Antonio Martí Campoy
amarti@disca.upv.es

Step 3. New movements

Methods and Tools for Software Quality

Introduction.

Once you are sure the hardware works, it is time to create new, a bit more complex, movements. The exercises you can find below will guide you in the creation of these new movements employing the functions you test in previous work. Carry out the exercises and record your source code and the answers to questions for your final report. Finally, if you have doubts or something behaves improperly, don't hesitate to ask teacher.

Exercise M1 (checkpoint)

The simplest movement is not moving ☺, so you are going to build the second simplest movement: the straight movement.
The objective of this exercise is to create a function that moves the robot in a straight line until a fixed number of steps/turns is reached.
You have to create a module called *blocking_movements* (remember, you need two files, one file .c with the code and another file .h where declare constants and prototypes of public functions, and don't forget the "header guards"). Within this module, create a function called *straight_move*. This function receives four parameters: first, direction of the movement (forward or reverse). Second, speed of motors in the range [0 , +100]. Third and fourth parameter, the number of turns and steps the robot must move (for example, move 10 turns and 5 steps).
The movement must be **incremental**, that is, the turns and steps passed by parameters have to be added to the actual number of turns and steps.
The function returns when distance is reached.
Don't forget to check errors in parameters and return a value to the calling function.
It would be very useful, during development stage, to have some debugging info in the display, like showing turns, steps, or any string you need in the display. This debugging code will never run in the final, deployed application, so following Rule 2.2 (required) from MISRA-C, this code must removed before compiling the final version of your program. Is there an automatic way to do that? Do you know conditional compilation?
Which encoder did you use to measure distance? Why?
Create a new project adding the library (or reuse the previous one) and add the module *blocking_move* to it.
In order to check your code works properly, from function main call several times the function ***straight_move*** with different speed, distance and direction. It would be nice that the beginning and finishing point be the same
Show your code to the teacher.

Exercise M2 (checkpoint)

The objective of this exercise is to create a non-blocking straight movement. That is, you call a function that starts to move the robot, and then you loop in main function waiting the end of the movement. This way you can execute other tasks while the robot is moving.

The main question is how to know that the movement has finished. Well, there are two options: interrupts or time-triggered polling (Time Triggered Architecture, TTA). In this case, we are going to use time-triggered polling because we can't access to a hardware interrupt that tells us that the robot must stop.

To carry-out this exercise you will use a module called *scheduler*. In this case, you have the source code of this module. In this module you will find the function *void callback_2ms(void).* This function executes every 2 milliseconds. This function just sets the variable *Tick_out*. Warning, this function is called from the ISR of one of the timers, so REMEMBER you are running with interrupts disabled and inside an ISR. You must be very careful if you want to add code into this function (like in any other ISR).

You need two new functions. The first one is a new version of *straight_move* called *nb_straight_move()*. This new function receives the same parameters than the old. This function must calculate the number of turns and steps the robot must reach and store these values in a variable of type *distance_type,* called, for example, *distance_to_stop*. Then run the motors and return, leaving the motors running.

The second function you need is called *check_stop_steps().* This function must check if the robot has reached the turns and steps to stop using the values stored in *distance_to_stop.* If the robot has to stop, stop it and return 0. In other case, leave the robot moving and return 1. Pay attention to the scope of *distance_to_stop* when you declare it.

Warning! Where would you add the new functions? In the module blocking_movements? In a new module called non_blocking? Discuss with the teacher your decision.

Finally, in function main you have to create a kind of finite state machine (fsm) to run the movements and check if they have finished. This fsm will be in an infinite loop. You will use the variable *in_movement* to check if the robot is not moving and then run a new movement or if the robot is moving and then check if it must stop. Also, variable *movement_nb* (number of movement) will store the number of the following movement to run.

Substitute your calls to *straight_move* with the following code:

```
while (1)
  {
   if (//To do: is the robot moving? ) // If robot is not moving run a new movement
   {
           switch(//To do: which movement we have to run?){
           case 0: //to do:Run non_blocking straight forward movement and update system state
                   break;
           case 1: //to do: Run non_blockin straight reverse movement and update system state
                   break;
           default:;
           }
   }
   else //The robot is moving, check if it must stop
   {
    if (Tick_out) //Time trigger It is time to check.
    {
         //To do: call function to check if it must stop and update variables
    }
   }

   //something to do at the same time the robot moves
   show_number (movement_nb);

  }// while (1)
```

Now, implement the new functions, complete function main with code and variable declaration, and show your code to the teacher. **Warning**! Check with the teacher the declaration of variable *Tick_out*.

Antonio Martí Campoy
amarti@disca.upv.es

Exercise M3 (checkpoint)

Is there a race condition in your previous code? Do you think race condition is an important hazard? Check en.wikpedia.org/wiki/Therac-25

In your code the race condition comes from the use of a shared variable between an ISR and main function. The way to avoid this race condition is to disable related interrupt before checking and clearing the variable. Add to module *scheduler* a function called *uint8_t check_and_clear_Tick_out(void).* This function must carry out the following operations: save state of IE bit for Timer A0 (bit TAIE in 16-bits SFR TA0CL) and clear it, check Tick_out and clear it, and restore state of IE bit. The function returns 1 if Tick_out was set, 0 otherwise.

Remember to change the scope of Tick_out from global to local to module.

Exercise M4 (checkpoint)

In this exercise you are going to create a new movement that makes spin the robot, that is, rotate upon its centre. Do you guess how to achieve this behaviour? I'm sure you know.
In module *blocking_movements* create a new function called *spin_steps.* This function receives four parameters: direction of rotation (LEFT or RIGHT), speed of wheels, turns and steps the robot has to rotate (like in previous exercise). Also, add by means of conditional compilation some debugging trough the display.
The function returns when distance is reached. Don't forget to check errors in parameters and return a value to the calling function.
Write a simple code in main function to test the new function *spin_steps*. Keep a copy of your previous function main, you will use it in following exercises.
Show your code to the teacher

Exercise M5 (checkpoint)

Like in previous straight movement, you have to convert *spin_steps* in a non-blocking function. Create in module *non_blocking* a new function called *nb_spin_steps* to start the spinning movement. Do you think you need a new function to check if the robot must stop? Or you can use the previous *check_stop_steps()*? In order to check this new function you have to use the function main of exercise M3, including calls to *nb_spin_steps*.

Exercise M6

Now you have to create a function called *new_bearing.* This function receives as parameters an unsigned integer (actual bearing) and a signed integer (change of course), and returns the sum of them as unsigned integer in the range of 0 to 3599. This function will add or subtract mill grades to a bearing, allowing us to work with relative bearing instead of absolute.

For example, if actual bearing is stored in variable *actual_bearing* and we want to turn left 90 degrees, we do the following call:

target_bearing = new_bearing (actual_bearing, -900);

Which module are you going to implement this function in?

Exercise M7 (checkpoint)

Regarding the function *new_bearing* of the previous exercise, do you think is needed to check the correctness of parameters? If you want to return both the new bearing and an error code to indicate the parameters were erroneous, how can you return an error code? Propose a new prototype for the function and implement it.

Exercise M8 (checkpoint)

Now is time to create a function called *spin_bearing*. The purpose of this function is to spin the robot until it reaches the bearing indicated by the parameter. The function receives three parameters: direction of rotation (LEFT or RIGHT), speed of wheels, new bearing (in the range from 0 to 3599). Include conditional compilation to add debugging trough the display. The function should return an error code if any of the parameter is out of range. Add this function to module *blocking_move*.
Note: don't try to *exactly match* the current bearing with the target bearing. In analogue variables it is close to impossible. Instead, check if new bearing is close to the target bearing.
Think carefully if it is a good idea to divide this function in sub-functions. Discuss with the teacher.
Check function *spin_bearing* by adding to function main several calls to *spin_bearing()* with both absolute and relative bearing (using function *new_bearing()*) and different speeds and direction of spin.

Exercise M9 (checkpoint)

Again, like in previous exercises, implement a non-blocking version of *spin_bearing*. You will need two new functions, *nb_spin_bearing()* and *check_stops_bearing()* You also need some variables to store the target bearing. Where are you going to implement these functions? In module *non_blocking* or in a new module? In order to check this new function you have to use the function main of exercise M3, including calls to *nb_spin_bearing()*.

Exercise M10 (checkpoint)

Create a new module called *Turn_path* and add to it a blocking function to create a curve path. The name of the function should be *turn_steps_basic* and receives five parameters: direction of the turn (LEFT of RIGHT), speed of the outer wheel, speed of the inner wheel, turns and steps to move for the outer wheel. You can add conditional compiled debug.
Check the relationship between the speed of the wheels and the radius of the turn.

Exercise M11 (checkpoint)

Create a new function that allows to turn choosing if the movement is forward or reverse. You can, even more, you must use calls to function *turn_steps_basic.* This new function called *turn_steps_plus* receives the same parameters than *turn_steps_basic* plus the direction of the movement. Did you create in exercise M1 a pair of constants like FORWARD and REVERSE? It would have been a good idea, do you agree?

Exercise M12 (checkpoint)

Are you able to convert turn_steps_plus in a non-blocking function? Let's go!

Exercise M13 (checkpoint)

Repeat the three previous exercises but passing as parameter a target bearing instead of turns and steps. This way, the robot will stop when reach the target bearing. Name functions in a similar way than in previous exercises. Build both blocking and non-blocking movements.

Exercise M14 (checkpoint)

Do you think you can perform any movement or path combining the functions you have developed? Yes? Are you sure?
I think we forget the simplest one: stop. In future exercises you will concatenate different movements (straight forward, spin, straight reverse, turn left…) and may be very interesting and even necessary to insert some stops in this list.
In fact, you can stop the robot setting speed to zero for any of the functions you have built before. But, how do you finish the stop? Counting turns? Reaching a new bearing? If the robot doesn't move, there are no changes neither in distance nor bearing.
Think for a while. Do you know the answer? Tell it to the teacher and implement this movement.

Exercise M14 (checkpoint)

Do you think the previous exercise was easy? Create a non-blocking stop movement.

Exercise M15 (checkpoint)

In this exercise you must concatenate at least one steps-based movement, one bearing-based movement and several stop movements, all of them using non-blocking functions. Did you need to add a new variable in function main?

Exercise M16

In this exercise you have to create a new movement called *spiral_path*. Its objective is to move the robot following the shape of a spiral. You have to decide and design the parameters the function needs, if you are going to use turns or bearing or both to develop the movement, and any other related question. Show to the teacher your proposal and implement it.

There are no more movements to implement. Now, choose one of the four following options and ask the teacher how to start to work with it:

1.- Complex movements: concatenation of basic movements in a complex structure. It is a pure software problem and will make the infinite loop infinitely simpler.

2.- Straight bearing-based movement: an auto-correcting straight movement using the compass reading to automatically correct the course of the robot in a straight displacement. Do you know PID controller? You can try with a simpler approach.

3.- Adding Low Power Modes: if you have nothing to do while waiting the robot to stop, you can save power sleeping the cpu.

4.- Modifying the library: you can add new functions to the library, for example a function to measure speed of wheels.