C SC 230 – Summer 2014 – Assignment 3
Due Wednesday July 23 at 6:00 pm; Total Marks = 50

The goal of this assignment is to develop your competence at programming good, robust and elegant ARM code, plus downloading it to a board to get real hardware to cooperate with you. Finally you will be giving a demo and receive one-on-one feedback on your work, in the hope that we can provide you with a constructive learning experience and good suggestions for the future.

Do check the web pages after the posting of this assignment, as changes and/or clarifications will be added as necessary.

The assignment overall comprises the following main tasks:

1. Write a new program for the control of a small embedded system, following specifications provided here.
2. Learn how to fix problems in your program and improve it with new features.
3. Test the program using the simulator with the board plug-in as the output peripherals.
4. Learn how to change existing working code to make it compatible with the Embest board, download it and run it on the board.
5. Give a demo of your work in person, discuss your design choices and receive feedback.

1. The Application' Requirements – An Embedded System for an Elevator

Software requirements explain what a piece of software is expected to do, that is, its functional requirements. *Software specifications* explain how the functional requirements should be designed and implemented, and they are presented in separate sections below.

We all have used elevators and would probably claim to have a pretty good idea of how they work. Yet there is a lot to learn about the different ways an algorithm can be designed for such an operation. The problem here is restricted to a single elevator travelling from floor 1 to floor 4.

The simplest single elevator algorithm can be summarized as follows:

> *Continue traveling in the same direction while there are remaining requests in that same direction.*
> *If there are no further requests in that direction, then stop and become idle,*
> *or change direction if there are requests in the opposite direction.*

The elevator algorithm has found an application in computer operating systems as an algorithm for scheduling hard disk requests. Modern elevators use more complex heuristic algorithms to decide which request to service next. An introduction to these algorithms can be found in the "Elevator traffic handbook: theory and practice" by G.C. Barney.

Because we don't have an actual elevator to control, we will program just the part of the embedded system responsible for handling user input and displaying information, so that the program will actually be a "simulation" of the movement of an elevator. Many details about the mechanics of the operation (opening doors, slow acceleration when first starting to move, slowing down when reaching a floor, speed, etc.) are ignored. In 4th year you might take the courses in Mechatronics and expand your competence by programming such a system plus attaching it to actual mechanical and electronic components.

Since the elevator controller is not as easy as one might think, a lot of design and implementation decisions have already been made for you, including giving some initial code and the pseudo code for the algorithm. Make sure to ask questions to your manager (i.e. your instructor) when you wish to change some items - it might be a great idea and there is flexibility, or it might cause troublesome side effects.

2. The Inputs and Outputs in the Elevator Simulation Program

Elevators have buttons for users' requests, including a button to signal an emergency and enter a manual operation mode. Our simulation program will also have an extra input to ask to stop the simulation gracefully.

1.1. The Elevator's Controls

As shown in Figure 1, one can initially visualize the functional interface to the following controls and mechanisms:

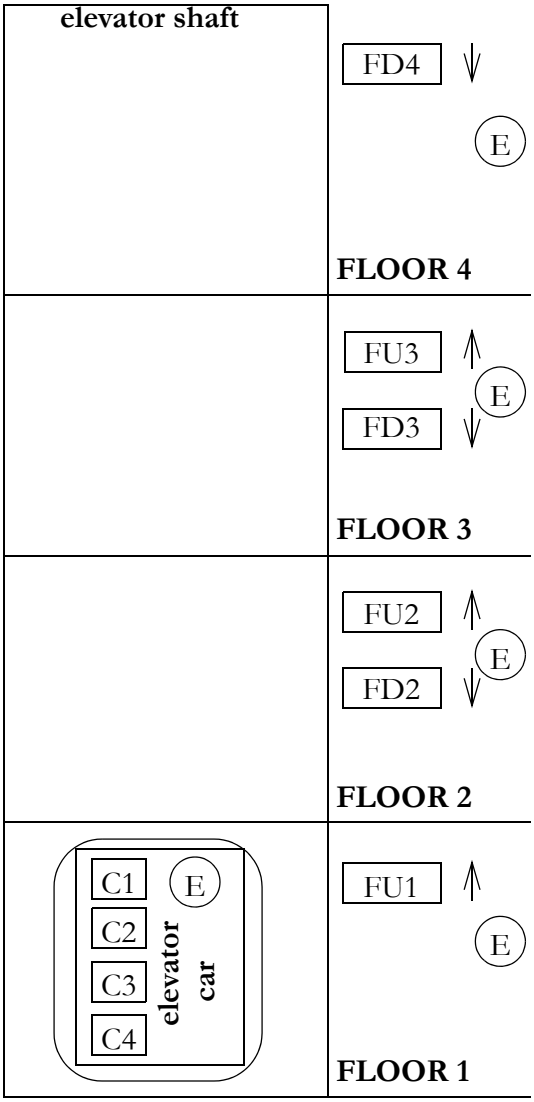
- There are four buttons inside the elevator car, labelled C1, C2, C3 and C4. They are pressed by users when inside the car to request a movement to a floor.
- There are two buttons at each floors 2 and 3 for users to request the elevator, labelled FU2, FU3, FD2 and FD3. Buttons FU2 and FU3 request the elevator to come to floor 2 or 3 with the intention of wanting to go to a higher floor. Similarly FD2 and FD3 request the elevator to come to floor 2 or 3 with the intention of wanting to go to a lower floor.
- There is one button at floor 1, labelled FU1, for users to request the elevator to come to floor 1 with the intention of wanting to go to a higher floor - the only possibility. Similarly there is one button at floor 4, labelled FD4, for users to request the elevator to come to floor 4 with the intention of wanting to go to a lower floor - the only possibility.
- There is an emergency button E available at each floor and inside the elevator car.

1.2. The Elevator's Display

The display of the simulated elevator shows the following information:

- the current program simulation time in seconds (constantly increasing every second),
- the current floor number,
- the current direction (moving up or down or idling).

Figure 1: Visualization of elevator input controls



The possibilities for inputs in this assignment are constrained by what is available on the Embest board and, thus, on the simulated version of the board too. The simulated board looks something like, and has the components shown in, Figure 2.

The inputs to be used are listed in Table 1. Any keys or buttons not listed below are *ignored* by the elevator program. For the purposes of explaining the program operation later, each key in the list has been given a name which corresponds to its intended function.

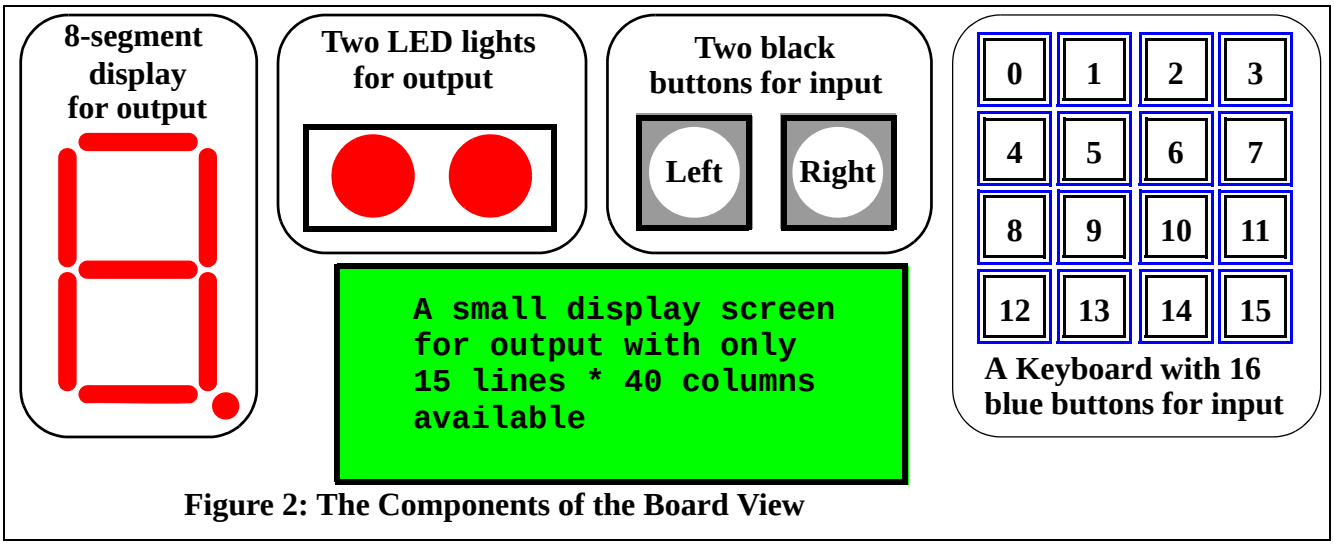


Figure 2: The Components of the Board View

Table 1: INPUTS

Left black button:	**Stop** – used to stop the simulation program.
Right black button:	**Emergency Stop** – used to stop the elevator immediately and quit the program manually.
Key 0: **C1**	button in elevator car for request to go to floor 1.
Key 1: **C2**	button in elevator car for request to go to floor 2.
Key 2: **C3**	button in elevator car for request to go to floor 3.
Key 3: **C4**	button in elevator car for request to go to floor 4.
Key 4: **FU1**	Request from floor 1, wanting possibly to move up.
Key 5: **FU2**	Request from floor 2, wanting possibly to move up.
Key 6: **FU3**	Request from floor 3, wanting possibly to move up.
Key 9: **FD2**	Request from floor 2, wanting possibly to move down.
Key 10: **FD3**	Request from floor 3, wanting possibly to move down.
Key 11: **FD4**	Request from floor 4, wanting possibly to move down.
Other keys:	**No effect.**

For the outputs, the LCD screen displays both static global program information and updates as the elevator moves. The static information include the identification of the programmer's name (top) and provide the user with a reminder of the buttons-to-key mapping (bottom). The dynamic information include the current floor number, the current simulation time and a pattern for the current direction. The floor number is increased or decreased as soon as the elevator reaches or passes by a floor. The simulation time is updated continuously every 1 second. The patterns are updated whenever a change in state (moving up or

down or idling) is encountered (see Figure 4). The template for the LCD screen is shown in Figure 3. The only difference for your program should be the first line. The code to change this display is provided to you. You are responsible for understanding it and call it correctly and appropriately.

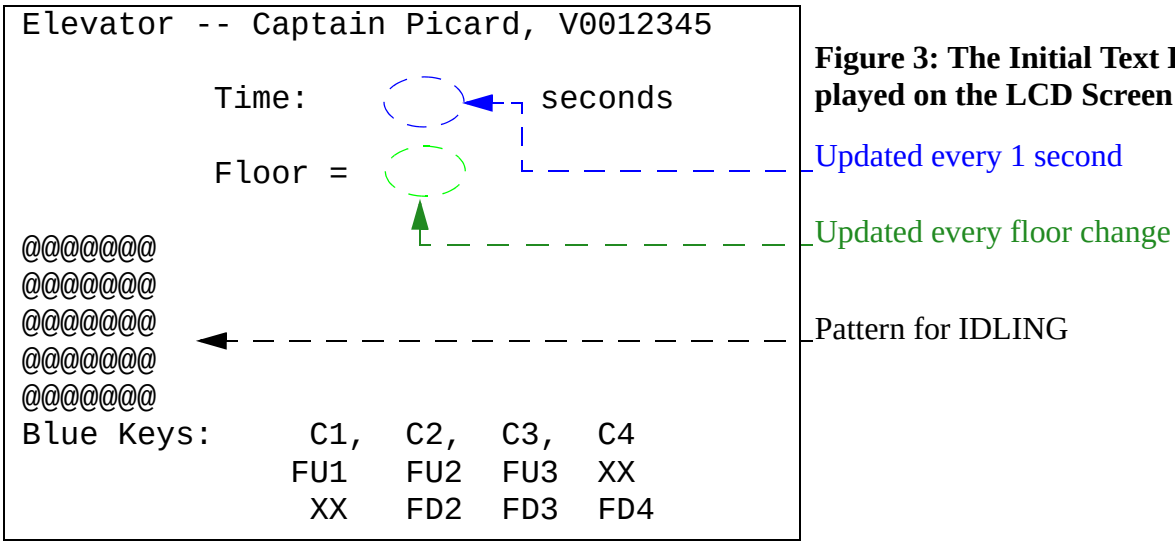


Figure 3: The Initial Text Displayed on the LCD Screen

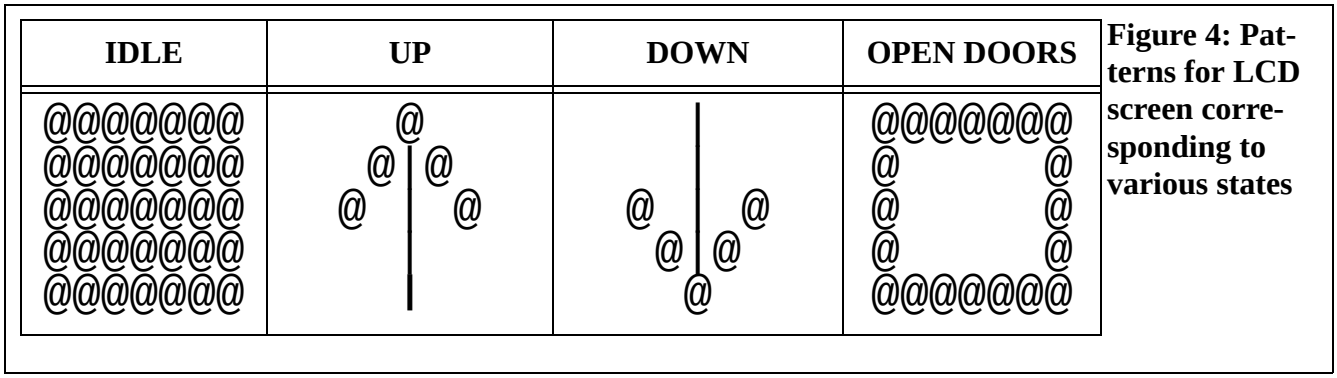


Figure 4: Patterns for LCD screen corresponding to various states

The 8-segment display shows the current floor number, synchronized with the LCD screen display. The "dot" is shown next to the floor number if the elevator is idling at that floor. The LEDs are both off when the elevator is idling. They are both on when the elevator has stopped at a floor and is opening the doors for a given time period. The left LED is on (and the right one off) when the elevator is moving up, while the right LED is on (and the left one off) when the elevator is moving down. Table 2 summarizes all the outputs.

Table 2: OUTPUTS

Action/State	8-segment display	LEDs	LCD Screen
Global identification in top line and buttons mapping in bottom lines remains unchanged			
IDLING	floor number *f* plus dot	both off	• IDLE pattern • floor number *f* • simulation time updates every second
MOVING UP	floor number *f* which updates to floor number *f* +1 when reaching it	left on	• UP pattern • floor number *f* then updated to *f* +1 • simulation time updates every second

Table 2: OUTPUTS (Continued)

Action/State	8-segment display	LEDs	LCD Screen
MOVING DOWN	floor number *f* which updates to floor number *f* -1 when reaching it	right on	• DOWN pattern • floor number *f* then updated to *f* -1 • simulation time updates every second
OPEN DOORS	floor number *f*	both on	• DOORS pattern • floor number *f* • simulation time updates every second
EMERGENCY	all segments blinking every 0.5 seconds (equivalent to the blinking number "8")	both blinking together every 0.5 seconds	• displays only the emergency message (provided)
SHUTDOWN	all off	both off	• displays simply the shutdown message (provided)

3. The Finite State Machine Diagram

The elevator is an example of a control system which is "*event driven*". Event-driven programming is a paradigm where the flow of the program is determined by sensor outputs, user actions or messages from other programs. It can be seen as a set of events, each divided into two portion: the first is the *detection* of an event and the second is the *handling* of an event. In embedded systems, this can be achieved either through a set of loops with "*polling"*, or with "*interrupts*"[1]. An event needs to be *captured* through some polling or interrupt programming and then it needs to be *examined* in order to decide what the appropriate action to follow should be. Every event should have a *deterministic* outcome. Conversely any action on the board or simulator which could cause an event must have a corresponding action attached to it, even if the action may be a "do nothing". In general, an *event* could be the push of any button or a timing-out signal and all possibilities are explained below in their context.

The *deterministic* outcome stated above is to be viewed in the context of the theory of computation, where a deterministic finite state machine is a finite state machine where for each state and possible input, there is one and only one defined transition to a next state, with an accompanying output. Thus at any point in time in the simulation of the elevator, the system is considered to be in a well defined state and every action corresponds to an event which may or may not change the state.

The description of the behaviour of the elevator is given as a set of finite state diagrams in Figure 5. The first task is to translate such diagrams into flowcharts corresponding more closely to the code in ARM. Thus the understanding now is to consider the elevator controller from the finite state machine point of view. Each circle is a "state" and the arrows show the transitions between states with the corresponding event as a label.

4. States and Actions - Specifications

At any time, the program driving the elevator is in one of several possible states. The behaviour of the program, and which inputs it accepts, is dependent on the current program states. *It should be emphasized*

1. Both concepts of polling and interrupts will be explained in the lectures.

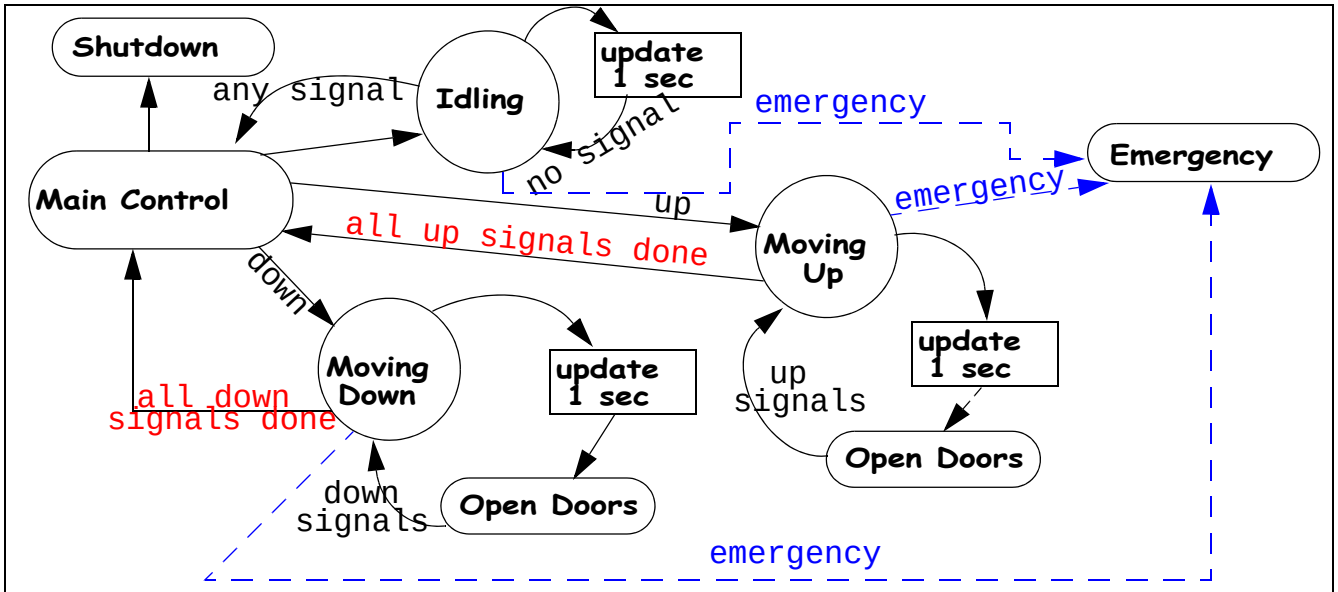


Figure 5: The Finite State Machine diagram for the elevator controller

that a state does not necessarily correspond to a subroutine! Such decisions about the program structure are discussed below. The states and actions are as follows.

Main Control. This is where all the decisions are made and where all the states above return to. The program here selects the next appropriate action depending on the current relationship of state and inputs. The displays change only after control moves to one of the states.

Idling. The elevator is not operating and it is stationary at a particular floor *f*, where *f* =1 when the program starts. The user can: push an elevator car button, C1 to C4 (assuming a person is in the elevator and presses a button); push any F#DW or F#UP buttons from the various floors; press the left black button to stop the simulation program; press the right black button to signal an emergency. Thus the program is continuously polling for inputs. When a button press is captured, the program returns to Main Control who will decide what to do next.

Moving Up. The elevator is moving up. It takes X seconds to move from one floor to the next floor. After moving up for 1 floor, the elevator may stop there and open the doors, or it may continue up, or change to moving down. In the X seconds time period while the elevator is moving, the user can press any button and the events are captured. Thus the program is continuously polling for inputs. The floor number is updated when the elevator reaches the next floor, stopping or not, after X seconds. The program may return to Main Control or stay in the Moving Up. The program *never* goes from the Moving Up state directly to the Moving Down state.

Moving Down. The elevator is moving down. It takes X seconds to move from one floor to the next floor. After moving down for 1 floor, the elevator may stop there and open the doors, or it may continue down, or change to moving up. In the X seconds time period while the elevator is moving, the user can press any button and the events are captured. Thus the program continues polling for inputs. The floor number is updated when the elevator reaches the next floor, stopping or not, after X seconds. The program may return to Main Control or stay in the Moving Down. The program *never* goes from the Moving Down state directly to the Moving Up state.

Open Doors. The elevator is moving up or down and has reached a floor where some signal is present. *Any* signal for that floor implies that a request was made and that the elevator should stop and open the

doors. The elevator must stop for Y seconds to simulate the doors opening. During these Y seconds the user can press any button and the events are captured. Thus the program continues polling for inputs.

Emergency. The user has pressed the right black button from any of the previous states/action during polling. The only way to end the program is to use the manual Stop command in ARMSim# or power off the board.

Shutdown. When the user presses the left black button, the program gracefully ends.

Table 3: Program States and Events Affecting the Program

State	Event	Action
Main Control	*Entry from start of program*	Initialize displays. Go to Idling.
	Return from Idling	A button has been pressed, decide on action (see next entries).
	Stop button	Go to Shutdown state. Clear displays. Programs ends.
	Blue button	Choose action (see section on Control Choices).
	Return from Moving Up	There are no more events from above the current floor (finished going up). Follow the pseudo code (below) and check if there are events for lower floors. If so, go to Moving Down, else go to Idling.
	Return from Moving Down	There are no more events from below the current floor (finished going down). Follow the pseudo code (below) and check if there are events for higher floors. If so, go to Moving Up, else go to Idling.
Idling	*Entry from Main Control*	Update displays.
	1 seconds has elapsed since last time check	The simulation time on the LCD screen is updated by 1 second. No other display changes.
	Stop key	Return to main control.
	Emergency key	Emergency exit.
	Blue button	Update global arrays and return to main control.

Table 3: Program States and Events Affecting the Program (Continued)

State	Event	Action
Moving Up	*Entry from Main Control*	Update displays.
	1 second has elapsed since last time check	The simulation time on the LCD screen is updated by 1 second. No other display changes.
	X seconds have elapsed (moved from floor *f* to *f*+1)	Update floor number. Event for "Checking Signals" is triggered.
	Stop key	Return to main control.
	Emergency key	Emergency exit.
	Blue button	The global array for signals is updated.
	"Checking Signals" event triggered	Check the global array. If there are requests for the current floor, then trigger Opening Doors. Then check if there are more move requests going up: if so, restart from state entry. Else return to main control.
Moving Down	*Entry from Main Control*	Update displays.
	1 second has elapsed since last time check	The simulation time on the LCD screen is updated by 1 second. No other display changes.
	X seconds have elapsed (moved from floor *f* to *f*-1)	Update floor number. Event for "Checking Signals" is triggered.
	Stop key	Return to main control.
	Emergency key	Emergency exit.
	Blue button	The global array for signals is updated.
	"Checking Signals" event triggered	Check the global array. If there are requests for the current floor, then trigger Opening Doors. Then check if there are more move requests going down: if so, restart from state entry. Else return to main control.
Opening Doors	*Entry from Moving up or down*	Update displays.
	1 second has elapsed since last time check	The simulation time on the LCD screen is updated by 1 second. No other display changes.
	Y seconds have elapsed	Return to caller.
	Any key	The global array for signals is updated.

5. Program Structure: the final detailed specifications

Note this carefully! The program states do *not* correspond necessarily to subroutines. They may correspond to subroutines or they may correspond to labelled blocks of assembler statements all contained

inside the same subroutine. Your code will use conditional branch instructions to go to labels *while remaining inside the same subroutine*, OR your code may have to return to the calling routine before entering a different state, OR you may transfer control by calling another function. You cannot/must not jump between subroutines.

5.1. The outputs

While writing code for the output settings is not difficult, it is often time consuming, especially regarding the LCD screen. Some code has been prepared for you to be used to make your overall project easier. The corresponding functions are listed in Table 4. Your work includes understanding their code and using them correctly. Check the parameters carefully. You are free to change them, if you wish., and be creative, but be mindful of the time needed.

Table 4: The functions already given for Output displays

`UpdateTime`	Displays the updated value of the current simulation time on screen and updates its variable.
`UpdateFloor`	Displays the value of the current floor on the LCD screen and in the 8-segment.
`UpdateDirectionScreen`	Displays the pattern for the elevator direction on the appropriate 5 lines on the LCD screen.
`Display8Segment`	Displays the number 0-9 on the 8-segment, with/without the point.
`ExitClear`	Clears the board and displays the exit message.
`Initdraw`	Draws the initial state of all displays.
`Wait`	Wait for some interval of time without polling.

5.2. Main Control

The main routine itself is the one which should contain the controls for the whole process - it is here where the decisions are made to switch between states. Other functions return here before any decision is made for a change in direction for the movement of the elevator. The initial code given to you implements part of this portion of the program and its complete pseudo code is given in Figure 6.

The only complex part is when a blue button has been pressed, as each of the 10 possible cases must be considered, given the current floor position, plus making sure that nothing is done when an invalid button is pressed. Further help and hints for this logic is given in the subsection below labelled “Control choices”, where in Table 5 it is shown that the 10 possibilities for the blue buttons events can be distilled into just 4 cases for actions (a lot of design thinking has been done on your behalf, take the time to understand it).

If the analysis implies that the elevator should move up, control passes there. On return from the up direction, it is guaranteed that the elevator is at floor f and that no request from floors above f exists, or f=4. Signals in the global array must then be checked to find if there is any request below the current floor. If none, the program goes back to the top of Main Control (and eventually to Idling), else it changes direction to going down. Read now carefully the pseudo code in Figure 6 and understand the logic, before starting to consider the implementation details of the data structures.

5.3. The global arrays for the signals: ButtonsCar, FloorUp, FloorDw

At this point it is useful to examine the global data structure containing the signals set by the blue buttons. Normally global variables are frowned upon, but in control systems or graphic applications some “state” variables available everywhere are essential. However it is important in program design to try and mini-

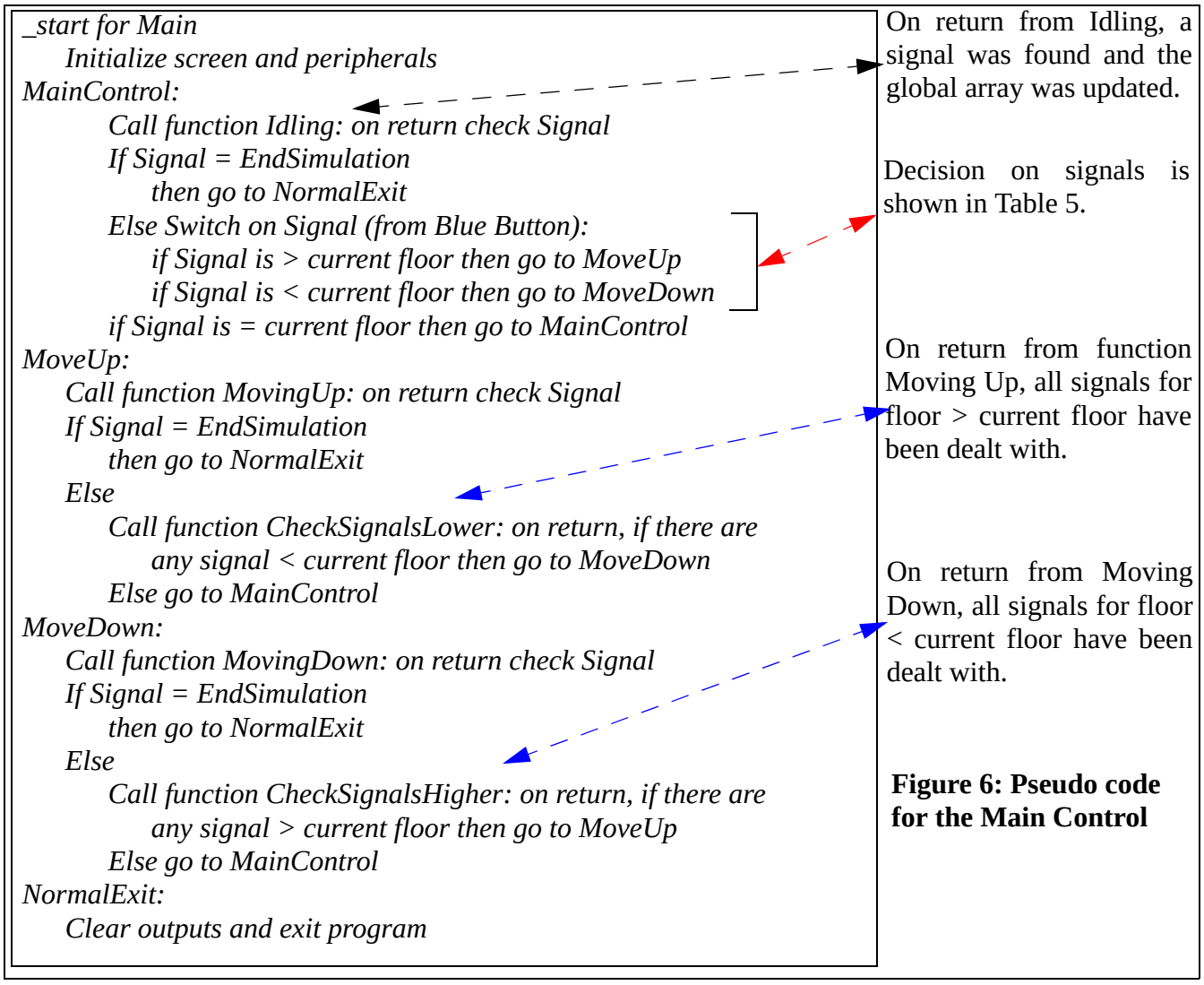


Figure 6: Pseudo code for the Main Control

mize their occurrence (tricky side effects can be hard to handle). In this program the following three arrays are declared:

```
ButtonsCar:.word    0,0,0,0   @ 4 signals for 4 car buttons
FloorUp:  .word     0,0,0,0   @ Up signals - floor 1,2,3, (4 unused)
FloorDw:  .word     0,0,0,0   @ Down signals - floor 2,3,4, (1 unused)
```

Here `ButtonsCar[0]`, `ButtonsCar[1]`, `ButtonsCar[2]` and `ButtonsCar[3]` correspond to the 4 buttons C1 to C4 inside the elevator. The entries for `FloorUp[0]`, `FloorUp[1]` and `FloorUp[2]` correspond to FU1, FU2 and FU3, while `FloorUp[3]` remains 0 at all times. Similarly `FloorDw[1]`, `FloorDw[2]` and `FloorDw[3]` correspond to FD2, FD3 and FD4, while `FloorDw[0]` remains 0 at all times. Whenever a blue button signal is captured, it is stored into this global array which can in turn be examined when necessary. To help you, the function `SetButtonsArray` is given to you with the following interface:

```
@ === Void SetButtonsArray(r0:blue button)
@   Inputs:r0=blue button request
@   Results:  none
@   Description:
@         Set the appropriate entry in the global array of requests
@         if it is a valid button
```

The global arrays are used for checking whether there are signals above or below or at a given floor. One needs at least the following functions. If the functions are given to you in the initial code, analyze and understand them carefully and learn to use them correctly. If they are not given to you, it is important to maintain their specified interface and write the code according to the commentary.

```
@ === Void ClearButtonsArray(r3: &floor)
@   Inputs:r3 = &floor
@   Results:  none
@   Description:
@         Clear the entries in the global array for a given floor
@ === CheckSignalsHigher(r3: &floor)
@   Inputs:r3 = & floor
@   Results:  R0 = 0 (no to higher signal); 1 (yes to higher signal)
@   Description:
@         Check global arrays for any signals at floor>given floor
@ === CheckSignalsLower(r3: &floor)
@   Inputs:r3 = & floor
@   Results:  R0 = 0 (no to lower signal); 1 (yes to lower signal)
@   Description:
@         Check global arrays for any signals at floor<given floor
@ === CheckFloorDoors(r3: &floor)
@   Inputs:r3 = & floor
@   Results:  R0 = 0 (no to signal at floor); 1 (yes to signal at
floor)
@   Description:
@         Check global arrays for any signals at current floor
```

5.4. Idling subroutine

The Idling function loops continuously and returns to Main Control only when a signal has been captured. Every second it also updates the simulation time on the screen. Figure 7 shows the overall pseudo code. Most of the code for Idling is given to you. *It is absolutely crucial that you understand how the polling is done and how the timing is precisely calculated*, as it will serve as your model for other parts of the code. Read carefully the section below on timing notes.

Idling:
    Set IDLE pattern on screen and peripherals
IdlePoll:
    Get Time T0
    RepeatUntilEvent:
        Poll Right Black Button - If pressed, exit directly to "EmergencyStop function"
        Poll Left Black Button - If pressed, return to Main Control with "EndSimulation"
        Poll Blue Buttons - If pressed, set global array and return to Main Control with the button number for the event
        Get Time T1
        If (T1 - T0) > 1 second, then go to UpdateSimulTime
        Else go back to RepeatUntilEvent
UpdateSimulTime:
    update time on screen and in variable
    Go back IdlePoll

Figure 7: Pseudo code for Idling

5.5. Moving Up subroutine

The program enters "Moving Up" from the main control and returns to main control only when all requests to move upwards have been satisfied. Moving Up simulates moving from floor f_i to floor f_{i+1} as many times as requested, taking Y seconds for the movement between each floor, and it polls continuously for more possible requests during this time period Y. Moreover, every second it also updates the simulation time on the screen. After reaching floor f_{i+1}, the floor outputs are updated and the arrays are checked to see whether a signal at this floor level is present. If so, the Open Doors action is triggered. Then a check is done on whether more signals above this new current floor are present. If so, the elevator continues moving upwards. Else it returns to the main control. The pseudo code is given in Figure 8 and the corresponding flowchart is given in Figure 9. The time Y to move between floors should be in the range of 3 to 5 seconds, your choice. While the longer delay of 5 seconds makes the program a bit tedious to watch, it helps tremendously in testing as one has the time to observe all details (and it teaches patience). Use an EQU statement for setting the time period.

```
Moving Up:
    Set LEDs, set LCD, refresh floor # on 8-segment without dot
TopMovingUp: (main loop)
    Move for 1 floor in X seconds ==> call WaitAndPoll
    On return: if R0=EndSimulation go to DoneMovingUp
    Update floor number on LCD, 8 seg., and in variable
    Call CheckFloorDoors to see if there is a signal at this floor
    On return: IF R0=no go to CheckTop
               ELSE call OpenDoors (which will call WaitAndPoll for X secs)
               On return: if R0=EndSimulation go to DoneMovingUp
                          else update LCD, LEDs
CheckTop:
    IF current floor = top floor go to DoneMovingUp
    ELSE call CheckSignalsHigher
    On return: If R0 = yes
        update direction on screen and go to TopMovingUp
DoneMovingUp:
    return to caller with appropriate R0 value
```

Figure 8: Pseudo code for Moving Up

5.6. Moving Down

The actions associated with moving downwards mirror the actions for moving upwards, with the appropriate changes.

5.7. Opening the Doors

The doors open at a floor in two cases: (a) when the elevator reaches the destination floor from a signal; (b) whenever it passes by a floor and, by checking the global arrays, there is a signal in one of the locations corresponding to that floor. For example, the elevator may have started at floor 1 and button C4 has been pressed. While moving between floor 2 and floor 3, somebody also pushes the FU3 button. When the elevator arrives at floor 2 it simply passes by and updates the display. When the elevator passes by floor 3, it detects a signal in the global array, stops at floor 3, opens the doors, updates the displays and clears the arrays for that floor. Then the elevator proceeds to floor 4 where again it will open the doors and clear the arrays for that floor.

The doors should stay open for a short time period in seconds, in the range of 3 to 5, your choice. While the longer delay seconds makes the program a bit tedious to watch, it helps tremendously in testing. Use an EQU statement for setting the delay for opening doors so it can easily be changed.

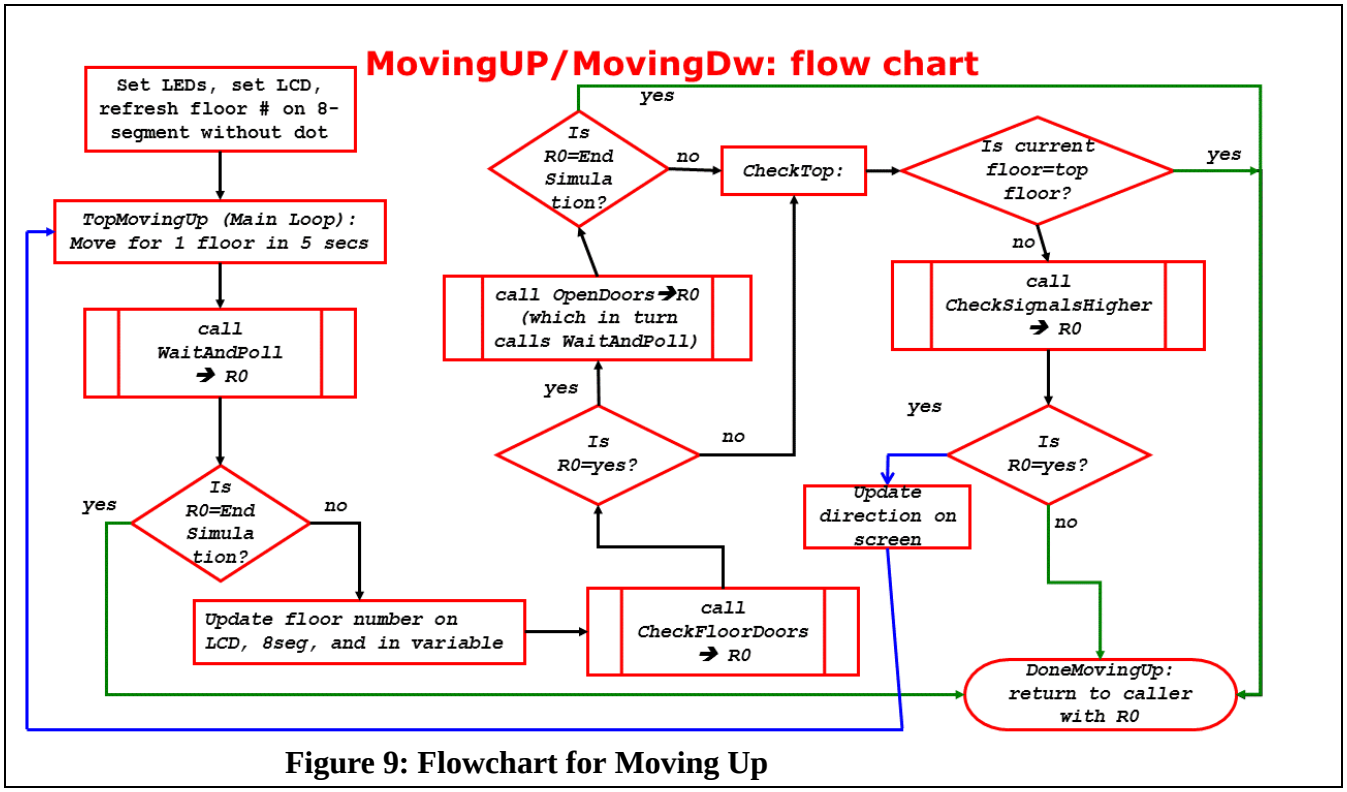


Figure 9: Flowchart for Moving Up

5.8. Control choices (suggestions) for Main Control

In the pseudo code for the main control one needs to analyze which blue button has been pressed in order to associate the appropriate action. Since there are 10 valid buttons, the code for the corresponding "switch" structure is not exactly elegant and, moreover, at first pass, it may appear that there are indeed 10 distinct actions, plus the "no action" for invalid buttons. However on close scrutiny one should be able to discern that the 10 cases can be grouped together into four cases as they are associated to equivalent set of actions. Table 5 shows the logic and should give ideas on simplifying the coding for this portion of the control.

Table 5: Four major action cases distilled from 10 possible blue buttons events

Valid Button	Associated actions with *f* denoting current floor number	Case	
Key 0 = C1	(f = 1)→do nothing; (f > 1) → Move Down	CF1UP	X1
Key 1 = C2	(f = 2)→do nothing; (f < 2) → Move Up; (f > 2) → Move Down	CF2UPDW	X2
Key 2 = C3	(f = 3)→do nothing; (f < 3) → Move Up; (f > 3) → Move Down	CF3UPDW	X3
Key 3 = C4	(f = 4)→do nothing; (f < 4) → Move Up	CF4DW	X4
Key 4 = FU1	(f = 1)→do nothing; (f > 1) → Move Down	CF1UP	X1
Key 5= FU2	(f = 2)→do nothing; (f < 2) → Move Up; (f > 2) → Move Down	CF2UPDW	X2
Key 6 = FU3	(f = 3)→do nothing; (f < 3) → Move Up; (f > 3) → Move Down	CF3UPDW	X3
Key 9 = FD2	(f = 2)→do nothing; (f < 2) → Move Up; (f > 2) → Move Down	CF2UPDW	X2
Key 10 = FD3	(f = 3)→do nothing; (f < 3) → Move Up; (f > 3) → Move Down	CF3UPDW	X3
Key 11 = FD4	(f = 4)→do nothing; (f < 4) → Move Up	CF4DW	X4

6. Timing, timing and more timing!

In any programming task, once you have to deal with timing issues, things get complex and tricky to implement and debug. On the topic of debugging, avoid placing breakpoints inside any timing loops, as the number of clock ticks being captured keeps updating while the simulation stops, creating problems. Place breakpoints just before or just after a timing loop to observe the state of registers.

There are two main complex item about timing with which a programmer has to deal: (1) how to solve various technical issues with the timer itself, depending on the processor used and the hardware platform on which to execute eventually, and (2) when and how to check for timing. The first issue is a hardware/ software interface one and the software needs to adjust for the hardware specifications. The second issue is an algorithmic one to be tackled through the correct logic to be develop in software.

6.1. Technical note on the timer

First of all the ARM processor (unlike other processors) does not have a built-in timer which can be used directly through instructions available in its ISA. Thus any timing must come from a separate hardware device and, in this case, it is accessed through software interrupts routines available both in the ARM-Sim# simulator and on the Embest board.

The timer in ARMSim# is the most general and it is implemented using a 32-bit quantity. The current time (as number of ticks) is accessed by using the SWI instruction with operand 0x6d (the corresponding EQU is set to be SWI_GetTicks). It returns in R0 the number of ticks in milliseconds. The timer on the Embest board uses only a 15-bit quantity and this can cause a problem with rollover. How long does each timer take before rolling over? How can one make a program work with both?

Assume you poll the time at a starting point T1 and then later at point T2, and you need to check whether a certain amount of time has passed, e.g. 2 seconds. Ideally you simply compute T2-T1 and compare whether this result is less than 2 seconds or not. The range in ARMSim# with a 32-bit timer is between 0 and 2^{32} -1 = 4,294,967,295. As milliseconds, this gives a range of about 71,582 minutes, which is normally enough to ensure that one can keep checking the intervals T2-T1 without T2 ever going out of range in a single program execution (at least in this course).

The range in the Embest board with a 15-bit timer is between 0 and 2^{15}-1 = 32,767, giving a range of only 32 seconds. When checking the interval T2-T1, there is no problem as long as T2>T1 and T2<32,767. However it can happen that T1 is obtained close to the top of the range (i.e. close to 2^{15}-1) and T2 subsequently has a value after the rollover, thus T2<T1. It is not enough to flip the sign as the following examples show. This is illustrated graphically in Figure 10.

Let T1 = 1,000 and T2 = 15,000. Then T2-T1 = 14,000 gives the correct answer for the interval. Subsequently let T1= 30,000 and the later T2 = 2,000 (after the timer has rolled over). If one simply calculates T2-T1 = -28,000 or even tries to get its absolute value, the answer is incorrect. The value for the interval should be: (32,767 - T1) + T2, that is, 32,767 -30,000 + 2,000 = 4,767, which represents the correct number of ticks which passed between T1 and T2.

Two things need to be done for correct programming. If executing only in ARMSim# (or on a board with a 32-bit timer), one may ignore the problem of rollover altogether. It does not lead to robust software as it could very well be the case that a simulation program is left to run for a long time to give a significant statistical view of the process. Thus, programs should include a solution to a rollover issue at all times, even if it appears unlikely.

Secondly one must design with compatibility of the program with the hardware platform in mind, or, in general, how to make a program compatible with any given platform. Thus the timing value obtained in 32 bits in ARMSim# should be "*masked*" to be only a 15 bit quantity, so that the code can work both in the simulator and on this particular board. This also implies that the testing for the interval must follow a more elaborate algorithm than a simple subtraction. The pseudo-code in Figure 11 should help and it is implemented in some of the code given to you.

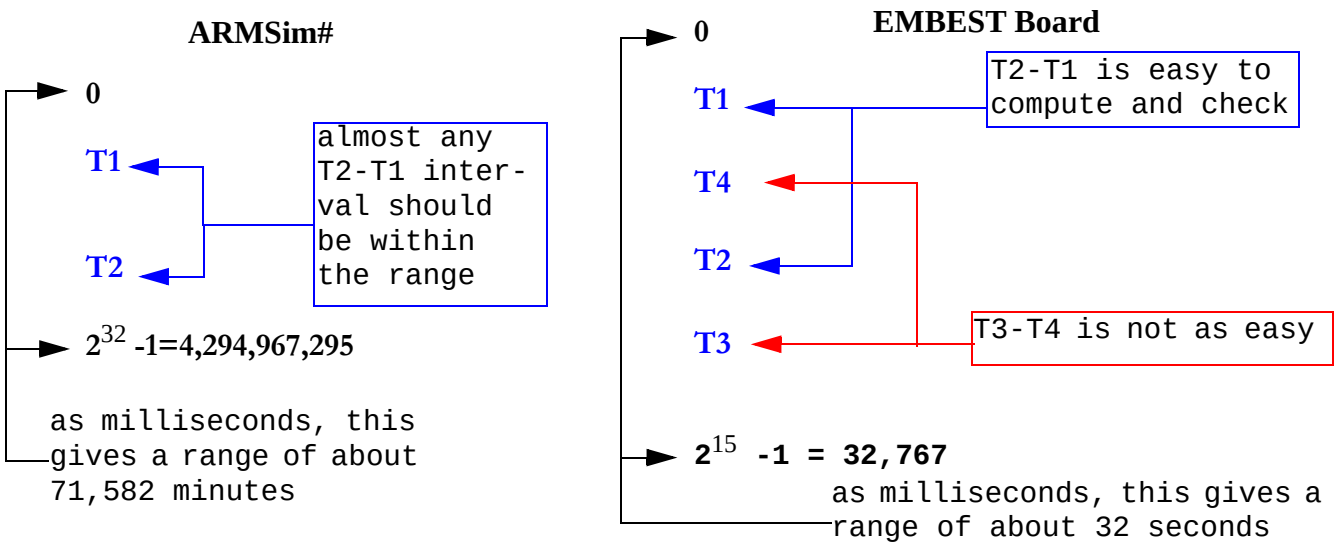


Figure 10: Timing intervals on ARMSim# and on the Embest Board

```
PSEUDO CODE FOR SIMPLE DELAY: void Wait (Delay)
Check that an INTERVAL has passed between actions
     T1 = get time with swi 0x6d
     T1 = adjust the time with a 15-bit with mask
     Repeat:
          check: has enough time passed?
               T2 = get time with swi 0x6d
               T2 = adjust the time with a 15-bit with mask
          IF T2 >= T1 then TIME = T2 - T1
          ELSE TIME = 32,767 – T1 + T2
          If TIME < INTERVAL go to Repeat
```

Figure 11: Pseudo code for a simple delay cycle

While the actual code to implement the masking and testing may have been given to you, you are *absolutely responsible* to understand it fully, as you *will* be quizzed on it during the demo or on the final exam.

6.2. Algorithmic and software issue for the timing.

How to check for a given delay? How to cause a program to "wait" for X seconds? How to cause a program to "wait" for X seconds while still polling for events? Some solutions are shown in the code already given to you, make sure you analyze it and understand it. Here is some further explanation.

The simplest timing process is the need to wait for a given period without anything else being done. For example, if all is happening is that LED lights, 8-segment and screen are set to a certain output for 5 seconds (for example, just before exiting the program altogether), then one needs a simple programmed delay for 5 seconds, while nothing else happens. This can be a straightforward "`Wait`" function, with one input parameter, namely the time delay needed, and no output. The code for this routine is probably part of your template and its pseudo code is illustrated in Figure 11.

The more complex timing process is when a program needs to go through a certain interval of time while still doing some actions (e.g. polling for events). For example, in this assignment, this is the case when going between floors and yet needing to check for any buttons being pressed during the movement of the elevator. The strategy is also useful when a program needs to go through a certain interval of time while still doing some actions and then implement yet another different action at the end of such an interval. For example, in this assignment, this is the case when during each interval of 1 second one continues to poll

for signals, while at the end of each interval one updates the simulation time on the screen. Graphically this is shown in Figure 12 and can be seen in some of the code given to you. One cannot simply program

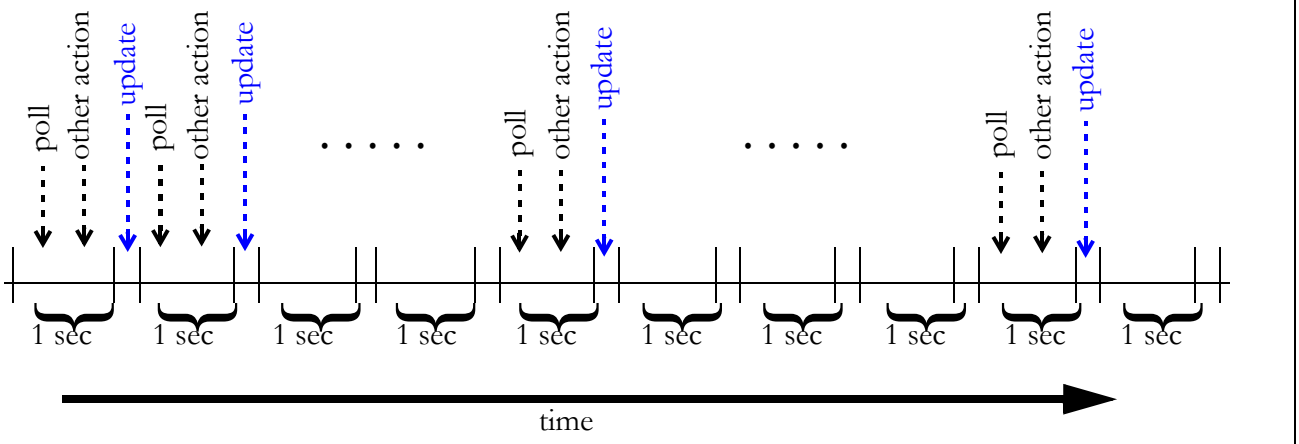


Figure 12: The time intervals with polling, actions and updates

a delay of X seconds and then check whether any button has been pressed, as it would be an unreasonable delay for a user who pressed a button right away. In this interval one must continue to poll for user buttons while also monitoring everything else.

The simple "`Wait`" routine given to you is not sufficient. One needs to develop a "`WaitAndPoll`" function, with one input parameter (the maximum time delay needed), and one output stating why it is returning: either because the maximum delay has taken place or because an interrupt has been captured. The pseudo code in Figure 13 summarizes the required logic.

In summary, make sure you understand what the strategy is before you do any programming including any timing. It can be complex and it can become quite tricky to debug. In particular, avoid placing breakpoints or stepping through the code instruction by instruction when timing is involved. Check the state of registers before or after timing loops.

```
CHANGE DELAY WITH EVENT POLLING: int WaitAndPoll (Delay)
     T1 = get time with swi 0x6d
     T1 = adjust the time with a 15-bit with mask
     Repeat:
          poll for events
          IF events happened, break and return with int = event number
          ELSE check delay
               T2 = get time with swi 0x6d
               T2 = adjust the time with a 15-bit with mask
               IF T2 >= T1 then TIME = T2 - T1
               ELSE TIME = 32,767 – T1 + T2
               IF TIME < INTERVAL go to Repeat
               ELSE break and return with code = 0  CODE
```

Figure 13: Pseudo code for a delay cycle with polling

To make things even more complex here, when an event is captured we don't really necessarily want to return from the `WaitAndPoll` subroutine. For example, while moving between two floor (it may take 5-10 seconds), one needs the delay while polling for other signals. If another blue button press is detected, the time spent moving between floor remains the same - the action associated at this point is not to stop an action. Instead it is to update the global array with the new signal so that it can be checked after returning from `WaitAndPoll`. If a Stop Simulation button is detected, you can decide whether to shorten the `WaitAndPoll` routine and return, or wait out the delay of moving between floors. If an

Emergency button is pressed, instead you should exit immediately! In any case, the pseudo code above needs some customization (given at the tutorial).

Try and write the pseudo code yourself for this task before the upcoming tutorial when a few more hints will be given. It is a great exercise for your learning strategy: if you find at the tutorial that you have solved the task perfectly, that's great - on your own you will have increased your understanding and everything else will be easier. If you have some confused logic, then the tutorial will help you sort it.

7. More on program design and structure

You are free to design your program as you choose. However it may be the case that you do not have much experience in control systems (as the elevator controller is) and in modular program design. When using assembly language this can lead to complex logic and unstructured code. Below are a few (strong) suggestions for functions and their interface which you should find useful. Most of all, remember that the only global access is to the global arrays for signals and that all other information should be passed through parameters. For example, most functions need to know the current floor number. It is tempting to make it a global variable since the floor number needs to be updated in many places, but it is not a good decision. Pass a pointer to it (that is, its address) to each function that needs its value or needs to update it in memory as well as in a register.
A possible program might contain the functions listed in Table 6. Note that some of them are given to you as help (learn the code!), others are given partially implemented and others are not given at all (and you may choose to design differently).

Table 6: Functions and interfaces in program design

Function Name	Parameters	Description
`Main/Start`	---	Main control of the program where decisions on movement are made. See the pseudo code in Figure 6.
`Idling`	*Inputs*: R3 = & floor; R4 = &simulation time. *Output*: R0 = Signal from button	Poll buttons continuously until an event happens. Every 1 second update the simulation time on screen.
`SetButtonsArray`	*Inputs*: R0 = blue button request *Output*: none	Set the flag in the global array of requests.
`ClearButtonsArray`	*Inputs*: R3 = & floor *Output*: none	Clear the signals in the global array for the current floor.
`CheckSignalsHigher`	*Inputs*: R3 = & floor *Output*: R0 = yes/no to signal	Check global arrays for any signals above current floor.
`CheckSignalsLower`	*Inputs*: R3 = & floor *Output*: R0 = yes/no to signal	Check global arrays for any signals below current floor.
`CheckFloorDoors`	*Inputs*: R3 = & floor *Output*: R0 = yes/no to signal	Check global arrays for any signals at current floor.
`MovingUp`	*Inputs*: R3 = & floor; R4 = & simulation time. *Output*: R0 = Signal for EndSimulation or EmergencyStop or 0.	Move up each floor until all UP requests done. Side effects: if events captured, global arrays updated.

Table 6: Functions and interfaces in program design (Continued)

Function Name	Parameters	Description
`MovingDown`	*Inputs*: R3 = & floor; R4 = & simulation time. *Output*: R0 = Signal for EndSimulation or EmergencyStop or 0.	Move down each floor until all DOWN requests done. Side effects: if events captured, global arrays updated.
`OpenDoors`	*Inputs*: R3 = & floor; R4 = & simulation time. *Output*: R0 = Signal for EndSimulation or EmergencyStop or 0.	Set the outputs for doors open, stay for some seconds while polling. Side effects: clears arrays at floor number.
`WaitAndPoll`	*Inputs*: R4 = & simulation time; R5 = time in seconds to wait while polling. *Output*: R0 = Signal for EndSimulation or 0.	Wait for some seconds while continuing to poll. Every 1 second update simulation time. Side effects: if events captured, global arrays updated.
`UpdateTime`	*Inputs*: R4 = & simulated time *Output*: none	Display the updated value of the current simulation time and update its variable.
`UpdateFloor`	*Inputs*: R3 = & floor number; R5 = 1(idle) or 0(moving) *Output*: none	Display the value of the current floor on the LCD screen and in the 8-segment.
`UpdateDirectionScreen`	*Inputs*: R5 = current direction of elevator movement *Output*: none	Display the pattern for the elevator direction on the LCD screen
`Initdraw`	*Inputs*: R3 = & floor *Output*: none	Draw the initial state of all outputs.
`EmergencyState`	*Inputs*: none *Output*: none	Set the configuration for all outputs.
`ExitClear`	*Inputs*: none *Output*: none	Clear the board and display the last message
`Wait`	*Inputs*: R10 = delay in milliseconds *Output*: none	Wait for some milliseconds doing nothing.
`Display8Segment`	*Inputs*: R0=number to display; R1=point or no point *Output*: none	Display the number 0-9 on the 8-segment, with/without the point.

7.1. Caveat and initial help

You are supplied with code that implements some of the trickier functions. There is no guarantee that the code is free of bugs. Any bugs are your responsibility to track down and fix. (You do not have to use the supplied functions.) However it is imperative that you fully understand the code you choose to use! You can expect questions on the whole code both at the demo and on the final exam.

8. Developing the Program

Unless you are a super programmer and don't need any advice, you *should* follow the stages of development shown in the separate document named "Development and Testing", which has been written to help you through the steps. This is the *only way* to ensure that you get partial credit for a program which is not perfect. The table decomposes the problem into a sequence of tasks. After completing any stage, you will have a working program. It won't perform all the actions required of the final program, but it will perform some of them. Moreover it is strongly suggested that you think in even smaller steps by dividing up each stage into components and test each one separately. The testing sequence which is to be used at the demo session is also described in the document and its steps are linked to the development stages.

8.1. Flowcharts (and pseudo-code)

Look at the 3 items given to you for the MovingUp function:
(a) an english description, precise enough;
(b) the pseudo code;
(c) a flowchart.
All three are an integral part of system design and development. Personally, your instructor needs all three of them to produce good programs. The flowchart is given there to show you an example of what you are supposed to be able to produce as well and in what style, so that you can end up coding directly from it. I strongly suggest that you take the example for MovingUp and use it as a template for all the other functions, no matter how trivial. Do the pseudo code an flowcharts before you start coding!

9. Submissions (plus paper copy!)

The program source code should be submitted electronically via the Assignment page in Connex. IMPORTANT NOTE and change from previous assignments: A paper copy of your assignment must be deposited in the ECS course box.

Important! *Make sure that your name and ID appears in a comment at the beginning of the program. This applies both to the electronic submission and to the paper printouts.*

10. What is going to happen at the demo and how?

- A sign-up sheet will be made available through Connex. Lab session times are included.
- You will need to make two independent appointments: one for the execution demo and one for the code evaluation. The latter appointment should come later in time from the former.
- At your first appointment, the demo on which will take place in the lab, your code will be tested on the ARMSim# simulator by an evaluator who will discuss with you these results. If there are any problems, a second test run may be rescheduled for another time. If you pass stage 9, you will show that you are able to build a project for the board and execute it.
- At yur second appointment, the code evaluation which will take place either in the lab or in an office, your source code will be examined on the printout by your instructor together with you. You will be asked about your design choices and expectd to give explanations of your sequence of instructions.
- You will have the chance to ask questions for any clarifications or feedback you wish to have.
- This is supposed to be a learning experience for you, plus a lot of fun to show off your good work!