

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

**ARTHUR ADOLFO
GABRIEL DE SOUZA SEIBEL**

MICROCONTROLADORES: RELATÓRIO FINAL

**Porto Alegre
2018**

**ARTHUR ADOLFO
GABRIEL DE SOUZA SEIBEL**

MICROCONTROLADORES: RELATÓRIO FINAL

Relatório técnico confeccionado para a
disciplina de Microcontroladores da
Universidade Federal do Rio Grande do Sul

Orientador: Walter Fetter

**Porto Alegre
2018**

Resumo

Este documento tem como objetivo reportar o projeto de um *shield* para a placa Intel Galileo Gen 2. Tal *shield* deve ter capaz de acionar uma junta do robô Quanser 2DSFJE. São mostrados todos os procedimentos realizados durante o desenvolvimento do projeto, desde a sua concepção até a sua implementação final - e são documentados os resultados e as conclusões sobre o projeto.

Sumário

1. Introdução	5
2. Hardware	6
2.1. Motor DC	6
2.2. Encoder	8
2.3. Sensores de Fim de Curso	9
2.4. Placa de Circuito Impresso (PCI)	9
3. Software	13
3.1. Controle do motor	13
3.2. Leitura do Decoder	15
3.3. Leitura dos Sensores de Fim de Curso	17
3.4. Script de Inicialização	18
3.5. Rotina de PID	19
3.6. Programa de Teste	21

1. Introdução

Este projeto consiste no desenvolvimento de um *shield* para a Intel Galileo Gen 2, construído em uma placa de circuito impresso, capaz de acionar uma junta do robô Quanser 2DSFJE. Os sinais enviados para acionar o motor devem ser do tipo PWM, de forma que quando o ciclo de trabalho for 0% o motor gire em velocidade máxima em um sentido; quando ele for de 50% o motor permaneça parado, e quando o ciclo for de 100% o motor gire em velocidade máxima no sentido contrário. Os *encoders* devem ser decodificados em quadratura e a contagem deve ser feita em *hardware*, para evitar perdas.

Além do *hardware*, deve ser desenvolvido um *software* que implementa um controlador PID. O *software* deve incluir uma biblioteca com uma API, de forma que o controlador possa comandar o motor pela tensão sobre ele (em volts), ler os *encoders* em radianos e ler os sensores de fim de curso. Esta biblioteca deverá executar no sistema Linux como um programa no espaço do usuário, sem privilégios de superusuário. Nos próximos capítulos serão apresentados os detalhes de funcionamento do *software* e do *hardware*.

2. Hardware

O *hardware* foi projetado em uma placa de circuito impresso e possui dois principais módulos. Um deles controla o motor DC, e outro que realiza a leitura do encoder em quadratura com uma interface SPI. Foi projetada ainda uma conexão digital para receber os sinais dos sensores de fim de curso e repassá-los para a Intel Galileo Gen 2 por meio de sinais GPIO.

A ferramenta de *design* utilizada foi o *Proteus*, um CAD que se mostrou muito útil. Com o seu uso, foram constituídos o esquemático lógico, o leiaute da placa de circuito impresso, e todos os arquivos de saída necessários (gerbers, netlist, *bill of materials* e *pdfs*).

2.1. Motor DC

Todo o processo de chaveamento do motor e controle das chaves é realizado pelo circuito integrado L6203, cujo esquemático está representado na Figura 1A. Este circuito lida com o chaveamento dos transistores MOSFET e supre a tensão necessária para o motor.

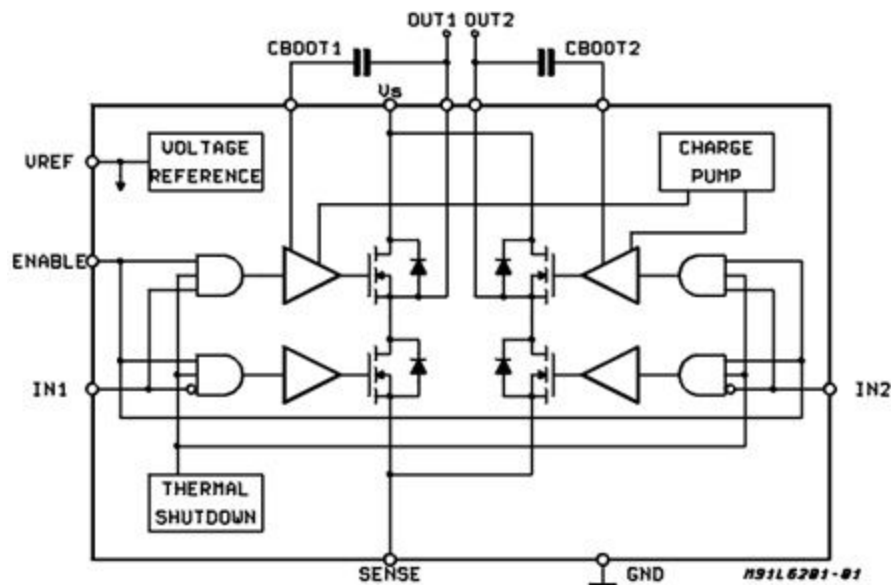


Figura 1A - Esquemático do CI L6203

É necessário ainda alimentar este circuito integrado com duas entradas, uma com um sinal PWM e outra com esse sinal negado, de forma a comutar os transistores de cada lado da ponte H. Para tanto, foi utilizado um sinal PWM gerado pela Intel Galileo Gen 2 e a negação do sinal ocorre por meio de um transistor NPN com uma resistência *Pull-up*, de forma que quando o sinal PWM for positivo, o transistor conecte o GND na saída do sinal negado, e quando o PWM for negativo, conecte o Vcc (5V fornecido pela Galileo) na saída do sinal negado. O esquemático deste mecanismo é representado na Figura 1B.

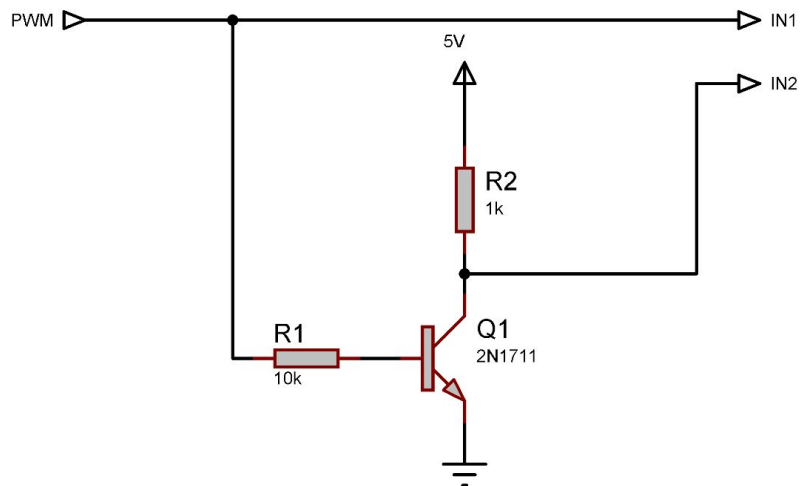


Figura 1B - Inversão do sinal PWM

Para alimentação do L6203 foi utilizada a fonte de 27V fornecida pelo professor no laboratório. A alimentação de 5V para o circuito que inverte o sinal PWM foi obtida diretamente da Galileo, do pino de power de 5V.

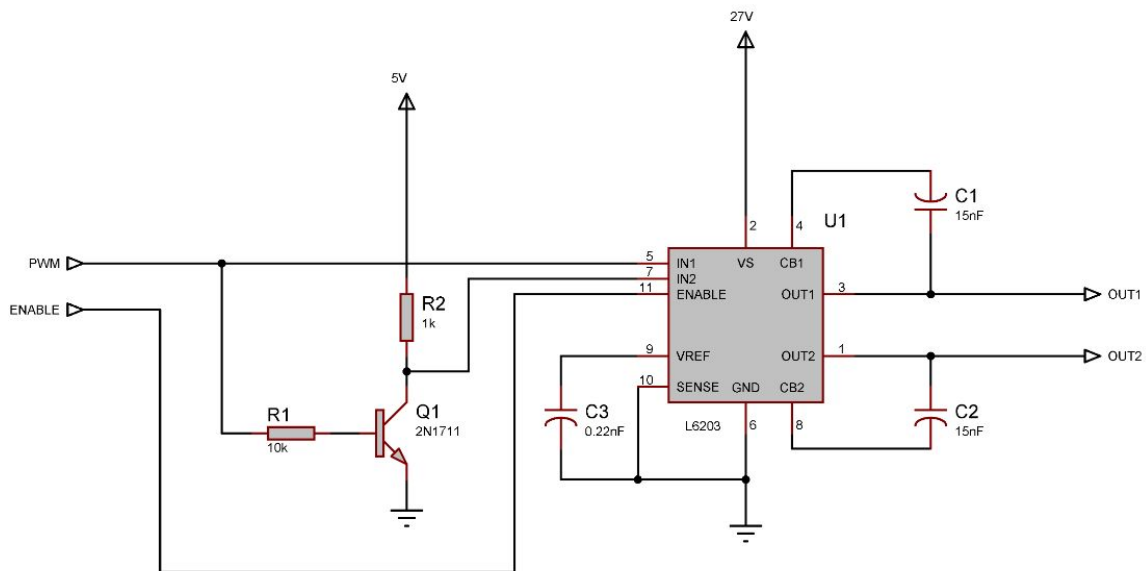


Figura 1C - Esquemático do módulo de acionamento do motor DC

2.2. Encoder

Para a contagem dos sinais do *encoder* foi utilizado o circuito integrado LS7266r - um decoder com interface SPI - ligado nos pinos de interface SPI da Galileo. Para a alimentação do circuito integrado foi utilizado o pino de power de 5V da Galileo. Na Figura 2 é possível ver o esquemático do circuito do decoder, com o respectivo circuito oscilador e sinais de controle ligados na Galileo.

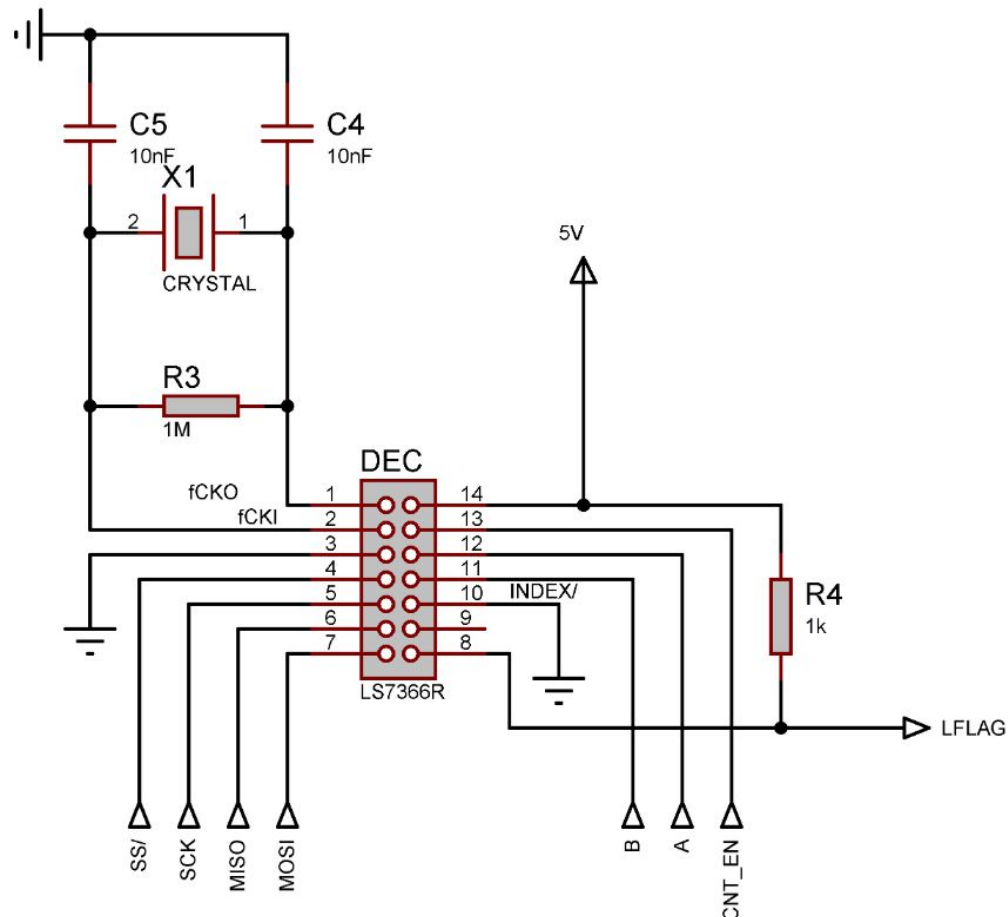


Figura 2 - Esquemático do Decoder

A ideia inicial seria utilizar um conector din 5 fêmea para a leitura do *decoder*, porém não foi possível encontrar tal conector à venda em Porto Alegre. Portanto, um cabo de 10 fios foi utilizado e dele foram soldados apenas 4 fios, um em cada uma das 4 conexões usadas pelo *encoder*.

2.3. Sensores de Fim de Curso

Para a leitura dos sensores de fim de curso, foi preparada uma conexão para o cabo SCSI, com 2 fileiras paralelas de 8 conectores cada. Uma fileira é totalmente ligada no GND da placa e a outra fileira possui dois pinos que representam os sensores do cotovelo do braço robótico, ligados diretamente em dois GPIO de entrada na Galileo. O esquemático desta parte do circuito pode ser visto na Figura 3.

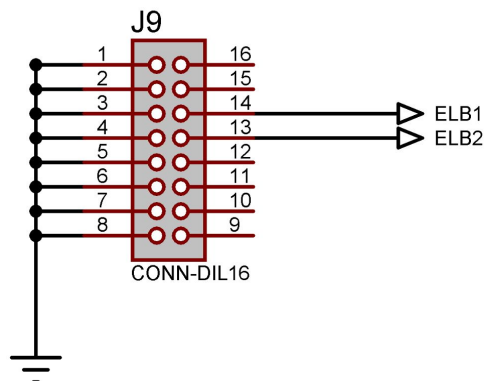


Figura 3 - Esquemático da leitura dos sensores de fim de curso

2.4. Placa de Circuito Impresso (PCI)

Após projetados os módulos, juntamos todos eles em um único esquemático, como pode ser visto na Figura 4A. A placa foi impressa numa placa de fenolite de face dupla, utilizando uma fresadora emprestada pelo professor da UFRGS Renato Ventura. A PCI projetada pode ser vista na Figura 4B, em camadas de duas dimensões, na Figura 4C, em três dimensões. Além disso, Figura 4D mostra uma foto da versão final da PCI confeccionada.

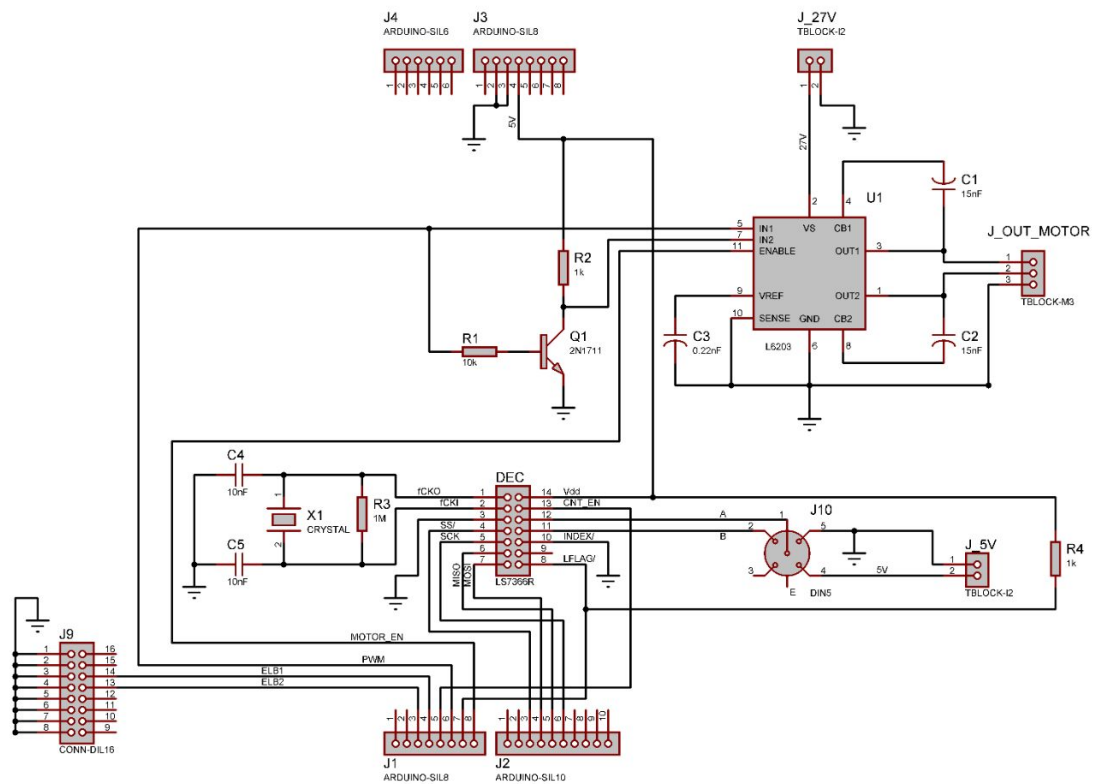


Figura 4A - Esquemático da shield no Proteus

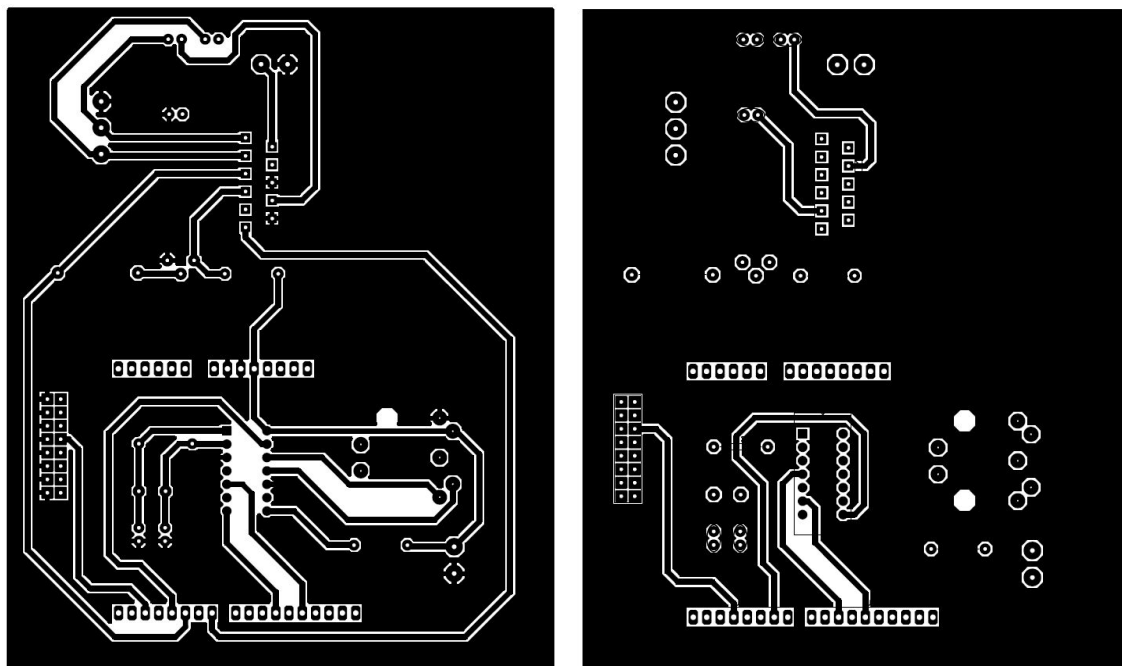


Figura 4B - PCI no Proteus. À esquerda, face inferior; à direita, face superior

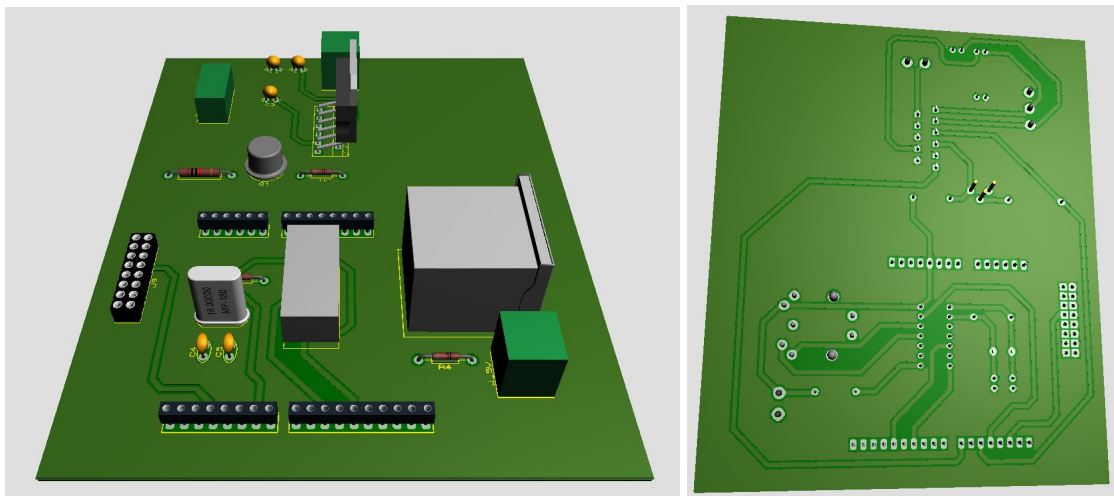


Figura 4C - Modelo 3D da PCI no *Proteus*

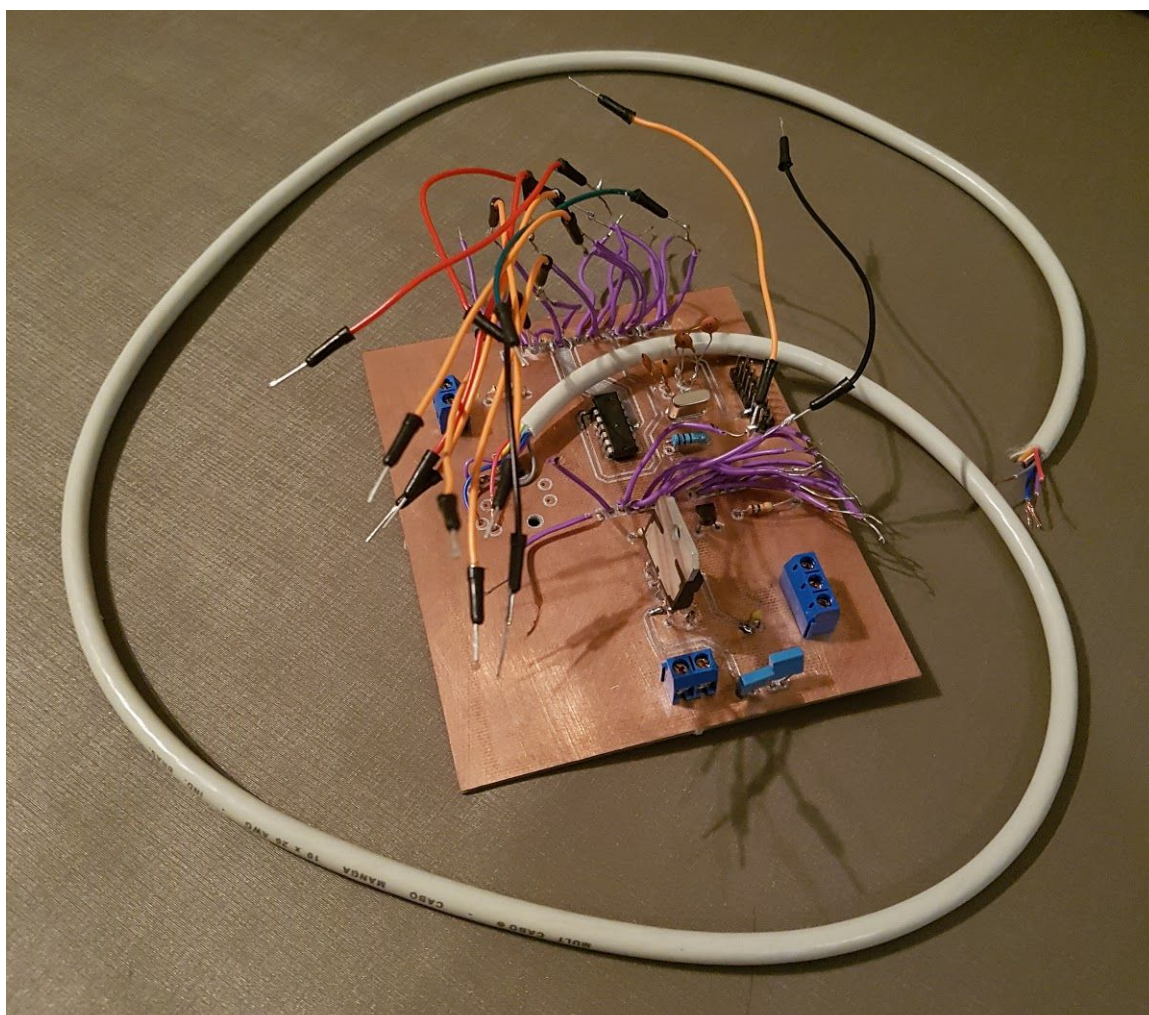


Figura 4D - PCI confeccionada

O esquemático pode ser interpretado mais facilmente de posse de uma referência de pinagem utilizada. Portanto, a seguir encontra-se a tabela de GPIOs utilizadas no projeto.

I02	ENTRADA	Leitura do sensor 1 de fim de curso
I03	ENTRADA	Leitura do sensor 2 de fim de curso
I04	SAÍDA	Sinal de habilitação de contagem do decoder
I05	PWM	Controle de chaveamento da ponte H
I06	ENTRADA	Leitura da flag de <i>borrow</i> ou <i>carry</i> do decoder
I07	SAÍDA	Sinal de liberação do motor DC
I010	SAÍDA	Sinal de <i>slave select</i> do decoder
I011	SAÍDA	MOSI
I012	ENTRADA	MISO
I013	SAÍDA	SCK

3. Software

Para que os recursos do *hardware* desenvolvido pudessem ser utilizados no espaço de usuário de um sistema Linux, foi implementada uma biblioteca de controle da *shield*, em C/C++. A abstração dos componentes foi feita com o paradigma orientado a objetos, com classes que representam cada um dos módulos principais. O uso dessa metodologia permitiu uma modelagem que se encaixa bem com as necessidades de inicialização, configuração, encerramento e modularização das partes da *shield*. O projeto de software também inclui arquivos de automatização de compilação, um *script* para inicializar os pinos utilizados pela placa, um controlador PID e um programa de testes da biblioteca.

Além da implementação da biblioteca em si, também foi gerada a sua documentação. Para tanto, foi utilizada a ferramenta Doxygen, que provê um acervo de descrições e referências de utilização de todas as classes baseados em declarações de blocos de comentário e comandos especiais dentro dos mesmos. Cabe ressaltar que foi feito uso da ferramenta de versionamento Git, com repositório hospedado no site Github em <https://github.com/gabrieelseibel1/QuanserControllerShield>. A seguir, serão brevemente descritas as classes, arquivos e rotinas principais do projeto de *software*.

3.1. Controle do motor

O controle do motor DC deveria se dar através de uma função que aceitasse como parâmetro uma tensão desejada, fazendo com que ela fosse transferida para o motor. Além disso, seriam necessárias algumas rotinas de inicialização, configuração e finalização desse componente. Tendo isso em vista, foi projetada a classe Motor, que tem suas definições principais mostradas na Figura 5.

Além da função recém mencionada de controle da tensão do motor em volts, foi implementada uma que recebe a porcentagem desejada de tensão a ser aplicada no motor (ou seja, um número de 0 a 100). O efeito da chamada dessa função é de acionar o motor na maior velocidade em um dos sentidos, quando recebe 0; acioná-lo na maior velocidade no sentido contrário, quando recebe 100; e mantê-lo parado quando recebe 50.

O controle da tensão aparente sobre o motor se dá por atribuições de valores ao pseudo-arquivo que representa o ciclo de trabalho do PWM. Desse modo, quanto mais similares de uma ondas quadradas simétricas forem os sinais recebidos pelo L6203, mais próximo de nulo será o movimento resultante da junta. Foi configurado um período padrão para o PWM de valor de 1 milhão de nanossegundos, que corresponde a uma frequência de 1KHz.

```

#define MAX_MOTOR_VOLTAGE 27
#define MIN_MOTOR_VOLTAGE (-27)

/**
 * F = 1000Hz -> P = 0.001 s = 1000000 ns
 */
#define PWM_PERIOD_NS "1000000"

/**
 * a = PWM_PERIOD / (MAX_VOLTAGE - MIN_VOLTAGE)
 */
#define DUTY_CYCLE_SLOPE 18518.5185185

/**
 * b = a*MAX_VOLTAGE
 */
#define DUTY_CYCLE_INTERCEPT 500000

#define MOTOR_ENABLE_FILENAME "/sys/class/gpio/gpio38/value"

#define PWM_DUTY_CYCLE_FILENAME "/sys/class/pwm/pwmchip0/pwm1/duty_cycle"

/**
 * DC motor actuation based on PWM
 */
class Motor {
public:

    /**
     * Makes first configuration for the motor
     */
    Motor();

    /**
     * Close files
     */
    virtual ~Motor();

    /**
     * Turns motor OFF by setting its enable pin to 0
     */
    int disable();

    /**
     * Turns motor ON by setting its enable pin to 1
     */
    int enable();

    /**
     * Sets motor voltage from a percentage (0-100) of the possible range
     * 0 -> full throttle to side A
     * 50 -> no movement
     * 100 -> full throttle to side B
     * @param percentage value from 0 to 100 that corresponds to percentage of voltage range
     * @return 0 if success, -1 if failed
     */
    int set_voltage_percentage(float percentage);

```

```

private:

    /**
     * File descriptor for duty cycle configuration
     */
    int duty_cycle_fd;

    /**
     * File descriptor for enabling the motor
     */
    int enable_motor_fd;

    /**
     * Sets apparent voltage of the dc motor. Effectively, sets PWM duty cycle accordingly.
     * @param voltage The desired voltage between MAX_MOTOR_VOLTAGE and MIN_MOTOR_VOLTAGE
     */
    int set_voltage(float voltage);

    /**
     * Turns PWM ON
     */
    int enable_pwm();

    /**
     * Turns PWM OFF
     */
    int disable_pwm();

    /**
     * Configures PWM period to default value PWM_PERIOD_NS_INT
     */
    int set_pwm_period();
};

```

Figura 5 -Declarações da classe Motor

3.2. Leitura do Decoder

A principal função a ser disponibilizada pela biblioteca com relação a identificação da posição da junta do 2DSFJE seria uma que fizesse a leitura da contagem do decoder e convertesse esse valor para radianos. Essa conversão se dá utilizando uma equação que considera o valor lido do LS7366R como em uma porcentagem do número de contagens por revolução, multiplicando essa porcentagem por 2π .

Para que dados dos registradores do decodificador (como o CNTR, que guarda a contagem) pudessem ser lidos pelo programa, foram implementadas funções que realizam comunicações pelo padrão SPI, além de suas necessárias configurações de inicialização. Abaixo, na Figura 6, podem ser lidas as declarações da classe Decoder, a abstração deste módulo da *shield*.



```
/**
 * LS7366R Decoder for reading the position of the arm joint
 */
class Decoder {
public:

    /**
     * Opens files and configures SPI communication
     */
    Decoder();

    /**
     * Close open files and end SPI
     */
    virtual ~Decoder();

    /**
     * Reads counting and converts to radians, indicating position of the joint
     * @return position of the joint in radians
     */
    float get_position_radians();

private:

    /**
     * Read counting from SPI
     * @return value of CNTR register
     */
    int get_count();

    /**
     * File descriptor for SPI
     */
    int spi_fd;

    /**
     * File descriptor for #SS
     */
    int _ss_fd;

    /**
     * Write an op code than a byte of data to the decoder
     * @param op_code the op code to define the register and the operation
     * @param data the data to be transmitted
     * @return 0 if success, -1 if failed
     */
    int write_byte(unsigned char op_code, unsigned char data);
```



```

/**
 * Write an op code than wait for the decoder to read byte data from the SPI bus
 * @param op_code the op code to define the register and the operation
 * @return a byte transmitted by the decoder
 */
unsigned char read_byte(unsigned char op_code);

/**
 * Set #SS to high
 * @return 0 if success, -1 if failed
 */
int _ss_high();

/**
 * Set #SS to low
 * @return 0 if success, -1 if failed
 */
int _ss_low();

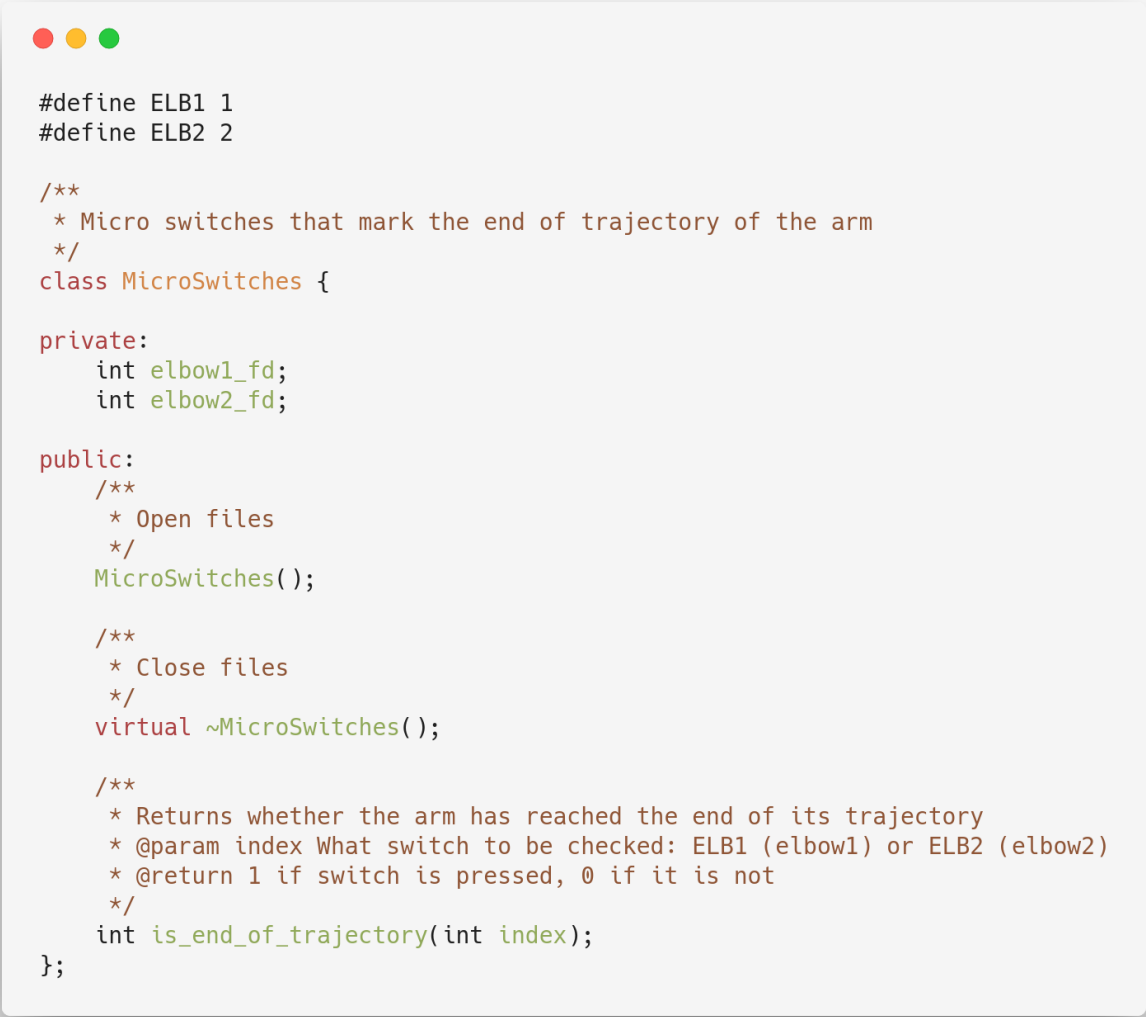
/**
 * Clear contents of CNTR register
 */
int reset_count();
};

```

Figura 6 -Declarações da classe Decoder

3.3. Leitura dos Sensores de Fim de Curso

Os sensores de fim de curso, referenciados na biblioteca como *microswitches*, são lidos com o uso simples de pinos GPIO da galileo. Portanto, basta que os devidos pseudo-arquivos sejam abertos, lidos sob demanda de um usuário da biblioteca, e fechados ao fim da execução - feito automaticamente pelo destrutor da classe. As declarações dessas funções, pertencentes a classe MicroSwitches, estão listadas na Figura 7.



```

#define ELB1 1
#define ELB2 2

/**
 * Micro switches that mark the end of trajectory of the arm
 */
class MicroSwitches {

private:
    int elbow1_fd;
    int elbow2_fd;

public:
    /**
     * Open files
     */
    MicroSwitches();

    /**
     * Close files
     */
    virtual ~MicroSwitches();

    /**
     * Returns whether the arm has reached the end of its trajectory
     * @param index What switch to be checked: ELB1 (elbow1) or ELB2 (elbow2)
     * @return 1 if switch is pressed, 0 if it is not
     */
    int is_end_of_trajectory(int index);
};

```

Figura 7 -Declarações da classe MicroSwitches

3.4. Script de Inicialização

No script de inicialização são configurados diversos pinos IOs. São eles: pinos IO3 e IO4 como GPIO de saída para leitura dos sensores do cotovelo do Quanser, pino IO5 como PWM para controlar chaveamento da ponte H que aciona o motor, pino IO6 como GPIO de entrada para ler LFLAG do decoder, pino IO7 como GPIO de saída para sinal de *enable* da ponte H e os pinos IO10 a IO13 como interface SPI para se comunicar com o decoder.

3.5. Rotina de PID

Para a rotina de PID foi feita uma classe PIDController para gerenciar os cálculos. A classe pode ser utilizada incluindo o arquivo PIDController.h. Detalhamento dessa classe pode ser encontrado na Figura 8.

```
/**
 * PID Controller
 */
class PIDController {
public:

    /**
     * Makes first configuration for the PIDController
     */
    PIDController(float kp, float ki, float kd);

    /**
     * Sets the max limit
     * @param value in voltage for max limit
     */
    void set_max_limit(float value);

    /**
     * Sets the min limit
     * @param value in voltage for min limit
     */
    void set_min_limit(float value);

    /**
     * Sets new point that motor needs to reach
     * @param new_point radian value
     */
    void set_point(float new_point);

    /**
     * Computes PID routine
     * @param delta time in radians
     * @param sample in radians
     * @return the necessary voltage to be applied to the motor
     */
    float compute(float delta, float sample);
```

```

private:

    /**
     * Point where motor must go (radians)
     */
    float point;

    /**
     * Proportional gain
     */
    float Kp;

    /**
     * Integral gain
     */
    float Ki;

    /**
     * Differential gain
     */
    float Kd;

    /**
     * Max limit in voltages
     */
    float max_limit;

    /**
     * Min limit in voltages
     */
    float min_limit;

    /**
     * Saves the last sample for integration
     */
    float last_sample;

    /**
     * Used to calculate the PID equation
     */
    float integrated;

};

```

Figura 8 - Declarações da classe PIDController

3.6. Programa de Teste

O programa teste utilizado retorna a posição do braço para um valor de referência, e após isso, move o braço para uma posição nova passada pelo programa no espaço de usuário. O programa utiliza a rotina PID para implementar o movimento do braço. Mais detalhes do programa podem ser vistos na Figura 9.

```

#define THRESHOLD 0.00001

/**
 * Script for testing the quanser controller shield
 * @return 0 if succeeded, -1 if not
 */
int main(int argc, char *argv[]) {

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <angle from 0 to 180>", argv[0]);
        exit(EXIT_FAILURE);
    }

    clock_t last, now;
    float angle, sample, new_voltage, delta, rad_angle;

    //initialize quanser modules
    QuanserControllerShield *quanser = new QuanserControllerShield();

    //initialize PID controller
    PIDController pid = PIDController(0.1, 0.2, 0.0005);
    pid.set_max_limit(MAX_MOTOR_VOLTAGE);
    pid.set_min_limit(MIN_MOTOR_VOLTAGE);

    angle = (float) atof(argv[1]);

    if (angle < 0 || angle > 180) {
        fprintf(stderr, "Usage: %s <angle from 0 to 180>", argv[0]);
        exit(EXIT_FAILURE);
    }

    rad_angle = (float) (angle * M_PI) / 180;

    //enable motor
    quanser->motor->enable();

    //send joint to reference position, so that the counting can begin
    quanser->send_joint_to_reference_position();

    //define the desired position of the joint
    pid.set_point(rad_angle);
    last = clock();
}
```

```

while (true) {

    //break if a micro switch is triggered
    if (!quanser->microSwitches->is_end_of_trajectory(ELB1) ||
        !quanser->microSwitches->is_end_of_trajectory(ELB1))
        break;

    //get current joint position
    sample = quanser->decoder->get_position_radians();

    delta = sample - rad_angle;
    delta = delta < 0 ? -1 * delta : delta;

    //convergence: joint arrived at desired position
    if (delta <= THRESHOLD) break;

    now = clock();

    //calculate and set the necessary voltage to be applied at the motor
    new_voltage = pid.compute((float) (now - last) / CLOCKS_PER_SEC, sample);
    quanser->motor->set_voltage(new_voltage);

    last = now;
}

//stop the motor
quanser->motor->set_voltage(0);
quanser->motor->disable();

//close files
delete quanser;

return 0;
}

```