Practical assignment of Digital Systems and Microprocessors

Course 2021-2022

Practice 2B

LSSudoku

	Login	Name
Students	arnau.sf	Arnau Sanz Froiz
	alfonso.sarro	Alfonso Sarró Larrey

	Board	Report	Grade
Delivery			

Date	

Table of Contents

Summary of the statement

For this practice we were required to develop a system that would serve as a controller to play sudokus off a Java interface.

In order to do that, we needed a 3x4 matrix keyboard and a joystick as the physical input medium and a 2x16 *LCD* screen as the output medium with the Sudoku Java software. In order to connect the physical medium and the Java program a serial connection had to be used (either over *USB to TTL* or *Bluetooth*).

The first interaction the user has with the LSudoku is with the Access Menu which allows them to Login or Register. The Login action lets the user in the Game Menu if their credentials are correct. The Register action makes the entered credentials valid in order to login with them later if desired. If Register is successful then the user is taken to the Login screen. If any of the actions is unsuccessful the user will be taken to the Access Menu again. A maximum of 8 users' credentials can be registered at any given time and they will be overwritten cyclically if a new user registers and the quota is met. Both menu options get the input by using the matrix keyboard in an *SMS* mode. Once in the Game Menu the user can choose to go into one of the following actions:

1. Play Game
2. Modify Time
3. Show Top 5 Scores
4. Logout
7. Show All Users

1. *Play Game* starts the game both in the physical system (*PIC18*) and in the *Java* program by sending the name of the user and a '\0'. Then the user can move around the Sudoku with the joystick and select the answer for the cell with the matrix keyboard while the Remaining Time is shown on the *LCD* screen. When the user finishes the sudoku (or gets frustrated and gives up), they press the asterisk key on the matrix keyboard. This will make the *LCD* screen show all the errors the user made and, finally, show the time that was left and the total score of this game. The user will press the hash key to go back to the Game Menu if desired.

2. *Modify Time* asks the user to input the time they want to have available when playing the game from then on. The *LCD* screen will show the title and the time as it's being typed through the matrix keyboard. The time will be in minutes and seconds.

3. *Show Top 5 Scores* shows, as its title implies, the top 5 scores of all times and the users that reached them one at a time in a cyclical marquee format on the *LCD* screen.

4. *Logout* option will go to the Access Menu and the user will need to login or register to go back to the Game Menu.

7. Show All Users shows, as its title implies, all of the users, using both rows in the LCD, one at a time per row in a cyclical marquee format. The logged-in user will be the first one to be shown and won't be shown again until the next cycle starts.

All options will need to be pressed with the hash key and the joystick will be used to move through the menu. Options that don't end by themselves will need to wait until the hash key is pressed in order to go back to the Game Menu.

System design

General functioning

The system is implemented following a cooperative methodology that allows for all the subsystems to work in a simulated parallel plane. This is achieved by splitting the necessary tasks in each of the subsystems into smaller quick tasks and switching between all the subsystems cyclically resulting in the aforementioned pseudo parallelism.

The subsystems are called motors and, along with the passive modules (they react but are not called every cycle like the motors), they form the ADTs. See *ADTs* for reference.

By following the cooperative methodology, our system is able to show a cyclical marquee on the screen while sending - receiving data and reacting to input by the joystick or keyboard all at the same time in practice.

The only interrupt we have this time is the Timer (*Timer0*) interrupt which is needed to have an accurate time measurement.

As a result of our design philosophy when creating the different *ADTs* (motors or not), they communicate with each other in a way that intrinsically avoids non-desired inputs or outputs at the wrong moments. P.e. the keyboard always communicates when a character is chosen but the menu logic chooses when to listen to this input medium.

We already implemented a virtually parallel set of subsystems in the last practice (*Phase 2A*) in order to have two (2) *PWM* signals working all the time but, this time, we implemented all systems to work in ‘parallel’ one with each other.

Input handling

There are two (2) different subsystems in charge of handling the 2 physical input mediums (user inputs): the *Keyboard* motor and the *Joystick* motor. See *ADTs: Keyboard, Joystick* for reference.

Both the keyboard and joystick are polled every cycle and, if some input needs to be managed, they take care of managing the input, debouncing and such letting the controller (See *ADTs: Logic* for reference) know whenever a final input has been reached (p.e. a character’s been chosen after debouncing and *SMS* are completed).

The other kind of input the system receives is through the serial connection with the Java program but it only communicates an acknowledgement when the game starts and the errors and score when the game ends making it a very situational input and not relevant to how it is handled in the grand scheme of things. It is handled directly from the controller by using the *Bluetooth* ADT. See *ADTs: Bluetooth* for reference.

Data management

The system stores some information in the *EEPROM* and in the *RAM* of the microcontroller. The choice of storage depends on if the value to store needs to stay between different games (or power-offs) or not.

Since the *RAM* is a volatile storage solution, the system uses it for variables and temporal data that contribute to the correct functioning of the program. Further details about these are given in the explanation for every ADT. See *ADTs* for reference.

For all the data we want to store 'permanently' the system uses the *EEPROM* within the *PIC18* divided into sections for every type of data.

Starting at address `0x00` we can see the first section of the *EEPROM* would be the 'control variables' consisting of two (2) bytes. The first byte (`0x00`) is the number of users that have been registered with values ranging from `0x00` to `0x08`. The second byte (`0x01`) is the address of the newest registered user with values ranging from `0x10` to `0x80` in increments of `0x10`.

The next data block we have set starts at address `0x10`. It consists of the registered users in the system. The block is split into 8 sections of 16 bytes each making each section's beginning range from `0x10` to `0x80` in increments of `0x10`. Within the 16 bytes section, we find the user's nickname and password taking up 8 bytes each. In the *EEPROM* reading shown below, the third user's nickname is stored from address `0x30` to `0x37` and their password from `0x38` to `0x3F`.

The last block consists of the top 5 scores. It starts at address `0xA0` and, even though every slot takes just 9 bytes (1 byte for the score and 8 for the user-that-achieved-it's name), the whole row of 16 bytes is reserved to simplify iteration when writing and reading since we can afford the space. So the slots start from address `0xA0` to `0xE0` in increments of `0x10`.

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00	03	30	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.0..... ....
10	41	4C	46	4F	4E	53	4F	00	31	41	00	00	00	00	00	00	ALFONSO. 1A.....
20	41	52	4E	41	55	00	00	00	31	00	00	00	00	00	00	00	ARNAU... 1.....
30	42	45	43	41	52	49	4F	53	31	00	00	00	00	00	00	00	BECARIOS 1.....
40	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
50	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
60	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
70	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
80	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
90	02	41	4C	46	4F	4E	53	4F	00	FF	FF	FF	FF	FF	FF	FF	.ALFONSO ....
A0	01	41	52	4E	41	55	00	00	00	FF	FF	FF	FF	FF	FF	FF	.ARNAU.. ....
B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....
F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	....

Screenshot of EEPROM reading after two games from two different users, having three registered users.

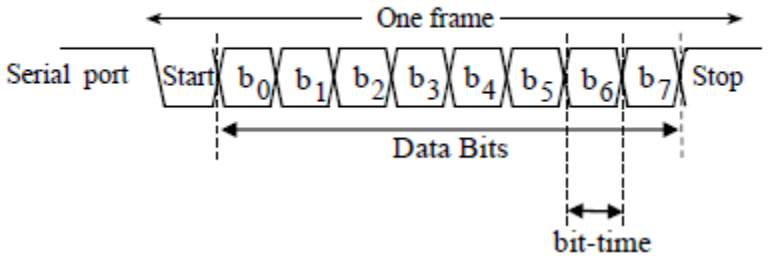
Microcontroller configuration

We have chosen to use a frequency of 40Mhz, by having a 10Mhz external crystal oscillator and setting the microcontroller oscillator configuration to *HSPLL* allowing for the 10MHz external signal (*HS*) and multiplying that by 4 (*PLL*).

```
#pragma config OSC = HSPLL
```

The reason behind this specific frequency lies mainly in the fact that we needed a high enough frequency to be able to manage a stable and accurate 1200 *baudrate* connection with the secondary *Serial* channel. In this phase, we didn't need a specific period or frequency for the *PWM* that powered the speaker since we didn't have a constraint on the specific sound, notes or frequencies it had to play making it adaptable to whatever frequencies we gave it to it as long as it did produce a human-hearable sound.

Having a stable and accurate *baudrate* is crucial because if it's not accurate even by 1% that would, very quickly, undermine the connection. The first few characters would be sent and understood by the receiver properly but, since the other party does have a stable *baudrate*, the rest would be misunderstood rendering the connection useless. A few characters every now and then would be understood correctly but the connection is as strong as the weakest point in it.



Serial protocol for sending a byte.

The process we followed was the following: we needed a 1200 *baudrate* which meant a 1200Hz signal when transmitting, translating to a 0.8333ms period. In order to get that period as accurate as possible, we thought why not use a Timer interrupt that adds 0.0001ms every time so we are sure we are nailing it. And so we did, to get a Timer interrupt of 0.0001ms = 0.1us we needed a *Tinstruction* of the same period which meant having a *Toscillator* 4 times as fast making it 0.025us. If we do the math to get the needed *Foscillator* we get 40 MegaHertz which is why we ended up using it. (Also because we already had the crystal from the last phase but that's a secret between us).

For the *Timer0*, we have chosen not to enable the pre-scaler, that way we can reach the 0.1us aforementioned with the desired *Fosc* since this time we were working cooperatively, and we don't mind the exact moment the code gets interrupted as long as time is counted accurately.

```
INTCONbits.GIE_GIEH = 1;
INTCONbits.PEIE_GIEL = 1;
INTCONbits.TMR0IE = 1;
INTCONbits.TMR0IF = 0;
T0CON = 0x08; //16 bits without Prescaler
//@ 40MHz (Tinst = 100nS), we want (0.8333ms/Tinst)=8.333 tics
//2^16-8.333=0xDF73 (0.833 bc BAUDRATE=1200)
TMR0H = 0xDF;
TMR0L = 0x73;
```

For the *BT-EUSART* we needed to configure a baud rate of 9600 so that it could be compatible with the computer. Apart from this, we also needed to enable the transmission, receiving and serial ports of the *EUSART* for it to function properly. This time we didn't use the receiving interrupt to know when we got something from the Java program. Instead, we polled the *RCIF* bit in the *PIR1* register.

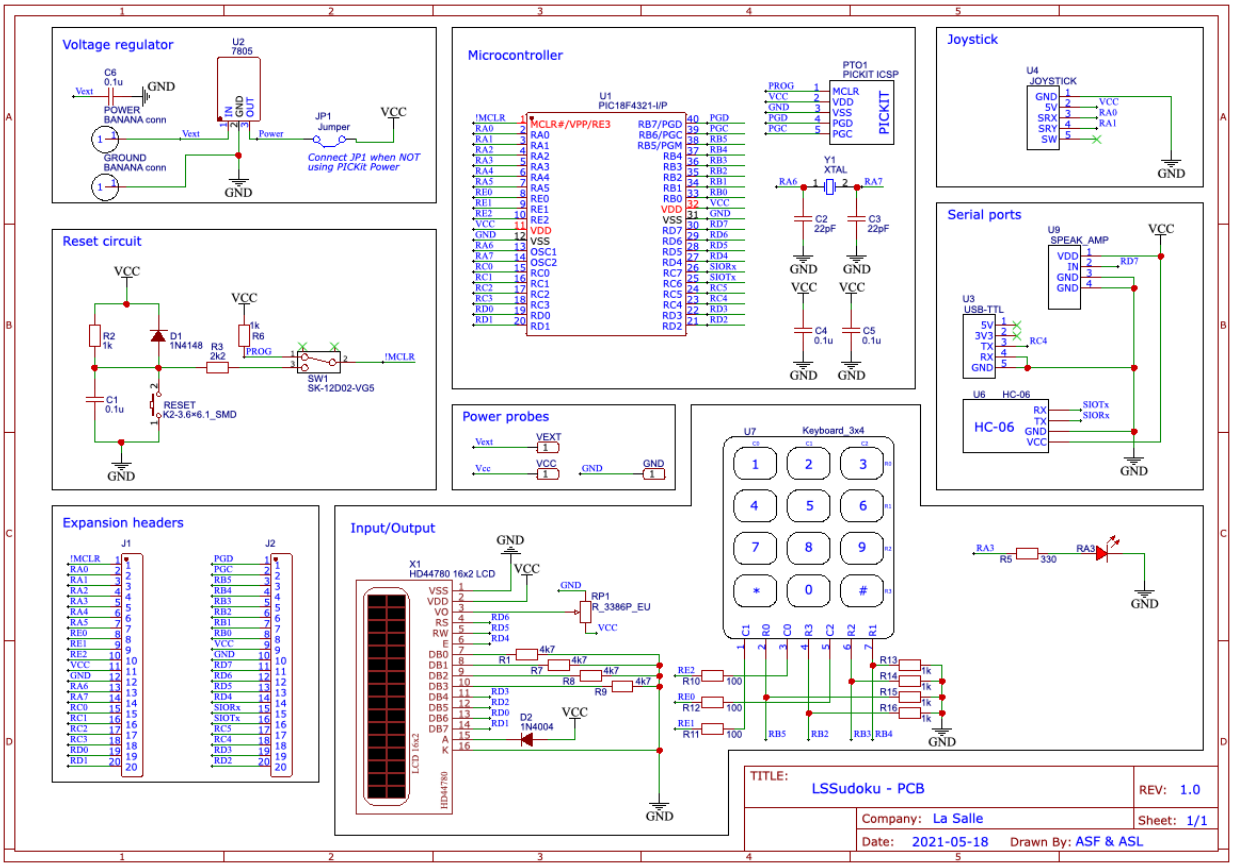
```
TRISCbits.TRISC6 = 1;
TRISCbits.TRISC7 = 1;
TXSTA = 0x20; //SYNC = 0; BRGH = 0;
RCSTA = 0x90; //RECEIVING ON
BAUDCONbits.BRG16 = 0;
SPBRG = 64;
```

We also needed to configure the ADC, so that pins *RA0* and *RA1* are interpreted as analog. Once the conversion is done we want to make it easier for us to compare its value, so we use left justification.

```
ADCON0 = 0x01; //b'00000001'
ADCON1 = 0x0C; //b'00001101'
ADCON2 = 0x89; //b'10001001'
TRISAbits.RA0 = 1;
TRISAbits.RA1 = 1;
```

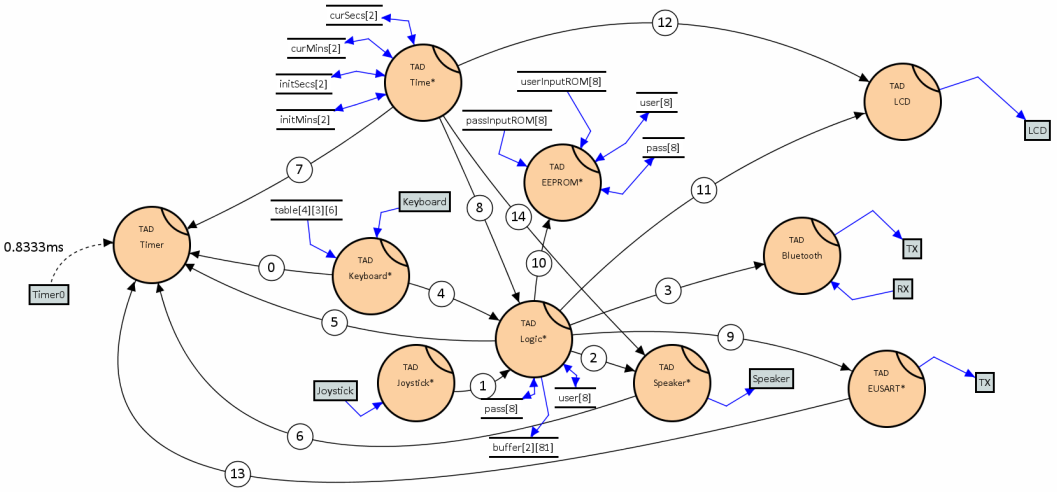
Electrical schematic

It's available also as a PDF in the delivered .zip



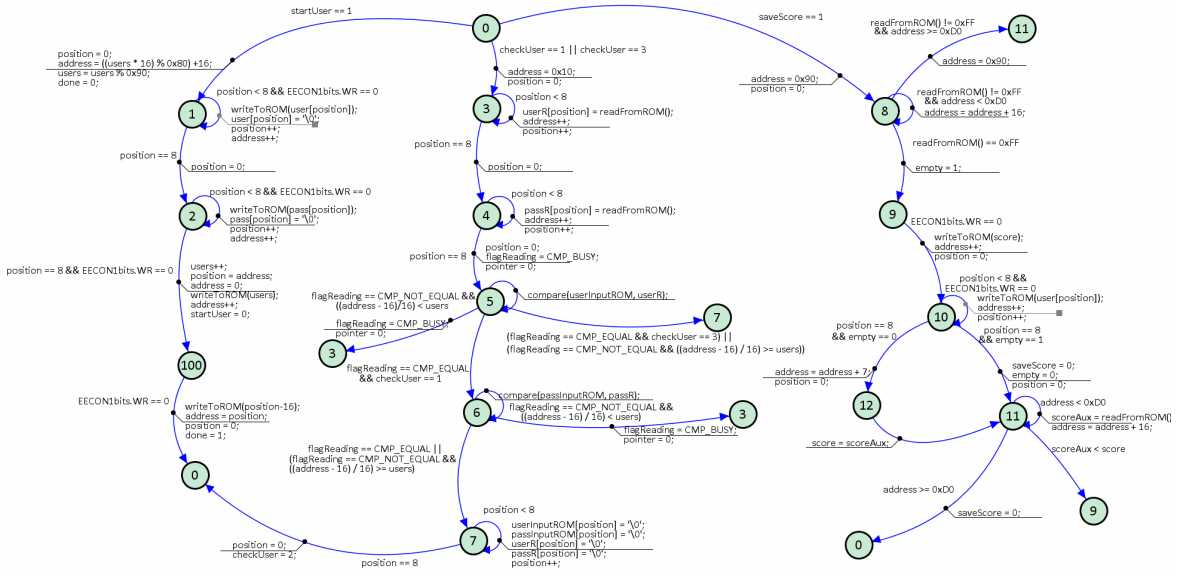
ADTs

The diagrams are also available as PNGs in the delivered .zip



This is the whole ADT diagram. We ended up using 10 ADTs, 7 of which are motors. We tried compartmentalizing the different modules as much as possible and we think we achieved a modular but efficient design.

EEPROM

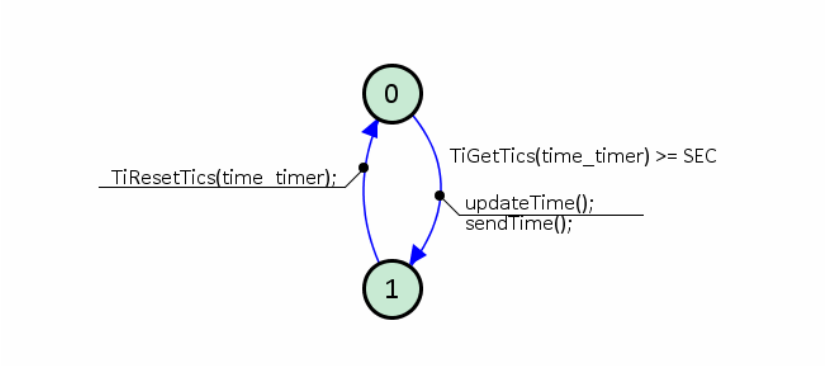


The *EEPROM* ADT handles all the permanent storage-related functionalities such as storing the users, the top scores and control variables that help the functioning of the ADT. It also handles the reading of the aforementioned data and the validation of users such as confirming a user exists and their password is correct in order for them to login or, on the contrary, confirming a user doesn't exist so they can register without any duplicates.

The ADT is split into the motor which handles the state machine, and the functions that either avoid code repetition (p.e. readFromROM()) or communicate with other ADTs (p.e. startReadingUsers()).

The state machine is divided into three sections or functionalities which consist of storing a user's information (username and password), validating a user's information for login or registration, and storing a score and the achiever's username.

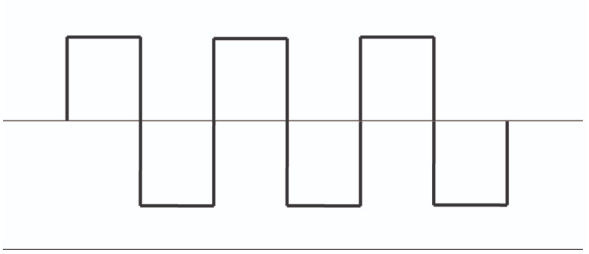
Time



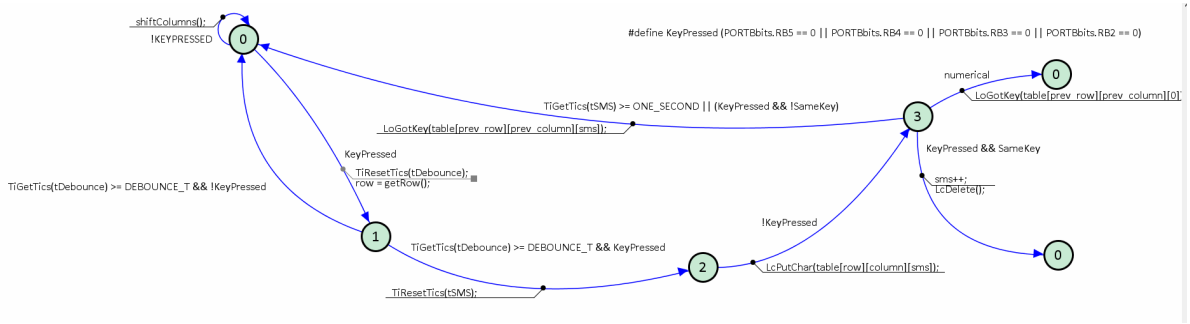
The *Time* ADT handles the time counting, sending, and showing during the playing of a game. Since it already is counting seconds and minutes it is also used to let the Speaker ADT when to start and stop playing (every minute) and when to change the playing note (every second). This is done inside the updateTime() function. It also tells the *EUSART* ADT when and what time to send over the serial connection with the terminal.

Speaker

The *Speaker* ADT handles the PWM signal sent to the amplifier which makes the speaker make a sound. It's told by the Time ADT when to start and stop playing and also when to change the note. We thought a nice way of making the 5 different notes would be just getting the tics from a *Timer* and getting one of the bits with a mask that would be shifted depending on the playing note. This produces a square symmetrical signal like in the image below which is what we want for the speaker.



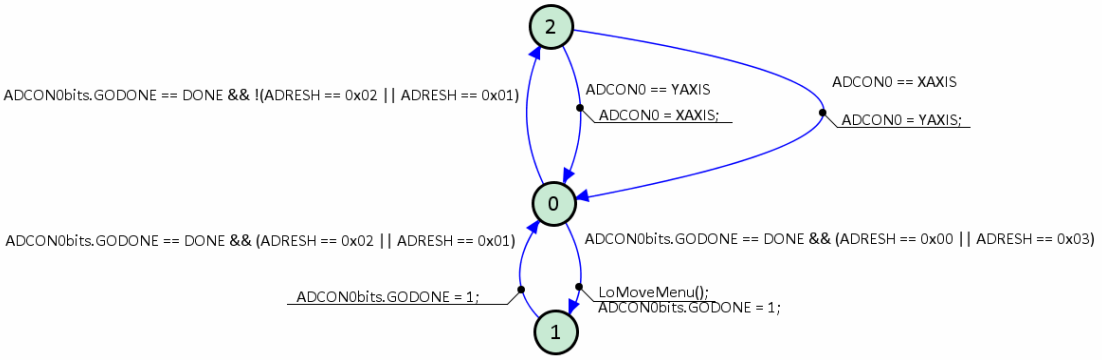
Keyboard



The *Keyboard* ADT handles all the input and showing of characters pressed on the keyboard when required. It is told by the *Logic* ADT when to be enabled or not, and when to be in *SMS* or numerical mode. When enabled, the ADT will tell the Logic ADT that a character has been selected (in the case of *SMS* mode it will only send the final character, not the whole *SMS* selection process).

The motor state machine takes care of the *SMS* looping process and the debouncing time for every keystroke it notices.

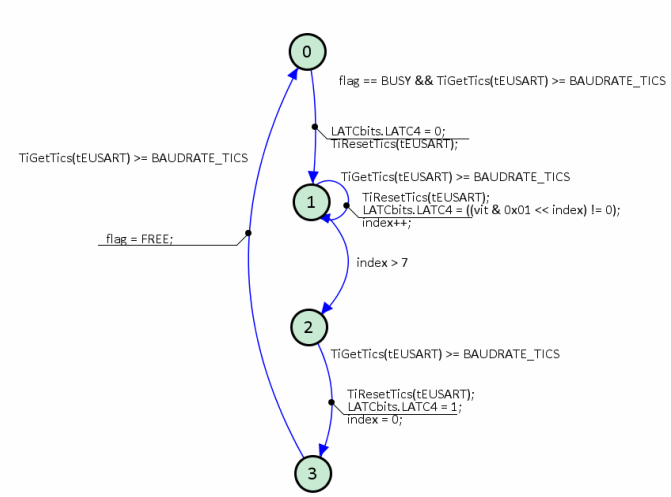
Joystick



The *Joystick* ADT handles all the movement sensed in the joystick and communicates its direction to the *Logic* ADT which, then, decides what to do with that information depending on if it's in the Game Menu or playing a game. The motor state machine implements a buffer as a virtual debouncing measure since if the user is not very fast at moving the joystick towards any given direction it can cause the joystick to see confusing results (p.e. when going up it can detect the Center position, then the Up position, then the center and up

positions again in a series due to having a rigid threshold between both positions and no margin or buffer area / position between them). See *Observed Problems* for reference.

EUSART



The *EUSART* ADT handles the sending of data from the microcontroller to the computer it's connected through the *USB2TTL* module. In the context it exists within, it is tasked with sending the remaining time every second during a game but because of how it has been implemented it could send any type of data as long as it's in blocks of 8 bytes.

LcLCD

The *LcLCD* ADT was given to us by the professors but we modified it in order to be able to print strings cooperatively and in a marquee format when needed. We have created the functions getLength(), LcPrintLine() and setLine(). Because of unplanned complications we ended up duplicating functions giving birth to getSecondLength() and LcPrintSecondLine(). See *Observed Problems* for reference.

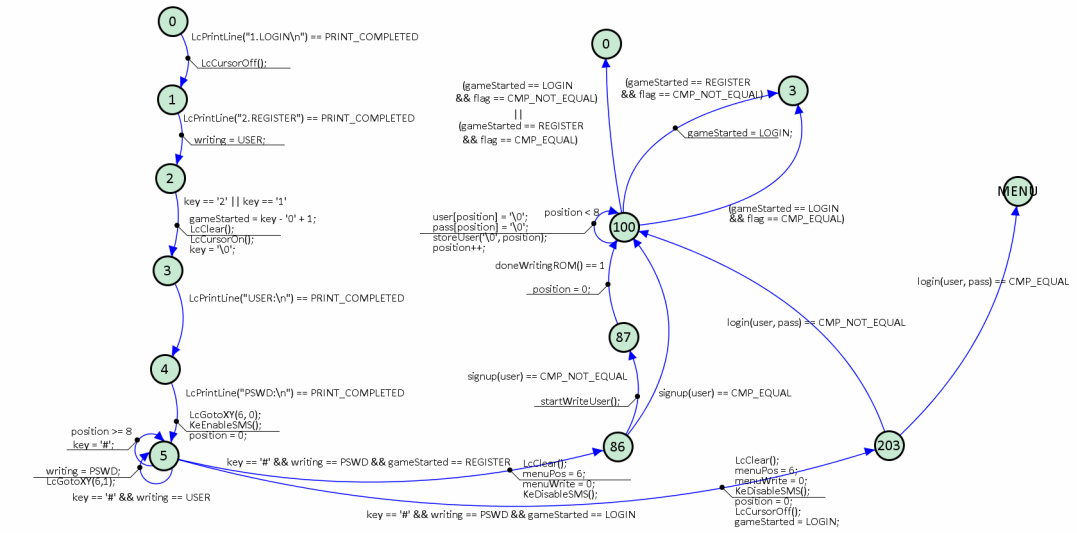
Bluetooth

The *Bluetooth* ADT handles the sending and receiving of data through a *HC-06 Bluetooth* module using the microcontroller's internal *EUSART* framework. It's the bridge between our code and the microcontroller's *EUSART* controller.

Logic

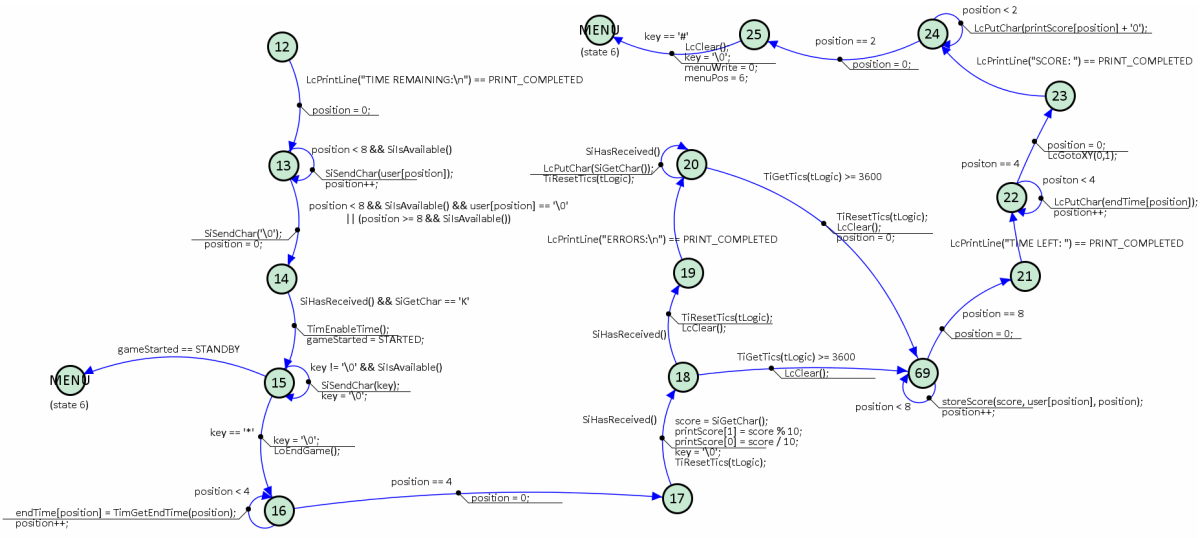
Login

The *Logic* ADT is the controller of the program. It knows where the program is at and it's what makes it go from one mode to another one. It's the bridge between every mode and takes care of enabling the ADTs that need to 'wake up' in order to do a specific thing. The motor state machine is divided into six (6) big modules which represent the available options in the menus.



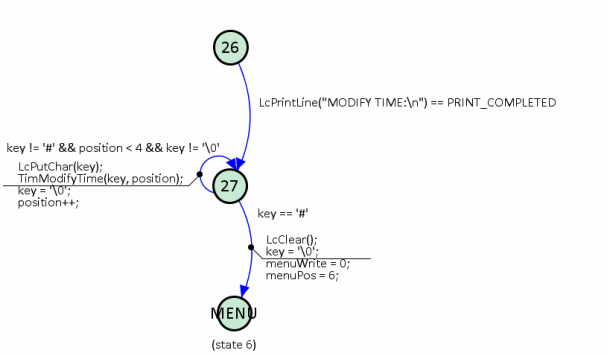
The *Login* module handles the Access Menu and its options making sure the appropriate constraints are met before storing the user data or allowing the user into the Game Menu.

Play a game



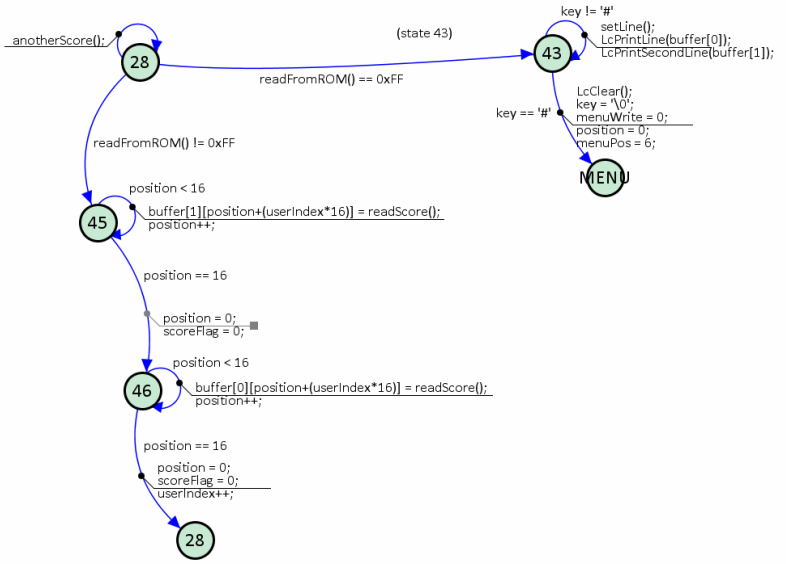
The *Play* module handles the input, output and logic for a game that is being played. It takes care of enabling the *Time* and *Speaker* ADTs, printing the remaining time, errors and score while also storing that through the *EEPROM* ADT and communicating with the Java through the *Bluetooth* ADT.

Modify time



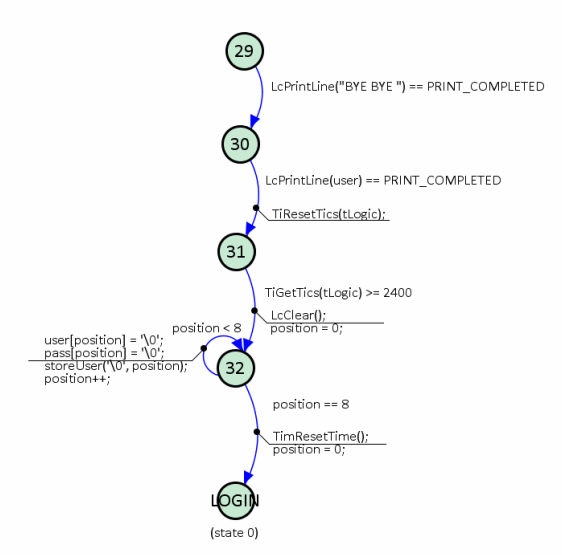
The Modify time module handles the option that allows the user to modify the starting time for the following new games. It communicates with the *Time* ADT.

Show top 5 scores



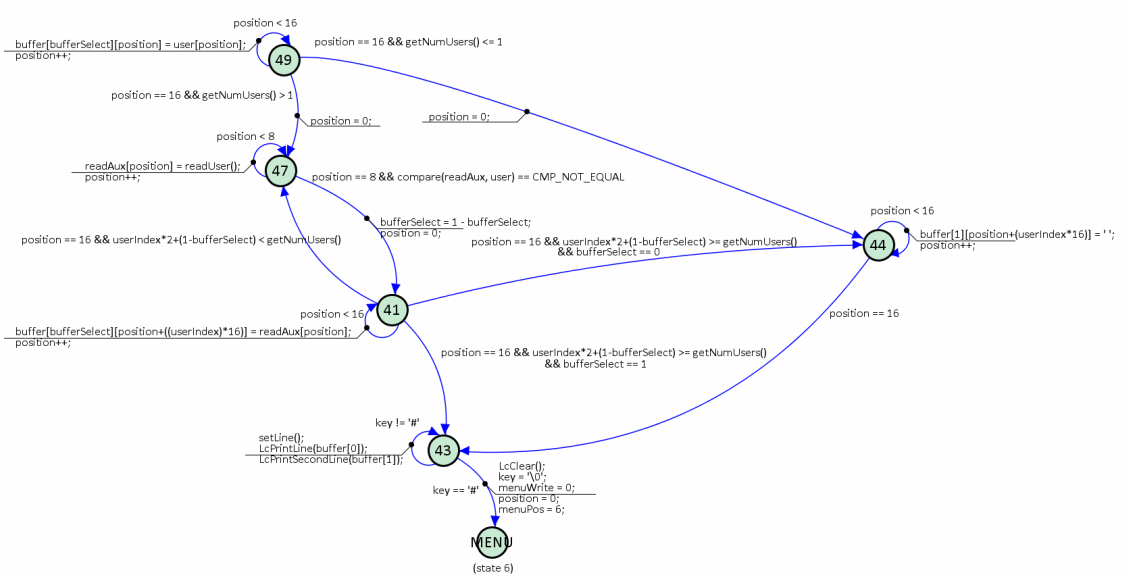
The Show Top 5 Scores module handles the showing of the top 5 scores and their achievers' usernames on the LCD screen in a marquee format if there's more than one score. It uses the *EEPROM* and *LcLCD* ADTs.

Logout



The Logout module handles the brief goodbye message to the user as they exit the Game Menu and go back to the Access Menu. It uses the *EEPROM* (to reset helper variables) and *LcLCD* ADTs.

Show all users



The Show All Users module handles the showing of all the registered users in the system by putting them on the LCD screen in order from the latest to register to the first to do so. The logged-in user will be put first regardless of when they were registered. It uses the *EEPROM* and *LcLCD* ADTs.

Observed problems

One of the biggest problems we encountered is that the *Bluetooth* module that we bought was not capable of being set to a 1200 baudrate due to firmware limitations. This forced us to use the *Bluetooth* module at a 9600 baudrate to communicate with the Java and use the *USB2TTL* module at 1200 baudrate.

Another issue we encountered is that when the joystick was moved to any direction, the joystick would detect that we had moved that same direction several times. This is due the little wiggle the joystick can experience if we move it not fast enough and us splitting the joystick axis ranges into 4 per axis (forcing us to go from the big middle zone to one of the extremes without a margin or buffer zone between them) since we only read the ADRESH for ease of use. To fix this we made a virtual buffer for the joystick to truly confirm direction movement when we were 100% sure we needed to. The virtual buffer made sure that the last ten (10) (tried different amounts and it's the lowest one that worked for us all the time) detected zones were the same waiting to be sent and it avoided the issue altogether.

We also encountered a flash problem, since it being limited by hardware forced us to optimize certain functions for the code to fit all inside 8K instructions. We are aware we could have optimized the code more than the final version shows like not having 2 functions which do the same thing but to different variables but it was the best way of fixing a problem and it worked without needing to change the whole structure of how our program communicated with the LCD controller.

Another issue we suffered had to do with the marquee. When the shifting occurred as many times as the length of the string that was being shifted the pointer went out of scope for that variable and started outputting random characters which were, in fact, the value of the following addresses in the RAM. This was solved by controlling the shift value and resetting it to 0 whenever it became equal to the length of the string.

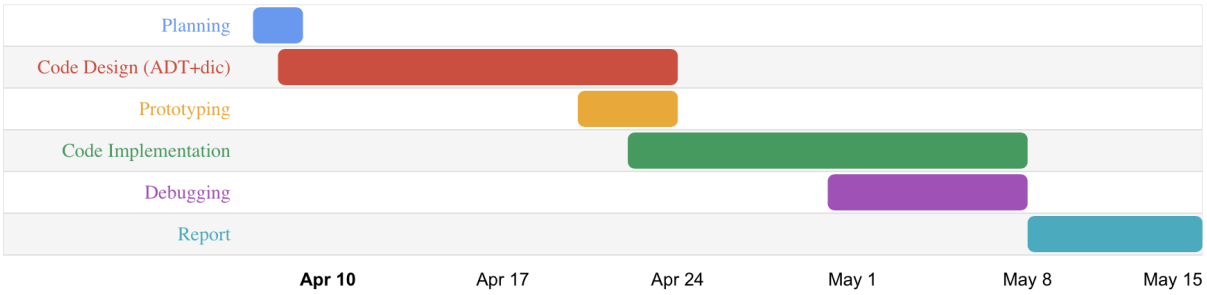
Also related to the LCD we had to tweak the original LcLCD code to add another WaitForBusy() call after the PutChar() code run to avoid making the LCD get overwhelmed if the function was called many, many times.

Last but not least, we also had a big issue with cleaning all the used variables each time a user logged out and a new user logged in since it would just plainly overwrite the variable instead of first cleaning it and, then, overwriting it. This made it so that if we logged in with the username "ARNAU", logged out and then logged in again with the username "ALFONSO" we ended up with the username "ARNAUSO". We fixed it by making sure all related variables to any given module were cleaned before usage.

Planning

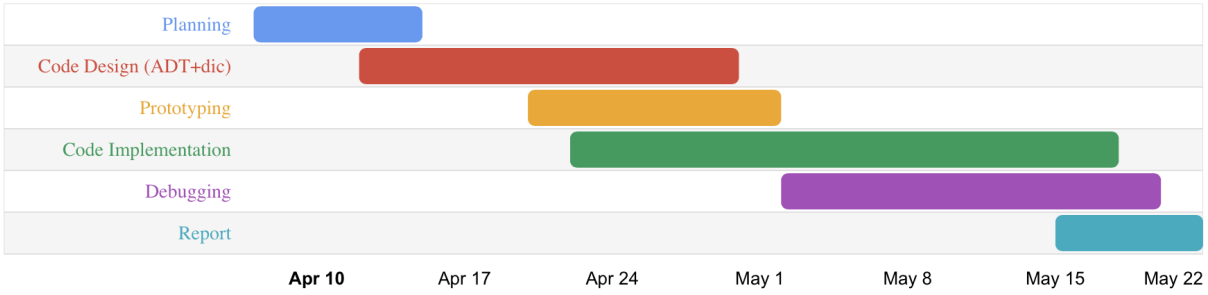
We started planning the practice right away and already came up with some ideas on how we were going to implement the main functionalities and how we were going to use the hardware we thought was appropriate. We even thought out which optional functionalities we were going to implement before the practice presentation ended.

Planned schedule:



As amazing as that sounds, we had other subjects to tend attention to and we ran up into problems (as you have already read). All of that combined with finding out that our implementation was less optimized than we would have liked, we had to spend some time optimizing it for the sake of not delivering some spaghetti code. We did have functioning code that would have gotten us through the interview, we just didn't feel that this should be the way to pass unless the optimization work hadn't ended up working. We also changed our mind about the optional and ended up implementing another one.

How the schedule turned out to be:



Conclusions

To conclude, we have seen the difference in working style compared to the first phase of the practice where we had to basically tell the microcontroller what to do literally step by step whereas now we have told the microcontroller what to do in a more conceptual, abstract way, in a higher layer. We have realized we prefer this style since it's easier to change the functioning of the system and to iterate in order to test it again making the whole workflow way smoother. We also find it easier to find errors this way since we can use advanced tools like the EEPROM reader and the Debugger within MPLab).

We think this practice reflected our improved organizational (even though Easter and Sallefest were in the middle oops) and problem-solving skills we have been working on since the beginning on the subject.

We have enjoyed way more working on this project than on the previous ones since we got way less frustrated with the mistakes we made due to the notable ease at solving them compared to having to desolder half the board and solder it again or having to write 100 lines of instructions for one simple functionality. (Also finding the mistakes has been a blessing compared to the rest of the course).