Antonio Martí Campoy
amarti@disca.upv.es

Introduction:

The purpose of the following exercises is to improve your skills in C programming, both for embedded systems as for general purpose applications. The focus is set in modular programming and best practices coding, that is, writing programs that are predictable, and also easy to read, understand and modify.

You are going to complete the implementation of a simple videogame. This videogame is a car driving game. Several parts of the videogame, like drawing the road, the car and detecting crash, for example, are already programmed. Figure 1 and Figure 2 show images of the two screens of the game.

Figure 1. Menu screen.

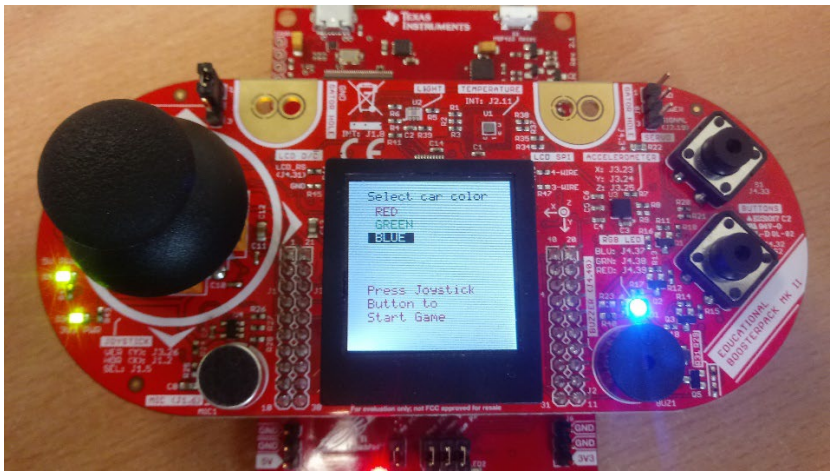
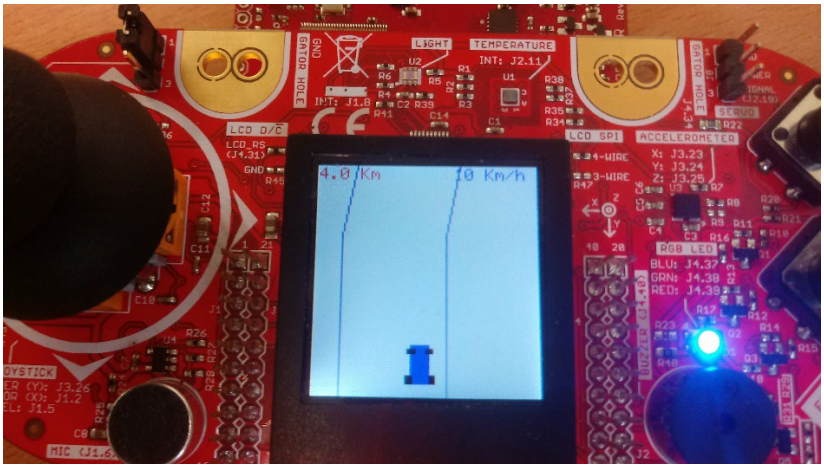


Figure 2. Game screen.

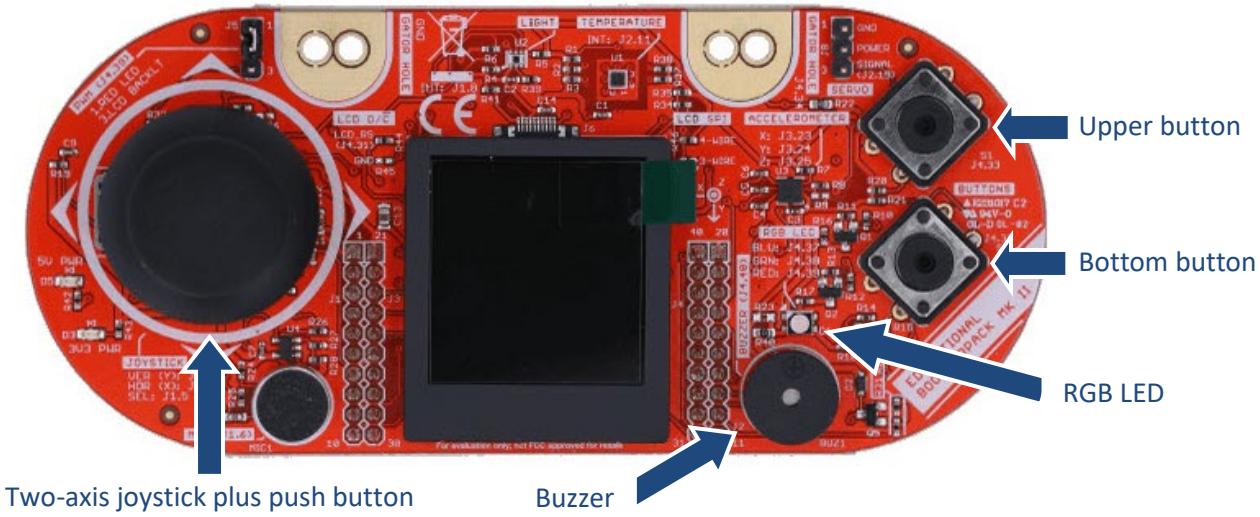


However, the control of the game and the car are still waiting to be implemented. This is your duty, write the code to allows the player to interact with the game.

Antonio Martí Campoy
amarti@disca.upv.es

You will write functions to read sensors like buttons and joystick, to drive the car. Also, you write the code to control some devices to make lights and sounds. Figure 1 shows the peripherals you are going to use.

Figure 3. MKII booster pack.



The already built part of the game is given to you as a library (game.lib), so you can call functions like *Show_Menu (option)* or *Steering_Wheel (value)*. You can find details of all available functions in the document *game_lib.pdf.*

Some tips: when starting a new exercise, **keep copy of previous exercises and functions by means of creating new functions.**

Laboratory marks come mainly from exercise evaluation in the lab. Be sure the teacher checks all your exercises.

As you can see the in Program 1 the main loop of the game is split in two parts: the menu and the game. The menu is managed in a polling loop that runs as fast as the processor can. The game is managed by a Time Triggered structure inside a loop. SysTick is used to create an event every 40 milliseconds.

First exercises are devoted to managing the menu using push buttons and an RGB led. However, you will learn how to poll, program a finite state machine (FSM), and use interrupts for inputs.

Following exercises will drive the car using buttons and joystick. It's time to play, but also to learn how to use global variables, how to NOT use global variables, and how to avoid race *conditions* because shared variables.

In last exercises you will add new behaviour to the game: feel free!

Antonio Martí Campoy
amarti@disca.upv.es

Program 1. Main function

```
void main(void)
{
    // Declaration of variables
    //WARNING: always declare variables at the very beginning of function
    uint8_t menu, menu_option, game, error;

    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;   // Stop WDT

    // Enable global interrupt and initialize
    _enable_interrupts();
    Init_Game(); //Requires interrupts

    while (1)  //Super Loop
    {
        Create_Menu();
        menu = 1;  //The menu is running
        while (menu)
        {
            error = Show_Menu(menu_option);
            if (error)
            {
                show_error("PARAM", "ERROR");
            }
            if (foo() ) //TODO: Go down the menu if bottom button is pressed
             {
                //TODO: Add 1 to menu_option taking care that there are 3 options
             }
            if (foo() )//TODO: Choose the color if the upper buttons is pressed
             {
               //TODO: Call library function Set_Car_Color. ATTENTON. You MUST check
                     the correctnes of the parameter before calling Set_Car_Color().
             }
            if (foo() )//TODO: if joystick button end menu loop and go to next loop
             {
               //TODO: How do you end the menu loop?
             }
        }
        Init_Game (); //
        game = 1; //Let's play!!!
        while (game)
        {
                if (check_SysTick_flag()) // SysTick event every 40ms
                   {
                    Draw_Road();
                    Draw_Car();
                    if (foo())//TODO: if upper button is pressed, update speed
                        {
                         //TODO: call function to increase speed one unit
                        }
                    if (foo())//TODO: if joystick button is pressed, end game loop
                        {
                        //TODO: How do you end the game loop?
                        }
                    //TODO: using the analog joystick, move car left and right
                   }
        }
    }
}
```

Antonio Martí Campoy
amarti@disca.upv.es

Exercise 1.

In this exercise you are going to work with one of the push buttons on the Booster Pack, the bottom button (see Figure 1). You will poll the state of the button using the digital GPIO pins.
The first step is to know the port and pin where the button is connected. Also, you need to know if the hardware circuit includes pull-up or pull-down resistors. All this info can be found in the following documents: MKII_start_guide.pdf, MSP-EXP432P401R_guide.pdf and MKII_users_guide.pdf.
The second step of this exercise is to create and add to your project a new module called *buttons*, with its header file *buttons.h* and the code *file buttons.c*. You can use the options of CCS File menu to create the files. When naming the files, don't forget add the proper extension .h or .c.
The third step is to implement one function to configure the button, and one function to read the state of the bottom button. The prototypes of the functions are:

```
void init_buttons (void);
uint8_t read_button_bottom (void);
```

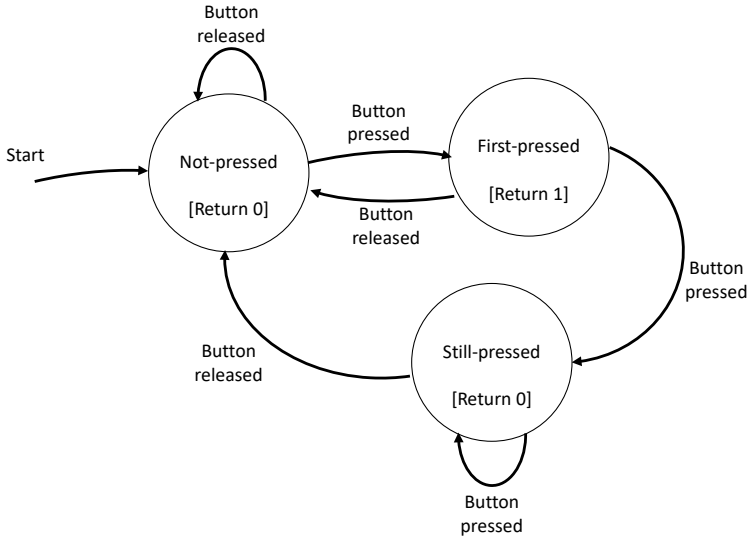
The function read_button_bottom() must return 0 if the button is not pressed and return 1 if the button is pressed.

To check your code call function init_buttons() in function main before the while(1), and call function read_button_bottom() in the first `if (foo())` . Also, in the body of the if, add the code to increment in one unit the variable menu_option.
If you get a message "PARAM ERROR" you forget that the number of options is 3. Do you know the operator %?
Don't worry if the behaviour is not very nice, we will solve It in following exercises.
Warning: Show your code to the teacher.

Exercise 2.

In the previous exercise is so hard to choose the colour because we check the state of the button faster than the user is able to press and release the button. This way you count one button activation as many.
In fact, we don't want to know the level (high or low) of the input, we want to detect the transition, that is, the edge on the input.
In this exercise we are going to solve this problem by software, programming a finite-state-machine (FSM), also known as state machine or finite automaton.
Basically, an FSM is an artefact (hardware or software) that stores a state (represented by a value) and produces an output. The output depends on the current state (Moore machine) and the new state depends on current state and current inputs.
Figure 4 (next page) shows the state diagram of the FSM we want to program. Below, an explanation.

Figure 4. State diagram



Our FSM will be a function, so the output of the state will be the returned value.
The initial state is *not-pressed*.

- If when we call the function, the state is *not-pressed* and the button is not pressed, we keep in the same state and return 0. However, if the button is pressed we change to a new state *first-pressed* and return 1.
- If when we call the function, the current state is *first-pressed* and the button is pressed, we change to a new state called *still-pressed* and return 0. If the button is released, we move to initial state *not-pressed* and return 0.
- If when we call the function, the state is *still-pressed*, and the button is pressed, we keep in this state. On the other hand, if the button is released we move to the state *not-pressed*. In both cases, we return 0.

Now, write a function button_bottom_fsm() to implement the previous FSM and call it from main (substituting previous call to read_button_bottom()). Where are you going to write this function? Yes, in module *buttons*, you are right!
Some tips for the implementation. You will need to declare the variable to store the state of the FSM as static, to keep the value between calls to the function. Use the switch-case structure, it will make easier to implement your program. And you can call your function read_button_bottom() to check the state of the button (reuse code).
Note: you still can appreciate that one activation is detected two or three times. This is due to hardware bounces. Hardware bounces can be minimised by means of software and eliminated using a low pass RC filter.
Warning: Show your code to the teacher.

Exercise 3.

In this exercise you will use interrupts to work with the bottom button. Interrupts can automatically detect the edge on an input signal, so you don't need to program a FSM.
Add to module buttons a function called *init_buttons_int* to configure the bottom button and enable its interrupt. In this function you can call your function *init_buttons* to configure the port and then configure and enable the interrupt. Don't forget to clear the interrupt flag before enabling the interrupt. And remember you need to enable both the interrupt for the pin and for the interrupt for the related port in the NVIC. In function main replace the call *init_buttons* with *init_buttons_int*.
You must add an ISR to handle the interrupt when the button is pressed. In this ISR you must read the PxIFG to check which pin has raised the interrupt and then clear the corresponding bit. PAY attention: you must clear ONLY the corresponding bit in PxIFG.
And now, the BIG question: How do you update the variable *menu_option*?
You can create a global variable called *button_bottom _flag* .and set it in the ISR. Then, in function main, check this variable instead calling *button_bottom_fsm()*. Should you clear the *button_bottom _flag*? Where?
Warning: Show your code to the teacher.

Exercise 4.

Is there a race condition in your previous code? Do you think race condition is an important hazard? Check https://en.wikipedia.org/wiki/Therac-25
In your code the race condition comes from the use of a shared variable between an ISR and function main. The way to avoid this race condition is to create an atomic section of code, that is, a code that will run with no interruptions. To create this atomic code, the interrupt is disabled before checking and clearing the variable shared between the ISR and function main. You can choose between disabling all the interrupts, o just disabling the related interrupt (pin5 of port 3). PAY ATTENTION. When you finish checking and clearing the variable, you have to restore, not simply enable, the interrupt. In order to restore the previous interrupt state, you have to save it before disabling the interrupt. Add to module buttons a function called uint8_t check_and_clear_bb_flag (void). This function must carry out the following operations: save in a variable the state of bit 5 from Interrupt Enable SFR of Port 3. Then disable the interrupt for that pin just clearing the bit. You are now in an atomic critical section, so you can copy to a local variable the value of button_ bottom _flag and, once, copied, clear it. Come out of critical section RESTORING (not just setting) the enable interrupt bit for pin 5 of Port 3. Finally, return the copy of button_bottom_ flag.
Remember to change the scope of button_bottom_flag from global to local to module and make it permanent (static).
You have to exchange the accesses to button_bottom_flag in function main by a call to check_and_clear_bb_flag.
Warning: Show your code to the teacher.

Exercise 5.

Add into buttons module the code and the functions to use the upper button and the joystick button by means of interrupts. The upper button will select the car colour, and the joystick button will finish the menu and pass to the game loop.

PAY ATTENTION. You must add two ISRs to handle the interrupts from upper and joystick buttons, **BUT** in this exercise you must use Px->IV. This way you will learn the two ways of dealing with interrupts. Remember that every time you read PxIV the register is updated, that is, you must read just once the register, so I suggest you implement a switch-case statement. In case of doubts, check related slides or ask the teacher.

Some tips:

You can configure all the GPIO pins in function init_button.

You can configure interrupts for the three buttons in function init_button_int.

You need to declare a different flag variable for each button. For example, button_upper_flag and button_joystick_flag.

You need to create a "check and clear" function to avoid race condition for each button.

Once you have implemented all the functions, use them to remove the calls to foo() in all of the if sentences, both in the menu loop and in the game loop. Check the game library documentation to known how to modify the speed of the car.

Tip: if you press the joystick button, you will enter in the game, so you could check the car colour.

Warning: Show your code to the teacher.

Exercise 6.

This is the prototype of function *Set_Car_Color*: `void Set_Car_Color (uint8_t color);`

You can see the function doesn't return any value. So, what happens if the parameter you pass to the functions is out of range? May be the result is a random colour, the absence of car or even the crash of the application.

A Best Practices requires to check the parameters passed to a library function before calling the function.

Modify your code to check the parameter is in the range [0, NUM_OPTIONS-1] and in case the parameter is out of range, call function show_error to display an error message.

Do you know what happens if you call *Set_Car_Color* with a wrong parameter? Try yourself! And let the teacher know the answer.

Warning: Show your code to the teacher.

Exercise 7.

Try to speed up and speed down the car. How do you feel with the response of the system? Yes, the acceleration is very poor because you have to press and release the button to increase/decrease the speed. It would be much better that, if you keep the buttons pressed, the speed changes continuously. It depends on the behaviour we want, we need to detect edges or level. How can you get this behaviour? Do you remember the function read_button_bottom() from exercise 1? Use this function instead check_and_clear_button_flag(). You can create a similar function for bottom button.

Warning: Show your code to the teacher.

Antonio Martí Campoy
amarti@disca.upv.es

Exercise 8.

Now it is time to use the digital outputs. The MKII booster pack include an RGB Led. Check in the technical documents to find out the pins and ports where the LED is connected. You are going to use this led in the menu interface, to signal the user the car colour chosen when the upper button is pressed.

You must create a new module called led_control with three functions. The first one called init_rgb() where you configure the GPIO pins to control the led. The second one, called led_on(colour) where you switch on the colour red related to the menu option choose by the user. And the third one, led_off(), where you switch off the three leds.

In function main, when you detect the activation of the upper button, in addition to set the car colour, switch on the corresponding led calling led_on (colour).

Add a call to led_off() when exiting of the menu loop and before entering the game loop.

Warning: Show your code to the teacher.

Exercise 9.

You will find new, exciting exercises in Execises_part2.pdf.

In these exercises you will learn to use the joystick and drive the car!!!!