C SC 230 – Summer 2012 – Assignment 3

Due Wednesday July 18 by 1 pm

Total Marks = 40

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.

The assignment overall comprises the following main tasks:

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

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. Moreover, further instructions will be given in lectures.

1. The Cellular Phone as an Event Driven Program

You are going to write the code to simulate the control for a *simple* cellular phone. The program is able to detect an incoming call, to accept the user's decision to answer or not, and to charge a small amount for air time if the call is answered. It also lets a user initiate a call, either local or long distance, and charge accordingly for air time.

The cellular phone 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 cellular phone, 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 cellular phone is given as a set of finite state diagrams. The first task is to translate such diagrams into flowcharts corresponding more closely to the code in ARM.

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

2. The Functionality of the Main Events

2.1. Initialization.

When the code is loaded into ARMSim# or the board, the program does not start automatically, as the phone is assumed to be turned off. The program starts only after receiving a signal through pressing the blue button = "14" on the keyboard, corresponding to a user turning on the phone. When the program starts, a short welcoming message is displayed and the lights are set in an initial state. The program then waits for an external signal to cause an *event* (this is typical in interactive graphics programs).

- **Start (phone on):** After loading into ARMSim# (or to the board), press Blue Button = "14".
- Initial settings:
  * Greeting message on LCD screen (include your student name and number) on LCD (lines 1 & 2).
  * Both LED lights off.
  * 8-segment displays only the dot.

2.2. Ending the Simulation

The simulation does not end when a single call has been simulated. The program is terminated by pressing the blue button = "15" on the keyboard, corresponding to a user turning off the phone.

- **End (phone off)**: Blue Button = "15".
- Outcome:
  * Closing message on LCD screen using lines below 2; the rest of the screen is cleared.
  * After a 5 seconds delay, both LED lights off; 8-segment displays blank; whole screen cleared.
  * execution ends.
- Note: it is possible for the program to be started (Blue Button=14) and then immediately ended (Blue Button=15) before any other event.[1]

2.3. Initial Events

Idle Mode. While nothing happens, the 8-segment displays only the dot, the LED lights are off and a single line is displayed on the LCD screen (line 5), stating: "*Waiting for Event*".[2] Bonus: have the line blinking on and off in the same position and change the code to use a "Wait and Poll" strategy (see below about timing).

Starting event. The starting event is the initiation of a call or the decision to turn off the phone. In the former case one needs to discriminate whether it is an incoming or outgoing call. A call is initiated by the push of one of the two Black Buttons.

Figure 1 shows the events and their connections as a *finite state machine* diagram. Each circle represents a *state* in which the simulation of the system can be. Each state is given an identifier label. Each state has associated a set of outputs, namely what the lights and the screen and the 8-segment are supposed to display. The outputs are described separately below and are shown as a label within the state in the figure. Each arrow represents a *transition* from a state to another. A transition is triggered by an input event which is described by the label on the arrow.

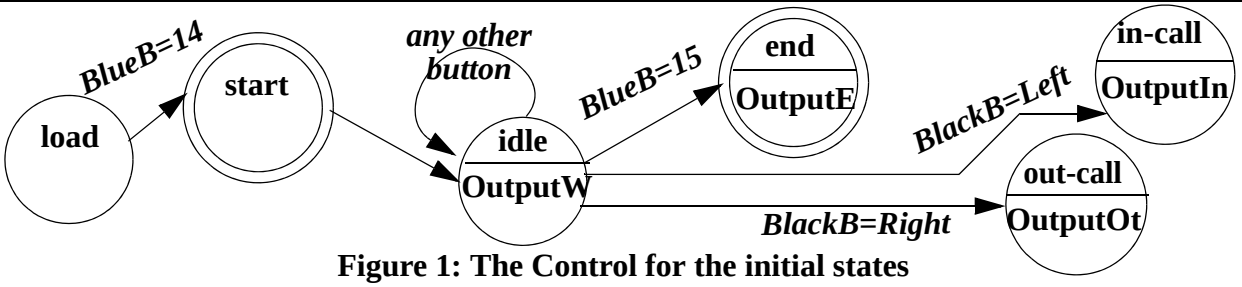

Figure 1: The Control for the initial states

1. I am sure you have occasionally turned on and off your phone without doing anything.
2. Feel free to be more creative and display something more interesting.

3. The Calls as Events

3.1. Incoming Call

A Left Black Button press is interpreted as an incoming call. One chooses whether to answer it or not, and the decision must be made within 5 seconds. The sequence of events is depicted in Figure 2 and is as follows:

- The Left Black Button is pressed and the event is detected.
- The 8-segment displays “0”.
- Both LED lights are turned on.
- The “DIAL” blue button “0” on the keyboard is checked for a maximum of 5 seconds. In this 5 second period:
  * the 8-segment displays the count of the seconds from 5 to 1, one number per second;
  * both LED lights stay on.
- If no signal (blue=0) is received for an answer, then the program returns to the Idle State at the end of the 5 seconds.
- If the blue button = 15 is pressed, the program returns to the Idle State with a flag to exit. Note that no function ever exits the program directly - everything must go back to the top and only `main` exits.
- If the “DIAL” blue button “0” is pushed within the 5 seconds, then:
  * a message is displayed stating the acceptance of the incoming call;
  * a timer starts in order to enable the computation of charges;
  * the 8-segment display is used to show the ticking of seconds, with wraparound (that is, it counts 1 to 9 to 1 to 9 continuously);
  * both LED lights blink on/off together (select your own blinking time).
- When the “HANG-UP” blue button “1” on the keyboard is pressed, the incoming call is officially terminated and:
  * both LED lights stop blinking and are on together;
  * the 8-segment displays “0”;
  * the cost is computed at a rate of $1 per second, and a message is displayed with the total amount;
  * the system returns to the Idle State after a delay of 5 seconds (i.e. no events possible in this period).

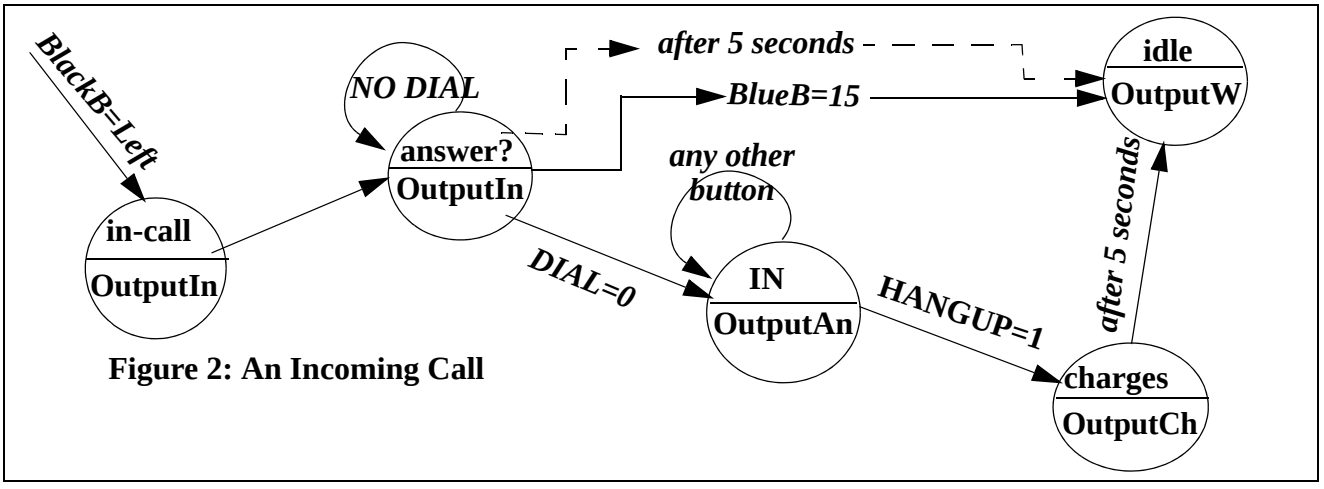

Figure 2: An Incoming Call

3.2. Outgoing Call

A Right Black Button press from the Idle State is interpreted as an outgoing call. The call can be local or long distance and this interpretation is given by the sequence of numbers entered on the keyboard before dialling the actual call. The actual start of an outgoing call, after numbers have been entered, is accomplished by pushing the “DIAL” blue button “0”. Ending an outgoing call is accomplished by pushing the

"HANG-UP" blue button "1". This is similar to what happens to receiving an incoming call. The sequence of events is depicted in Figure 3 and is as follows:

- The Right Black Button is pressed while in the Idle State and the event is detected.
- The 8-segment displays "0".
- Both LED lights are on.
- The blue buttons are checked for the dialing sequence - see the instructions below for local or long distance calls.
- If no blue buttons are pressed for 5 seconds the system returns to the Idle State.
- If the blue button = 15 is pressed, the program returns to the Idle State with a flag to exit. Note that no function ever exits the program directly - everything must go back to the top and only `main` exits.

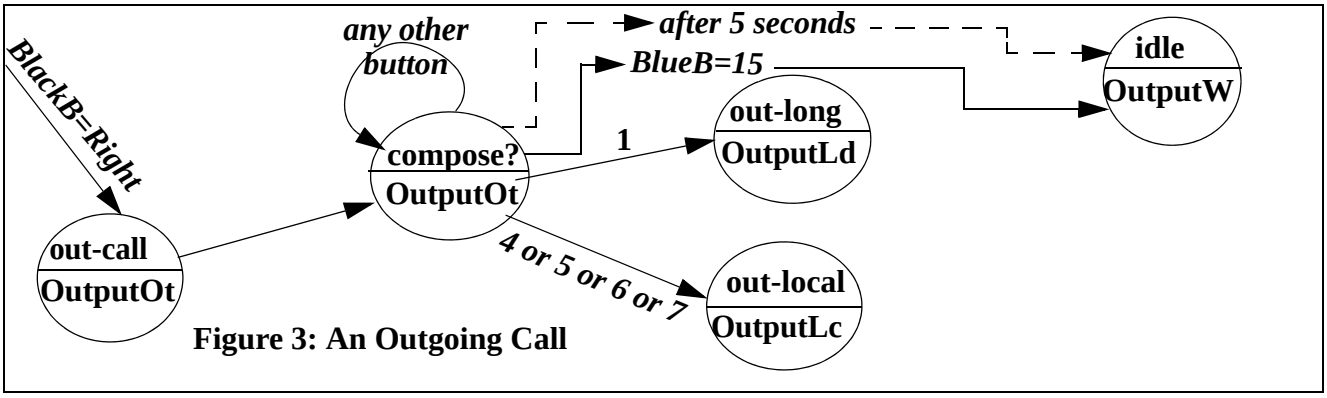

Figure 3: An Outgoing Call

3.2.1. Outgoing Local Call

A local call includes the selection of 4 digits for the number, in the range 4 to 7, in any order, with repetitions acceptable. The system detects exactly 4 digits and will not accept any more. The system should not accept any digit out of range (ignore). The sequence of events is depicted in Figure 4 and is as follows:

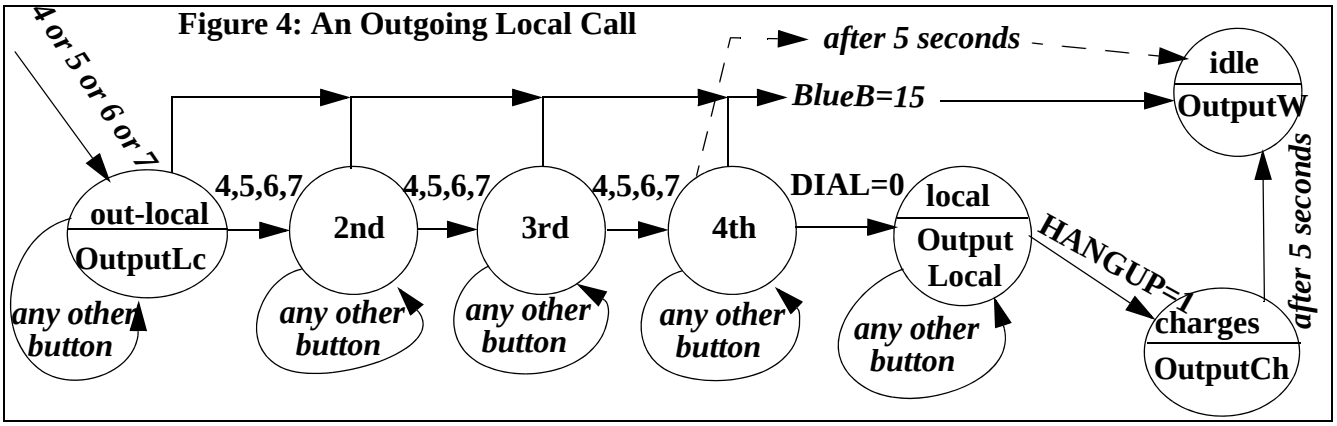

Figure 4: An Outgoing Local Call

- Four numbers in the range 4-7 are entered using the blue keyboard buttons.
- Each number is displayed on 8-segment display as soon as entered.
- The LEDs do not change (i.e. both on) until the "DIAL" blue button "0" is detected.
- The "DIAL" blue button "0" on the keyboard is checked for a maximum of 5 seconds. In this 5 second period:
  * the 8-segment displays the count of the seconds from 5 to 1, one number per second;
  * both LED lights stay on.
- The "DIAL" blue button "0" is checked to start the actual call.
- If the "DIAL" blue button "0" is not pushed within 5 seconds, the system returns to the Idle State.

- If the blue button = 15 is pressed at any point, the program returns to the Idle State with a flag to exit. Note that no function ever exits the program directly - everything must go back to the top and only `main` exits.
- If the "DIAL" blue button "0" is pushed within 5 seconds, then:
  * the left LED light is then turned on, while the right LED light is off;
  * a message is displayed on the LCD screen identifying a local call;
  * a timer starts in order to enable the computation of charges;
  * the 8-segment display is used to show the ticking of seconds, with wraparound (that is, it counts 1 to 9 to 1 to 9 continuously).
- When the "HANG-UP" blue button "1" on the keyboard is pressed, the outgoing local call is officially terminated and:
  * both LED lights are on together;
  * the 8-segment displays "0";
  * the cost is computed at a rate of $2 per second, and a message is displayed with the total amount;
  * the system returns to the Idle State after a delay of 5 seconds (i.e. no events possible in this period).

3.2.2. Outgoing Long Distance Call:

A long distance call includes the selection of the digit "1" followed by 4 digits for the number, in the range 4 to 7, in any order, with repetitions acceptable. The system detects the "1" for long distance and then detects exactly 4 digits and will not accept any more. The system will not accept any digit out of range (ignore).The sequence of events is depicted in Figure 5 and is as follows:

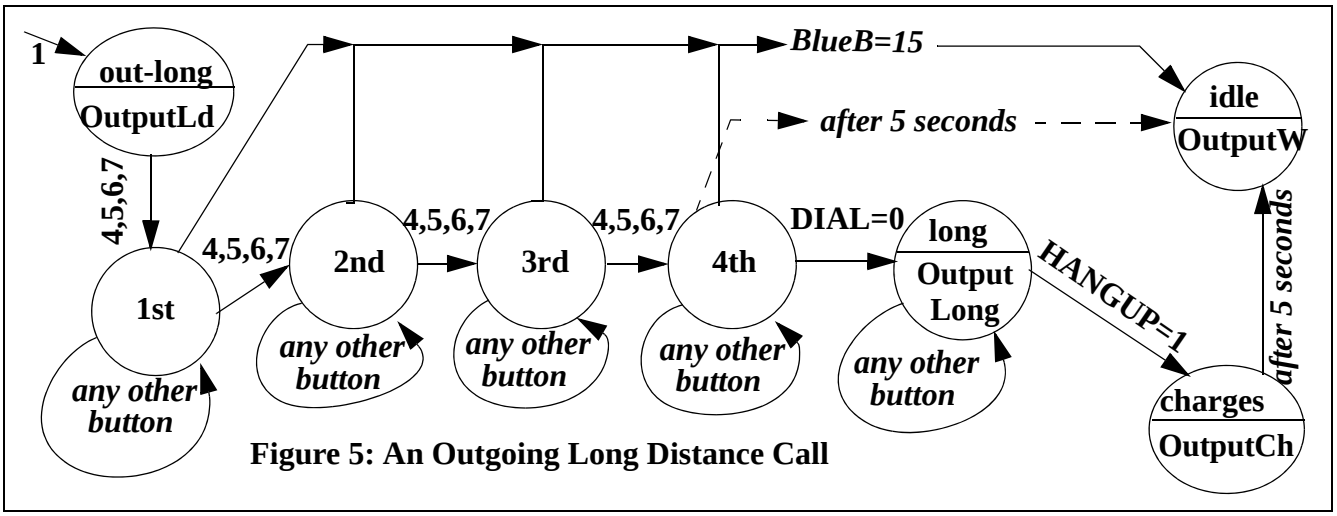

Figure 5: An Outgoing Long Distance Call

- The number "1" is pressed.
- Four numbers in the range 4-7 are entered using the blue keyboard buttons.
- Each number is displayed on 8-segment display as soon as entered.
- The LEDs do not change (i.e. both on) until the "DIAL" blue button "0" is detected.
- The "DIAL" blue button "0" on the keyboard is checked for a maximum of 5 seconds. In this 5 second period:
  * the 8-segment displays the count of the seconds from 5 to 1, one number per second;
  * both LED lights stay on.
- The "DIAL" blue button "0" is checked to start the actual call.
- If the "DIAL" blue button "0" is not pushed within 5 seconds, the system returns to the Idle State.
- If the blue button = 15 is pressed at any point, the program returns to the Idle State with a flag to exit. Note that no function ever exits the program directly - everything must go back to the top and only `main` exits.

- If the “DIAL” blue button “0” is pushed within 5 seconds, then:
  * the right and left LEDs blink alternatively (i.e. left on, right off, then left off, right on, select your own blinking time); another choice is to have the left LED off and the right one on without any blinking - implement this easier choice first!
  * a message is displayed on the LCD screen identifying a long distance call;
  * a timer starts in order to enable the computation of charges;
  * the 8-segment display is used to show the ticking of seconds, with wraparound (that is, it counts 1 to 9 to 1 to 9 continuously);
- When the “HANG-UP” blue button “1” on the keyboard is pushed, the outgoing long distance call is officially terminated and:
  * both LED lights are on together;
  * the 8-segment displays “0”;
  * the cost is computed at a rate of $3 per second, and a message is displayed with the total amount;
  * the system returns to the Idle State after a delay of 5 seconds.

4. The Physical Components: the peripherals to be programmed and their output

The Inputs and Outputs available for signals to commence or terminate events are best described as they appear on the ARMSim# Board view and on the board itself, as shown in Figure 6. The explanation and the code and examples on how to program the components can be found in the supporting documentation about SWI Codes for ARMSim# and in the tutorials in the lab sessions. There are 5 peripheral components which need to be programmed to support the simulation and their diagrammatic view is shown in Figure 6. They are:

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

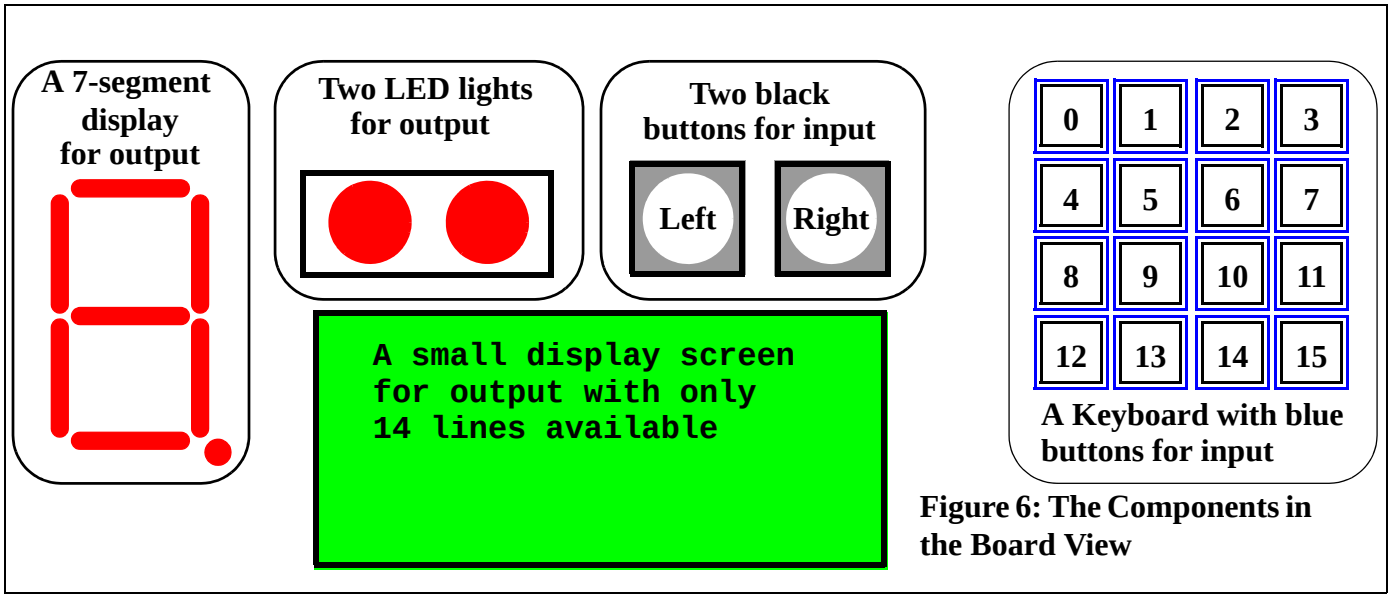

Figure 6: The Components in the Board View

4.1. The 8-Segment Display

- The 8-Segment display shows the passage of time as phone calls are taking place or while waiting for a signal by counting up every second.
- The display must be programmed to show a number from 0 to 9 and the count wraps around.

- The “Point” segment at the bottom right corner is turned ON (lit) by itself only at initialization, before any event has been detected, and is off otherwise.
- The code to display numbers in the 8-Segment Display is available in the documentation for ARM-Sim# and sample code is given in the initial template.

4.2. The LEDs

- The two red LED lights indicate the type of call.
- Left LED = ON $\Rightarrow$ Outgoing local call is taking place (extra choice: make it blinking).
- Right LED = ON or blinking right and left alternatively $\Rightarrow$ Outgoing long distance call is taking place.
- Both LEDs ON $\Rightarrow$ (Incoming call before answer) or (outgoing calls before numbers are dialed) or (all calls after hang-up while charges are computed).
- Both LEDs blinking ON/OFF $\Rightarrow$ Incoming call is taking place.
- No LEDs ON $\Rightarrow$ Idle State.
- The blinking time interval is your choice. Not too fast, not too slow.

4.3. The Black Buttons

- The two black buttons control the start and end of calls.
- Pressing the left black button starts a possible Incoming call.
- Pressing the right black button starts a possible Outgoing call.

4.4. The Blue Buttons

- Only a subset of all blue buttons have any effect, the others are ignored.[1]
- The blue button “14” starts the overall simulation program.
- The blue button “15” ends the overall simulation program.
- The blue button “0” denotes the actual beginning of a call (accepting an incoming or starting an outgoing).
- The blue button “1” ends a call.
- The blue button “1”, after an outgoing call request has been detected, implies a long distance call.
- The blue buttons “4,5,6,7” in some sequence are the dialling for an outgoing call.

4.5. 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 7 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. This part can be used to exercise your creative side.

4.5.1. Displaying Messages

- When executing the program on the ARMSim# simulator, messages can be displayed on the console as done previously. They can also be displayed on the virtual small LCD green screen on the *board view.*
- When executing on the board, messages must be displayed on the LCD screen, which only allows 14 lines of output, each line with a maximum of 40 characters. Moreover, there is no scrolling, no automatic overwriting of characters, no wraparound of text.
- Each line of display must be handled separately, given a text string or an integer plus the line number as parameters.
- On the LCD screen, the following protocol should be followed for this program:
  - Lines 1 and 2 are used to display the student number and name and whatever else you deem appropriate. This never changes after initialization.

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

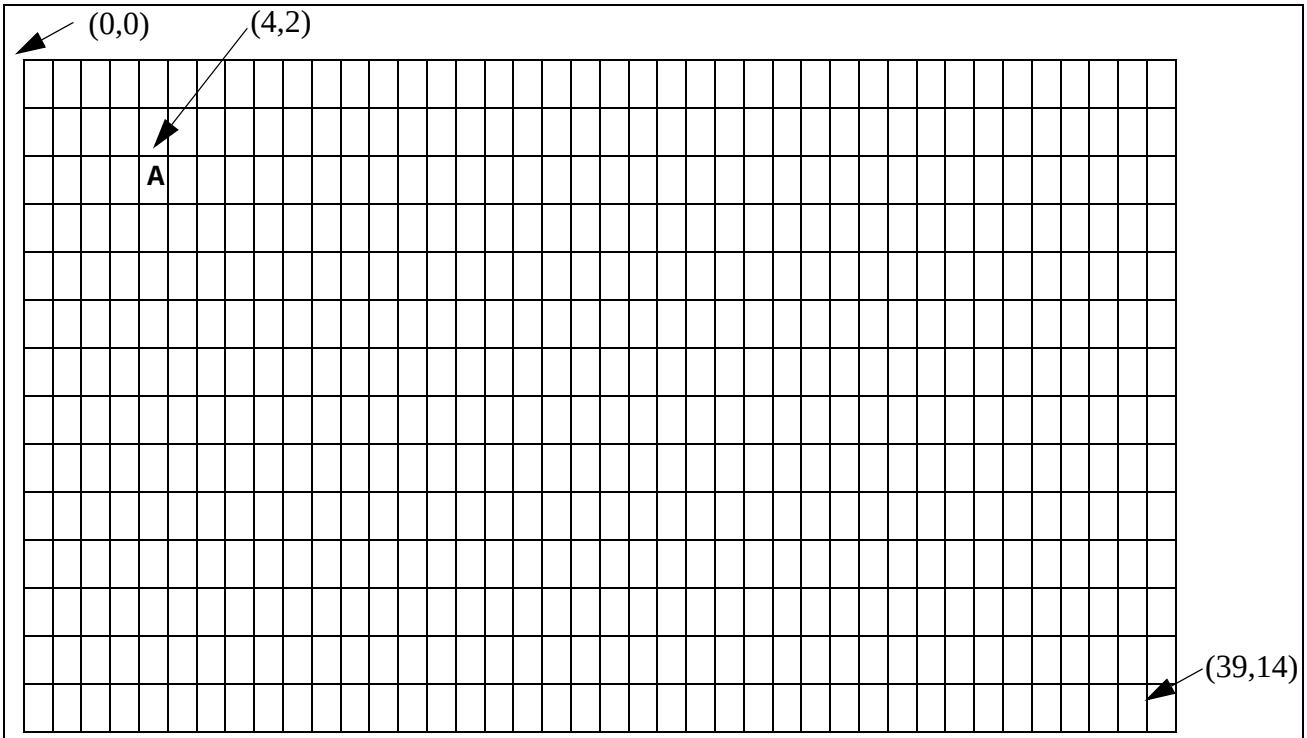

Figure 7: The LCD screen grid

- The other lines are used to display various messages as expected for the functionality, but you are free to place them as you wish.
- The LCD screen should leave the display as is (i.e. from the previous event) while in the Idle State.
- The LCD screen should be cleared and made ready for new displays as soon as a new event is enacted. In this context, enacted means when the call actually takes place (accepting an incoming call or dialing an outgoing call) and not while pending. However lines 1 and 2 with the identification should always be displayed, thus if the whole screen is cleared then it should be rewritten.
- Feel free to be more creative and enhance the display if you wish, as long as you maintain the same information content.

• It is probably a good idea to be able to print blank lines in order to clear a line in the display (a strong hint).

5. The Structure of the Program

The program *cannot be* just one main routine. The final structure will be the result of your design decisions and will be evaluated accordingly. 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.

5.1. 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. Many of them are given in the template file provided.

Which = WaitForBlueStartStop( ); [See Figure 1.]

It checks if one of the blue buttons is pressed to start the program (turn phone on) or exit immediately. It returns R0 = 0 when the Blue = 14 is pressed (turn phone on); it returns R0 = 1 when the Blue = 15 is pressed (exit). While any other button is pressed or none, the program waits here in a loop for an event.

Which = WaitForBlackBlue( ); [See Figure 1.]

It checks if one of the black buttons is pressed to start a call or a blue one to exit immediately. It returns R0 = 0 when the blue button = 15 is pressed (exit); it returns R0 = left or right Black button pattern when a black button has been pressed. While any other button is pressed or none, the program waits here in a loop for an event.

Init( ); Initialize all peripherals after the program starts (after Blue = 14 has been pressed).

ClearAll( ); Clear all peripheral after the program ends (5 seconds after Blue = 15 has been pressed).

int InCall( ); [See Figure 2.]

It takes care of all tasks related to an incoming call. It should call other sub functions to handle all the steps.

int OutCall( ); [See Figures 3, 4 and 5.]

It takes care of all tasks related to an incoming call. It should call other sub functions to handle all the steps.

Which = WaitInCall( ); [See Figure 2.]

It keeps checking, for 5 seconds, for Blue = 0 to be pressed. When it is pressed it calls other functions to implement all the steps for the incoming call functionality. When all is finished with a complete incoming call, it returns R0 =0. If Blue = 0 has not been pressed within 5 seconds, it returns R0 = 1. If Blue = 15 is pressed, it returns R0 = -1 to exit the program (eventually). This should call at least a *WaitAndPoll* function (see below). It should call other sub functions to handle all the steps.

Which = WaitOutCall( ); [See Figure 3.]

It keeps checking, for 5 seconds, for a blue button = 1, 4, 5 6, 7 to be pressed to start the outgoing call. When they are pressed it calls other functions to implement all the steps for the outgoing call functionalities. When all is finished with a complete outgoing call, it returns R0 =0. If no such blue button has been pressed within 5 seconds, it returns R0 = 1. If Blue = 15 is pressed, it returns R0 = -1 to exit the program (eventually). This should call at least a *WaitAndPoll* function (see below). It should call other sub functions to handle all the steps.

Display8Segment(NumberToDisplay:R0);

It displays an integer, given in R0, on the 8-segment display.

SetLEDs(Which:R0);

It controls the LED lights using the two rightmost bits in R0, that is, bits in positions 0 and 1, which are cleared to 0 when both lights are off. The left LED light is on if bit 1 is set to 1, the right LED light is on if bit 0 is set to 1.

Seconds = Counter( );

It returns in R0 the number of seconds elapsed for any call.

Cost = ComputeCharges(Seconds,Rate);

Given the number of seconds elapsed for a call and the rate, it returns the total cost in R0.

ClearLineLCD(LineNumber:R0);

It blanks out a line on the LCD screen, where the line number is given in R0.

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; otherwise appropriate values to denote which other event (i.e. which button press) has occurred interrupting the delay. More than one such functions might be written, customized for different contexts if desired.

Note: In our solution more choices were made of further subroutines (functions or procedures) useful for the structure of the program. Make sure you document such decisions and write subroutines to complement the ones listed here (for which you have no choice).

5.2. 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)!

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 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 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 8.

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.

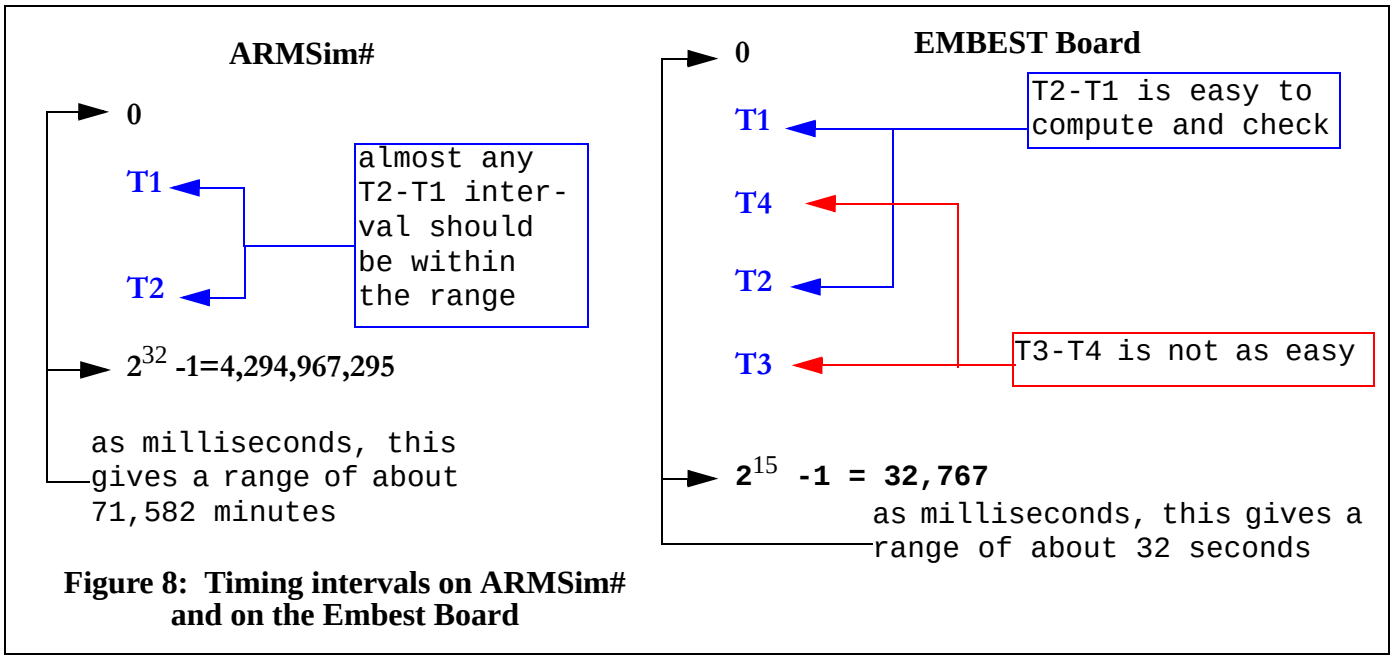

Figure 8: Timing intervals on ARMSim# and on the Embest Board

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 9 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 *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.

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

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 9.

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 waiting for a maximum of 5 seconds to see whether an incoming call is accepted. One cannot simply program a delay of 5 seconds and then check whether the blue button = 0 has been pressed (or the blue button = 15 to exit), as it would be an unreasonable delay for a user who pressed a button right away. Thus the 5 second cycle can be shortened any time by a user. In this interval one continues to poll for user buttons 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 10 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.

```
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 10: Pseudo code for a delay cycle with polling

7. Testing for a Successful Simulation

The testing scheme to be used for the evaluation (see below) will be posted and available. Note that you will get better marks by implementing successfully one subset of the simulation instead of trying to cover all of it with errors.

Follow the testing table and use it to plan your strategy for designing and implementing your program. Once you manage the first portion the rest becomes quite similar and you can re-use code and call common functions already available. This shows you one of the first most important lessons in software engineering and design: first devise your testing strategy then design your program.

8. The Flowcharts

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 Tuesday July 10, at 14:00 in the ECS course box and will be returned to you within 48 hours 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 FSM. Use the appropriate flowchart symbols (not any design you like!).

9. 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 11 shows the call graph for the subroutines specified so far. An example of call graph from assignment 1 is also available in the lab notes. The call graph must be handed in with your printout.

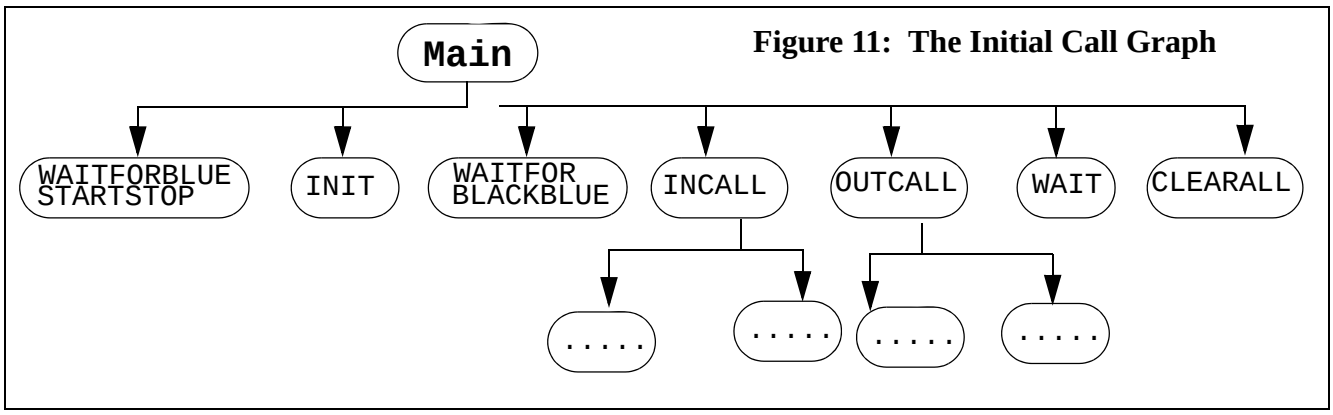

Figure 11: The Initial Call Graph

10. Evaluation [40 marks]

The evaluation will proceed as follows and will take place during the demo.

- 2 mark for the flowchart.
- 1 mark for the call graph.
- 15 marks for the code itself, as in:
  * 3 for documentation;
  * 2 for using an appropriate number of functions and procedures, including the specified ones;
  * 5 for avoiding side effects in subroutines (with correct use of STM and LDM) and no global variables;
  * 5 for the overall design and structure.
- 20 marks for the correct execution of test cases on the ARMSim#.

- 2 marks for the correct execution of test cases on the Embest board.

Please note that the relative weights of the marks may change if the evaluators decide on a better scheme.

11. Expected Deliverables

1. A flowchart of your program with all the details of the various subsystems. It is good to draw a series of interconnected small flowcharts, with precisely labelled connectors between them. **DUE: Tuesday July 10, at 14:00** in the ECS course box. It will be returned marked within 48 hours.
2. A call graph of your code, similar to what you have seen for the sample answer program of assignment 2 and in other examples given in the lectures or in the labs. See more examples on the web pages. **DUE: with the printout on July 18**.
3. The program as it works on the ARMSim#, called `CellSim.s`, in two parts. **DUE: Wednesday July 18, at 1:00 pm**.
   (a) electronic submission of the code, through Connex;
   (b) printout of the code in the ECS course box.

The deliverables of parts 1, 2, 3(b) and 4(b) are on paper, and should be submitted via the CSC 230 box in the ECS building. The deliverable 3(a) should be submitted electronically via Connex.

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

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

This is supposed to be a learning experience for you, plus a lot of fun to show off your good work!

- A sign-up sheet will be made available for you to select *two appointments* for a with an evaluator. Lab session times are included to minimize disruptions to schedules.
- First demo appointment in the lab, probably during lab times:
  * Your code will be tested on the ARMSim# during the first demo appointment on a lab machine. You will sit with the evaluator who will discuss with you these results. The evaluator will run the test. If there are any problems, a second test run may be given (or rescheduled for another time).
  * You will download the program to the board, show that you know how to package a project using the Embest IDE and test it with appropriate commands.
- First demo appointment in the lab, probably outside lab times:
  * Your code will be examined and evaluated. The evaluator will ask about your design choices and expect some explanation of your sequence of instructions.
  * You will have the chance to ask questions for any clarifications or feedback you wish to have.