Lab#3 Embedded application design project using a RTOS (Part 1)

Leonardo Fusser (1946995)

Objectives:

- Build a multi-source-file project respecting the encapsulation principle.
- Design a vending machine from a requirement document using a RTOS.
- Use Mutex and global shared data in a RTOS environment.

Hardware: Explorer 16/32 with pic32 PIM, 1 USB cables.

To hand in

On GitHub:

- Answer to the lab's questions.
- All diagrams.
- An indented and commented C code. See software requirement below.

Multi file system and software requirement:

The following items must be respected/included:

- File headers with date, name, author, description, and version.
- A multiple file system (see Table 1).
- Use preprocessor instruction to enable/disable debugging.
- You must use names that are representative for all variables, macros, tasks, and functions.
- The software must respect the following qualities: modularity, maintainability (readability, understandability) and portability.

You must follow the following encapsulation and modularization rules:

- Each task and each ISR should have its own "C" source file.
- The "C" source files hold the public function bodies, private function bodies, private variables, mutexes, semaphores and queues.
- All variables, local functions (e.g., setter-getter, tasks), mutexes, semaphores and queues must be declared as static to make them private.
- All interface functions must be declared public (e.g., `startTaskX()`).
- Public functions that access critical sections must be mutex protected. Otherwise, they must be declared static.

For example, the following `totalCredit` variable is accessible through a public function:

```
static int totalCredit;
int getCredit(void){
    xSemaphoreTake( xCredit, portMAX_DELAY );
    int s;
    s = totalCredit; // Critical Section
    xSemaphoreGive( xCredit);
    return s;
}

void setCredit(int s){
    xSemaphoreTake( xCredit, portMAX_DELAY );
    if(s >= 0)  totalCredit =s; // Critical Section
    xSemaphoreGive( xCredit);
}
```

- All queues must be accessible through public functions only.
- Any ISR handler must be located in the same source file as its ISR.

No global variables are allowed.

Not respecting the instructions will result in a refusal and a re-submission.

Table 1: example multi file system

Files	**Content**
`initBoard.c`	Includes its own header file: `initBoard.h` Contain all functions related to initializing the board: i.e. `#pragma, initTimer(), initOscillator(), initIO(),...`
`taskX.c`	Each task includes its own header file. A private `taskX()` A public `startTasX()` to create the task. Private variables, public setter and getter functions. External global variables are prohibited.
`public.h`	Includes device header file: `xc.h` Includes all dependencies and function prototypes. Includes all common macros.
`database.c`	Contains all common data.

Also:

- The name of the source file should be the task's name (e.g., `taskPrint.c` for `taskPrint()`).
- The queue's name must be suffixed with the file holding the queue (e.g., `queuePrint`).
- The file holding a queue, a mutex or a semaphore must be the receiving task file (e.g., `taskPrint.c` holds `queuePrint` queue).

Additional information:

Any damage part will be penalized by **mark lost.**

Failure to submit diagrams will result in **10 points** lost.

The hook function must toggle a LED to assess the amount of spare processing capacity. Not enough spare processing capacity will be penalized. Use a crude loop to help visualize:

```
for( ul = 0; ul < DELAY_LOOP_COUNT; ul++ );// Reduces LED frequency
LED_HOOK ^=1;
```

Deadlines:

User interface transactions – current lab	week 3 of the project
Final product – next lab	week 5 of the project

Embedded application

By building upon your previous courses' labs, you are to create a vending machine embedded control system by respecting the requirement document.

Requirement document:

The final product is a vending machine embedded control system using Explorer 16/32.

The user interface comprises three push buttons and an LCD.

Select increment	**Try vending**	**NOT USED**	**Add a quarter**
○	○	○	○
S3	**S6**	**S5**	**S4**

The user can navigate through a selection of different soda flavours using S3.

S4 simulates putting a quarter. Each time a user press S4, a proportional credit is added.

S6 simulates the actual vending operation. When S6 is pressed, the selected soda is distributed to the user if the user has previously put enough credit in the machine.

S3 and S4 buttons must have an auto repeat functionality.

The following transaction examples will clarify the different user interface functionalities.

Examples of user interface transactions:

When the user does not interact with the push buttons the following LCD menu is provided:

```
Select items
Press S3++
```

If S3 is pressed once orange soda information is displayed:

```
Orange Price:9Q
Credit: 6Q
```

An orange soda costs 9 quarters, and the current credit is 6 quarters.

And if S3 is pressed again lemon soda information is displayed:

```
Lemon Price:7Q
Credit: 6Q
```

A lemon soda costs 7 quarters, and the current credit is 6 quarters.

For example, if S4 is pressed a quarter is added to the credit:

```
Lemon price:7Q
Credit:7Q
```

And if S4 is pressed again another quarter is added to the credit:

```
Lemon price:7Q
Credit:8Q
```

Now if S6 is pressed and if there is no more lemon soda is stock:

```
Sorry
Sold out
```

But if there is enough credit the vending operation goes through:

```
Vending …
```

The message `Vending …` must be displayed for 2 seconds.

After vending operation, the credit must be reduced proportionally. In the example the credit is reduced to 1 after the vending operation:

```
Lemon price:7Q
Credit:1Q
```

Now if S6 is pressed there is not enough credit:

```
Insuffi credits
Missing 6Q
```

After 3 second of inactivity, the display always returns to the main menu:

```
Select items
Press S3++
```

Also, at all times the number of quarters in the machine must be tallied.

If too much money inserted in the machine:

```
Coin returned!
Max credit is 5$
```

Additional information:

Soda product parameters must be packed into structures within an array.

PBs must be responsive and properly debounced.

A task cannot hog the system.

The vending machine has a minimum of three items (e.g., Orange, Cherry, and Lemon).

Lab Work

Part 1: Tasking

Implement the user interface inside one task.

Notes:

- The task can be implemented using state machines.
- Also, if you prefer, you could split a task into multiple tasks to reduce code complexity.
- Draw a task diagram by downloading *task_diagram_template.vsd*
- Draw a flowchart or a state machine for the task.

Part 2: Code composing and debugging

- Write your program incrementally by regularly testing the task. To save time, you must use the simulator when debugging.
- Make sure you are logged into GitHub. Copy-paste the following URL to accept an invitation for this lab:

  https://classroom.github.com/a/SDcUoGIk

  You will get a private new repository for this lab. You will push your project to this repo at the end of this lab:

  https://github.com/Embedded-OS-Vanier/lab_vending_w22-your_name

Part 3: Test

- When the “user interface” is fully functional, give a demo in TARGET mode.
- Push it to GitHub.

Approval and demo required.

After lab questions:

1- Explain what you learned in this lab. What went wrong and what did you learn from your mistakes. Give specifics please.

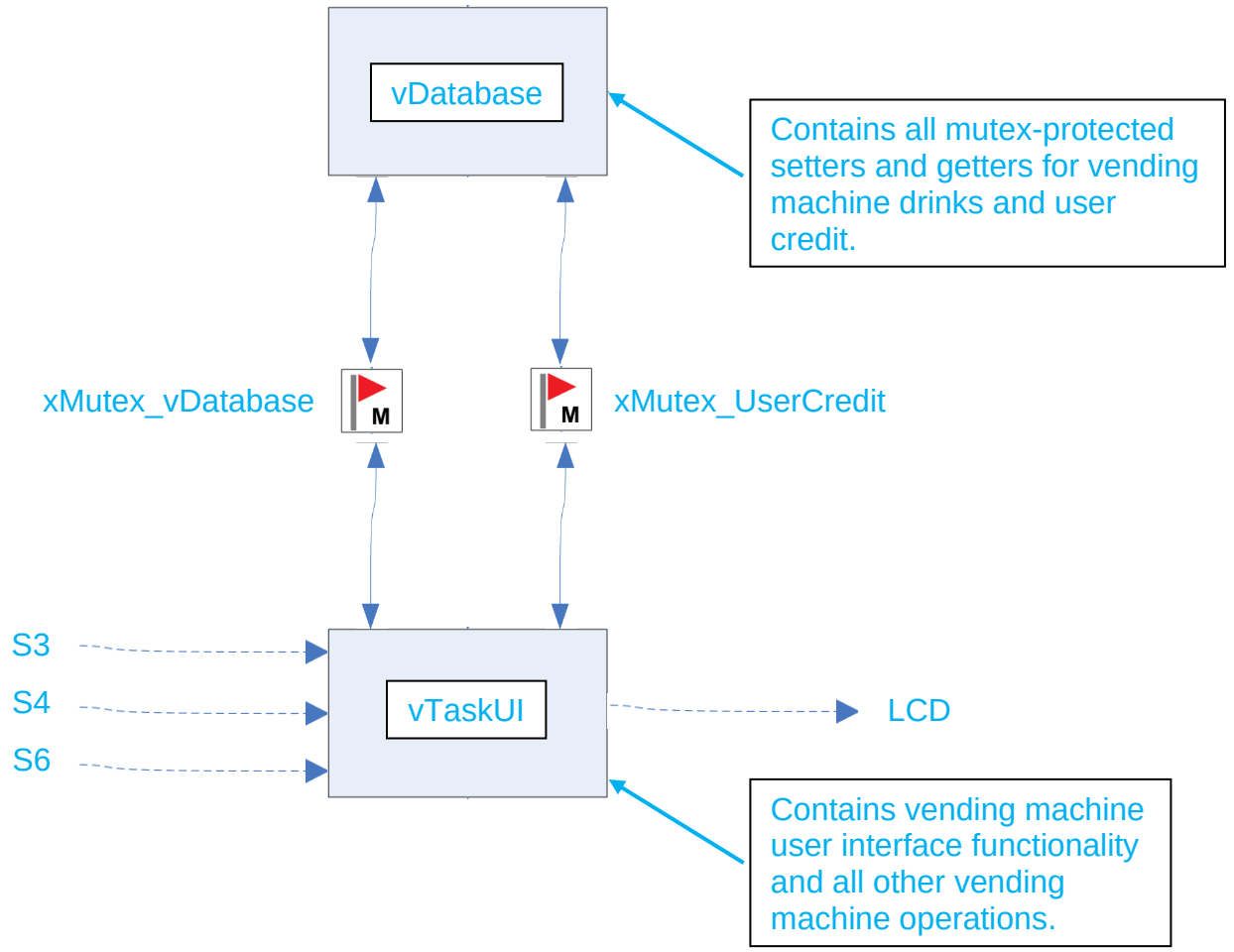
- Initially, there were issues with my vDatabase file. The purpose of this file was to hold all the mutex-protected setters and getters for the vending machine drinks and user credit. The way the vending machine drinks were initialized was done through a task routine inside the vDatabase file. The issue this caused was the task routine would run, but only once before attempting to return. The program did not work as expected, and after some debugging, it was determined that the task routine needed to be deleted after running once. Furthermore, an option in the FreeRTOS config file needed to be enabled to allow this to happen. Afterwards, the program worked as expected.

  Another issue that was encountered was inside the vTaskUI file. The main issue was that the timestamping was not done correctly. For example, after waiting for around three seconds, the LCD would still not show the vending machine main menu. Once debugging was done, the locations where timestamping needed to be done was set correctly. Afterwards, the program worked as expected.

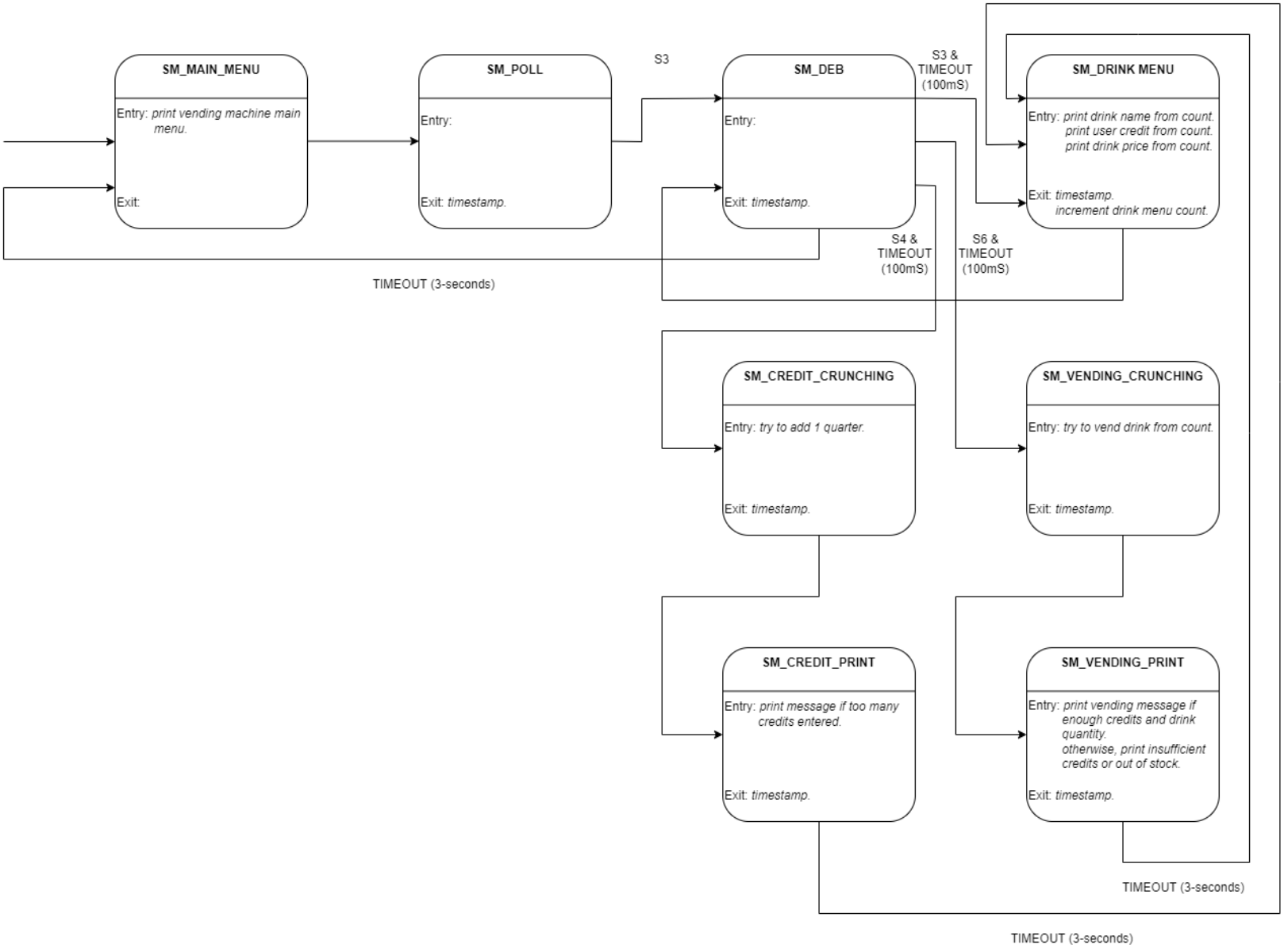
  The main takeaways here were: 1) ensure that the task routine can never return and 2) ensure that timestamping is done in the correct places of the code.

Diagrams

Task Diagram



UML State Diagram (for vTaskUI)



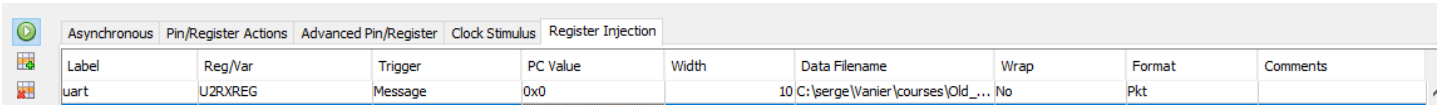
Annex

Common RTOS APIs

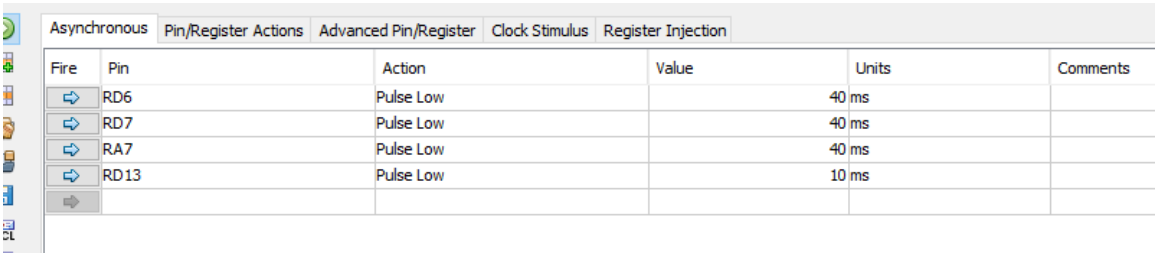
```
/******* Semaphore and mutex *****************/
Creates a mutex type semaphore. A semaphore must be explicitly created
before it can be used.
SemaphoreHandle_t xMutex;
xMutex = xSemaphoreCreateMutex();

Takes(or obtains) a semaphore that has previously been created
xSemaphoreTake(xMutex, portMAX_DELAY);

Gives(or releases) a semaphore that has previously been created
xSemaphoreGive(xMutex);

/******* Delays ********************************/
Places the task that calls vTaskDelay() into the Blocked state for a fixed
number of tick interrupts.
vTaskDelay( 100 / portTICK_RATE_MS );// 100 mS delay

/************* Queues ***********************/
Declare a variable of type QueueHandle_t.
This is used to store the handle to the queue that is accessed by all
tasks.
QueueHandle_t xMyQueue ;

Creates a queue. A queue must be explicitly created before it can be used.
The following example creates a 5-element deep queue.  Each element is the
number of bytes required for type int.
xMyQueue = xQueueCreate( 5, sizeof( int );


Send the value to the queue. The first parameter is the queue to which data
is being sent.  The second parameter is the address of the data to be sent
(e.g., the address of x). The third parameter is the Block time (e.g., 0
mS).
// Fills the queue with x without delay
xQueueSendToBack(xQueue,&x,0);

Versions of the xQueueSendToBack() API function that can be called from an
ISR. Unlike xQueueSendToBack(), the ISR safe versions do not permit a block
time to be specified.
xQueueSendFromISR( xQueueCtl, &cChar, 0 );


Receive data from the queue. The first parameter is the queue from which
data is to be received. The second parameter is the buffer or the address
of the variable into which the received data will be placed (e.g., the
address of x). The last parameter is the block time

Store the received value into y. Blocks without a timeout
xQueueReceive( xQueue,&y,portMAX_DELAY);

Blocks with a timeout
xQueueReceive( xQueue,&y, 100/portTICK_RATE_MS);
```

MPLAB X stimulus

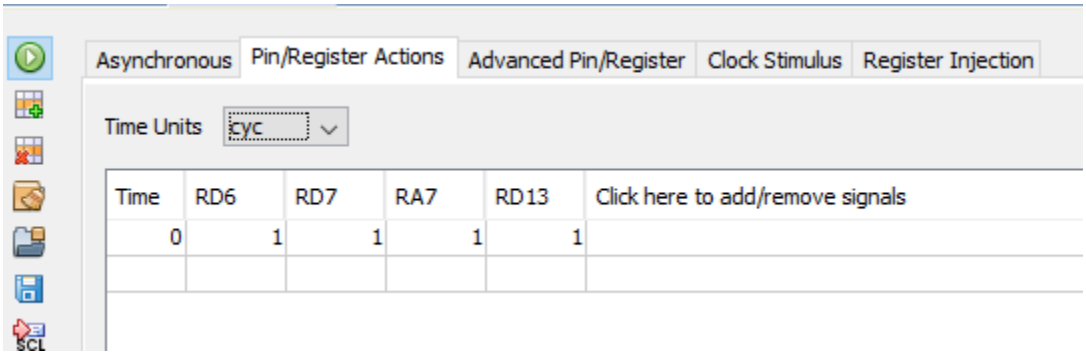
Serial reception buffer:



Asynchronous reception:



Simulating pull-up resistors:



Marking grid for lab3 and lab4

Criteria	Mark lost/gain	Comment
Deadline respect		
Encapsulation		
Comments, prolog headers		
Critical section mutex		
Use of queues		
Task priorities		
Encapsulation and modularization		
Responsiveness		
User friendly interface		
Software rules respected		
Damaged parts	-10	
Functionality		
Questions		
Versioning		
Diagrams	-10	
Spare processing capacity.		
Extra features	+10	
Modify/add features in a limited period of time		