Eurobot 2019 manual for EE team

Sergei Vostrikov

November 5, 2018

Introduction

Eurobot is an international robotics competition for students. The team RESet of Skoltech regularly participates in the contest since 2014. Each year on a basis of competition ISP lab selects new team members from the 1^{st} year MSc students to form new team. In such conditions the key component of team's development is an effective transfer of knowledge between the generations. This document is a short guide for a STM 32 firmware that was developed by the Eurobot 2018 team. The paper is organised in a following way. In a first section the author makes an overview of used architecture to presenting breakdown structure of the firmware. In the following sections subsytems are described in a more detailed way.

1 Breakdown structure of the firmware

The breakdown structure of the firmware for STM32f4 microcontroller is represented in the figure 1.

The pattern of a superloop with interrupts is used to implement all necessary functionality. After power on all the peripheral modules are initialized at beginning of `int main()`. Then inside a "`while`" loop (***main.c***) we can observe only functions that are responsible for two tasks:

- Communication with high level.
- Global shutdown.

All other tasks are implemented inside interrupt handlers of timer modules. They are:

- Movement control.
- Collision avoidance control.
- Task execution of servo manipulators.
- Global timer update.

In the next subsections each subsystem will be shortly described and mapped to concrete files where the functionality is implemented.

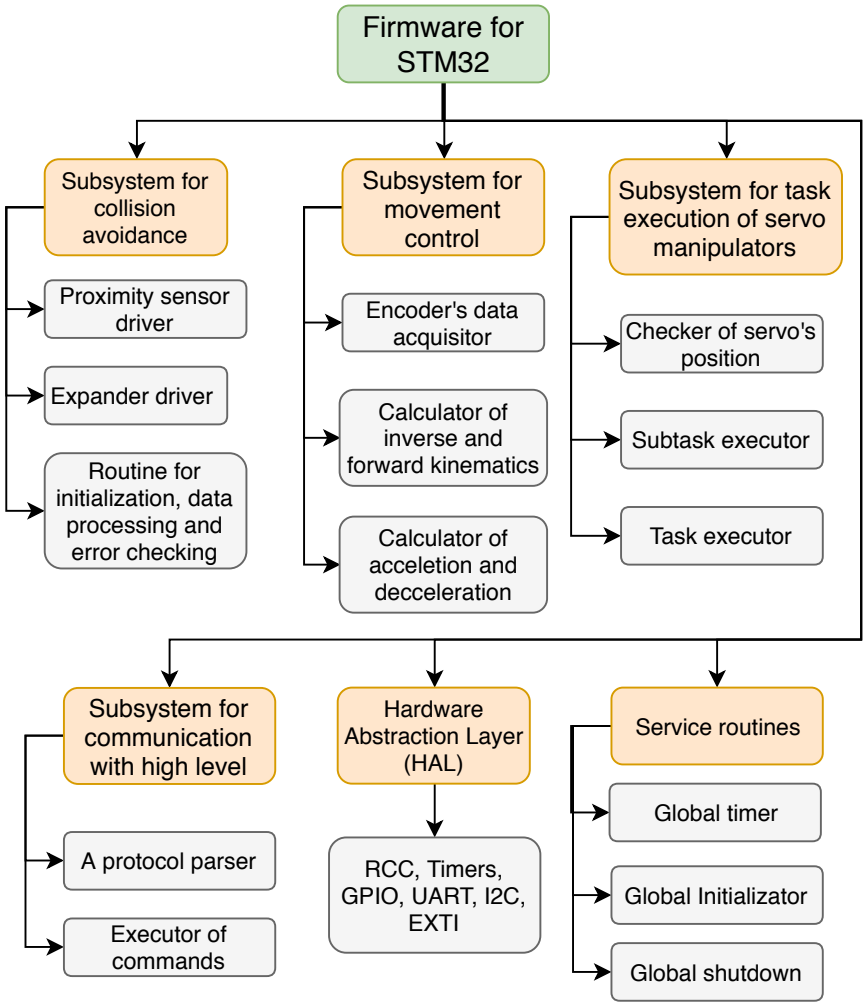

Figure 1: The breakdown structure.

1.1 Service Routines

There are basically three service routines that are depicted in the scheme 1. They supports all functionality of the firmware and will be described in details in the following paragraphs.

Global initialization. Global initialisation routine is a piece of code that performs the whole initialization of peripheral modules and sensors after a power on. To make mapping for peripheral modules more intuitive a bunch of macros are defined in ***Board.h***. Then macros are used everywhere in the firmware and especially in a function `void boardInitAll()` that is responsible for global initialization and is implemented in ***Board.c***

Global timer. Global timer is a main timer in the firmware that is used in all time-based functions. The quantization step of a timer is **1/10 of a millisecond**. Main three functions to work with global time are the following:

```
uint32_t getLocalTime(void);
uint32_t getTimeDifference(uint32_t startTime);
uint8_t checkTimeout(uint32_t startTime, uint32_t timeout);
void delayInTenthOfMs(uint16_t delay);
```

The first function returns global time in tenth of a millisecond (dimension of the output is **ms/10**). The second function return time difference between current time and `startTime`. The third function returns ***1*** if ***timeout*** was exceeded and ***0*** otherwise. The last function makes "stupid" delay for a desired time interval.

Global shutdown. This service routine is very important because it provides a predictable behaviour of the robot after the end of a match regardless of the high level code. The code of the function `void turnEverythingOff()` that is implemented in ***Robot.c*** is provided below:

```
void turnEverythingOff()
{
   // Global disable of interrupts

   __disable_irq();

   // Turn off dynamixels
   gpioPinSetLevel(SERVO_REBOOT_PORT, SERVO_REBOOT_PIN,
       GPIO_LEVEL_HIGH);

   // Turn off maxons
   uint8_t i;
   for (i = 0x00; i < ROBOT_NUMBER_OF_MOTORS; i++)
   {
      // Change PWM
      timPwmChangeDutyCycle(motorPwmCh[i].timModule,
          motorPwmCh[i].channel, 0.0f);
   }
   while (1)
   {
   }
}
```

As it can be noticed from the comments the function turn off the interrupts globally,stops all the actuators and enters infinite cycle. The firmware call the function after successful comparison between time of robot's start and macro `ROBOT_TIME_OF_MATCH_TENTH_OF_MS` inside ***main.c***.

1.2 Subsystem for collision avoidance

The collision avoidance system of the robot is based on a set of proximity sensors ***VL6180X*** from ***ST*** company. It is a time of flight (TOF) sensor that has an approximate precision (noise) of 2 mm and typical accuracy (range offset for surfaces with different reflectance) of about 13 mm. The sensor has digital I2C interface with a fixed I2C adress that can be changed after power on. As we are using a set of sensors with the same initial adress the subsystem also contains GPIO I2C expander ***MCP23017***. The expander allows to perform subsequent initialization of sensor's array and local reinitialization of particular sensor in case of an error. The maximum frequency of sensor's data acquisition for one sensor depends on the reflectance of the target, but it could be configured with constrained convergence time (see the datasheet for ***VL6180X***). In the firmware a custom method of sensor's initialization is used that allows to achieve $50Hz$ data acquisition frequency and reduce the sensitivity cone for the proximity sensors.

The source code files for ***VL6180X*** driver:

- ***VL6180x.h***
- ***VL6180x.c***

The source code files for ***MCP23017*** driver and other subroutines:

- ***Collision_ avoidance.h***
- ***Collision_ avoidance.c***

Periodical data acquisition of ranges is implemented in the interrupt handler of a timer called `COLL_AVOID_TIM_MODULE`. The code of the handler is located in ***Interrupts.c***.

1.3 Subsystem for movement control

The subsystem for movement control is responsible for encoder's data acquisition, calculation of kinematics (inverse and forward), smooth small movements with acceleration. All necessary functions are implemented in source code files:

- ***Robot.h***
- ***Robot.c***

The functions are separated into several groups. A short description of each function can be found in table 1.

Table 1: Functions of the subsystem for movement control

Function	Short description
Odometry data acquisition	
`readEnc (void)`	Reads data from encoders, calculate coordinates and speeds of each wheel, calculates instanteneous coordinates and speeds in robot's coordinate system.
`calcGlobSpeedAndCoord (void)`	Calculates robot's speed components and coordinates in global coordinate system by using speeds in robot's coordinate system.
Motor control	
`setMotorSpeed (uint8_t motorNumber, float speed)`	Sets speed of particular wheel. It is comfortable to use for diagnostics.
`setMotorSpeeds (void)`	Sets received target speeds for all motors.
Calculation of kinematics	
`calcForwardKin (void)`	Calculates forward kinematics (from speeds in robot's coordinate system to motors' speed).
`checkSaturation (float* targetSpeed)`	Check saturation of motors and change speeds if it is needed (speeds in robot's coordinate system).
`calcInverseKin (void)`	Calculates inverse kinematics (from wheels' speeds to speeds in robot's coordinate system).
Movement with acceleration	
`startMovementRobotCs1 (float* distance, float* speedAbs, float accelerationAbs[3])`	Starts short distance movement with acceleration in robot's coordinate system. Input accelerations are corrected inside the function in order to be syncronized.

`calculateTrajectParameters (uint8_t coordinateNumber, float* accelerationAbs)`	Inner function for calculation of points of accelleration and decceleration.
`syncAccelerations (float* distance, float* speedAbs, float* accelerationAbs)`	Inner function for syncronizing accelerations.
`speedRecalculation (void)`	Speed recalculation based on current speed and coordinate in robot's coordinate system.
`checkIfPositionIsReached (void)`	Function checks if we reached desired position or not and changes mode of movement when the robot reaches transition points.
`checkIfPositionIsReachedCoord (uint8_t i, float* robotCoordBuf)`	Subfunction for previous function.

An implementation of the particular function should be clear from the source code. However, a short description of movement with acceleration should be provided. Firstly, this feature is implemented to provide short and precise displacements of the robot. Therefore robot's coordinate system is used as a reference. When a high level sends a command for a short displacements, the firmware calls the function `void startMovementRobotCs1(float* distance, float* speedAbs, float accelerationAbs[3])`. Arguments of the function are distances (***x, y, angle***), absolute speeds that robot should reach after acceleration and absolute values of accelerations. Inside the function `void startMovementRobotCs1(...)` the following sequence of steps are implemented:

1. ***Normalization of absolute speeds***

   (Absolute speeds are converted into motors' speeds and scaled if the values exceed maximum speed);

2. ***Normalization of accelerations***

   (Absolute accelerations are scaled to make the time of acceleration and deceleration for all coordinates equal. It is necessary to keep the direction of movement constant during acceleration/deceleration);

3. ***Filling of the struct for odometry movement with necessary parameters***

(This step resets the coordinates in robot's system to zero, assign right signs for speed and acceleration, based on input distances. Moreover it optimizes the rotation based on input target angle, calculates speeds' increments, coordinates of points, where acceleration should stop and where deceleration should start).

After the initialization in the interrupt handler of motor control timer firmware calls the function `checkIfPositionIsReached (void)` to check if the robot has already reached destinations point **OR** if it is time to start stable movement / deceleration **OR** if the timer is expired. Then the firmware calls `void speedRecalculation(void)` to recalculate speeds depending on the current step.

Apart from the functions mentioned in the table 1 there are some supplementary funtions for matrix calculations (see ***Matrix.c***, ***Matrix.h***), angle normalization (see `static void normalizeAngle(float* angle)` of ***Robot.h***)) and other trivial procedures.

1.4 Subsystem for task execution of servo manipulators

1.5 Subsystem for communication with high level

1.6 Hardware abstraction layer (HAL)