Universidad de Barcelona

Arthur Font Gouveia 20222613

Ángel Rubio Giménez 20222484

Programación de Arquitecturas Empotradas

Proyecto Final

Barcelona

2020

Índice

1. Proyecto

1.1 Objetivos

El objetivo de este proyecto es desarrollar un movimiento autónomo por parte del robot, evitando posibles colisiones con objetos en este espacio. Para eso, el robot deberá realizar las siguientes funciones:

- Inicialmente buscar una pared.
- Recorrer la pared en un sentido determinado.
- En el caso encuentre un bloqueo, debe rodearlo y continuar el movimiento en el mismo sentido cuando sea posible.
- En el caso encuentre un camino sin salida, debe tornar o hacer marcha atrás y salir.

A fin de lograr el objetivo, disponemos de una librería de funciones para controlar robot desarrollada en la práctica anterior y conocimientos adquiridos al largo de todo el curso.

1.2 Recursos utilizados

Como el proyecto es una evolución de la práctica anterior, hemos reutilizado los siguientes recursos:

- Los módulos del robot Dynamixel AX-12 (motores, con ID1 y ID2) y AX-S1 (sensores, con ID3).
- La emulación de una UART que nos permite simular la comunicación del microcontrolador con los módulos del robot.

Además, el código ofrecido por parte del profesorado implementa un thread destinado a la emulación de los módulos dynamixel mediante un simulador, en el cual podemos simular el funcionamiento del movimiento del robot en un espacio virtual y también visualizarlo gráficamente.

1.3 Análisis y configuración de los recursos

De la práctica anterior tenemos que:

- Los módulos Dynamixel utilizan la comunicación asíncrona Half-duplex (tienen una línea para transmitir y recibir) mientras que la UART en principio es Full-duplex (tiene una línea para transmitir UCAxTXD y una para recibir UCAxRXD). Por lo tanto necesitamos decir al microcontrolador si queremos transmitir o recibir datos. Esto lo conseguiremos controlando la señal mediante el valor de pin P3.0 (DIRECTION_PORT) por medio de las funciones **Sentit_Dades_Rx_emu()** y **Sentit_Dades_Tx_emu()**.

- Para establecer comunicación segura entre los módulos del robot y la UART la función void TxUAC2_emu(byte bTxdData) hace esperar al procesador hasta que el buffer de datos esté listo para la transmisión de datos.

- Una vez disponemos de los recursos descritos arriba, debemos implementar la funciones **RxPacket()** y **TxPacket()**, que son responsables de enviar y recibir la información. Para eso, tenemos que tener claro el protocolo de comunicación entre los módulos del robot y el microcontrolador, que se basa en: el microcontrolador envía un "InstructionPacket" (formado por varios bytes) al módulo del robot y éste le contesta con un "Status Packet".

Sigue abajo la información detallada del “InstructionPacket" y del "Status Packet":

INSTRUCTION PACKET

Format dels Paquets: seqüència de bytes enviat pel microcontrolador

OXFF	0XFF	ID	LENGTH	INSTRUCTION	PARAMETER1	...	PARAMETER N	CHECK SUM

0xFF, 0xFF: Indiquen el començament d’una trama.

ID: Identificador únic de cada mòdul *Dynamixel* (entre 0x00 i 0xFD).
El identificador 0xFE és un “*Broadcasting ID*” que van a tots els mòduls (aquests no retornaran *Status Packet*)

LENGTH: El número de bytes del paquet (trama) = Nombre de paràmetres + 2

INSTRUCTION: La instrucció que se li envia al mòdul.

PARAMETER 1...N: No sempre hi ha paràmetres, però hi ha instruccions que si necessiten.

CHECK SUM: Paràmetre per detectar possibles errors del comunicació, es calcula així:

Check Sum = ~(ID + LENGTH + INSTRUCTION + PARAMETER 1 + ... + PARAMETER N)

***Figure 1**. Instruction Packet details*

STATUS PACKET

Format dels Paquets: seqüència de bytes amb que respon el mòdul

OXFF	0XFF	ID	LENGTH	ERROR	PARAMETER1	PARAMETER2	...	PARAMETER N	CHECK SUM

0xFF, 0xFF: Indiquen el començament d'una trama.

ID: Identificador del mòdul.

LENGTH: El número de bytes del paquet.

ERROR:

Bit	Name	Details
Bit 7	0	-
Bit 6	Instruction Error	Set to 1 if an undefined instruction is sent or an action instruction is sent without a Reg_Write instruction.
Bit 5	Overload Error	Set to 1 if the specified maximum torque can't control the applied load.
Bit 4	Checksum Error	Set to 1 if the checksum of the instruction packet is incorrect.
Bit 3	Range Error	Set to 1 if the instruction sent is out of the defined range.
Bit 2	Overheating Error	Set to 1 if the internal temperature of the Dynamixel unit is above the operating temperature range as defined in the control table.
Bit 1	Angle Limit Error	Set as 1 if the Goal Position is set outside of the range between CW Angle Limit and CCW Angle Limit.
Bit 0	Input Voltage Error	Set to 1 if the voltage is out of the operating voltage range as defined in the control table.

PARAMETER 1...N: Si es necessiten.

CHECK SUM: Paràmetre per detectar possibles errors del comunicació, es calcula així:

Check Sum = ~(ID + LENGTH + ERROR + PARAMETER 1 + ... + PARAMETER N)

***Figure 2**. Status Packet details*

1.4 Algoritmo de movimiento autónomo

Una vez disponemos de la configuración necesaria, trabajamos en el desarrollo del algoritmo del movimiento autónomo por parte del robot, utilizando la librería de funciones (documentada en el siguiente apartado).

A la siguiente página mostraremos y comentaremos nuestro algoritmo para controlar el movimiento del robot:

```
void start_ride(){
    uint8_t left, center, right = 0; // Modificación Arthur
    get_left_sensor(&left);
    get_front_sensor(&center);
    get_right_sensor(&right);//Coger el valor de los 3 sensores
    printf("TEST: Sensors -> (L: %d, C: %d, R: %d)\n", left, center, right);

    if((left < 100 && left > 20) || (right < 100 && right > 20)){//Comprobar si hay alguna pared cerca para seguirla
        if(center > 20){//Si se puede avanzar
            move_forward();//Avanzamos
        }else if(left <= right){//Si hay una pared mas cerca a la izquierda que a la derecha
            turn_right();//Girar a la derecha
        }else if(right < left){//Si hay una pared mas cerca a la derecha que a la izquierda
            turn_left();//Girar a la izquierda
        }
    }else if(left > 100 || right < 20){//Si hay una pared lejos a la izquierda o una muy cerca a la derecha
        turn_left();//Girar a la izquierda
    }else if(left < 20 || right > 100){//Si hay una pared lejos a la derecha o una muy cerca a la izquierda
        turn_right();//Girar a la derecha
    }else{//Si no se cumple ninguna condicion
        turn_left();//Girar a la izquierda
    }
}
```

Primero cogemos el valor de los tres sensores del robot para poder hacer cálculos con ellos, después comprobamos si hay alguna pared entre 100 y 20, si es así y no tenemos pared delante avanzaremos, en caso de que no se pueda ir hacia adelante giraremos hacia el lado contrario de la pared más cercana, pues es la que estábamos siguiendo. En el caso de que no haya ninguna pared cerca para seguir comprobaremos si hay una pared a la izquierda que esté más lejos de 100 o si nos acercamos más de 20 a la pared de la derecha, si es el caso giramos a la izquierda, después comprobamos lo mismo pero con los lados opuestos.

Por último si no se cumple ninguna de estas condiciones el robot girará a la izquierda hasta que se cumpla alguna, esto puede provocar que se quede dando vueltas en círculos si no hay ninguna pared lo suficientemente cerca.

1.5 Documentación de la librería

La librería que hemos implementado esta compuesta por diversas funciones. Primero hemos analizado las siguientes funciones para así poder entender y completar las funciones TxPacket(), que envía un paquete de datos a un módulo concreto, y RxPacket(), que recibe y guarda el paquete de respuesta de los módulos.

- Sentit_Dades_Rx_emu (): Esta función imprime por pantalla el mensaje de que se ha cambiado la direción para el sentido de recepción de datos (RX). Pero realmente debería cambiar el valor del pin P3.0 (DIRECTION_PORT) para ponerlo en el sentido de recepción de datos.

- Sentit_Dades_Tx_emu (): Esta función imprime por pantalla el mensaje de que se ha cambiado la dirección para el sentido de transmisión de datos (TX). Pero realmente debería cambiar el valor del pin P3.0 (DIRECTION_PORT) para ponerlo en el sentido de transmisión de datos.

- TxUAC2_emu (byte bTxdData**):** Esta función guarda y pone el byte que queremos enviar al registro UCA2TXBUF para poder establecer la comunicación a los módulos del robot. Luego completamos la implementación de las funciones TxPacket() y RxPacket() para la transmisión de datos, explicadas a seguir con más detalles:

- TxPacket (byte bID, byte bParameterLength, byte bInstruction, const byte * Parametros**):**

Esta función es responsable por enviar byte a byte un paquete de información que queremos que reciba el módulo con el que nos comunicamos. Este paquete que enviamos (I*nstruction Packet*), está compuesto por:

- Dos bytes que indican el comienzo de una trama (0xFF)
- Un byte con el ID del módulo con el que nos comunicamos (el ID de broadcast es el 0xFE, los motores tienen los ID de 0x01 al 0x04, y el sensor tiene el ID 0x64, pero en esta práctica hemos utilizado el ID 0x01)
- Un byte con el número total de bytes del paquete (*length*)
- Un byte con la instrucción a transmitir al módulo (0x02 es READ, 0x03 es WRITE, 0x04 es REG_WRITE y 0x05 es ACTION)
- N bytes con los parámetros necesarios para la instrucción.
- Por último, un byte con el valor referente al checksum del paquete.

- RxPacket(): Esta función es responsable por recibir un paquete de información de un módulo que responde a un Instruction Packet. Este paquete, que se llama Status Packet, permite hacer lecturas de los registros propios de los módulos y también se puede saber si ha pasado algún error en la transmisión de los datos . Este paquete de respuesta, de forma similar que los paquetes de transmisión, está compuesto por:

- Dos bytes de aviso (0xFF).
- Un byte con el ID del módulo que envía el paquete.
- Un byte con la longitud total del Status Packet (*length*)
- Un byte con el código de error.
- N bytes de parámetros adicionales.
- Por último, el byte del checksum, para comprobar que hayamos recibido el paquete correctamente y no ha ocurrido ningún error de transmisión.

Finalmente, ya tenemos implementadas las funciones necesarias para comunicarnos con el robot. Por lo tanto, para controlar el robot hemos creado la siguiente librería de funciones.

Por parte de los motores, las funciones disponibles son:

- move_forward (): Esta función escribe en los registros Moving Speed (0x20) de los motores ID1 y ID2 en el sentido debido para que el robot se mueva hacia adelante.
- move_backward (): Esta función escribe en los resgitros Moving Speed (0x20) de los motores ID1 y ID2 en el sentido debido para que el robot se mueva hacia atrás.
- turn_right (): Esta función escribe en los registros Moving Speed (0x20) de los motores ID1 y ID2 en el sentido debido para que el robot gire a la derecha.
- turn_left (): Esta función escribe en los registros Moving Speed (0x20) de los motores ID1 y ID2 en el sentido debido para que el robot gire a la izquierda
- stop_movement (): Esta función escribe en los registros Moving Speed (0x20) de los motores ID1 y ID2 para que el robot pare totalmente su movimiento.
- fast_move_forward (): Esta función escribe en los registros Moving Speed (0x20) de los motores ID1 y ID2 en el sentido debido para que el robot se mueva adelante con una velocidad elevada.

Para cada función hay otra que sirve para comprobar que los registros tienen los valores correctos para realizar el movimiento deseado.

Por parte de los sensores, las funciones disponibles son:

- get_front_sensor (): Esta función lee el sensor frontal del módulo AX-S1 del robot y imprime el valor en la pantalla LCD. Para eso, lee el valor del registro CCW Compliance Margin (0x1B).
- get_right_sensor (): Esta función lee el sensor derecho del módulo AX-S1 del robot y imprime el valor en la pantalla LCD. Para eso, lee el valor del registro CW Compliance Slope (0x1C).
- get_left_sensor (): Esta función lee el sensor izquierdo del módulo AX-S1 del robot y imprime el valor en la pantalla LCD. Para eso, lee el valor del registro CW Compliance Margin (0x1A).

- set_front_sensor (): Esta función escribe en el sensor frontal del módulo AX-S1 del robot y imprime el valor en la pantalla LCD. Para eso, lee el valor del registro CCW Compliance Margin (0x1B).
- set_right_sensor (): Esta función escribe en el sensor derecho del módulo AX-S1 del robot y imprime el valor en la pantalla LCD. Para eso, lee el valor del registro CW Compliance Slope (0x1C).

- set_left_sensor (): Esta función escribe en el sensor izquierdo del módulo AX-S1 del robot y imprime el valor en la pantalla LCD. Para eso, lee el valor del registro CW Compliance Margin (0x1A).

Para cada sensor hay también una función test que sirve para comprobar que el valor del sensor es el correcto.

1.5 Resumen del código

El código se encuentra debidamente comentado. Todavía comentaremos algunos puntos fundamentales a seguir:

El código dispone de tres threads: el main loop, el de emulación del joystick y botones y el de emulación de los módulos Dynamixel, que ahora también actúa como simulador.

En el primer thread, se inicializan las ***queues,*** se crean los demás threads y se inicia un bucle que contiene la función **start_ride()**, que es responsable por el movimiento autónomo del robot . Las queues son estructuras que permiten almacenar los datos de para comunicarse de manera segura en formato FIFO (First in, first out) circular. Para asegurarse de que apenas un thread se ejecute a la vez se utiliza un ***mutex***.

Sigue abajo un resumen del main.c :

```
/** main.c */
int main(void) {
   pthread_t tid, jid;
   // Init queue for TX/RX data
   init_queue(&q_tx);
   init_queue(&q_rx);
   // Start thread for dynamixel module emulation
   // Passing the room information to the dyn_emu thread
   pthread_create(&tid, NULL, dyn_emu, (void *) datos_habitacion);
   pthread_create(&jid, NULL, joystick_emu, (void *) &jid);
   while (!simulator_finished) {
      start_ride(); // Movimiento autónomo del robot
}
```

Dado las circunstancias actuales y el enfoque en la simulación del movimiento autónomo del robot, el thread de emulación del joystick y botones (lectura del teclado) no fue

utilizado. Todavía si dispusieramos de placa sería interesante utilizarlo a fin de agregar más funcionalidades al robot.

Por fin, el tercer thread es responsable por emular los módulos Dynamixel y el simulador. Destacaremos los puntos principales:

- Para manejar y facilitar la recepción y transmisión de datos por parte de los módulos disponemos de una máquina de estados (`fsm_state`).
- La creación de una matriz que emula la memoria de los módulos Dynamixel (`dyn_mem`).
- La función `init_movement_simulator(world)` inicia el simulador en el entorno definido anteriormente (en nuestro caso la habitación).
- El bucle principal es responsable por actualizar el simulador hasta su finalización. En el caso que tx_queue no esté vacía y rx_queue no esté llena, actualizamos la máquina de estados que implementa la recepción y transmisión de datos.

Sigue abajo un resumen del dny_emu.c :

```
/** Thread to emulate the Dynamixel communication on dyn_emu.c */
void *dyn_emu(void *vargp) {
   // Initialization of the state machine
   FSM_t fsm_state = FSM_RX__HEADER_1;
   world = (uint32_t *) vargp; // Obs: Equal to init_world()
   p = (uint8_t *) dyn_mem; // Matriz que emula la memoria de los módulos
   // Initialize the ID field of the dynamixel
   for (i = 0; i < N_DEVICES; ++i) {
       *(p + i * DYN_MAX_POS + 3) = i + 1;}
   // Add SIGTERM handler to kill the current thread
   signal(SIGTERM, handler);
   init_movement_simulator(world); // World must have been initialized
previously from main() by calling init_world()
   while (true) {
       update_movement_simulator_values(); // Actualización del simulador
       if (is_rx_state) {
           if (queue_is_empty(&q_tx)) {
               continue;
           }
       } else {
           if (queue_is_full(&q_rx)) {
               continue;
           }
```

```
        }
        // State machine to implement the data transmission/reception
        switch (fsm_state) {
        case FSM_RX__HEADER_1:
       // printf("\n Waiting for new packet\n");
       case FSM_RX__HEADER_2:
          tmp = recv_byte();
          assert(tmp == 0xFF);
          break;
        ...
        }
   }
}
```

1.6 Problemas

tenido dificultad en hacer que el robot gire exactamente el necesario y que mantenga una distancia entre 2 mm y 10 mm de la pared, una vez que muchas veces se distancia más de que debería.

Otros problemas que tuvimos fueron algunas configuraciones del simulador, como el número de pasos y el retraso de actualización.

La mayoría de los problemas fueron solucionados gracias a nuestro empeño, al material disponible en el Campus Virtual y principalmente al soporte del profesorado vía e-mail y fórum.

1.7 Conclusiones

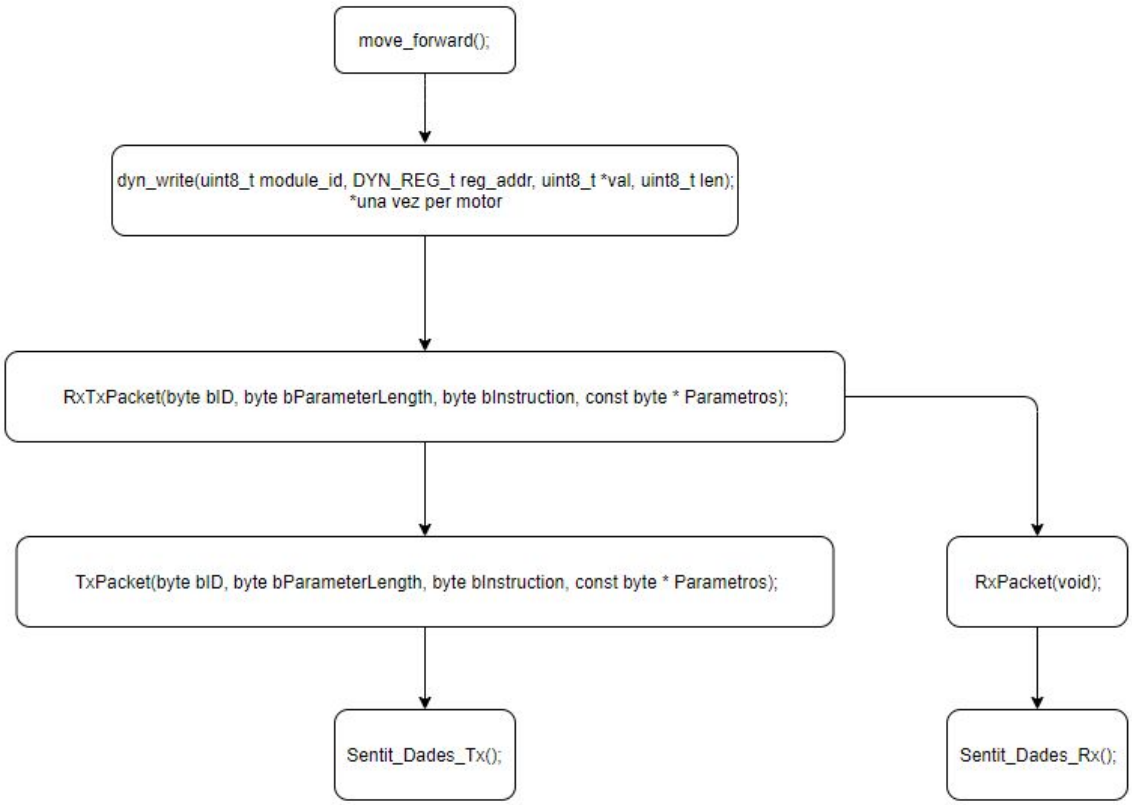
Realizar este proyecto nos ha enseñado cómo funciona un sistema planeado desde una arquitectura empotrada. Más específicamente, hemos comprendido toda la mecánica de comunicación de un robot Dynamixel, que es fundamental para la implementación de un algoritmo de movimiento autónomo por parte del robot.

Por fin conseguimos implementar un algoritmo eficaz de movimiento autónomo del robot mediante el uso del sensor AX-S1 (para obtener las distancias de los obstáculos) y del motor AX-12 (para controlar la dirección y velocidad del movimiento).

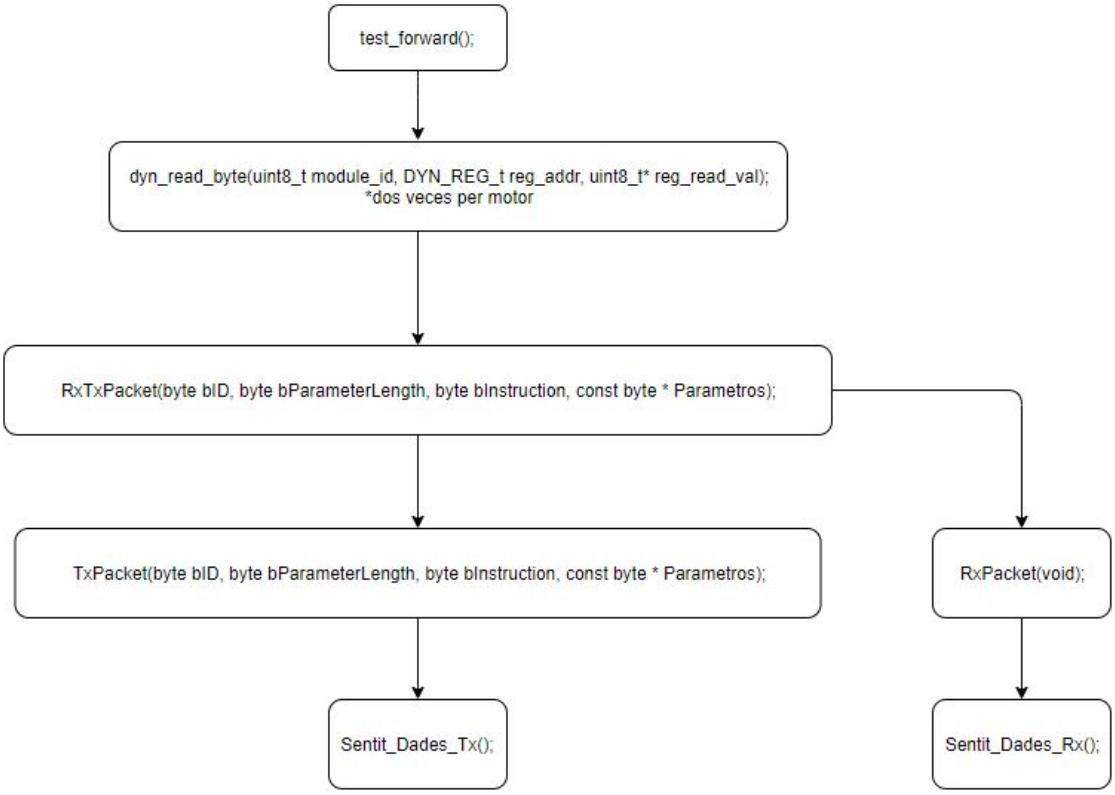
Los resultados obtenidos fueron satisfactorios y correspondientes al esfuerzo dedicado a la asignatura.

2. Diagrama de bloques

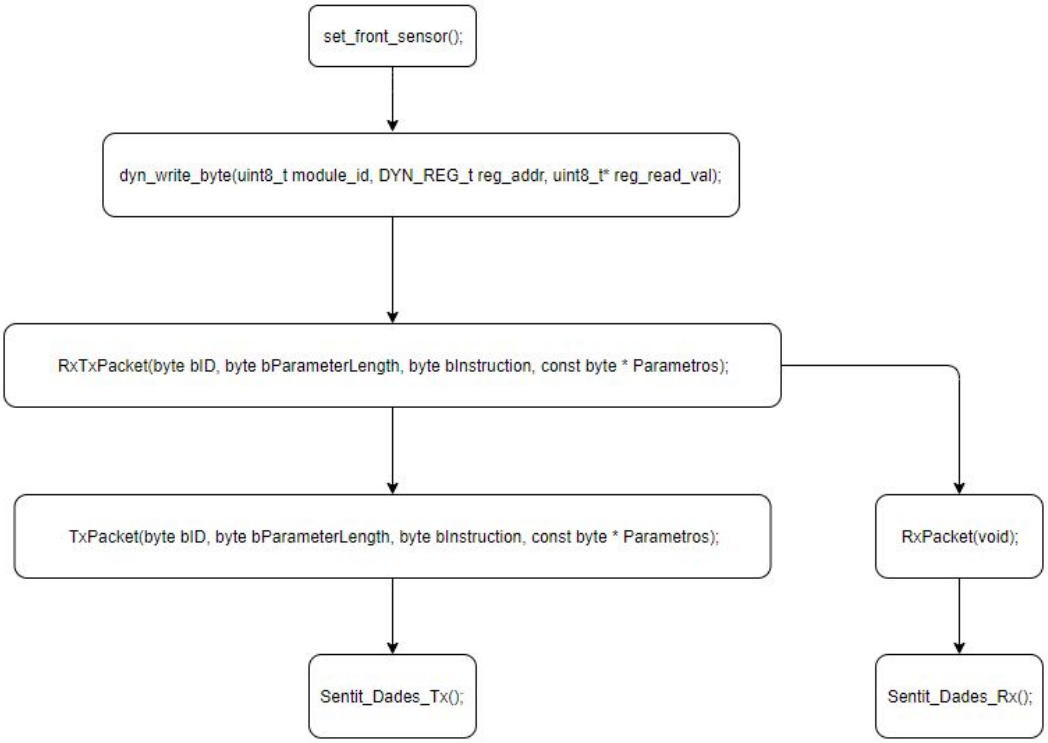
1) Ejemplo de diagrama para las funciones que hacen mover el robot.



2) Ejemplo de diagrama de las funciones que testean el movimiento del robot



3) Ejemplo de diagrama para cambiar el valor de un sensor.



El resto de funciones utilizan uno de los esquemas anteriores para escribir o leer valores de los registros pero cambiando los parámetros de entrada de la función.

4) Diagrama de flujo de nuestro algoritmo de movimiento del robot.