

Universidade Federal do Rio Grande do Norte

Departamento de Engenharia de Computação e Automação
DCA0119 - SISTEMAS DIGITAIS

RELATÓRIO DO 2º PROJETO DE UNIDADE

Sistema embarcado em um microcontrolador para o controle de
um secador de grãos

Integrantes:

Felipe Oliveira Lins E Silva - 20180154889

Emerson Wendlingger Dantas Sales - 20180154851

Luís Gabriel Pereira Condados - 2015093091

Luís Henrique Matias Viana - 20180155026

Professor orientador: Sérgio Natan

Natal-RN
2018

Universidade Federal do Rio Grande do Norte

Departamento de Engenharia de Computação e Automação
DCA0119 - Sistemas Digitais

RELATÓRIO DO 2º PROJETO DE UNIDADE

Relatório apresentado à disciplina de Sistemas Digitais, correspondente a 2º unidade do semestre 2018.2 do 7º período do curso de Engenharia de Computação e Automação da Universidade Federal do Rio Grande do Norte, sob orientação do **Prof. Sérgio Natan Silva**.

Natal-RN
2018

Conteúdo

1	INTRODUÇÃO	1
2	ESPECIFICAÇÕES DO PROBLEMA	2
3	Informações Técnicas	3
4	Implementação	5
4.1	thread principal- Main	5
4.2	Class Controller	5
4.3	Comunicação	8
4.4	Controle dos pinos	10
4.5	Módulo de leitura dos sensores	11
4.6	Módulo das interrupções	12
5	Curvas	13
6	Resultados	15
7	vídeo demostrativo	17

1 INTRODUÇÃO

Este projeto consiste na implementação de um secador de grãos utilizando um microcontrolador com o kit de desenvolvimento da Intel, mais especificamente o **Intel Galileo** da segunda geração[?]. Este, recebe dois sinais analógicos de entrada, respectivamente de temperatura e luminosidade, em sequência, assim que a chave CH é pressionada, inicia o processo de secagem. LEDS serão acesos, conforme as especificações, durante o processo. O microcontrolador emite um sinal PWM ($Q(t)$), que vai para um opto-acoplador, que associa a saída PWM ao circuito do motor.

O sinal $Q(t)$ obedece a esta formula:

$$Q(t) = \alpha \cdot Z(t) \cdot T(t) + \beta \cdot L(t)$$

Onde, α e β são constantes; $Z(t)$ corresponde ao formato da curva a qual queremos que modele o comportamento do motor; $T(t)$ e $L(t)$ são os valores obtidos, respectivamente, pelos sensores de temperatura e luminosidade.

2 ESPECIFICAÇÕES DO PROBLEMA

1. O sistema deverá ter dois sinais analógicos de entrada: um que tem como origem o sensor de Temperatura ($T(t)$), e outro que vem do sensor de luminosidade ($L(t)$).
2. O sistema deve possuir uma chave CH1 (HiZ e terra), a qual, quando em terra, dá início ao processo de secagem.
3. O sistema deve ter um LED (LED1) que tenha sua luminosidade proporcional a $Q(t)$ - o qual é o sinal de saída, que controlará o motor.
4. O sistema deve ter um LED (LED2) o qual a luminosidade será proporcional a $T(t)$ ou $L(t)$ - situação do secador.
5. O sinal emitido pelo controlador deverá estar em uma frequência entre 100 Hz e 100 kHz .
6. O sistema deve possuir pelo menos duas funções($Z(t)$) que regem o comportamento geral das saídas PWM.

3 Informações Técnicas

Segue a baixo a placa utilizada no projeto.

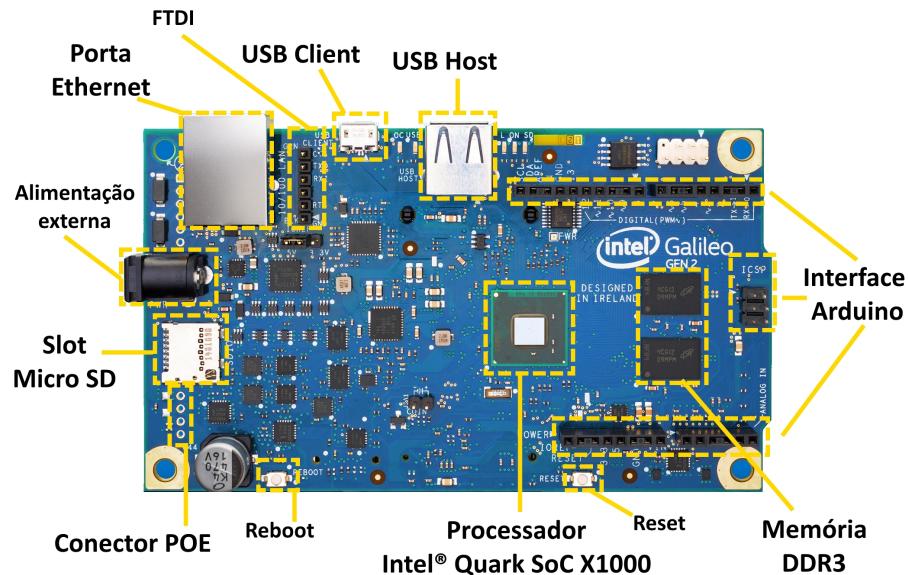


Figura 1: Intel Galileo Gen2, recursos

Segue abaixo o esquemático do projeto e a montagem:

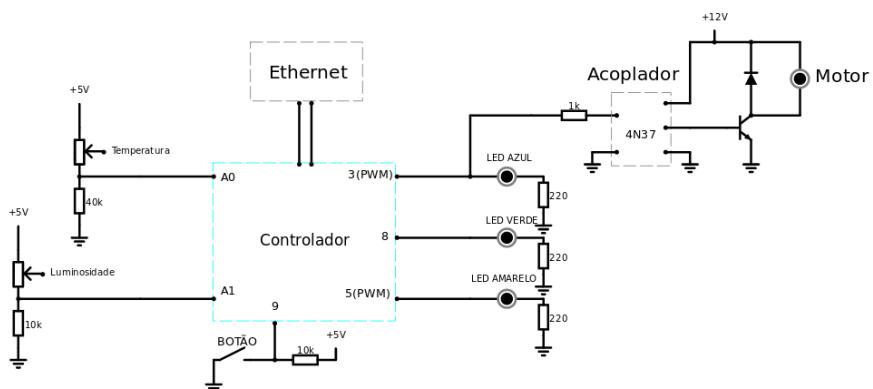


Figura 2: Esquemático

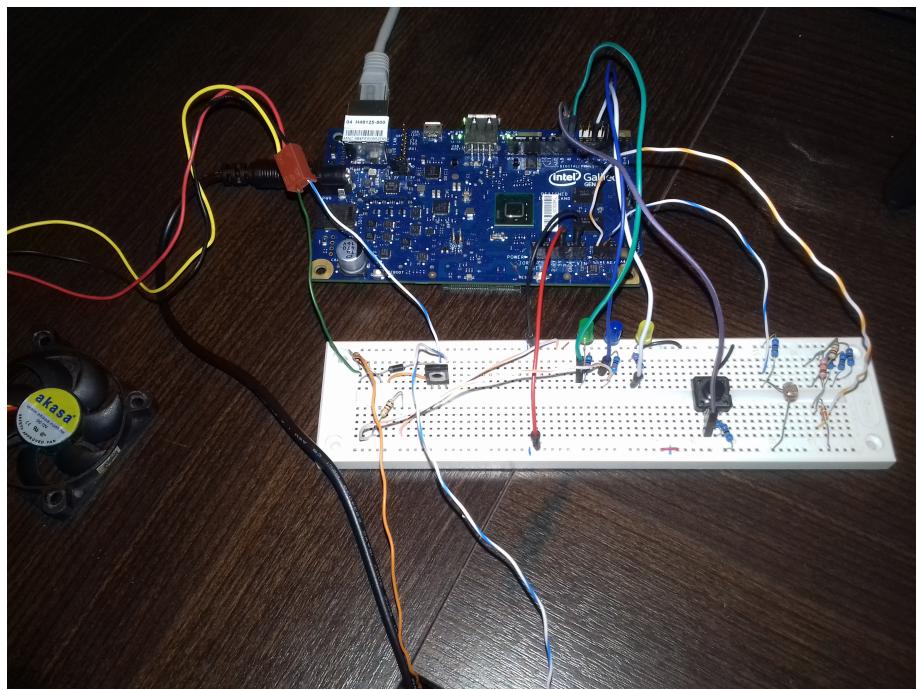


Figura 3: Montagem do circuito

4 Implementação

Na elaboração do código, optamos por dividi-lo em módulos - para que o desenvolvimento colaborativo seja o mais proveitoso possível. Os módulos são os que são executados de forma paralela graças a utilização de Threads, os módulos são os seguintes: Comunicação; Leitura dos sensores; Controle das saídas PWM; Interrupção do botão; programa principal.

Para programar a plataforma de desenvolvimento da Intel, utilizamos a API para C++ [colocar ref. da MRAA], essa API fornece as ferramentas necessárias para trabalharmos com todos os recursos da plataforma utilizando a linguagem de programação C++.

Abstraímos o problema, através de classes em C++, em que um objeto Controller fica responsável por controlar todo o processo envolvido no sistema do secador de grãos, reduzindo assim a complexidade do problema.

4.1 thread principal- Main

A seguir é mostrado o código do laço principal.

```
1 #include <iostream>
2
3 #include "controller.h"
4
5 int main(){
6     Controller ctrl_Fan;
7     ctrl_Fan.start();
8     std::cout << "Sistema online" << '\n';
9
10    while(ctrl_Fan.inOperation()){
11        //codigo paralelo ao controle
12        std::this_thread::yield(); //para nao bloquear o processo
13    };
14    std::cout << "Sistema finalizado" << '\n';
15
16    return 0;
17 }
```

4.2 Class Controller

```
1 //*** CLASSE PRINCIPAL ***/
```

```

2 #include <thread> // std :: thread
3
4 #include <mraa/common.hpp>
5 #include <mraa/gpio.hpp>
6 #include <mraa/pwm.hpp>
7 #include <mraa/aio.hpp>
8
9 #include <socket4L.h> // class Socket_4Linux
10 #include <system_tools.h>
11
12 #define PORT 28500
13 //para minimizar o efeito do debounce no botao de acionamento
14 #define TIMEDEBOUNCE 0.3 //segundos
15 #define REF_4_TEMP 27.0 //graus Celcius
16 #define REF_4_LUMI 500.0 //unidade de luminocidade
17 #define ALPHA 1.0/54 // inverso da maxima temperatura
    esperada
18 #define BETA 1.0/1000 // inverso da maxima luminancia
    esperada
19 // Modos de operacao
20 // Defininem o comportamento do secador de graos
21 enum MODE{
22     DEFAULT, //gera a curva padrao
23     MODE1, //igual ao modo Default
24     MODE2
25     // ...
26 };
27
28 class Controller
29 {
30 private:
31     ServerSocket serverCtrl;
32     MODE mode;
33     bool finish; //Flag para desligar o sistema
34     bool enable; //Flag para ativar/desativar o sistema , True:
            habilitado , False: desabilitado
35
36     mraa::Gpio gpioLedEnable;
37     mraa::Gpio gpioButton;
38     mraa::Aio aioLumi;
39     mraa::Aio aioTemp;
40     mraa::Pwm pwmLedSensor;
41     mraa::Pwm pwmDriveFan;
42
43     double timeRef_debounce;
44     volatile std::atomic<bool> newData; //Ha dados dos sensores
        para serem lidos
45     volatile std::atomic<uint8_t> temp; //valor de temperatura em
        graus celcius

```

```

46 volatile std::atomic<uint8_t> lumin; // valor da luminancia
47 volatile std::atomic<float> pwmValueDrive;
48
49 // thread para leitura de dados
50 std::thread thr_acquisition;
51 //thread que realizara o contro dos pinos
52 std::thread thr_pinController;
53 //thread responsavel pelo recebimento e envio de dados
54 std::thread thr_comunication;
55
56 //Funcao que sera associada a interrupcao do botao
57 void button_interrupt();
58
59 enum PIN{
60     LedEnable = 8, // Pino do led para indicar o estados do
61     sistema: ON/OFF
62     LedSensor = 5, // Pino do Led associado a a leitura de um
63     dos sensores
64     Button = 9, // Pino associado ao botao
65     DriveFan = 3, // Pino para PWM do motor
66     SensorLumi = 1, // Pino para leitura AD do sensor de
67     luminancia
68     SensorTemp = 0 // Pino para leitura AD do sensor de
69     temperatura
70 };
71
72 //Funcoes que ditam o comportamento da saida PWM para o motor
73 //retornam o valor do PWM
74 //equivalente ao Z(t) no projeto
75 float curve(const float& t); //este eh o metodo que deve ser
76     chamado para o retorno final(ele chaveia entre os demais, de
77     acordo com o modo atual)
78 float curve01(const float& t); //curva padrao
79 float curve02(const float& t); //curva 2
80
81 //Metodos para as threads:
82 void pinController();
83 void communication();
84 void acquisition();
85
86 //Funcoes auxiliares (para permitir usar metodos nas threads)
87 friend void func_pinController(const void*X);
88 friend void func_comunication(const void*X);
89 friend void func_acquisition(const void*X);
90 friend void func_button_interrupt(void*X);
91 //desliga o sistema e encerra o processo
92 // void shutdown();
93
94 public:
95     Controller();
96     ~Controller();

```

```

89 //Retorna o status do sistema
90 inline bool inOperation() const {return !finish;}
91 //A principio so sera usado por comandos via socket, logo n
92     sera necessario este metodo para o main
93 MODE getMode() const {return mode;}
94 void setMode(const MODE &new_mode); //A principio so sera usado
95     por comandos via socket, logo n sera necessario este metodo
96     para o main
97 //retoma o sistema da onde parou, ou inicia do zero
98 inline void start(){ this->enable = true; };
99 //Suspende o sistema temporariamente
100 //ao ser religado com start(), o sistema retoma
101 //da onde parou
102 inline void stop(){ this->enable = false; }; //so desabilita o
103     sistema, mas n encerra o processo
104 //desliga o sistema e encerra o processo
105 void shutdown();
106 };

```

4.3 Comunicação

Esta Thread tem como objetivo fornecer um meio de entrada e saída de informações via socket, permitindo o controlar remotamente o controlador.

```

1 void Controller :: communication(){
2     ServerSocket newSock;
3     std::string command;
4     std::string arg;
5
6     serverCtrl.create();
7     serverCtrl.bind(PORT);
8
9     while (!finish){
10         serverCtrl.listen();
11         serverCtrl.accept(newSock);
12
13         newSock >> command;
14         tools :: str2upper(command);
15
16         if (command == "REQUEST")
17         {
18             while (newData){} //aguarda a thread de acquisition
19             terminar de adquirir novos dados
20             newData = false;
21             mtx.lock();
22             newSock << std :: to_string(pwmValueDrive);
23             mtx.unlock();

```

```

23 }
24 else if (command == "START")
25 {
26     enable = true;
27 }
28 else if (command == "STOP")
29 {
30     enable = false;
31 }
32 else if (command == "SHUTDOWN")
33 {
34     finish = true;
35     enable = false;
36 }
37 else if (command == "MODE_1")
38 {
39     this->setMode(MODE1);
40     std::cout << "Curva 1 selecionada" << '\n';
41 } else if (command == "MODE_2")
42 {
43     this->setMode(MODE2);
44     std::cout << "Curva 2 selecionada" << '\n';
45 } else if (command == "TEMP")
46 {
47     mtx.lock();
48     newSock << std::to_string(temp);
49     mtx.unlock();
50 } else if (command == "LUMIN")
51 {
52     mtx.lock();
53     newSock << std::to_string(lumin);
54     mtx.unlock();
55 }
56 else
57 {
58     newSock << "INVALID_COMMAND";
59 }
60 newSock.disconnect();
61 }
62 }
```

4.4 Controle dos pinos

Este módulo é responsável por alterar os valores das saídas PWM de acordo com a equação $Q(t)$, e os LEDs de acordo com o que foi estabelecido para o comportamento do sistema.

```
1 void Controller :: pinController () {
2
3     while (!finish)
4     {
5         while (!enable) // aguarda o sistema ser habilitado
6         {
7             if (finish) break;
8             std :: this_thread :: yield ();
9         }
10        // funcionamento do sistema habilitado
11        gpioLedEnable . write (1);
12        double t_start = tools :: clock ();
13        while (enable){
14            // pedido de leitura dos sensores
15            newData = true;
16            while (newData){
17                if (finish) break;
18                std :: this_thread :: yield ();
19            }; // aguarda a leitura dos dados
20            // entrando na zona critica
21            mtx . lock ();
22            pwmValueDrive = curve (tools :: clock () - t_start ) * temp * ALPHA
23            + lumin * BETA;
24            // saindo da zona critica
25            pwmDriveFan . write (pwmValueDrive);
26            pwmLedSensor . write (lumin * BETA);
27            mtx . unlock ();
28        };
29        // sistema passou de habilitado para desabilitado
30        gpioLedEnable . write (0);
31        pwmDriveFan . write (0);
32        pwmLedSensor . write (0);
33        pwmValueDrive = 0;
34    }
```

4.5 Módulo de leitura dos sensores

Este módulo implementa uma rotina para a leitura de dados dos sensores de forma paralela.

```
1 void Controller :: acquisition (){  
2  
3     while (! finish ){  
4         while ((this->newData == false) && ! enable)//aguarda pedido  
5             de novos dados  
6         {  
7             if (finish )break;  
8             std :: this_thread :: yield ()//parar a execu    o do thread  
9             atual e dar prioridade a outros processos  
10            }  
11            //zona critica  
12            mtx.lock ();  
13            this->temp      = aioTemp.read ()*(REF_4_TEMP/512.0);  
14            this->lumin     = aioLumi.read ()*(REF_4_LUMI/512.0);  
15            this->newData = false;  
16            //fim da zona critica  
17            mtx.unlock ();  
18        }  
19    }  
20}
```

4.6 Módulo das interrupções

Este módulo implementa o tratamento da interrupção provocado pelo acionamento do botão.

```
1 void Controller :: button_interrupt() {
2     if(( tools :: clock() - timeRef_debounce ) > TIME_DEBOUNCE) {
3         enable = !enable;
4     }
5     timeRef_debounce = tools :: clock();
6 }
```

Devido a problemas intrínsecas ao botão o código acima apresenta uma forma de tratar o problema do debounce, para um funcionamento mais próximo do esperado, ao ser pressionado o sistema aciona esse método por meio da interrupção e ignora ocorrências da interrupção em um certo período de tempo menor do que um tempo pré-determinado, para se considerar um acionamento válido.

5 Curvas

Para cumprir o requisito de gerar pelo menos dois comportamentos distintos, o sistema desenvolvido neste trabalho está implementado com a capacidade de apresentar essas duas saídas e foi escrito de tal forma que pode ser facilmente acrescido de novas curvas ($Z(t)$).

Algoritmo que descreve o comportamento da Curva 1.

```
1 float Controller::curve01(const float& t){  
2     if( t <= 10 && t >= 0)  
3         return (0.35/10)*t;  
4  
5     if( t > 10 && t <= 15)  
6         return 0.35;  
7  
8     if( t > 15 && t <= 20)  
9         return (0.06)*(t-15) + 0.35;  
10  
11    if( t > 20 && t <= 25)  
12        return 0.65;  
13  
14    if( t > 25 && t <= 30)  
15        return (-0.13)*(t-25) + 0.65;  
16    //outros casos, erro ou final da curva  
17    // std::cout << "Fim da curva 1" << '\n';  
18    this->stop();  
19    return 0;  
20 }
```

Algoritmo que descreve o comportamento da Curva 2.

```
1 float Controller::curve02(const float& t){  
2     if( t <= 10 && t >= 0)  
3         return (t*t*0.004);  
4  
5     if( t > 10 && t <= 15)  
6         return 0.4;  
7  
8     if( t > 15 && t <= 18)  
9         return (0.1334)*(t-15) + 0.4;  
10  
11    if( t > 18 && t <= 20)  
12        return 0.8;  
13  
14    if( t > 20 && t <= 25)  
15        return (-0.08)*(t-20) + 0.8;
```

```

16
17 if( t > 25 && t <= 30)
18     return 0.4;
19 if( t > 30 && t <= 40)
20     return (-0.04)*(t-30) + 0.4;
21 //outros casos, erro ou final da curva
22 // std::cout << "Fim da curva 2" << '\n';
23 this->stop();
24 return 0;
25 }
```

O código a seguir permitiu acrescentarmos novas curvas de forma flexível e simplificada.

```

1 float Controller::curve(const float& t){
2     switch (mode){
3         case MODE1:
4         case DEFAULT:
5             return this->curve01(t);
6             break;
7         case MODE2:
8             return this->curve02(t);
9             break;
10        default:
11            // std::cerr << "Modo de operacao invalido" << '\n';
12            return 0;
13            break;
14    }
15 }
```

6 Resultados

Para visualizarmos o comportamento do sistema, desenvolvemos um client.cpp, que por meio de socket controlamos remotamente o sistema embarcado e monitoramos os valores dos PWM, assim como as leituras dos sensores em tempo real.

Com este cliente geramos os gráficos que correspondem ao comportamento esperado e assim confirmamos o correto funcionamento do sistema.

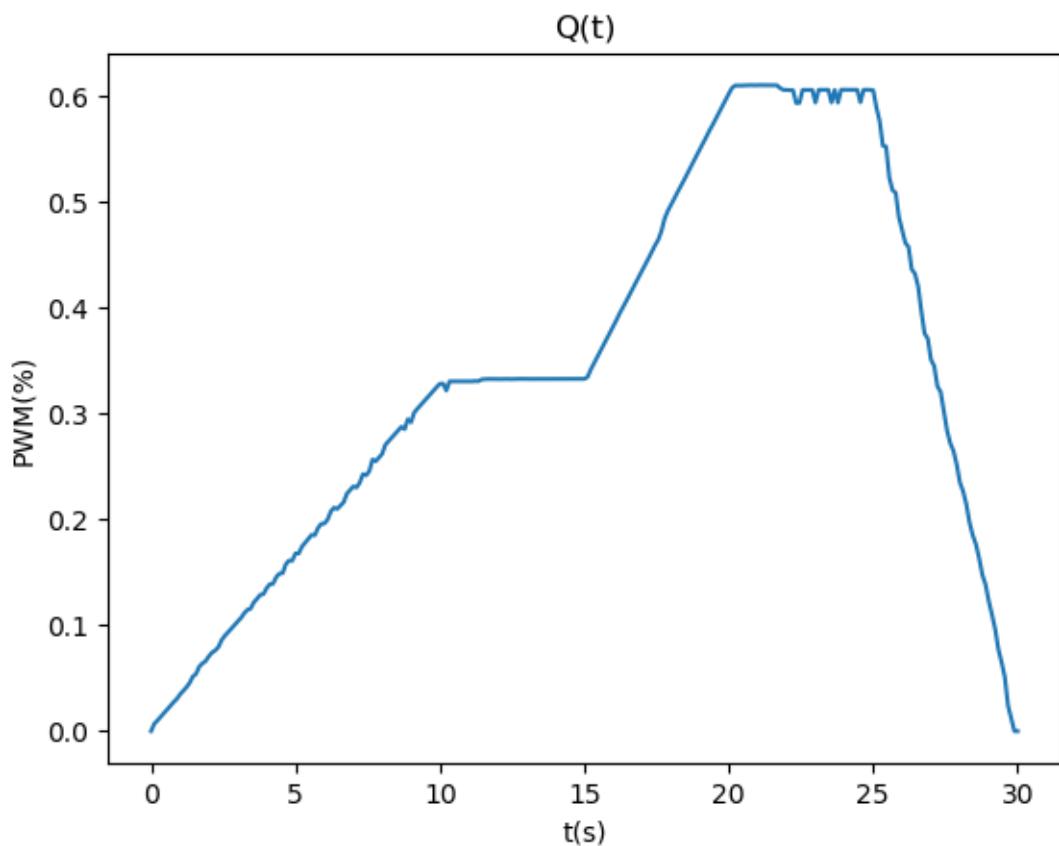


Figura 4: Curva 1

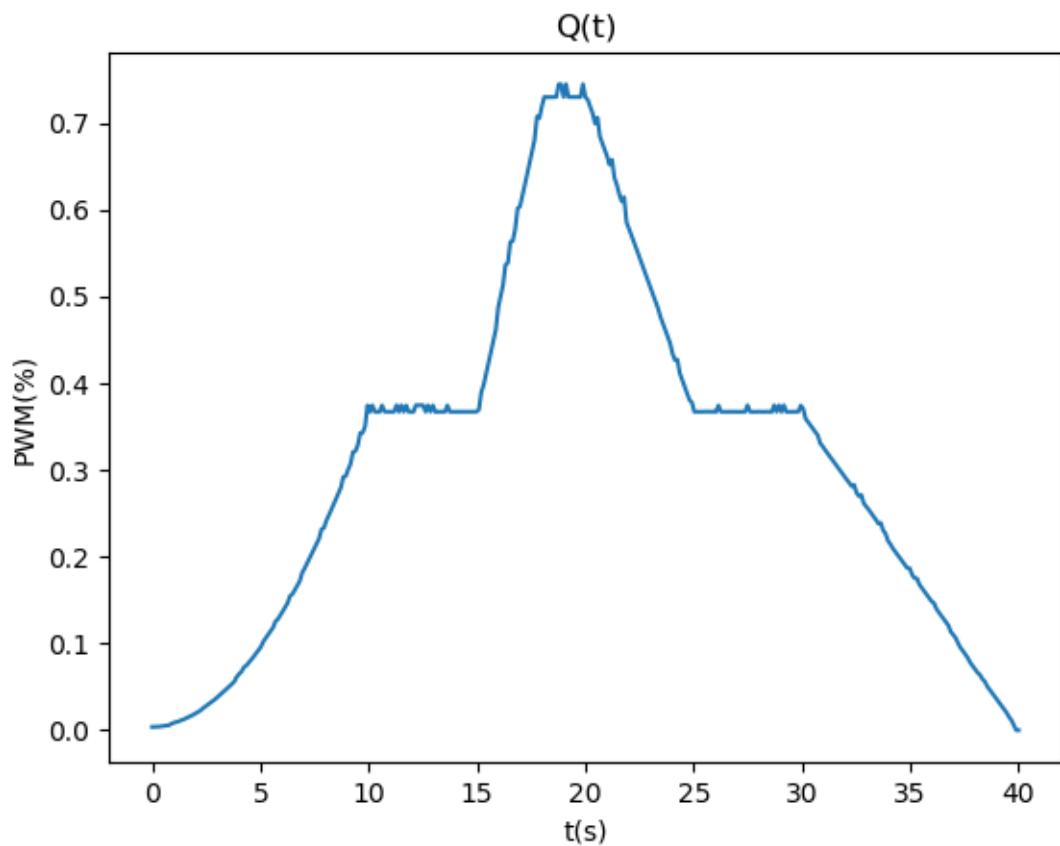


Figura 5: Curva 2

As curvas apresentam um comportamento bem muito ruidoso devido a escolha das constantes, elas foram escolhidas de forma a minimizar o efeitos dos sensores, por fins didáticos não atribuímos um significado para isso, apenas queríamos visualizar a curva de forma mais clara e ainda assim praticarmos o controle dos pinos, tanto a leitura ADC quanto para saídas PWM.

7 vídeo demonstrativo

Segue aqui o link para o vídeo de apresentação dos resultados:

https://www.dropbox.com/s/1cz3b8w5l6l7au1/videoDemonstrativosSDPU2.mp4?dl=0

Link para o repositório onde se encontra o código desenvolvido para este projeto.

https://github.com/Gabriellgpc/SDPU2