C SC 230 – Spring 2013 – Assignment 3 - Total Marks = 100

(1) Flowchart is due March 6th in the lecture; (2) Code is due Monday, March 25 by 11:55pm by electronic submission

The goal of this assignment is to develop your competence at writing 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 demonstration and receive one-on-one feedback on your work, in the hope that we can provide you with a constructive learning experience and give you some good suggestions for the future.

It is imperative to check the web pages after the posting of this assignment, as more details and/or clarifications will be added as necessary to the specifications in this document.

The assignment overall comprises the following main tasks:

1. Write a new program for the control of a small embedded system, following the specifications. Test the program using the simulator with a graphics plug-in for the output peripherals.
2. Download the program to run on the Embest board with the real physical peripherals.
3. Give a demo of your work in person, discuss your design choices and receive feedback.
4. Prepare good documentation for your design and implementation decisions and for your code.

1. The Story Line

You are to write an ARM assembly language program that simulates the operation of a traffic light controller for an intersection, where the movements of both cars and pedestrians are controlled by lights and buttons.

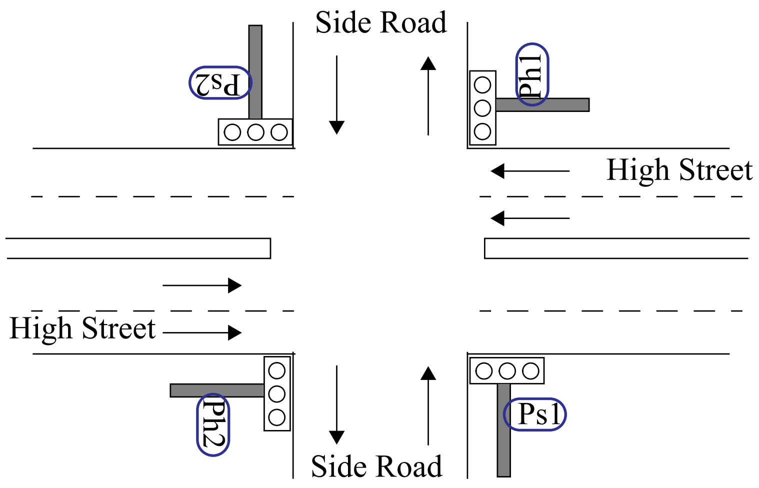


Figure 1: Diagram of the Road Intersection

Figure 1 illustrates the intersection of a major street, labelled “High Street”, with a secondary road, labelled “Side Road”. There is a traffic light at every corner. The light gives more crossing time to the vehicles on High Street than to the vehicles on Side Road, to take different volumes of traffic into account. The traffic light controller goes through a main cycle which allows traffic to flow on High Street for a minimum of 12 seconds and a maximum of 18 seconds. It then changes to allow traffic to flow on Side Road for 6 seconds. This two cycles repeat themselves forever in this manner.
On every pole holding a traffic light there is also a button, which can be pressed by pedestrians. If any of the four buttons for pedestrians are pressed, the car traffic cycles are eventually suspended and all vehicle traffic stops, allowing pedestrians to cross in all four directions for a maximum of 6 seconds. The four buttons are labelled “*Ps1*”, “*Ps2*”, “*Ph1*”, and “*Ph2*”, depending on their position with respect to the intersection. As an

aid to pedestrians, the traffic light also shows the countdown in seconds of the time available for crossing. The overall program for the simulation is started by a user who can also decides when to exit the program.

2. Functionality

2.1 Starting Up

When the program starts execution, the following should happen:

1. Both LED lights should be turned on.
2. On the LCD screen the student number should appear on line 1 as in: "Student Number *V00123456*".
3. The 8-Segment Display should light the *point segment* (labelled "P", see below).
4. The program waits for the user to press the left black button to start the simulation. When the left black button is pushed, the simulation of the traffic lights should start (see below), together with its outputs.
5. Pressing the right black button before pressing the left one, that is, before starting the overall simulation, should exit the program. The LCD screen is cleared and all lights and displays are turned off.
6. Pressing any blue button before pressing the left black button should have no effect.

2.2 Ending the simulation and the program.

A user can end the program execution by pressing the right black button. The following events should be programmed within the exiting module.

1. Pressing the right black button while the simulation is in progress should cause the program to end. However this event is recognized and acted upon only at certain points in the simulation cycle, namely the same points when the pedestrian buttons are also being recognized (see details on pedestrian polling and cycle below).
2. Ending the program implies:
   (a) returning to `main`;
   (b) clearing the LCD screen and then printing a final message;
   (c) turning off all lights and displays;
   (d) showing the cleared board for at least 2 seconds before exiting.

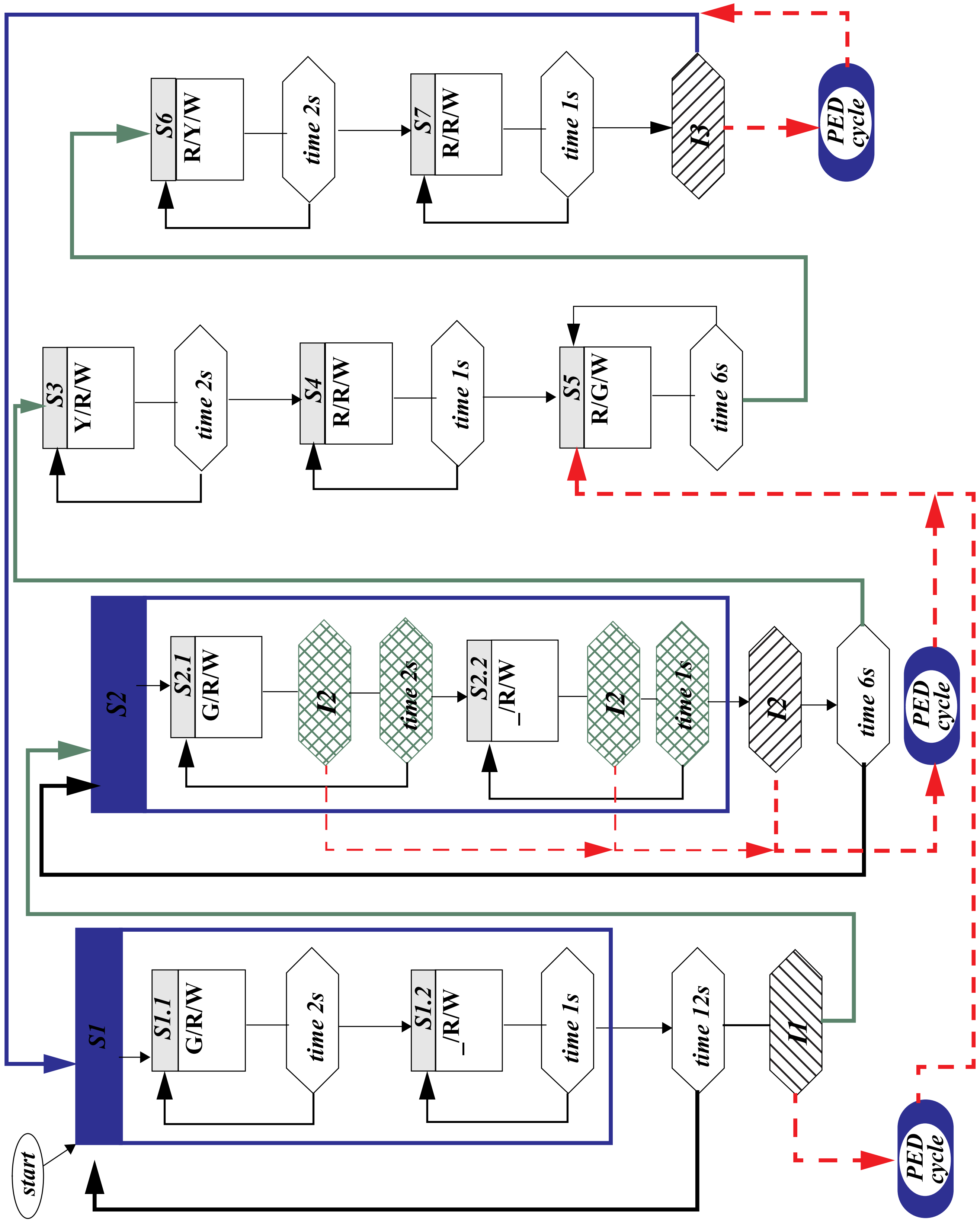
2.3 The Simulation.

The events for the traffic lights simulation include a cycle of red and green lights alternating between High Street and Side Road, with High Street using the longest time. Pedestrians can press buttons to request a crossing light. When the request is accepted, by polling the appropriate buttons, lights for car traffic turn to *red* in all directions and pedestrians are allowed to cross both High Street and Side Road. A display shows the number of seconds remaining in the pedestrian crossing cycle.

The car cycle then resumes. During the car cycle, the setting of the four traffic lights are displayed in a (crude) graphical form on the LCD screen in a given format, to show the user the current stage of the simulation. Figure 2 can be formally described as an Algorithm State Machine (ASM) chart. It represents the car traffic cycle and the patterns of expected outputs. The explanation follows. The colours used in the diagram do not have any semantic meaning.

We use a triples notation to represent the traffic light settings. A triple has the form $\mathbf{L_h}/\mathbf{L_s}/\mathbf{L_p}$ where $\mathbf{L_h}$ specifies the lights displayed for traffic on High Street, $\mathbf{L_s}$ the lights for traffic on Side Road, and $\mathbf{L_p}$ the lights shown to pedestrians. The possible values for $\mathbf{L_h}$ and $\mathbf{L_s}$ are: "**R**" (red is displayed), "**Y**" (yellow is displayed), "**G**" (green is displayed) and "_" (no light is displayed). The possibilities for $\mathbf{L_p}$ (displayed as graphic pictures to the pedestrians) are "**W**" meaning "wait -do not cross", "**X**" meaning "cross now" and "**!**" meaning "hurry up, the traffic is about to resume".

Figure 2: Algorithm State Machine for car traffic cycle



3. Understanding the ASM in Figure 2 for the Car Traffic Cycle.

3.1 State S1.

The traffic lights simulation in Figure 2 starts by showing *Green* for High Street and *Red* for Side Road, the pedestrian lights all display the ‘do not cross’ picture; this is state ***S1***. In this, the settings of the traffic lights are displayed in the ASM as **G/R/W**. As a special feature, the “**G**” signal for High Street flashes on and off, with an *on* period of 2 seconds and an *off* period of 1 second. The flashing corresponds to the traffic light settings alternating between **G/R/W** and **_/R/W**. In Figure 2, this behaviour is depicted by the controls between sub-states ***S1.1*** and ***S1.2*** contained within state ***S1***. The flashing lasts for 12 seconds without any possibility of changes, i.e. it must show four **G/R/W** periods and four **_/R/W** periods alternating for High Street, while the *Red* stays on for Side Road. If a pedestrian pushes a button during this period or the user pushes the right black button to end the simulation, there is *no* effect *during this period of time*.

However, a pedestrian may indeed press a button to request a crossing during this time. The built-in library software (for the simulator and for the board) captures this event and stores a signal appropriately. At the end of state ***S1***, the program needs to poll to ascertain whether any pedestrian request needs to be handled. This is shown in Figure 2 by the “***“I1”***” event (see more below).

Note that in the ASM representation used in Figure 2, every state has an associated “timer” which describes the length of time to keep the state, and its commands, active. The system passes control from one state to another only after the specified delay. For example, at entry to state ***S1.2*** at time T_i the action to be performed is to turn off the green light for High Street. Then the actions are continued (perhaps by refreshing the display) until time T_j where $(T_j – T_i)$ is equal to the desired delay – one second in this case.

3.2 State S2.

The maximum time for traffic flow on High Street is for 18 seconds, which includes State ***S1*** (for a fixed 12 seconds) and State ***S2*** (for a maximum of 6 seconds). State ***S2*** includes a second set of 2 *on* and 2 *off Green* flashes (6 seconds), as shown in the ASM. State ***S2*** in turn includes sub-states ***S2.1*** and ***S2.2*** similarly to State ***S1***. However, within this second set ***S2,*** the High Street cycle can be terminated by a pedestrian pressing one of the buttons or by an end of simulation request. That behaviour is explained and incorporated later in the ASM in Figure 3. This request can happen any time during the states ***S2.1*** and ***S2.2*** and needs to be polled and handled immediately. This implies that instead of a simple timing delay, one needs to program a timing delay which is also capable of polling (see more below). This is shown in the diagram by the arrows at ***“I2”***. That is, a pedestrian cycle or an end of simulation request can be recognized and acted upon only after 12 seconds in State ***S1,*** or at any time in State ***S2***.

3.3 States S3, S4, S5, S6, S7.

After states ***S1*** to ***S2*** have completed, with or without a pedestrian cycle, a new sequence starts for the lights to allow traffic flow from Side Road, with the possibility of a pedestrian request at the end of the car cycle for the Side Road. This is shown in states ***S3*** to ***S7***, and has the following effect on the lights:

(i) for 2 seconds, the traffic light settings are **Y/R/W,** that is, yellow for High Street;
(ii) for 1 second, the traffic light settings are **R/R/W,** that is, no traffic flow in any direction;
(iii) for 6 seconds, the traffic light settings are **R/G/W**, that is, Side Road now has traffic flowing;
(iv) for 2 seconds, the traffic light settings are “**R/Y/W**”, that is, yellow for Side Road;
(v) for 1 second, the traffic light settings are “**R/R/W**”, that is, no traffic flow in any direction;
(vi)a pedestrian cycle may occur at this point and should be tested for;
(vii) the lights return to the original “**G/R/W**” pattern, restarting the cycle in state ***S1***.

A pedestrian request or an end of simulation request can happen during states ***S5*** to ***S7***, but it is acted upon by polling only after state ***S7***. This is shown in the diagram by the event labelled ***“I2”***. This protocol ensures that the Side Road gets a minimum amount of traffic moving before a pedestrian interruption can take place (similarly to the event at ***“I1”***).

3.4 States S1 to S7 without pedestrian cycles.

The sequence above continues repeatedly without any pedestrian cycle, until the person monitoring the system decides to stop the program execution. Any deviations from the sequence of states can only occur after the minimum 12 seconds in state ***S1*** (at ***"I1"***), or after 7 seconds at the end of state ***S7*** (at ***"I3"***), or at any point during state S2 (at ***"I2"***), or by a user terminating the whole simulation (right black button polled at events ***"I1"*** or ***"I2"*** or ***"I3"***).

4. Understanding the ASM in Figure 3 for the Pedestrian Cycle.

The subset ASM for the pedestrian cycle only is shown in Figure 3. In this implementation, the system *polls* to see if a pedestrian has pressed a button. Polling takes place and is acted upon only at the points labelled ***"I1"*** or ***"I2"*** or ***"I3"*** or ***"I4"*** in the diagram, where***"I1" is*** at the end of state ***S1, "I2"*** is during state ***S2*** , ***"I3"***at the end of state ***S7*** and ***"I4"***at the end of state ***P5.*** If a pedestrian has pressed any of the four buttons "*Ps1*", "*Ps2*", "*Ph1*"or "*Ph2*", the normal sequence ***S1*** to ***S7*** changes to include a detour through states ***P1*** to ***P5***. Note that at the end of the pedestrian cycle called at ***"I1"*** or ***"I2"***, the return is to state ***S5,*** while at the end of the pedestrian cycle called at ***"I3"***, the return is to state ***S1.*** The pedestrain cycle may return also with the indication that a right black button has been pressed, thus requesting an end of the simulation. It is important to understand that a pedestrian button (or a right black button) can be pressed at any time and the built-in software "*captures*" the event and posts a signal. However the simulation program (and the real embedded system of traffic lights) only "*polls*" the event at certain times in the cycles, else repeated interruptions would impeded the flow of normal traffic.

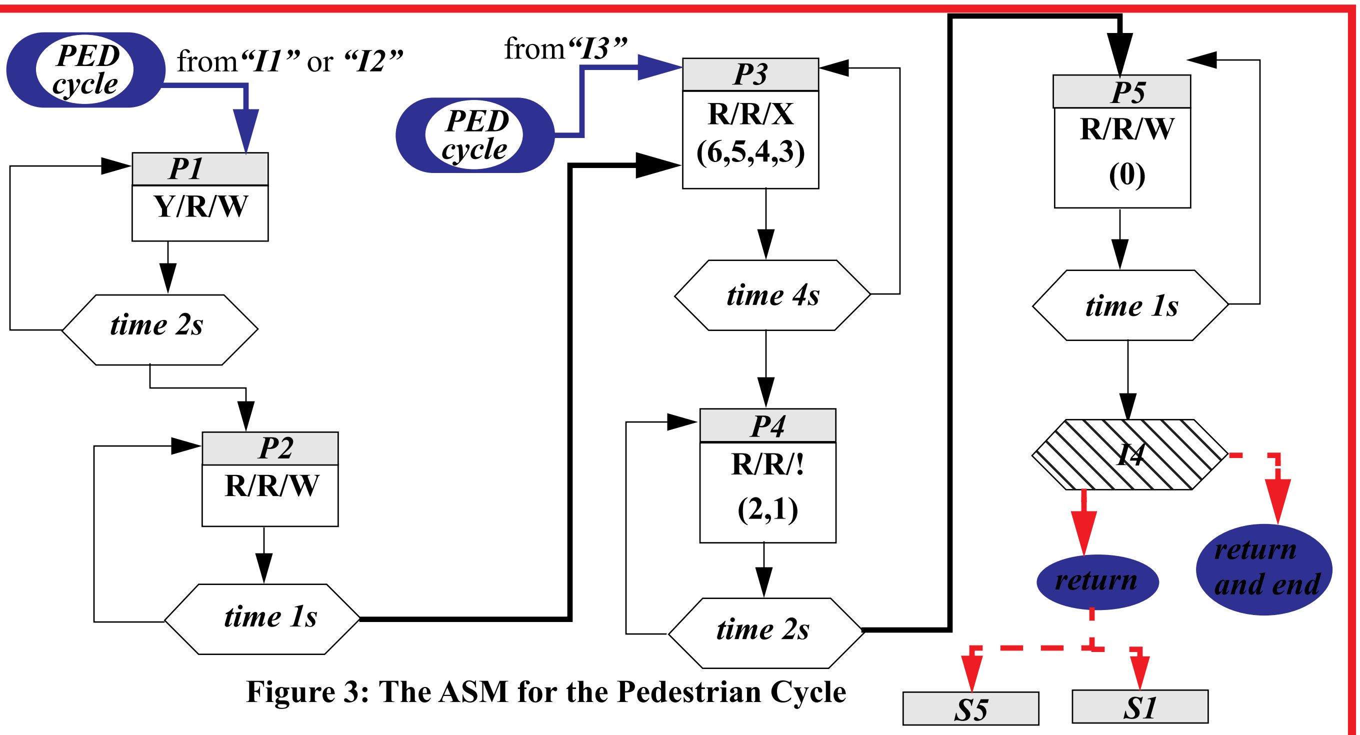


Figure 3: The ASM for the Pedestrian Cycle

4.1 States P1 to P5.

To denote the switch to the pedestrian sequence, the display should change as in the following description.

1. **CASE 1:** the pedestrian cycle was detected at ***"I1"*** or ***"I2"***
   (i) **P1:** for 2 seconds the traffic light settings are **Y/R/W,** that is, yellow for High Street;
   (ii) **P2:** for 1 second the traffic light settings are **R/R/W,** that is, no traffic flow in any direction;
   (iii) **P3:** for 4 seconds the traffic light settings are **R/R/X**, that is, pedestrians can cross in any direction; during this period the 8-Segment display counts down from the value "6" to the value "3";
   (iv) **P4:** for 2 seconds the traffic light settings are **R/R/!**; during this period the 8-Segment display counts down from the value "2" to the value "1";

   (v) **P5:** for 1 second the traffic light settings are **R/R/W,** that is, no traffic flow in any direction; during this period the 8-Segment display counts down to the value “0”;
   (vi) **S5:** the traffic light settings resume with the **R/G/W** pattern beginning at state ***S5***, independent on having been called from ***“I1”*** or ***“I2”***.
2. **CASE 2:** the pedestrian cycle was detected at ***“I3”.***
   (i) **P3:** for 4 seconds the traffic light settings are **R/R/X**, that is, pedestrians can cross in any direction; during this period the 8-Segment display counts down from the value “6” to the value “3”;
   (ii) **P4:** for 2 seconds the traffic light settings are **R/R/!**; during this period the 8-Segment display counts down from the value “2” to the value “1”;
   (iii) **P5:** for 1 second the traffic light settings are **R/R/W,** that is, no traffic flow in any direction; during this period the 8-Segment display counts down to the value “0”;
   (iv) **S1:** the traffic light settings resume with the **G/R/W** pattern beginning at state ***S1***.
3. **CASE 3:** same as either of the two cases above, but after P5 a right black button event is detected to end the simulation.
   (i) Return to the calling routine CarCycle and from there return all the way back to “`main`” through “`StartSim`”.

5. The Physical Components: the peripherals to be programmed

There are 5 peripheral components which need to be programmed to support the simulation and their diagrammatic view is shown in Figure 4. They are:

1. One 8-segment display.
2. Two red LED lights.
3. Two black buttons.
4. Four of the sixteen blue buttons arranged in a keyboard 4 x 4 grid.
5. One LCD display screen.

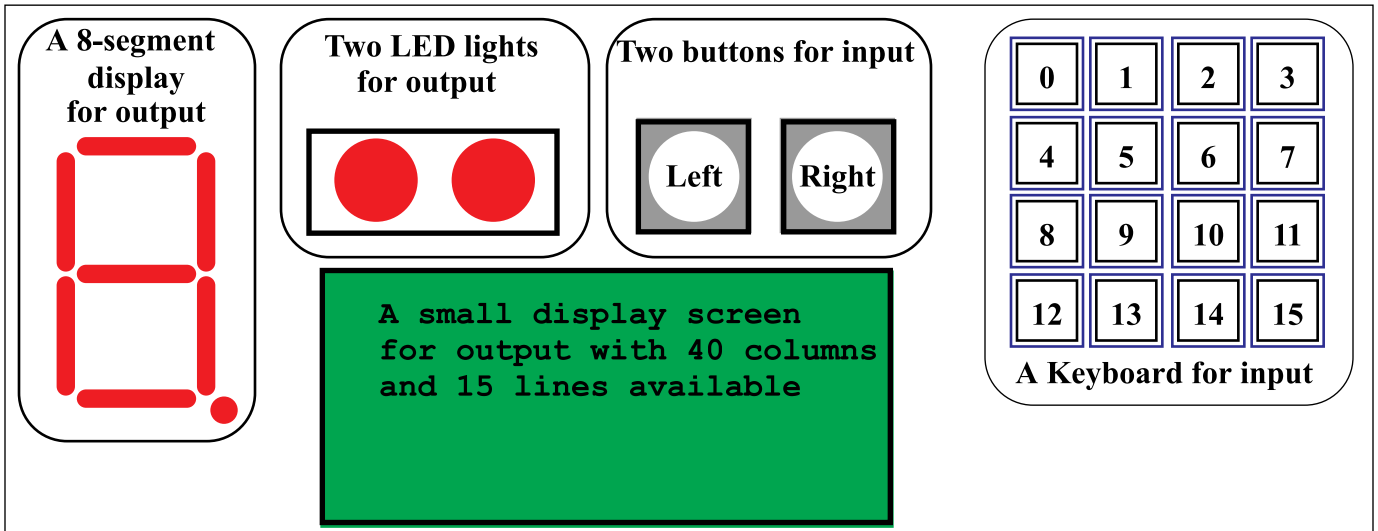

A 8-segment display for output
Two LED lights for output
Two buttons for input
Left
Right
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
A Keyboard for input
A small display screen for output with 40 columns and 15 lines available

Figure 4: A diagrammatic view of the available controls and displays

5.1 The 8-Segment Display

- The 8-Segment display shows the crossing time for pedestrians, counting down from 6 to 0 seconds.
- The display must be programmed to show a number from 6 to 0.
- The “Point” segment at the bottom right corner is turned ON (lit) to indicate that car traffic is allowed. When it is not lit it means that the pedestrian cycle is taking place.
- The code to display numbers in the 8-Segment Display is available in the documentation for ARMSim#.

5.2 The LEDs

- The two red LED lights indicate the direction of car traffic.

- Left LED = ON and right LED = OFF, ⇒ green on High Street and red on Side Road.
- Left LED = OFF and right LED = ON, ⇒ red on High Street and green on Side Road.
- Both LED = ON and steady (not blinking), ⇒ red in both directions, with no pedestrian cycle.
- Both LED = OFF ⇒ red in both directions, with a pedestrian cycle.
- BONUS: Both LED = ON and blinking, ⇒ yellow light in one directions.

5.3 The Black Buttons

- The two black buttons control the start and end of the program.
- Clicking the left black button starts the overall simulation program at State **S1**.
- Clicking the right black button stops the simulation program.
- The right black button for ending the simulation is polled only at points ***"I1"*** or ***"I2"*** or ***"I3"***.
- After clicking the right black button and ending the simulation, the program must exit by resetting all components (see section 2.2).

5.4 The Blue Buttons

- Only 4 blue buttons have any effect, namely "8" to "11", the others are ignored.[1]
- The buttons are polled and are acted upon only at points ***"I1"*** or ***"I2"*** or ***"I3"*** in the overall cycle.
- The blue button "8" implies that a pedestrian has pressed the "*Ph1*" button on High Street.
- The blue button "9" implies that a pedestrian has pressed the "*Ph2*" button on High Street.
- The blue button "10" implies that a pedestrian has pressed the "*Ps1*" button on Side Road.
- The blue button "11" implies that a pedestrian has pressed the "*Ps2*" button on Side Road.

The reason to distinguish between the 4 buttons is that a further improvement to the system may be to handle two distinct pedestrian crossing cycles (for High Street and Side Road), depending on the button pressed.

5.5 The LCD Screen Display

The LCD Screen Display is a grid of 40 columns by 15 rows of individual cells. Each LCD cell can display one ASCII character (byte). The coordinates for each LCD cell are specified by a {column, row} pair. The top-left cell has coordinates {0,0}, while the bottom-right cell has coordinates {39,14}. The diagram in Figure 5 shows this grid. The displayed letter **A** is at location {4,2} as an example.

The LCD screen displays the current state of the simulation to the user. Only a few lines are actively used, which should be displayed somewhere in the centre of the LCD screen, giving a somewhat weird and crude representation of the status of the 4 traffic lights. On the top right corner the state number must be displayed. While this would not normally be an integral part of a traffic light, it is essential for development and testing. This part can be used to exercise your creative side and can gather extra bonus points, if you think that you can design and implement a more realistic, meaningful and catchy display to simulate the traffic lights on the LCD screen. The best way to understand the expected behaviour on the LCD screen is to examine column 5 of Table 1.

5.6 Summary of the Overall Outputs

Unfortunately there aren't enough lights on the board components to simulate a traffic light for four corners. The available components are shown schematically in Figure 4. The simulated mappings for the traffic light controller are summarized in Table 1 (see the Appendix) for each possible output expected in the various states. Column 1 lists the states, while column 2 lists the corresponding *logical* outputs. Columns 3 and 4 show the expected outputs, for each state, of the two red LED lights and of the 8-segment display. The 8-segment display is active during a pedestrian crossing cycle, when it displays a countdown of the seconds remaining for crossing, starting at the value 6 down to 0, then turning off. Column 5 shows the 3 lines which should be displayed in the centre of the LCD screen, giving a somewhat weird and crude representation of the status of the 4 lights. This part can be used to exercise your creative side and can gather extra bonus points, if

1. "Ignoring" the others means actively writing code that does nothing in case they are pressed.

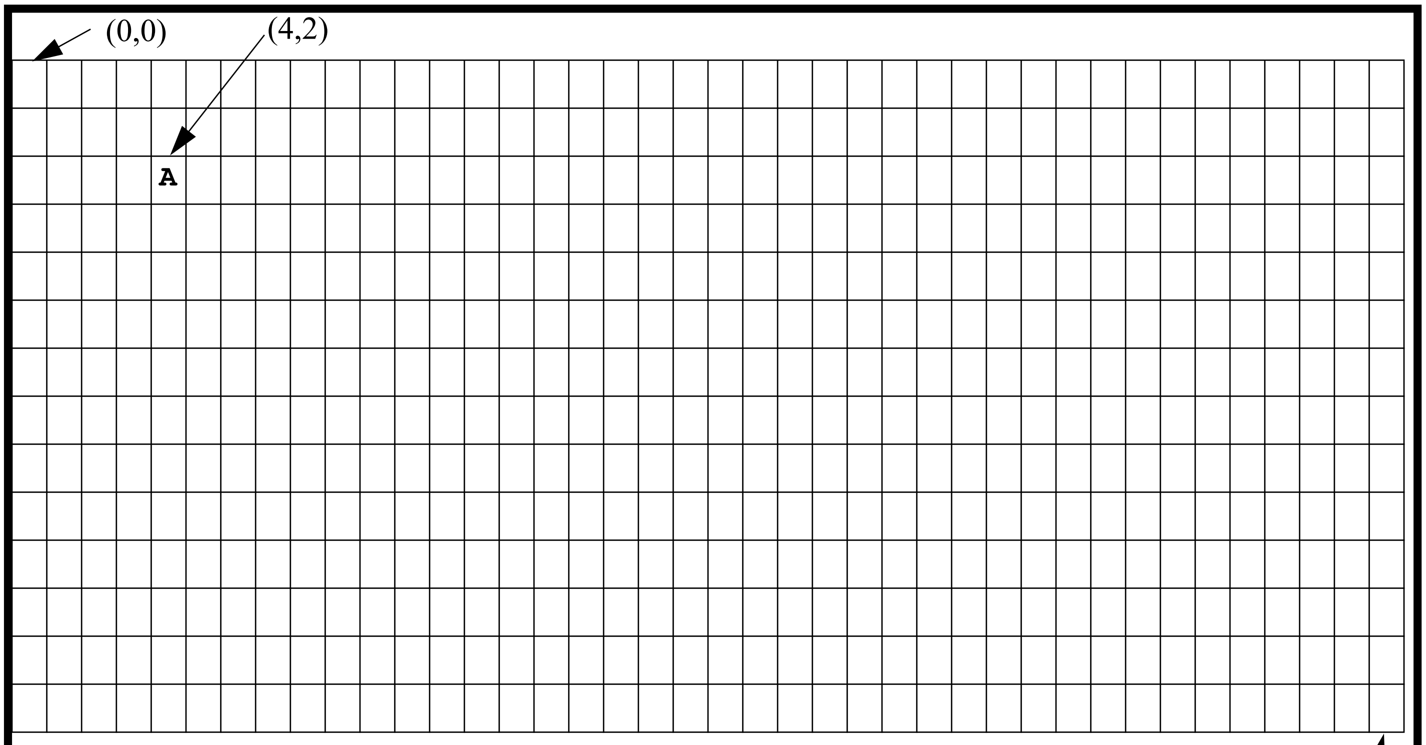


Figure 5: The LCD screen grid

you think that you can design and implement a more realistic or at least a more meaningful and catchy display to simulate the traffic lights on the LCD screen.

6. Testing for a Successful Simulation

You must design your own sequence of events to prove the correctness of your design and of your implementation. A full testing scheme which will be used during the demo will be posted just before the demo itself. As an initial suggestion, the program should be tested by checking the following sequence of events:

1. Start with state ***S1*** and go all the way to state ***S7*** at least a couple of times, checking the timing.
2. Repeat the sequence above and test ending the program at point ***"I1".*** While in ***S1*** a right black button should be pressed and after 12 seconds (before ***S2***), the program should end the simulation.
3. Repeat the above from point ***"I3"*** appropriately**.**
4. Repeat the sequence above and test a pedestrian cycle at point ***"I1".*** While in ***S1***, a pedestrian button should be pressed and its cycle should take place from ***"I1"***, followed by resuming from state ***S5***.
5. Repeat the sequence above and test a pedestrian cycle at point ***"I3".*** While in ***S5-S7***, a pedestrian button should be pressed and its cycle should take place from ***"I3"***, followed by resuming from state ***S1***.
6. Repeat the sequence above and test a pedestrian cycle at point ***"I2".*** While in ***S2***, a pedestrian button should be pressed and its cycle should take place immediately, followed by resuming from state ***S5***.
7. Repeat the sequence above and test ending the program at point ***"I2".*** While in ***S2***, a right black button should be pressed and the simulation should end immediately.

7. The Flowcharts [10 marks]

The first milestone is for you to draw *a set of detailed and interconnected flowcharts* for the program you are about to implement. The flowcharts are due on March 6th at 15:30 in the lecture and will be returned to you by the end of the week with feedback. The flowcharts should be detailed enough that you can program directly from them with almost no extra work. They are also to be used to check for correctness of the requirements as they are the translation into implementable steps from the higher level ASM. Use the appropriate flowchart symbols (not any design you like!).

8. Specifications and Implementation Requirements

While there are a number of choices for the programming of this simulation, some strict specifications for certain portions are given. They are intended to help you, and lead you towards a good software design. First, developing a set of flowcharts is important to define the various steps of coding later. By handing in your flowchart you will receive some timely feedback to help you program. Then an initial call graph is given to show the modularity of the program in terms of some expected subroutines. You are handing in a final call graph of your design to aid the evaluation and the demo and show your design decisions.

8.1 Interface Specifications for Subroutines

In these specifications only a minimum list is given, not necessarily sufficient. Any additional routines in your design must be documented and justified. All parameters are passed through registers - you must document exactly which ones both at the call point and at the start point of a subroutine. Here is the list of functions (procedures) which are certainly required, called from `main.` Many of them are given in the template file provided.

void Init(no parameters);
Set the initial display when the program starts and before the simulation begins. Called by `main`.

void StartSim(no parameters);
After the left black button has been pressed, start the traffic light simulation. Called by `main` after `main` detects a left button press. Returns to main

void EndSim(no parameters);
After the right black button has been pressed, stop the traffic light simulation, clear all displays and exit the program. Called by `main` after a right button press has been detected (somewhere) and control has returned to `main`.

int:R0 CarCycle(State:R1);
Given the number of the state in the cycle where to begin (choices are S1 or S5), simulate the cycle for car traffic and return a value denoting the reason for terminating the cycle. The return value must be: “1” for a pedestrian cycle called at ***“I1”***; “2” for a pedestrian cycle called at ***“I2”***; “3” for a pedestrian cycle called at ***“I3”***;“0” for the user ending the simulation (a right button press has been detected).

int PedCycle(CallPosition:R1);
Given the position in the cycle where the pedestrian cycle was called from (***“I1”*** or ***“I2”*** or ***“I3”***), simulate the pedestrian cycle. The return value must be: “0” when the pedestrain cycle ends normally; “1” if a right black button request has been detected during the cycle. Called by `StartSim`.

void Wait(Delay:R10);
Given an amount of time in milliseconds, cause a delay with nothing happening. Takes care of 15-bit timer adjustments (you are responsible to understand this code completely even if it is given to you).

int WaitAndPoll(Delay:R10);
Given an amount of time in milliseconds, cause a delay while continuously polling for other events (e.g. a button press). Takes care of 15-bit timer adjustments. The return value must be: “0” when the full time delay has taken place; “-1” if the function returns after a right black button press (interrupting the delay); “*blue button code*” if the function returns after a blue button press (interrupting the delay).

void Display8Segment(Number:R0; Point:R1);
Displays the number 0-9 given in R0 on the 8-segment. If R1 = 1, the point is also shown

void DrawScreen(PatternType:R10);

Displays on LCD screen the 5 lines denoting the state of the traffic light.

8.2 Global Variables

Using any global variable is normally bad practice. It should be possible to design this simulation without any. If any global variables are used, they must be documented explicitly and additional documentation attached stating the reason for their use.

- If global variables are used and a reasonable justification is given, the evaluator may accept without penalty or with a small penalty after showing an alternative implementation solution.
- If global variables are used without documentation and justification, then grade penalties can be expected.
- Ask questions on this kind of decisions - you are expected to consult your instructors (or managers)!

8.3 Call Graph

A call graph shows the connections between subroutines. It shows which subroutines can call which other subroutines. It does not show the logic of operations and it does not show the interface specifications (e.g. the parameters being passed or the results returned). Its purpose is to display clearly the expected structure and modularization of the code. As an example, Figure 6 shows the call graph for the subroutines specified so far. An example of call graph from assignment 1 will also be available in the lab notes.

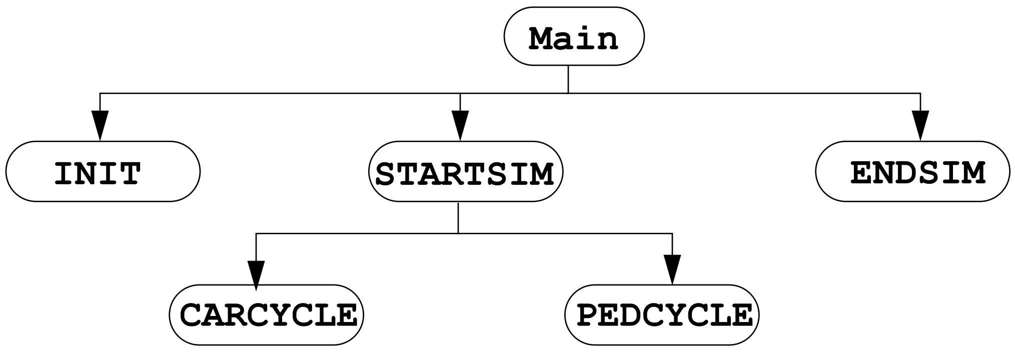


Figure 6: The Initial Call Graph

9. Deliverables and Due Dates

1. The first deliverable is a flowchart of your program design, due on Wednesday March 6 at 15:30 in the lecture and will be returned to you by the end of the week with feedback. Suggestions may help you change your code in progress. The flowchart is worth 10% of the assignment marks, that is, 10 marks.
2. You are to demonstrate your project to your instructor. The demo will take place from March 12 to March 15. A schedule will be organized. The demo is not optional. If you do not attend the demo, your assignment will still be evaluated with a penalty of-30%. There are 3 parts to the demo session:
   (a)the execution of your program using ARMSim#, worth 40 marks;
   (b)the execution of your program on the Embest board, worth 5 marks;
   (c)the discussion and evaluation of your code, worth 45 marks of which: 20 for the structure of the code; 10 for the documentation; 10 for technical issues (e.g. parameters); 5 for the call graph.
3. The source code of your program must be handed in by electronic submission via Connex no later than Monday March 11 at 11:55 p.m.
4. A printed source code listing must be handed in by no later than Monday March 11 at 11:55 p.m in the CSC 230 box in the ECS building.
5. A design of the call graph of your code must be handed in with the printout of the source code no later than Monday March 11 at 11:55 p.m in the CSC 230 box in the ECS building.
6. Collaboration between students is *not* permitted., consultation is encouraged, especially with instructors.

7. Total marks = 100: 10 (flowchart) + 45 (execution on ARMSim# and Embest Board) + 45 (code evaluation).

10. Appendix 1: Table summary of the Overall Outputs

Table 1 summarizes all the outputs for the various states. They are grouped when they are the same for distinct states. Most of the code for the actual graphical outputs for the LCD screen is given in the template given and may change (check the postings).

11. Appendix 2.

In the explanation for this assignment a few concepts were used whose more formal definition may be important in order to exploit them precisely and fully. The explanations are paraphrased from Wikipedia.

The ***Algorithmic State Machine (ASM)*** is a method for designing finite state machines. The ASM diagram is like a state diagram but less formal and thus easier to understand.

A ***flowchart*** is a schematic representation of an algorithm or a process, including all the details in the decision making and in the paths which needs to be followed. The symbols used in a flowchart have a meaning according to their shape.

A ***call graph*** is a directed graph that represents calling relationships between subroutines in a computer program. Specifically, each node represents a procedure and an edge indicates that a particular procedure calls another procedure. Thus, a cycle in the graph indicates recursive calls. Call graphs are a basic program analysis result that can be used for human understanding of programs, or as a basis for further analyses. One simple application of call graphs is finding procedures that are never called. A dynamic call graph is a record of an execution of the program, e.g., as output by a profiler. Thus, a dynamic call graph can be exact, but only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program.

12. Appendix 3: 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.

12.1 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 ARMSim# 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 232 -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 215-1 = 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 215-1) 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 7.

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.

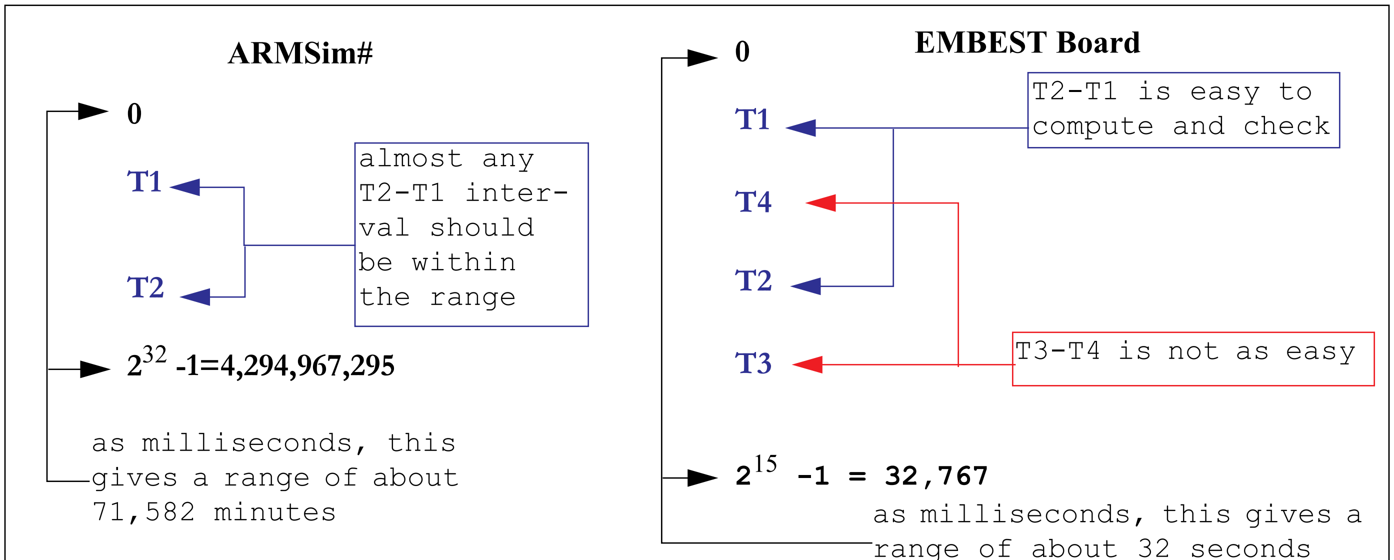

ARMSim#
0
T1
T2
almost any T2-T1 interval should be within the range
2^{32} -1=4,294,967,295
as milliseconds, this gives a range of about 71,582 minutes
0
EMBEST Board
T2-T1 is easy to compute and check
T1
T4
T2
T3
T3-T4 is not as easy
2^{15} -1 = 32,767
as milliseconds, this gives a range of about 32 seconds

Figure 7: Timing intervals on ARMSim# and on the Embest Board

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 8 should help and it is implemented in some of the code given to you.

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

```
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 8: Pseudo code for a simple delay cycle

12.2 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 3 seconds (as is the case in one of the states for the traffic light), then one needs a simple programmed delay for 3 seconds in between changing the output settings, 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 8.

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 during State 2 the program delays up to 6 seconds with Green on the High Street, but the cycle can be interrupted and shortened any time by a pedestrian request. In this interval one continues to poll for pedestrian requests while also monitoring the intervals on the output signals. The simple “`Wait`” routine from above 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 9 summarizes the required logic.

```
PSEUDO CODE FOR 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
```

Figure 9: Pseudo code for a delay cycle with polling

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.

Table 1: Mapping for Outputs

1	2	3	4	5
State	**State Output**	**LED Lights**	**8-Segment Display**	**LCD Screen on 3 lines**

S1.1	**G/R/W**	Left LED: ON.
		Right LED: OFF.
			Dot “P” on	`R W S1`
				`GGG W GGG W`
				`R W`

S1.2	**_/R/W**	Left LED: ON.
		Right LED: OFF.
			Dot “P” on	`R W S1`
				`W W`
				`R W`

S2.1	**G/R/W**	Left LED: ON.
		Right LED: OFF.
			Dot “P” on	`R W S2`
				`GGG W GGG W`
				`R W`

S2.2	**_/R/W**	Left LED: ON.
		Right LED: OFF.
			Dot “P” on	`R W S2`
				`W W`
				`R W`

S3	**Y/R/W**	Left LED: ON.
		Right LED: OFF.
		Bonus: both LEDs blinking
			OFF	`R W S3`
				`YYY W YYY W`
				`R W`

S4	**R/R/W**	Left LED: ON.
		Right LED:ON.
			OFF	`R W S4`
				`R W R W`
				`R W`

S5	**R/G/W**	Left LED: OFF.
		Right LED: ON.
			Dot “P” on	`GGG W S5`
				`R W R W`
				`GGG W`

S6	**R/Y/W**	Left LED: OFF.
		Right LED: ON.
			OFF	`YYY W S6`
				`R W R W`
				`YYY W`

S7	**R/R/W**	Both LEDs: ON.	OFF	`R W S7`
				`R W R W`
				`R W`

P1	**Y/R/W**	Left LED: ON.
		Right LED: OFF.
		Bonus: both LEDs blinking
			OFF	`R W P1`
				`YYY W YYY W`
				`R W`

P2	**R/R/W**	Both LEDs: ON.	OFF	`R W P2`
				`R W R W`
				`R W`

P3	**R/R/X**	Both LEDs: ON.	Countdown:
			(6,5,4,3).
				`R XXX P3`
				`R XXX R XXX`
				`R XXX`

P4	**R/R/!**	Both LEDs: ON.	Countdown:
			(2,1).
				`R !!! P4`
				`R !!! R !!!`
				`R !!!`

P5	**R/R/W**	Both LEDs: ON.	Display: “0”	`R W P5`
				`R W R W`
				`R W`

13. Appendix 4: changes

- On the top right corner the state number must be displayed. While this would not normally be an integral part of a traffic light, it is essential for development and testing. Table 1 has been updated.
- The schedule for development, aiming towards maximizing correctness, minimizing debugging time and increasing your grade, should follow these steps (which are similar to the testing in section 6):
  1. [40%] Program an infinite sequence which goes through states S1 to S7 without interruptions.
  2. [10%] Add the stop request at points ***“I1”*** and ***“I3”***.
  3. [30%] Add the pedestrian request and the pedestrian cycles at points ***“I1”*** and ***“I3”***.
  4. [10%] Add the stop request during state S2 (at ***“I2”***).
  5. [10%] Add the pedestrian request and the pedestrian cycles during state S2 (at ***“I2”***).

The above shows how the execution will be evaluated. A full testing protocol will be posted before the demo.

- Make sure to keep versions of your work at the various stages above, so you can always submit something which has no assembly errors.