{midfloor}$	P0.21
L❷	P0.10	R❷	P0.14	2▼	P0.18	$\overline{R_{ajar}}$	P0.22
L❸	P0.11	R❸	P0.15	3▼	P0.19	$R_{midfloor}$	P0.23

Table 1. The input pin connections. "L" refers to the left-side elevator, and "R" refers to the right-side one.



(a) *(b)*

Fig 3. One possible way to generate the ajar signal. The white box indicate a cut-away view looking into the wall. The gray rectangle shown in the white box is a metallic plate that is attached to the top of the door and which cannot be seen because it is behind the wall. The red diagram is a voltage divider circuit. When the metallic plate slides over the contacts, the circuit will be completed and the indicated output terminal's voltage will drop to 0V. The size of the plate is such that no contact occurs when the door is fully closed (*a*) or fully open (*b*). That is why neither (*a*) nor (*b*) shows the red circuit touching the plate.

L⇔	P1.16	R⇔	P1.20	1▲	P1.24	control codes for L	P1.28
L❶	P1.17	R❶	P1.21	2▲	P1.25		P1.29
L❷	P1.18	R❷	P1.22	2▼	P1.26	control codes for R	P1.30
L❸	P1.19	R❸	P1.23	3▼	P1.27		P1.31

Table 2. The output pin connections. "L" refers to the left-side elevator, and "R" refers to the right-side one.

Now we discuss the output of our system. P1.16-P1.27 are the LEDs corresponding to the buttons. That leaves just 4 pins (P1.28-P1.31) to control the elevators. Thus, we have only 2 pins per elevator (P1.28-P1.29 for L & P1.30-P1.31 for R). With 2 pins, we can only issue 4 commands. Fortunately we only need four commands: 00 - do nothing; 01 - go up; 10 - go down; 11 - change the door state (i.e., if it is currently open, then close it; if it is currently closed *or closing*, open it). Table 2 summarizes the output pins.

Signal generator description:

You will need to create 12 buttons: "L<|>", "L1" , "L2" , "L3", "R<|>", "R1" , "R2" , "R3", "1^", "2^", "2v", and "3v". Each of these buttons should behave like the noisy 50ms button press from the previous homework. They all must be implemented by the same function in the ini file. That function must not have a loop; it always ends after 53ms.

You will also need to create a signal function to drive P0.20-P0.23. This function is executed by creating a "start" button. Once the toolbox button is pressed, start() loops and never exits. This start() function responds to changes in the elevator states. The state changes are indicated by changes in P1.28-P1.29 (for the left elevator) and in P1.30-P1.31 (for the right elevator). These changes are responded to with signals on P0.20-P1.21 and P0.22-P1.23, respectively. Because of this symmetry, I will here only describe the behavior for the left elevator.

In discussing the left elevator's signal behaviors, the first issue is how to respond to broken input. For example, an elevator with open doors can't go down. If your program tries to do this, it means your LPC2129 program has a bug. And it will not be the job of the ini file to look for these bugs; it will assume commands are logical.

A second issue is when the LPC program will issue its commands. The answer is that the commands are held until the acknowledgment of completion is received. So, for example, when the LPC2129 issues a command for the left elevator to go up (P1.28=0 & P1.29=1), it anticipates a response that the $L_{midfloor}$ sensor will go high for a period of time, *and then go low again*. The going low is the indication that the next floor was reached. This indication is controlled by the ini file and serves as an acknowledgement signal for the LPC program. Hence, the LPC will hold the 01 on P1.28-29 until this acknowledgment is received. At that point it *must* change the value on the pins (either to 00, 10, or 11). It must do this because the ini file will not consider a new command to be issued unless it is different than the previous command.

The above issue leads into the third issue. This issue is: what if the LPC decides to send the elevator straight from 1 to 3, without stopping on 2? The answer is that the LPC must first allow the elevator to stop at 2 (but with the doors staying closed) for 5ms. In other words, a 00 command will be sent and held for 5ms, before the next 01 command is sent. (Technically, real elevators are optimized to save time by not slowing down on floors that aren't requested. We will not implement such an optimization.)

The fourth issue is that a closing door may be commanded to reopen before it finishes closing. The way that the LPC program will indicate this by changing from 11 to 01. Ordinarily, a 01 command would indicate that the elevator should go up. The ini file knows, however, that the elevator door is not yet closed, so the $\overline{\text{L}}_{ajar}$ pin is still low, so the LPC cannot actually want to go up. Thus the ini file instead recognizes that this is, in fact a door-open command. (The reason that we need this special rule is because regular door opening and closing both use the 11 command.) The inputs and responses are summarized as follows:

Command (P1.28-29)	Secondary Condition	Response:		
		Pin:	Becomes:	For this long:
10 (↓)		P0.21	1	8 sec
01 (↑)	P0.20=1	P0.21	1	8 sec
01 (↑)	P0.20=0 (↤↦)	P0.20	0	Exactly as long as P0.20 already has been low. (The idea is that the door will take as long to reopen as it has so far taken to partially close.)
11 (↤↦/↦↤)		P0.20	0	1.5 sec (if P1.28 stays high the whole time)

Table 3. Sensor responses from the ini file. This table is for the left elevator. The right has different pins.

Project implementation and testing details:

The first step is to get pins to work. This means performing actions like you saw me do in class:
1) Make a copy of the measure project.
2) Add "while();" before the printf() in main().
3) Make the first two statements of main() set IODIR0 and IODIR1 to allow P0.8-23 to be inputs (*ie*, just set all 32 bits to 0) and P1.16-31 to be outputs (*ie*, set the top 16 bits to be high).
4) Remove everything from tc0() except for the last 2 lines.
5) Add into tc0() code to: read P0.8-23, then SHIFT it down by 8, then AND it with 65535, then put that value onto P1.16-32.
6) Modify measure.ini to have a new signal function that puts a random value onto pins P0.8-23, then holds it for 0.1sec, then loops.
7) Create a button to execute your new signal function.
8) Display GPIO0 and GPIO1.
9) Position the two GPIO windows the way I did it in class, so that that P0.8-23 is directly above P1.16-31.
10) Begin running the LPC program and hit the toolbox button.
12) Verify that the two ports are displaying the same values.
13) Remove unused parts of measure.c (a little at a time, and re-verifying the output after each removal.) Do not remove the ability to use printf().

The second step is to implement just one elevator. You will not use a loop, because tc0()

executes every 1ms. You also will not put your petrinet *directly* into tc0(); instead, you will call a function to implement it (and remember: I said not to use any loops).

Your implementation will require places. These will be stored in two 32-bit bit vectors. One bit vector will hold the places for the tokens of the signals and buttons: let P0.8-23 correspond to the same positions in the bit vector, and put all other signals into any of the other remaining bits. Let the second bit vector hold the sixteen elevator states: (3Fclosed, 3F⇤⇥, 3F↦↤, 3Fopen, 2Fclosed, 2F⇤⇥, 2F↦↤, 2Fopen, 1Fclosed, 1F⇤⇥, 1F↦↤, 1Fopen, 2.5F↓, 1.5F↓, 1.5F↑, 2.5F↑). Although this implementation may seem to be wasting bits, remember that we'll be adding a second elevator later, and that will use the extra bits.

You implementation will require tokens. Some of these are input from port0. The top of the tc0() function should update these tokens. You should be able to do it with just one line of code. The next lines of code should remove bits for buttons you would ignore (eg, ⇤⇥ while the elactor is moving). After these lines, use an IOSET1 to put 12 button token values onto the corresponding LED pins (remember to shift up by 8 bits). The only input token that is not received from port 0 is the timer that keeps the elevator door open for five seconds. This signal is generated inside tc0, by counting to 5000 from the time of the door becoming open (you won't need to put this token into the bit vector, as you can just use the counter directly in the relevant part of the petrinet).

You implementation will require transitions. This is where most of the code will occur. From our lectures, we saw how a DFA can be easily implemented as a switch statement. That is exactly what you will do here. The only difference is that the transitions are sometimes nondeterministic. Your code should prefer the transition with the most input places. Among ties, you are free to choose however you wish. Remember to remove tokens from their old places (unless the transition actually puts them back) and remember to also put them in their new places.

Your testing should involve printing what the LPC thinks the system's state is, whenever it changes. This must be done in main(), not tc0(). This should print pictures like the following:

```
           [v]

 [12 ]     [^]
  |<=>|


           [ ]
```

The meaning is:
On the 2nd floor with doors closing. Inside, 1&3 are pressed. Outside, 3F's↓ and 2F's ↑are pressed.

```
           [v]

 [1  ]     [ ]
  |===|


           [ ]
```

The meaning is:
Now the doors are open. There's a problem: the guy from 2F expects to go up, but the elevator still needs to go down to 1. Ignore this problem.

```
           [v]

 [1 3]     [ ]
   >|<


           [ ]
```

The meaning is:
The doors are now closing, and the guy who just walked in has pressed 3.

```
           [v]

 [1 3]     [ ]
    |


           [ ]
```

The meaning is:
Now the doors are closed.

```
           [v]
 [1 3]
 ^^^^^     [ ]


           [ ]
```

The meaning is:
Now we are going up.

For your testing, you should print similar pictures in the ini file, for comparison that the two systems are agreeing on what is occurring. (But we saw in class that printf *might* cause problems in the ini file.)

The only remaining concern is initial conditions. The system starts with the elevator on the first floor with the doors closed. You have to therefore start, in main, by putting a token in this place. This also means that your tokens must be global variables.

As for the second elevator, that will come in the final homework.