Sistema de Controle de Abastecimento para Foguete de Competição Universitário

Wemerson Fontenele Sousa, Lucas Gonçalves Campos

***Resumo*—Este documento apresenta a concepção e desenvolvimento de um controlador de abastecimento para foguete de competição. O projeto se baseia na estrutura do foguete da equipe Capital Rocket Team, utilizando-se de uma MSP430 para controle da temperatura do *venti* - válvula que ejeta gases de óxido nitroso, de forma a fechar o abastecimento ao atingir uma temperatura limite.**

***Index Terms*—Microcontroladores, Assembly, Eletrônica Embarcada, MSP 430.**

I. Introdução

TRATA-SE de um problema comum às equipes de competições de lançamento de foguetes o controle de abastecimento à distância - ponto cobrada pelas organizadoras das competições. No caso estudado para este projeto, a equipe possui uma válvula (*venti*) de escape para o óxido nitroso em fase gasosa. Quando o tanque está preenchido, a válvula em questão dispensa a mesma substância em fase líquida. A partir disto, foi possível pensar em um controle a partir da temperatura na saída desta válvula. Através da medição contínua da temperatura, é possível marcar uma temperatura limite para fechamento do abastecimento, assim como verificar bruscas variações da temperatura (decorrente da mudança de fase do óxido nitroso).

Não foi encontrado projeto parecido em pesquisas para referência, mas foram encontrados alguns códigos para realizar algumas de suas funções. Primeiro, foi observada a existência de projeto de medição de temperatura a partir da MSP430 utilizando o sensor LM35, cujo preço e velocidade de resposta o tornam ideal para o projeto [1]. A partir deste projeto foi necessário alterá-lo de modo a identificar alterações bruscas na temperatura. Também foi encontrado um projeto para controle de motor de passo, modelo 28BYJ-48, com o uso de um *driver* ULN2003, este sendo necessário, pois a MSP430 não consegue fornecer corrente suficiente para a movimentação do motor [3]. Esta fase do projeto será responsável por abrir a válvula, que estará conectada ao motor. Portanto, é possível adaptar este projeto facilmente, sendo necessário apenas a alteração do momento no qual o motor deve ser ativado. Por fim, foi encontrado um projeto onde se realiza a comunicação da MCU (responsável pela medição e controle do motor), com um celular a partir de um módulo bluetooth HC-06 [2], que se comunica com um aplicativo presente no celular. O que se aplica a nossa proposta, pois deve ser realizada a comunicação entre o microcontrolador e uma *ground station*, para que sejam analisados os dados de medição e de controle de abertura da válvula, sendo necessário apenas a modificação dos dados a serem enviados. Logo, é possível interligar estes projetos, de modo a conseguir implementar o que foi proposto.

Wemerson Fontenele Sousa é estudante de Graduação em Engenharia Eletrônica pela Universidade de Brasília - UnB, Brasília, Brasil. Email: wemersonfont@gmail.com - Matrícula: 17/0024130

Lucas Gonçalves Campos é estudante de Graduação em Engenharia Eletrônica pela Universidade de Brasília - UnB, Brasília, Brasil. Email: lucassgcampos@gmail.com - Matrícula: 17/0016757

II. Desenvolvimento

A. Descrição do Hardware

O Hardware do projeto foi estruturado visando atender às funções explicitadas na Figura 1.

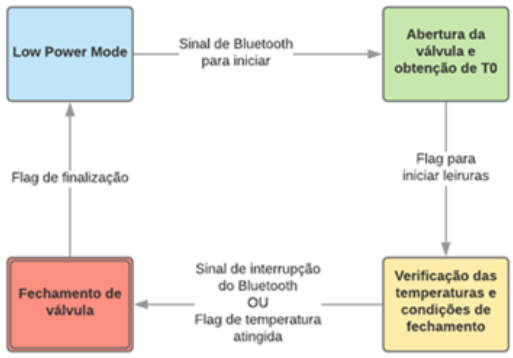

Figura 1. Diagrama de blocos do funcionamento do software desenvolvido.

Assim, para atender à condição do fechamento e abertura da válvula, usou-se um motor de passo, considerando a necessidade de um controle preciso. O modelo 28BYJ-48 foi determinado para a função por conta do seu custo-benefício. Para a medição e obtenção das temperaturas citadas no esquemático, foi necessário a obtenção de um sensor de temperatura, o qual optou-se pelo LM35 devido a seu custo e maior disponibilidade no mercado - é muito utilizado para protótipos em microcontroladores. Quanto ao Módulo Bluetooth, este foi selecionado para realizar a comunicação entre a *ground station*, sendo determinado o modelo HC-06 pelos mesmos motivos do LM35.

Desta maneira, a figura 2 expõe os componentes e respectivos modelos, assim como suas ligações e direção de comunicação entre eles. As ligações entre os componentes foram montados sobre uma *protoboard* através de *jumpers*.

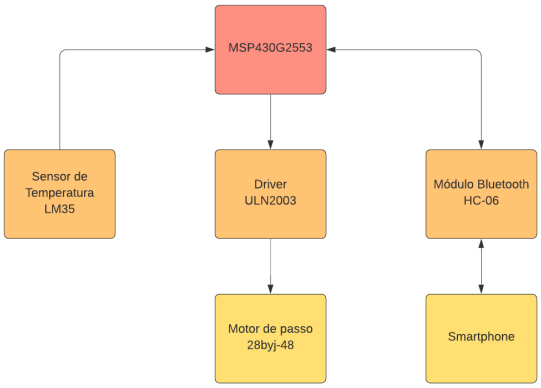

Figura 2. Diagrama de blocos do Hardware do projeto.

B. Descrição do Software

Para iniciar o projeto, foram organizadas as ideias e desenvolvida uma máquina de estados simples para servir de base para a programação do projeto. Tal diagrama está representado na Figura 1.

Através do diagrama de blocos é possível verificar que foi definido que a MSP se manteria em *Low Power Mode* até o recebimento do sinal de *bluetooth* enviado pelo usuário através da *ground station*, que no caso se trata de um aplicativo presente no celular. Após isso, serão realizados pelo sistema uma primeira leitura de temperatura (obtenção de T0) e a abertura da válvula. Após isso, o circuito entra em um *loop* onde ele realiza leituras consecutivas de temperatura, até a temperatura variar em relação a T0 um valor ΔT (o que ativaria a *flag* de temperatura atingida) ou ser recebido via *bluetooth* um sinal de desligamento emergencial. Após isso, é realizado o fechamento da válvula e o retorno ao estado inicial (*low power mode*).

Em se tratando do código em si, este foi dividido em várias funções para diminuir sua complexidade e facilitar seu entendimento. Tendo isso em vista este foi desenvolvido da seguinte maneira:

Primeiramente, foram desenvolvidas três funções convenientes para o controle do motor de passo: uma função para girá-lo em sentido horário (abertura), uma função para girá-lo em sentido anti-horário (fechamento) e uma função de *delay* de X ms, sendo X determinado pelo usuário, sendo esta função necessária para o controle de velocidade do motor, além de ser usada para controlar o tempo entre as medições de temperatura, o que será explicado posteriormente. A função atraso foi desenvolvida em *assembly*, por se tratar de uma função crítica para várias etapas do funcionamento do projeto, e fazê-la nesta linguagem traz grande otimização ao código como um todo, além de trazer um maior precisão sobre o atraso por estar se comunicando em nível de máquina com a MSP.

Após isso, foi desenvolvido uma função (leitura) para obter a temperatura medida pelo sensor LM35, para tal, esta se utiliza de um conversor AD (também desenvolvido em assembly para se tornar mais confiável), e de uma operação matemática que converte esse valor do conversor para o valor de temperatura em graus celsius. Este código realiza o processo de leitura vinte vezes, e após isso, faz a média aritmética dessas medições para se obter maior precisão dessas medidas, sendo essa média necessária para aumentar a precisão da leitura. Sendo esta função configurada de maneira a ser realizada uma medida inicial que é guardada para comparações e após isso, entrando em um *loop* de medições, até que seja detectada uma interrupção que acionaria o fechamento da válvula, ou que a temperatura varie um determinado ΔT, que também acionaria o fechamento da válvula. Entre as leituras, também foi usada a função de atraso para diminuir erros por alta velocidade de processamento.

Após isso, foram desenvolvidas as funções necessárias para o uso do módulo *bluetooth* (HC-06), sendo elas:

- A função Init_Uart, responsável por iniciar as flags necessárias para o uso das portas RX e TX para tal meio de comunicação e escolher o *baud rate* para o módulo bluetooth HC-06, no caso, 9600;
- A função Receive_Data, responsável por realizar a leitura de cada caractere dos dados enviados à placa.
- A função Read_string, responsável por traduzir os dados obtidos pela função anterior em uma *string*;
- A função Send_Data responsável pelo envio de dados;
- A função Send_String, usada para enviar mensagens maiores para o aplicativo;
- A função Send_int responsável pelo envio de dados numéricos do tipo Int para o aplicativo;
- A função Send_float que é responsável por enviar variáveis do tipo float para o aplicativo (esta se utiliza da função Send_int).

Após isso, foi realizada a união de todas estas funções. Para isso, foi desenvolvido o código de maneira a placa permanecer em LPM1, até receber uma interrupção. A única rotina de interrupção vem do bit RX, recebido quando enviada uma mensagem pelo aplicativo de celular, essa rotina identifica os dados recebidos e através de uma nova função (cmp_str), que identifica se é uma mensagem para a válvula ser aberta, fechada; após isso, essa rotina compara o comando enviado e o estado atual e identifica se este comando deve abrir a válvula, fechá-la, ou se este é um comando inválido (seja por estar em um estado errado, ou pela mensagem ser inválida).

Todos os códigos desenvolvidos estão disponíveis nos apêndices do relatório. Estando eles distribuídos da seguinte maneira:

- Figura 3 contendo o código desenvolvido em C, a *main.c*, que contém a maioria das funções utilizadas no projeto, como já descrito anteriormente *bluetooth*.
- Figura 4 contendo o código em *assembly* responsável por realizar a conversão AD dos dados obtidos pelo sensor de temperatura
- Figura5 contendo o código em *assembly* responsável por realizar o atraso de X ms utilizado para gerar atrasos em áreas críticas já explicadas anteriormente.
- Figura 6, contendo o *header* desenvolvido para realizar a ligação dos códigos desenvolvidos em *assembly* e o código *main* desenvolvido em C;

III. Resultados

Após a conclusão dos códigos foram realizados testes para validar o funcionamento do projeto, iniciando pela parte de abertura da válvula, testando inicialmente a conexão módulo *bluetooth*-celular. Primeiramente, com o envio de um comando qualquer (que não deveria fazer nada) e o programa respondeu corretamente: "Comando Inválido!). Após isso, foi testado o envio do comando de abertura do motor, e o programa respondeu corretamente: "Abrindo Válvula!", além disso, é possível reparar o motor girando no sentido horário (sentido de abertura da válvula, já que o motor está posicionado na parte superior da válvula que abre no sentido anti-horário). Portanto, no que diz respeito ao funcionamento do código para abrir a válvula, este está funcionando corretamente.

Após isso, foi testado o funcionamento do código com a válvula aberta, sendo primeiro testado o recebimento dos dados de temperatura, que responderam próximos ao esperado. Então foi testado o sistema de fechamento da válvula, inicialmente testando o envio de um fechamento de emergência, efetuado pela própria *ground station* (novamente, um envio qualquer, retornando "comando inválido!", e após isso, um envio correto de fechamento) e o programa respondeu como esperado, girando o motor no sentido anti-horário, ou seja, fechando a válvula. Após isso, foi posicionado um potenciômetro no lugar do sensor de temperatura (para simular uma variação brusca de temperatura) e foi testado inicialmente com o potenciômetro parado (a válvula não fechou), e então foi girado ele de maneira a simular a variação na temperatura e o motor girou no sentido anti-horário, representando o fechamento da válvula. Além disso, após o fechamento da válvula, o código não mais envia dados de temperatura para a "ground station", o que indica sua volta ao low power mode e o funcionamento correto do código.

IV. Conclusão

Ao contemplar o material exposto neste documento, é possível notar que o projeto é de grande valia para a resolução do problema proposto. Os testes realizados demonstram um funcionamento dentro das expectativas, com as médias das temperaturas capturadas sendo visualizadas com uma precisão de até duas casas decimais. A temperatura inicial é capturada no início da operação e a válvula é fechada por interrupção através de botão ou através de uma variação, para o caso dos testes, de 6^{o}C, como o esperado.

Algumas alterações e adequações trariam melhores resultados quanto à precisão e acurácia dos dados, como a adição de novos sensores e a inclusão de outros, os quais serviriam para cruzar dados e adquirir novos parâmetros para diminuir potenciais erros de medição. Também seria possível incluir um tratamento e análise dos dados recebidos dos sensores para aumentar a precisão dos dados e para considerar características como o gradiente de temperatura, pressão, umidade, etc. Uma válvula solenoide poderia substituir a válvula de abastecimento, de maneira a permitir maior praticidade e controle do abastecimento e assim facilitar a integração com um controlador geral, por exemplo.

Referências

[1] Karuppuswamy. Temperature Monitor using MSP430 Launchpad and LM35 Temperature Sensor. Disponível em: <http://karuppuswamy.com/wordpress/2016/10/15/temperature-monitor-using-msp430-launchpad-and-lm35-temperature-sensor/>. Acesso em 10/12/2019.

[2] Boruro. MSP430G2553 conectado al módulo bluetooth HC-06. <http://boruro.com/blog/2016/04/30/msp430g2553-conectado-al-modulo-bluetooth-hc-06/>. Acesso em: 10/12/2019.

[3] Circuit Digest. Interfacing Stepper Motor with MSP430G2. Disponível em: <https://circuitdigest.com/microcontroller-projects/interfacing-stepper-motor-with-msp430>. Acesso em: 04/10/2019.

APÊNDICES

A. Main.c

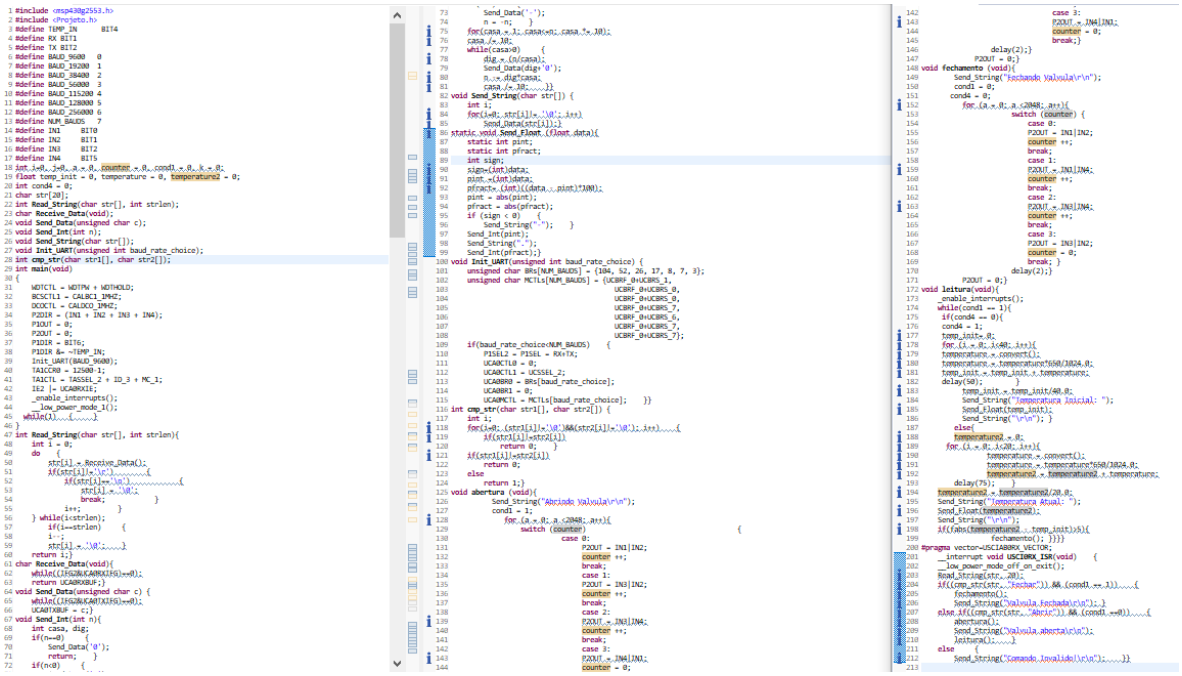

Figura 3. Código principal do projeto, desenvolvido em liguagem C.

B. Conversor AD

```
1            .cdecls C,LIST,"msp430.h"       ; Include device header file
2            .text                           ; Assemble into program memory.
3            .global convert                 ; define convert como global para ser usada
4                                            ;em c
5  convert:
6               mov.w    #SREF_0+ADC10SHT_2+ADC10ON, ADC10CTL0   ;declarando conversor AD
7               mov.w    #INCH_4+SHS_0+ADC10DIV_0+ADC10SSEL_0+CONSEQ_0, ADC10CTL1; SELECIONANDO P1.
8               mov.w    #BIT4, ADC10AE0                        ; selecionando p1.4
9               add.w    #ENC, ADC10CTL0
10 LEITURA:
11              add.w #ADC10SC, ADC10CTL0                       ; inicia conversão
12 LOOP:
13              bit.w    #BUSY, ADC10CTL1                       ; espera terminar conversão
14              jnz      LOOP
15              mov.w    ADC10MEM, R12                          ; após converter, salva em R12 (padrão
16              ret                                             ; finaliza subrotina
17              .end
```

Figura 4. Código desenvolvido em *assembly* responsável por realizar a conversão AD dos dados de temperatura.

C. Atraso de X ms

```
1            .cdecls C,LIST,"msp430.h"       ; Include device header file
2            .text                           ; Assemble into program memory.
3            .global delay                   ; define delay como global para ser usada
4                                            ;em c
5  delay:
6               mov.w    #999,   TACCR0  ;999        define variáveis do clk
7               add.w    #TACLR,     TACTL   ;       para obter delay de 1 ms
8               mov.w    #TASSEL_2+ID_0+MC_1, TACTL;
9
10 comp:        ; função para identificar se já se passou o tempo pedido pelo usuário
11              bit.w    #65535, R12     ; entrada do tempo de delay vem em R12 (padrão do CCS)
12              jz       end ; se for zero, vai pra função end, caso contrário vai para loop
13 loop:
14              bit.w    #TAIFG, TACTL ; testa se já se passou 1 ms
15              jz       loop ; fica em loop até ter rodado 1 ms
16              add.w    #65535, R12; subtrai 1, 65535 é o complemento de dois de 1
17              jmp      delay; após subtrair 1, volta para a comparação
18 end:
19              add.w     #MC_0, TACTL; ativa mc0 pra economizar energia
20              ret                     ; finaliza subrotina
21              .end
```

Figura 5. Código desenvolvido em *assembly* responsável por realizar o atraso de X ms.

D. Header

```
#ifndef PROJETO_H_
#define PROJETO_H_
//header responsável por chamar as funções em assembly para serem usadas em C
void    delay(int T_ms);
int     convert();
#endif
```

Figura 6. Código desenvolvido para realizar a ligação entre os arquivos em C e os arquivos em *assembly*.