SENSORIAMENTO NÃO-INTRUSIVO PARA UM SISTEMA DE FRENAGEM

GUILHERME GIL DE BRITO CAMPBELL MARQUES

TRABALHO DE GRADUAÇÃO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

Sensoriamento não-intrusivo para um Sistema de Frenagem

GUILHERME GIL DE BRITO CAMPBELL MARQUES

Trabalho final de graduação submetido ao Departamento de Engenharia Elétrica da Faculdade de Tecnologia da Universidade de Brasília, como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista.

APROVADA POR:

Prof. Daniel Chaves Café, D.Sc. (ENE-UnB)
(Orientador)

Prof. Ricardo Zelenovsky, D.Sc. (ENE-UnB)
(Examinador Interno)

Prof. Adson Ferreira da Rocha, D.Sc (FGA-UnB)
(Examinador Interno)

Brasília/DF, Julho de 2017.

FICHA CATALOGRÁFICA

GUILHERME GIL DE BRITO CAMPBELL MARQUES

Sensoriamento não-intrusivo para um Sistema de Frenagem. [Distrito Federal] 2017.

xiii, 33p., 210 x 297 mm (ENE/FT/UnB, Engenheiro Eletricista, Engenharia Elétrica, 2016).

Trabalho de Graduação – Universidade de Brasília, Faculdade de Tecnologia.

Departamento de Engenharia Elétrica

1. Sistema Embarcado
2. MSP430
3. Comunicação Serial
4. MPU6050

I. ENE/FT/UnB

II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

MARQUES, G. G. D. B. (2016). Sensoriamento não-intrusivo para um Sistema de Frenagem, Trabalho de Graduação em Engenharia Elétrica, Publicação 2017, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 52p.

CESSÃO DE DIREITOS

AUTOR: Guilherme Gil de Brito Campbell Marques

TÍTULO: Sensoriamento Não-intrusivo para um Sistema de Frenagem.

GRAU: Engenheiro Eletricista ANO: 2017

Guilherme Gil de Brito Campbell Marques

Departamento de Eng. Elétrica (ENE) - FT

Universidade de Brasília (UnB)

Campus Darcy Ribeiro

CEP 70919-970 - Brasília - DF - Brasil

AGRADECIMENTOS

Agradeço aos meus pais, Lúcia Clara e Mauro Luiz, pelo apoio e carinho me proporcionando as condições para me dedicar aos estudos.

À minha irmã Manoella por seu apoio e amizade que me ajudaram em tempos difíceis.

À minha namorada Letícia, que no pouco tempo que estamos juntos trouxe grandes mudanças à minha vida. A ela também agradeço à inestimável ajuda na conclusão deste trabalho.

Aos professores e funcionários da Universidade de Brasília por proporcionarem as ferramentas e instrução para minha formação.

Ao professor Daniel, pela orientação neste trabalho permitindo ultrapassar dificuldades e aprofundar conhecimentos e experiência.

Aos colegas e amigos da Universidade de Brasília que juntos, conseguimos ultrapassar barreiras dividindo experiências e conhecimento.

Aos meus queridos amigos Gabriel, Paulo, Alexsandro, Jeremy e Iago pelo companheirismo, amizade e auxílio nos últimos anos.

RESUMO

Este projeto tem como objetivo apresentar uma solução para a luz de freio de uma bicicleta utilizando métodos não invasivos e contidos em apenas uma caixa, acoplada ao canote de selim. Para isso será usado um sistema embarcado composto por um sensor com acelerômetros e giroscópios conectado a um microcontrolador por meio de comunicação serial. A plataforma que possui os sensores é o MPU6050 (MPU), fabricada pela InvenSense. Já o microcontrolador deste projeto é o MSP430F5529 (MSP), fabricado pela Texas Instrument (TI). Por fim, para realizar a programação é utilizado o software Code Composer Studio, também da TI.

Palavras-chave: microcontrolador, MSP430, MPU6050, comunicação serial, sinalização de freio, não-invasivo.

ABSTRACT

This project aims to present a solution to the brake light of a bicycle using noninvasive methods and contained in one box, attached to the saddle seatpost. For this purpose it is used an embedded system comprises a sensor with accelerometers and gyroscopes connected to a microcontroller via serial communication. The platform which has sensors is MPU6050 (MPU), manufactured by InvenSense. The microcontroller of this project is the MSP430F5529 (MSP), manufactured by Texas Instruments (TI). Finally, to make the programming the software Code Composer Studio software, also from TI, is used.

Keywords: microcontroller, MSP430, MPU6050, serial communication, brake signal, non-invasive.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

GLOSSÁRIO

MPU	MPU-6050 ou MPU-60X0
MSP	MSP430 ou MSP430f5529
CCS	*Code Composer Studio*
UnB	Universidade de Brasília

CAPÍTULO 1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO DO TEMA

No Brasil, o uso do modal ciclístico aumentou nos últimos anos. A Organização Não Governamental Transporte Ativo e o laboratório de mobilidade da UFRJ realizaram uma pesquisa em dez capitais brasileiras, foi registrado que 45% dos entrevistados em 2015 eram novos ciclistas [1] . Estes novos usuários do modal são pessoas que passaram a utilizar a bicicleta como meio de transporte nos últimos dois anos. Além disso, pesquisas do IBOPE que houve aumento de 50% do número de ciclistas regulares, na cidade de São Paulo, entre 2012 e 2017 [2].

Com o aumento do número de ciclistas, há naturalmente o crescimento da procura de acessórios. De acordo com o Código de Trânsito Brasileiro, capítulo IV, artigo 105, sessão VI, é obrigatório utilizar em bicicletas: campainha, sinalização noturna dianteira, traseira, lateral e nos pedais, além do espelho retrovisor do lado esquerdo.

A sinalização noturna é mais eficiente usando dispositivos que, ao contrário do tradicional refletor olho de gato, emitem luz própria. Dentre esses dispositivos, os mais utilizados são os sinalizadores baseados em LED, devido à alta luminosidade e baixo consumo de energia. As tradicionais luzes para bicicleta emitem padrões pré-definidos de cores com o objetivo de chamar a atenção para a presença do ciclista. Outros sistemas mais sofisticados alteram seus padrões luminosos de acordo com uma instrução. A maioria das instruções são para indicar conversão lateral (setas) ou freio.

A problemática do sinal de freio é a forma de sinalizar ao dispositivo que está ocorrendo a frenagem. Algumas soluções já são bem conhecidas e disseminadas. Entre elas está o uso de cabos ligados às pastilhas ou aos manetes de freio. Quando as pastilhas e/ou manetes de freio

[1] http://g1.globo.com/jornal-nacional/noticia/2016/01/cresce-o-uso-da-bicicleta-como-meio-de-transporte-em-cidades-brasileiras.html

[2] http://folhanobre.com.br/2017/02/09/ciclismo-em-sao-paulo-teve-aumento-de-50-segundo-ibope/44709

são acionados ocorre o envio de sinal de frenagem para o dispositivo, que reage de acordo com a sua instrução.

O método citado acima apresenta uma clara problemática. É intrusivo, necessitando instalação de cabos em espaços limitados. Se esse processo for mal executado ou ocorrer desgaste como o tempo, isso pode interferir no uso da bicicleta. Os cabos podem atrapalhar o uso do guidão, do *headset* ou dos pedais. De forma geral a instalação de cabos não é atrativa.

Outro método é a utilização de comunicação sem fio. No guidão da bicicleta é instalado um dispositivo de comunicação sem fio com o farol traseiro. O dispositivo do guidão possui sensores de pressão que são instalados no manete de freio. Quando eles são acionados, o farol recebe o sinal. Esta forma é menos intrusiva, mas ainda requer a instalação de alguma parte móvel.

Considerando os pontos acima, não há produtos que estejam contidos apenas na caixa presa ao canote de selim. Todos eles requerem algum tipo de instalação.

1.2 OBJETIVOS DO TRABALHO

O projeto tem como objetivo apresentar uma solução para a luz de freio de uma bicicleta utilizando métodos não invasivos e contidos em apenas uma caixa, acoplada ao canote de selim. Para isso foi utilizado um sistema embarcado composto por um sensor com acelerômetros e giroscópios conectado a um microcontrolador por meio de comunicação serial.

CAPÍTULO 2

METODOLOGIA E FERRAMENTAS

Como ferramentas para o projeto será utilizado o microcontrolador MSP430F5529 acoplado à plataforma MPU-6050 por interface de comunicação serial. A programação e *debug* será realizada com o programa /tesxtitCode Composer Studio. Por fim, as saídas serão obtidas a partir de uma Máquina de Estados.

2.1 TECNOLOGIA USADA

Para tratar os dados e implementar a lógica deste projeto, será utilizado o microcontrolador MSP430 da Texas Instruments. Microcontroladores são pequenos sistemas compostos tipicamente por uma unidade central de processamento (CPU), memória para o programa, memória de dados, portas de entrada e saída, e outros periféricos. Microcontroladores possuem espaço de memória finito o que limita sua capacidade ao mesmo tempo que os tornam baratos e simples.

O MSP430F5529 (para abreviar, será usado o termo MSP430 OU MSP) é um microcontrolador de quinta geração da família MSP430 que inclui na sua arquitetura o processador CPUX e um barramento de endereços de 20-bits, fabricado pela Texas Instruments. Ele é utilizado principalmente em aplicações de baixo consumo de energia por ter a capacidade de desligar individualmente seus periféricos através de diversos modos de baixo consumo. Entre seus principais recursos está a utilização de um sistema de *clock* configurável podendo atingir até 25MHz, cinco *timers*, múltiplas interfaces de comunicação serial, diversos modos de *lowpower* e um conversor analógico para digital.

2.1.1 Memória

A memória *flash* é uma memória não-volátil, ou seja, quando seu suprimento de energia é cortado, suas informações são retidas, sem ocorrer perdas. Este tipo de memória é especialmente útil para a escrita do programa e de constantes. Ao contrário da tradicional memória não-volátil do tipo ROM (*read only memory*), a tipo *flash* pode ser reescrita. Para reescrever, é necessária a aplicação de uma tensão acima da normal de operação. O MSP possui memória *flash* de 128KB. Como o sistema deste projeto não ficará energizado a todo momento, por exemplo na troca de bateria, é essencial ter uma memória de código não-volátil e a possibilidade de reescrevê-la para ajustes no código.

O MSP também possui memória 10KB de memória RAM. Este tipo de memória é volátil, significando que perde suas informações quando ocorre corte de suprimento de energia. Além disso, com essa memória há a possibilidade de escrever e ler com igual facilidade.

A memória RAM pode ser estática ou dinâmica. A primeira requer seis transistores por

bit, o que se traduz em maior demanda por área no semicondutor. Já a segunda requer apenas um transistor, porém precisa ser revisitada periodicamente para não perder seus dados. Em microcontroladores, a RAM estática é mais comumente empregada, pois, apesar de precisar de mais espaço, é mais econômica em termos de energia e, caso o *clock* seja interrompido, não perde suas informações (DAVIES, 2008). Para o projeto, a memória RAM será utilizada para armazenar dados coletados pelo sensor e variáveis de controle.

2.1.2 Temporizador

O MSP possui cinco temporizadores (*timers*) sendo que apenas o *timer* A será utilizado. Esta funcionalidade é bastante versátil por ser um contador de 16 bits com até sete comparadores e módulos de captura. Além disso o timer permite a escolha de clock e possui várias opções de interrupções (DAVIES, 2008).

O *timer* A pode ser dividido em duas partes principais: o bloco de *timer* e os canais de captura e comparação (DAVIES, 2008). O primeiro apresenta a escolha de fonte de *clock*, sendo que a frequência percebida pelo *timer* pode ser reduzida. Com cada subida do *clock* selecionado, ocorre o incremento ou decremento do registrador de 16 bits TAxR, a depender se foi selecionada a contagem progressiva ou regressiva. O sinal de *clock* pode ser alterado dividindo sua frequência por 2, 4 ou 8 inicialmente e, em seguida, novamente por 2, 3, 4, 5, 6, 7 ou 8, permitindo versatilidade na rapidez de contagem. Este bloco possui como saída uma *flag* (TAIFG) que é colocada em nível lógico alto quando o contador retorna a 0.

O segundo bloco contém os módulos de comparadores e de captura. Para este projeto será utilizado apenas a parte de comparadores com o bloco de *timer* em *up mode*. Neste modo, o *timer* conta progressivamente até o valor escolhido para o registrador TAxCCR0, e faz TAxR = 0. Como mencionado anteriormente este evento torna TAIFG verdadeira.

Outro *timer* que merece ressalva é o temporizador de 16 bits *Watchdog*. Esta função tem como objetivo reiniciar o sistema caso ocorra falhas no *software*. Quando o contador alcança seu valor máximo ocorre a reinicialização, sendo necessário interrompê-lo caso não seja utilizado. Este *timer* é útil como medida preventiva a estados inesperados de *software*.

2.1.3 Comunicação Serial

O MSP430 possui módulos de comunicação serial capazes de selecionar alguns protocolos de comunicação serial. Entre eles o padrão I^2C será utilizado na interface com o módulo externo MPU6050. De acordo com Davies (2008), o protocolo para o microcontrolador tem como atributos:

- Conformidade com a Especificação I^2C NXP para semicondutores;
- Chamada de endereços de 7-bit e 10-bit;
- *General call*;
- START/RESTART/STOP;
- Modo de múltiplos mestres transmissores e receptores;
- Modo escravo transmissor/receptor;
- 100kbs em operação normal e 400kbs em modo rápido;
- Fonte de *clock* programável;
- Projetado para *low power*;
- Na condição de escravo, pode aguardar em modo *low power* pelo START.

A comunicação com o MPU6050 será mais aprofundada em sua própria sessão.

2.1.4 Programação do MSP430

O MSP possui diversos recursos como *timers*, *clocks*, interfaces de comunicação serial e conversor analógico para digital. Para programar corretamente esses recursos fazemos uso dos registradores internos.

Cada recurso necessita de ajustes para sua correta implementação. O INSTRUMENTS (2015) fornecido pela fabricante, Texas Instrument, apresenta um excelente manual sobre quais registradores e procedimentos devem ser adotados para aproveitar de forma plena os recursos embutidos. O guia apresenta o mapa de registradores com a descrição da função de cada bit.

Para permitir que as portas de entrada/saída sejam utilizadas para comunicação serial, por exemplo, deve-se checar quais possuem opção para a comunicação desejada (I^2C). No caso de se escolher as portas P3.0 e P3.1., segundo INSTRUMENTS (2015) faz-se necessário implementar

a função dedicada alterando os bits do registrador P3SEL correspondentes a P3.0 e P3.1 para nível lógico alto (*high*, 1, verdadeiro, *true*). Para comunicação no protocolo I^2C é necessário que as linhas fiquem em *high* enquanto nenhum dispositivo esteja às utilizando. Para isso, deve-se empregar resistores externos de *pull-up* ou os próprios registradores das portas, alterando P3REN e P3OUT para *high* em P3.0 e P3.1. Com essas alterações as portas utilizadas para comunicação serial estão corretamente implementadas.

2.2 O CODE COMPOSER STUDIO

O Code Composer Studio (CCS) é um ambiente de desenvolvimento integrado (IDE) que suporta microcontroladores e microprocessadores do catálogo da Texas Instruments (TI) (CODE...,). O CCS possui compilador C/C++, *debugger* entre outras ferramentas. O programa é gratuito e pode ser encontrado no sítio da TI, www.ti.com.

2.2.1 Tutorial

Após instalar o CCS, inicie-o. Será solicitada a pasta onde você deseja que fique o projeto, como mostrado na Figura 2.1. Selecione a pasta e clique em "OK". Aguarde a inicialização.

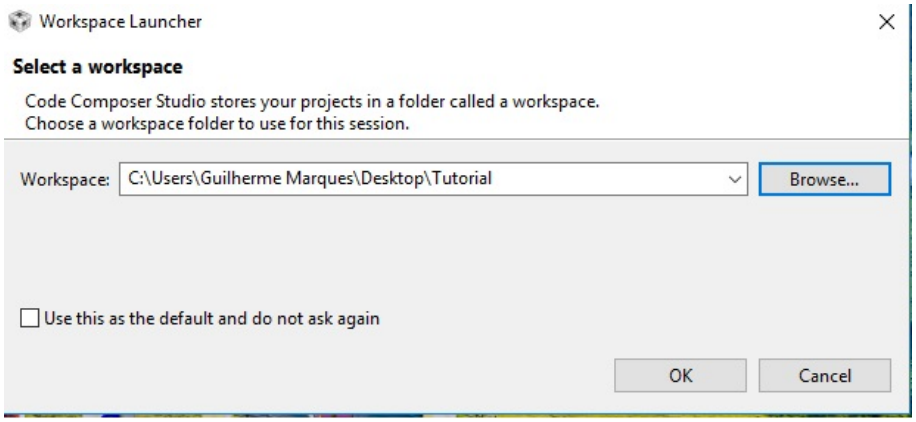


Figura 2.1. Inicialização CCS. Escolha de pasta.

Para criar um novo projeto vá em Project » New CCS Project, conforme a Figura 2.2.

A janela apresentada em Figura 2.3 será aberta. O microcontrolador utilizado neste projeto é o MSP430f5529 e é isso que será colocado em "Target:", como está no retângulo vermelho. Escolha um nome para seu projeto e escreva na caixa "Project name:". Para este caso o programa será escrito na linguagem C, então, na caixa destacada em verde selecione "Empty Project (with main.c)" ou algum dos códigos exemplos mais abaixo. Nas próximas imagem, será considerado que o código exemplo "Blink the LED" foi selecionado. Em seguida, clique em "Finish". O seu projeto está criado.

O projeto "Blink the LED"apresenta um código que faz com que o LED vermelho pisque. Para implementar este código basta clicar em "debug", representado por um ícone de besouro e destacado por círculo vermelho, como apresentado na Figura 2.4.

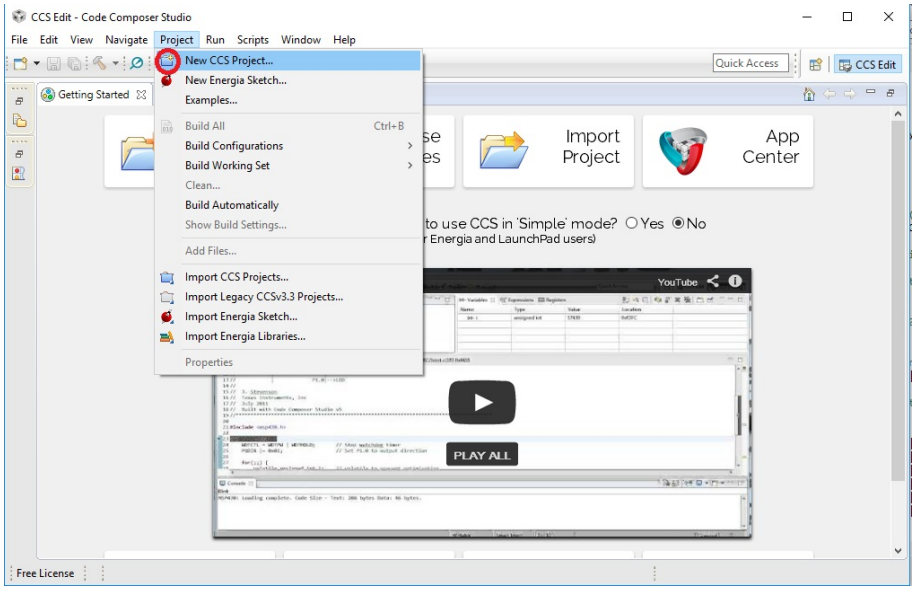


Figura 2.2. Tela inicial do CCS.

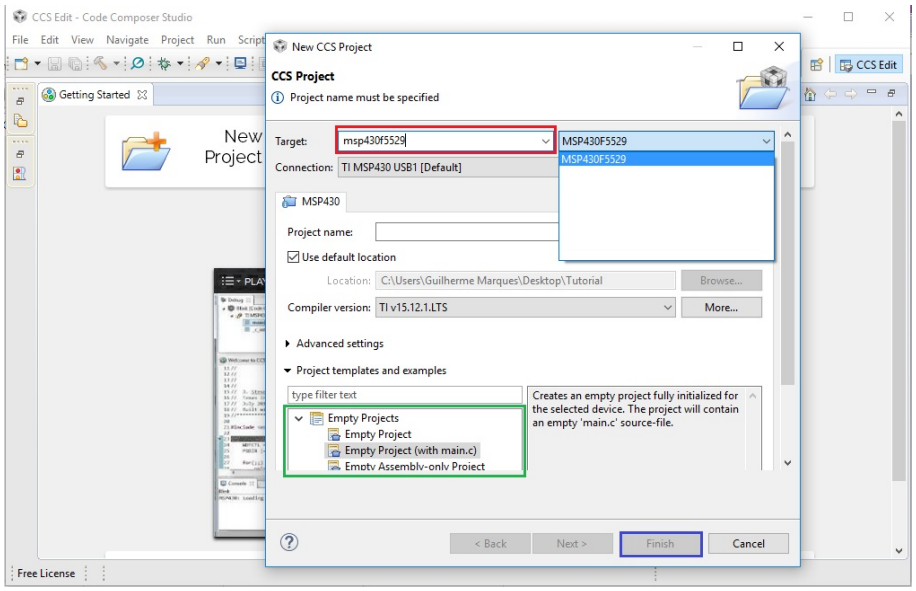


Figura 2.3. Criando projeto no CCS

Caso não haja nenhum erro no código, avance e aguarde o término do *debugger*. Quando chegar à janela apresentada na Figura 2.5, utilize os comandos de "Resume(F8)", "Suspend(Alt+F8)"e "Terminate(Ctrl+F2)"destacados no retângulo vermelho. Valores das variáveis locais, expressões, variáveis globais e registradores, só poderão ser corretamente lidos quando o programa está em pausa (*Suspend*). É possível ver em que linha de código o MSP atualmente se encontra pela pequena seta azul, que está melhor sinalizada através da grande seta azul.

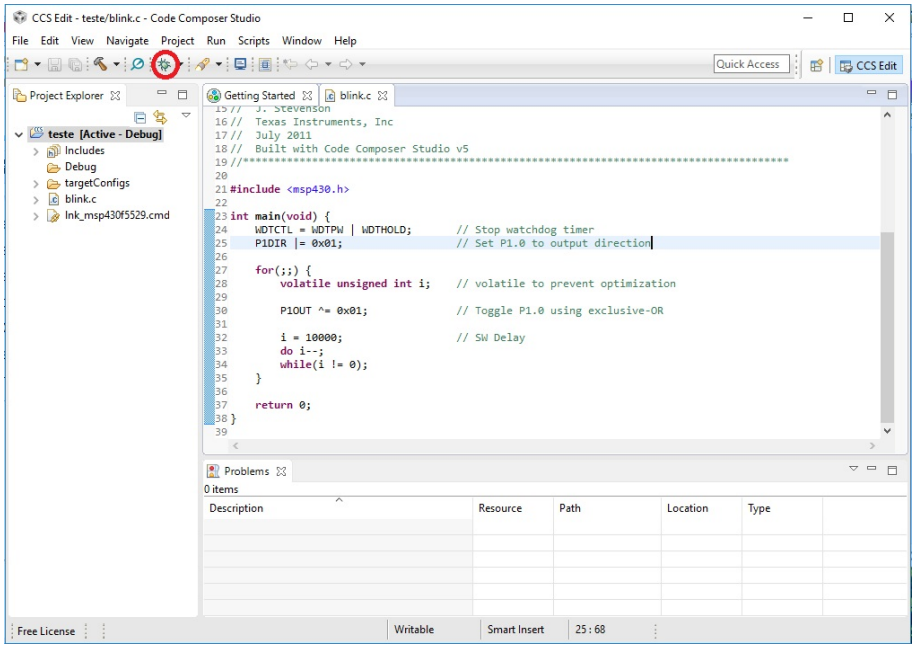


Figura 2.4. Ativando *debbuger* no CCS

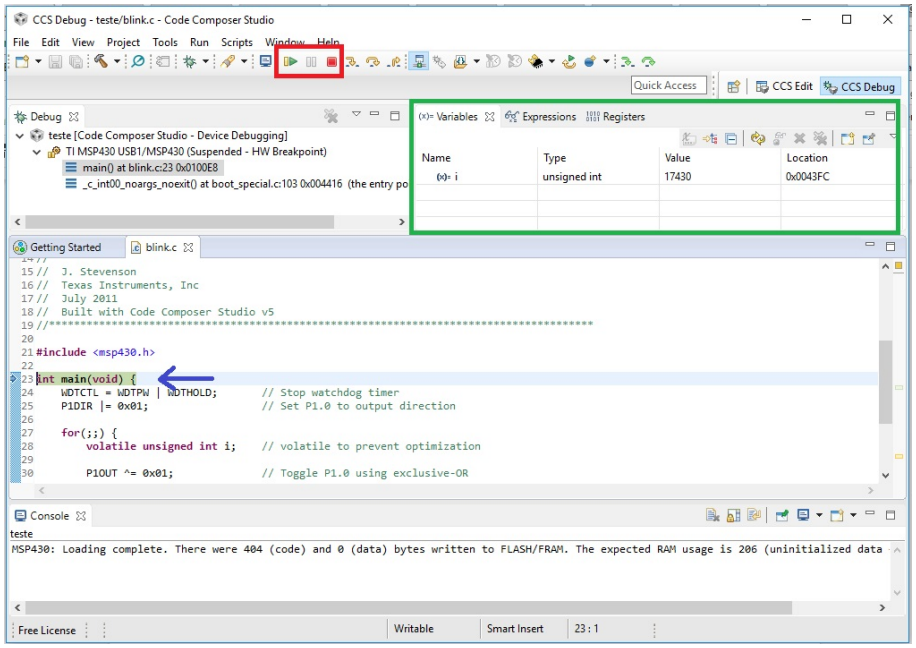


Figura 2.5. Tela de *debbug*

2.2.2 Etapas de programação de um sistema embarcado

Para o projeto de um sistema embarcado é importante lembrar que a memória é limitada, sendo necessário ter atenção ao tamanho do código e de área para variáveis. Funções demoradas precisam de mais memória e mudanças do *clock*, devendo ser evitadas. O CCS auxilia nesse quesito, informando funções, operações e *loops* problemáticos, que consomem muita memória ou tempo. Conhecer o tamanho necessário para cada variável também é importante para reduzir

a capacidade utilizada.

Os microcontroladores, ao contrário dos microprocessadores, para economizar memória, não possuem sistema operacional. Com o objetivo de contornar esta dificuldade e transferir o programa para o MSP, o CCS utiliza compilação cruzada. Este tipo de compilação produz um código executável em uma plataforma diferente da utilizada pelo compilador.

O *linker* permite realizar a ligação entre os arquivos objetos criados pelo compilador em um arquivo executável. Comumente apenas as funções efetivamente utilizadas são as aplicadas pelo *linker* para o arquivo executável. Essa capacidade permite economia de memória, importante para o microcontrolador.

Uma importante parte para o projeto é utilizar o *debug*. De acordo com (4) para o cenário de debug, o CCS tipicamente segue os procedimentos abaixo:

1. O *debugger* lê o arquivo de configuração do alvo (alvo sendo o MSP), cria a configuração de *debug*. Com essas informações, se conecta ao JTAG *debugger* (interface entre o CCS e o alvo) e se comunica com o MSP.

2. Quando a comunicação é estabelecida, o *debugger* do CCS executa a inicialização do hardware.

3. Se um executável está sendo carregado, o *debugger* do CCS divide a informação:

   (a) O código e os dados são enviados pelo JTAG e alocados nas posições apropriadas de memória do alvo. Este processo segue as diretrizes do arquivo *linker*.

   (b) Os símbolos de *debug* são guardados no computador para realizar a correlação de memória entre o alvo e o código.

4. Por fim, o *debugger* adiciona *breakpoints* de acordo com o estabelecido na função main() e inicia o alvo até que este ponto seja atingido.

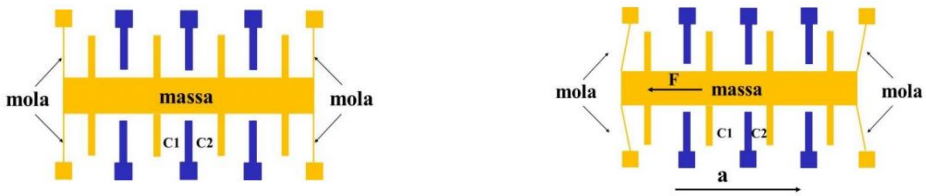
Para realizar a interface de *debug* entre o CCS e o MSP, é utilizado o eZ-FET. Este dispositivo é não-intrusivo, ou seja, permite ao usuário utilizar o microcontrolador alvo a velocidades normais de operação e utilizar interrupções no código (*breakpoints*)(EZ-FET...,).

2.3 O MPU-6050

Utilizaremos o acelerômetro do MPU-6050, uma plataforma que possui giroscópios e acelerômetros em 3 direções, bem como um processador digital de movimento e um sensor de temperatura. A plataforma possui como características atraentes: o seu tamanho, que é de apenas 4x4x0.9mm e o seu baixíssimo consumo. A economia de energia se deve ao fato do processador ler os dados dos sensores e, em seguida, entrar em modo *low-power*; enquanto isso, os sensores coletam novos dados.

2.3.1 Explicação física dos sensores

O acelerômetro do MPU-6050 utiliza a tecnologia conhecida por Sistema Micro-Eletromecânico, ou MEMS (*Micro Electro Mechanical System*). Este sistema permite a criação de sensores de tamanhos extremamente pequenos capazes de medir acelerações com frequência de 0 Hz (contínuas) (INTRODUCTION...,). A Figura abaixo contém uma representação simplificada de um acelerômetro MEMS.



(a) (b)

Figura 2.6. Acelerômetro MEMS (a) Em repouso; (b) Submetido a uma aceleração "a" (ZELENOVSKY; MENDONCA,).

Na figura, há uma massa e duas "molas". As placas da porção fixa e da porção móvel formam diversos pares de capacitores. Os capacitores são denominados de C1 e C2 (a figura apresenta apenas um par de C1 e C2). Ao estar em repouso essas capacitâncias devem ser idênticas. Porém, quando a peça é submetida a uma aceleração, da esquerda para a direita, como está na figura, a inércia faz a parte móvel se movimentar na direção oposta, com isso, a capacitância de C2 fica maior que a de C1. Pela relação entre os esses dois capacitores, pode-se estimar a aceleração que a peça está submetida, vide figura 2.6.

Para o giroscópio é utilizado um MEMS com o princípio da Aceleração de Coriolis. Ela é

consequência de uma pseudo-força que se manifesta perpendicular ao objeto em movimento, caso este seja submetido a um giro. Assim, podemos utilizar o mesmo sistema do acelerômetro se o submetermos a um movimento oscilatório perpendicular à direção de medida do MEMS e ao eixo de rotação que deseja-se medir.

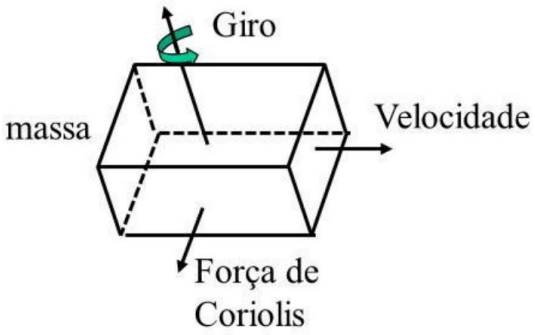


Figura 2.7. Ilustração da Força de Coriolis agindosobre uma massa que foi submetida a um giro enquanto estava em movimento(ZELENOVSKY; MENDONCA,).

2.3.2 Atributos dos sensores

Segue abaixo os atributos relevantes do acelerômetro e do giroscópio:

- Giroscópios:
  - Saída digital em três eixos de 16 bits (2 bytes);
  - Escolha de amplitude de escala programável de ±250, ±500, ±1000, ±2000 graus/segundos;
  - Aprimorado ajuste de sensibilidade à temperatura;
  - Aprimorada resposta a baixas frequências;
  - Filtro passa-baixa programável;
  - Corrente de operação de 3.6 mA;
  - Corrente de aguardo de 5 uA;
  - Auto-test (*self-test*) para usuários.
- Acelerômetros:
  - Saída digital em três eixos de 16 bits (2 bytes);

  - Escolha de amplitude de escala programável de ±2, ±4, ±8, ±16 g;
  - Corrente de operação normal de 500 uA;
  - Em modo de *low-power* a corrente de operação é reduzida;
  - Interrupções programáveis;
  - Auto-test (*self-test*) para usuários.

- Atributos adicionais:
  - *MotionFusion* de 9-Eixos com o Processador Digital de Movimento (DMP - *Digital Motion Processor*);
  - Corrente de operação de 3.9 mA com todos os 6 sensores e o DMP operantes;

Mais atributos e informações específicas podem ser adquiridas seu *DataSheet* (MPU-6000..., a).

O Processador Digital de Movimento (DMP) é uma poderosa ferramenta embutida no MPU-6050. O objetivo desse sistema é retirar a necessidade do cálculo de certos parâmetros do microcontrolador associado a ele. Infelizmente não há muitas informações referentes a esse processador, sendo necessária engenharia reversa para conseguir utilizá-lo. O motivo dessa dificuldade é porque a fabricante, InvenSense, incentiva o uso de seu *software* proprietário. No entanto, com pesquisa, é possível conseguir alguns exemplos de instruções. Por exemplo, através do DMP podemos conseguir os ângulos de inclinação do MPU.

2.3.3 Pinagens

Detalhamento das pinagens:

- VCC - Alimentação de 3 a 5V, sendo recomendado utilizar 5V;
- GND - Ground (referência);
- SCL - Clock da comunicação serial I^2C;
- SDA - Data da comunicação serial I^2C;

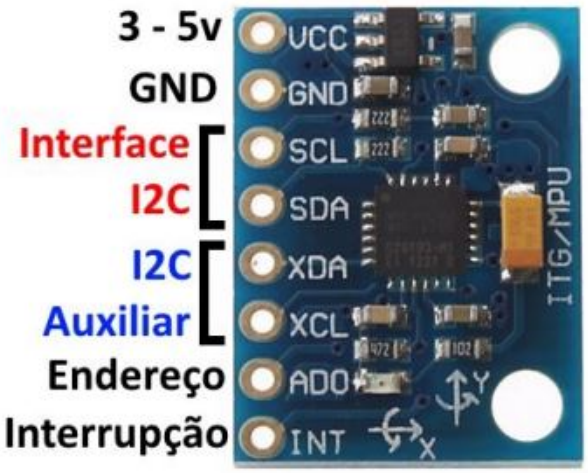


Figura 2.8. Pinos do MPU6050

- XDA - Data da comunicação serial auxiliar I^2C;
- XCL - Clock da comunicação serial auxiliar I^2C;
- ADO - Indica o valor do LSB do endereço do MPU. Se não for utilizado este pino ou for colocado em nível lógico baixo, o endereço será 0x68 (b1101000). Caso seja colocado em nível lógico alto terá endereço 0x69(b1101001). Este pino permite a utilização de múltiplos MPU-6050 em um mesmo circuito;
- INT - Utilizado para modos de interrupção.

2.3.4 Comunicação com MPU

Será apresentado o comportamento do MPU-6050 em relação a comunicação serial I^2C, esta por sua vez, não será detalhada. A frequência máxima permitida pelo processador é de 400kHz no modo rápido. Para facilitar o entendimento, será utilizada a legenda apresentada na tabela 2.1.

Para obter os endereços dos registradores, ver o mapa de registradores ((MPU-6000..., b)) do MPU-6000 ou MPU-6050.

As tabelas as tabelas 2.2 e 2.3 apresentam formas de se escrever e de se ler os registradores internos.

Na sequência da tabela 2.3, o escravo, no caso o MPU 6050, continuará escrevendo DATA até

Tabela 2.1. Legenda para comunicação I^2C

Símbolo	Descrição
S	Condição de *Start*
AD	Endereço do MPU-6050 (0x68 ou 0x69)
W	Requerimento de escrita (*write*)
R	Requerimento de leitura (*read*)
RA	Endereço do registrador
ACK	Condição de (*acknowledge*)
NACK	Condição de (*not acknowledge*
DATA	Um byte de informação
P	Condição de Stop

Tabela 2.2. Escrever em um registrador (*Single-Byte Write Sequence*) (MPU-6000..., b).

Mestre (*Master*)	S	AD+W		RA		DATA		P
Escravo (*Slave*)			ACK		ACK		ACK	

Tabela 2.3. Leitura de dados em sequência (*Burst Read Sequence*) (MPU-6000..., b).

Mestre (*Master*)	S	AD+W		RA		S	AD+R		
Escravo (*Slave*)			ACK		ACK			ACK	DATA

ACK		NACK	P
	DATA		

que ocorra um NACK do mestre seguido de um P. O primeiro DATA escrito será a informação do RA escrito, sendo o DATA seguinte o contido em RA+1 e assim por diante. Essa operação é extremamente útil para a leitura de dados, pois os valores dos sensores (depois de processados) são divididos em dois registradores de um byte cada (total de 16 bits).

2.4 MÁQUINA DE ESTADOS

Sistemas embarcados geralmente recebem estímulos vindos dos sensores ou da temporização e realizam funções dentro de um limitado número de possibilidades. Essas características favorecem o uso de máquinas de estados. Isso se aplica a este projeto.

Máquinas de Estados para programação é uma técnica de desenho de *software*, que permite a transição entre cenários. Os cenários são chamados de estados e as transições ocorrem de acordo com certas variáveis ou estímulos determinados. Nesse *design*, a máquina pode estar em apenas um estado de cada vez e abordar todas as transições possíveis de acordo com a situação proposta.

O mapa da Máquina de Estados, com entradas, saídas e transições deste projeto será apresentado mais adiante.

Para realizar as mudanças de estado será utilizada a função switch/case com uma variável de controle para identificar o estado atual.

CAPÍTULO 3

SOLUÇÕES

3.1 ESQUEMÁTICO DE CONEXÕES

O projeto possui as conexões apresentadas na Figura 3.1. O fio verde representa o CLK da comunicação I^2C e o fio amarelo representa o DATA. Para alimentação, nesse projeto, será aplicada a tensão de 5V pelo fio vermelho, com o referencial aplicado pelo fio azul.

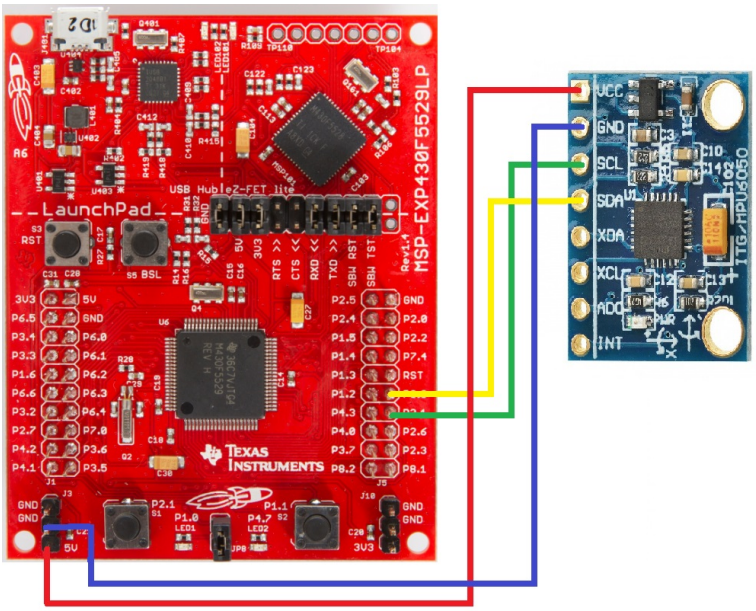


Figura 3.1. Conexão entre MSP430 e MPU6050

Para auxiliar no *debug* e realizar alguns testes, foi adicionado um *display* de LCD. Isto permite que o processo seja acompanhado sem precisar pausar o andamento do código, apresentando palavras chaves quando um ponto de controle é alcançado.

3.2 ARQUIVOS E FUNÇÕES AUXILIARES

Na construção do código foram criados arquivos de livrarias com o objetivo de facilitar e melhorar a organização. Eles possuem funções e *defines* para: portas lógicas, comunicação serial, *clock* e *timer*. A seguir serão esclarecidos alguns pontos sobre os conteúdos de cada arquivo.

Os arquivos com os códigos utilizados podem ser encontrados em <https://github.com/GuilhermeGilCampbell/Mpu-6050_with_Msp-430>.

3.2.1 Arquivos de portas

Funções e definições para as portas são apresentadas no arquivo ports.c e ports.h. Estes arquivos mostram algumas declarações para tornar o código mais intuitivo, dentre eles a função que realiza o *setup* das portas e a função que facilita o comando dos LEDs.

3.2.2 Arquivos de *clock*

É necessário realizar os ajustes de *clock* para frequências conhecidas. Segundo o guia de usuário (6), o *master clock* (MCLK) é utilizado pelo sistema e pela CPU. Neste projeto, a frequência determinada é de 16MHz. O *clock* auxiliar (ACLK) será utilizado no *timer* e é ajustado para 32.768Hz. Os ajustes apresentados acima são implementados pela função de *setup* descrita em clock.c.

3.2.3 Arquivos de *timer*

O *timer* A será utilizado ao longo do código para algumas funções. Em timer.c é aplicado os ajustes necessários pela função de *setup*. Neste arquivo, também existe uma função, que tem como objetivo aguardar a quantidade desejada de batidas do *clock* auxiliar, esta é denominada: (waitFor(entrada em ms)). Ela faz com que o MSP entre em modo de *lowpower* e aguarde que o contador chegue ao valor determinado em milissegundos. Vale ressaltar que a contagem é aproximada, não sendo necessária grande precisão em relação ao tempo real.

3.2.4 Arquivos de comunicação serial

É preciso realizar a configuração da comunicação serial I^2C. Isto ocorre na função de *setup* do arquivo serial.c. Neste arquivo, além do *setup*, são apresentadas diversas funções para facilitar o uso da comunicação serial. Elas serão aplicadas no arquivo mpu.c, que apresenta um grupo de funções para a comunicação dedicada para MPU, como explicado em 2.3.4. Para facilitar a utilização do arquivo de leitura mpu.h foram realizadas definições do mapa de registradores.

3.3 PROCEDIMENTO DE INICIALIZAÇÃO DO MPU

O MPU possui alguns procedimentos de inicialização que são necessários ou úteis para sua utilização. Os procedimentos são os seguintes:

1. Retirar o MPU do modo sleep;
2. Testar comunicação;
3. Auto teste (Self-Test);
4. Calibração;
5. Configuração para operação.

Abaixo serão detalhados cada um dos procedimentos.

3.3.1 Retirar o MPU do modo *sleep*

Este é o único procedimento obrigatório para a inicialização. Quando ocorre a energização, o MPU encontra-se em modo de baixo consumo, *sleep*. Para retirá-lo deste modo, o Bit6 do registrador PWR_MGMT_1 (*Power Management* 1) deve ser colocado em nível lógico baixo.

A fonte de *clock* utilizada ao energizar é o oscilador interno, porém, no mapa de registradores (MPU-6000..., b) é altamente recomendado que a fonte seja alterada para um dos giroscópios para melhorar a estabilidade. Para realizar essa alteração utilizam-se os bits de seleção de *clock*

(CLKSEL) do mesmo registrador, o PWR_MGMT_1. Para escolher o giroscópio do eixo x, utiliza-se CLKSEL = 1.

3.3.2 Testar Comunicação

É importante saber se as leituras estão sendo feitas corretamente. Para isso, podemos utilizar o registrador WHO_AM_I. O valor retirado da leitura deste registrador interno deve ser o endereço do MPU para a comunicação serial, ou seja 0x68. Este valor será 0x69 se o pino ADO estiver energizado.

Para testar a comunicação basta comparar o valor do registrador WHO_AM_I com o endereço utilizado.

3.3.3 Autoteste

O autoteste (*self-test*) é uma ferramenta utilizada para testar as partes mecânicas e elétricas do acelerômetro e do giroscópio. Em ambos os casos, quando o *self-test* é ativado, a eletrônica embarcada atua sobre o acelerômetro e giroscópio simulando forças, fazendo com que as massas dos sensores se desloquem por distâncias pré-determinadas (MPU-6000..., b).

Para realizar o procedimento, deve-se utilizar como escalas ±8g para o acelerômetro e ±250 dps para o giroscópio. Essas alterações podem ser realizas alterando nos registradores GYRO_CONFIG e ACCEL_CONFIG os bits 4 e 3 para 0 e 2, respectivamente. Nesses mesmos registradores pode-se ativar o *self-test* colocando os bits 7, 5 e 6 em nível lógico alto para testar todos os eixos.

O teste consiste em comparar a variação das leituras (com e sem *self-test*) com o valor de corte de fábrica (*Factory Trim* ou FT).

$$STR\,(Resposta\ do\ self-test) = Output\ self{-}test\ -\ Output\ sem\ self{-}test \tag{3.1}$$

$$Alterao\ do\ self{-}test\ em\ relao\ ao\ factory\ trim(\%) = \frac{STR - FT}{FT} \tag{3.2}$$

O valor do *factory trim* é obtido através de equações fornecidas em (MPU-6000..., b). Para conseguir os valores necessários para seu cálculo, devemos coletar os valores dos registrado-

Tabela 3.1. Self Test Registers (MPU-6000..., b).

Registrador (Hex)	Registrador (Decimal	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
9D	13	XA_TEST[4-2]			XG_TEST[4-0]				
9E	14	YA_TEST[4-2]			YG_TEST[4-0]				
0F	15	ZA_TEST[4-2]			ZG_TEST[4-0]				
10	16	Reservado		XA_TEST[1-0]		YA_TEST[1-0]		ZA_TEST[1-0]	

res de *self test* (*Self Test Regiters*) conforme mostrados na tabela. XA_TEST, YA_TEST, ZA_TEST, XG_TEST, YG_TEST e ZG_TEST devem ser interpretados como valores sem sinal (*unsigned*).

Com esses valores calculamos o *factory trim* a partir das equações abaixo.

$$\left\{ \begin{array}{ll} \text{FT[Xg]} = 25 \cdot 131 \cdot 1046^{(XG\_TEST-1)} & \text{if XG\_TEST} \neq 0 \\ \text{FT[Xg]} = 0 & \text{if XG\_TEST} = 0 \end{array} \right\} \tag{3.3}$$

$$\left\{ \begin{array}{ll} \text{FT[Yg]} = -25 \cdot 131 \cdot 1046^{(YG\_TEST-1)} & \text{if YG\_TEST} \neq 0 \\ \text{FT[Yg]} = 0 & \text{if YG\_TEST} = 0 \end{array} \right\} \tag{3.4}$$

$$\left\{ \begin{array}{ll} \text{FT[Zg]} = 25 \cdot 131 \cdot 1046^{(ZG\_TEST-1)} & \text{if ZG\_TEST} \neq 0 \\ \text{FT[Zg]} = 0 & \text{if ZG\_TEST} = 0 \end{array} \right\} \tag{3.5}$$

$$\left\{ \begin{array}{ll} \text{FT[Xa]} = 4096 \cdot 0.34 \cdot \frac{0.92}{0.34}^{\frac{(XA\_TEST-1)}{2^5-1}} & \text{if XA\_TEST} \neq 0 \\ \text{FT[Xa]} = 0 & \text{if XA\_TEST} = 0 \end{array} \right\} \tag{3.6}$$

$$\left\{ \begin{array}{ll} \text{FT[Ya]} = 4096 \cdot 0.34 \cdot \frac{0.92}{0.34}^{\frac{(YA\_TEST-1)}{2^5-1}} & \text{if YA\_TEST} \neq 0 \\ \text{FT[Ya]} = 0 & \text{if YA\_TEST} = 0 \end{array} \right\} \tag{3.7}$$

$$\left\{ \begin{array}{ll} \text{FT[Za]} = 4096 \cdot 0.34 \cdot \frac{0.92}{0.34}^{\frac{(ZA\_TEST-1)}{2^5-1}} & \text{if ZA\_TEST} \neq 0 \\ \text{FT[Za]} = 0 & \text{if ZA\_TEST} = 0 \end{array} \right\} \tag{3.8}$$

Em resumo, os passos do *self-test* devem ser:

1. Coletar os *outputs* dos giroscópios e acelerômetros com *self-test* desabilitado;
2. Coletar os *outputs* dos giroscópios e acelerômetros com *self-test* habilitado;
3. Coletar os dados dos registradores de *self-test*;
4. Calcular o *Factory Trim*;
5. Calcular as porcentagens de alteração;
6. Checar se as porcentagens de alteração estão dentro dos limites
7. estabelecidos, geralmente de 14%.

3.3.4 Calibração

A calibração consiste em realizar diversas medições com o MPU em repouso e calcular a média entre elas. Essa média deve ter seu valor armazenado e subtraído das medições do dispositivo já em operação. Para reduzir a complexidade das contas para o MSP, serão realizadas 256 medidas (2^8). Dessa forma pode-se acumular os valores de medidas em uma variável de 32 bits (4 bytes) e para tirar a média basta realizar um deslocamento (*shift*) para a direita de 8 bits.

Outra forma de otimização é minimizar o tempo entre leituras. Para isso será utilizada a técnica de *pooling*. Isso significa que toda vez que um novo dado é escrito nos registradores de *output* dos giroscópios e acelerômetros, será realizada uma leitura. Aplicando 1 ao bit0 do registrador INT_ENABLE, o MPU sofrerá interrupção quando um novo dado é escrito nos registradores de *output*. Para perceber que houve a interrupção deve-se ler o registrador INT_STATUS. Quando este registrador é lido, seu conteúdo muda para 0 e acaba a interrupção. Portanto, a calibragem se dará da seguinte forma:

1. Com a interrupção para *pooling* habilitada, aguardar que o bit0 de INT_STATUS se torne verdadeiro;
2. Realizar a leitura dos *outputs* dos acelerômetros e giroscópios;
3. Somar às leituras anteriores;
4. Repetir o procedimento acima 256 vezes;
5. Retirar a média de cada *output*;
6. Para os valores futuros coletados, subtrair a média encontrada.

Vale lembrar que a calibração depende da escala de operação.

3.3.5 Configuração para operação

Com os procedimentos de testes e calibragem concluídos, deve-se configurar o MPU para a operação. Para isso é interessante realizar um *reset* alterando o bit7 do registrador PWR_MGMT_1

para nível lógico alto. Após o *reset*, deve-se aguardar um pouco para o sistema se estabilizar. Em seguida, deve-se retirar o MPU do modo *sleep* mais uma vez.

O próximo passo é alterar as taxas dos acelerômetros e giroscópios pelo registrador CONFIG. Em seguida, ajustar a taxa de amostragem no registrador SMPLRT_DIV. Por fim, realizar os ajustes de escala nos registradores GYRO_CONFIG e ACCEL_CONFIG.

3.4 MÁQUINA DE ESTADOS

Após o fim das etapas de configuração tem-se início a máquina de estados. A máquina possui duas configurações. Uma para detecção de frenagem e outra para adicional de estado de alerta.

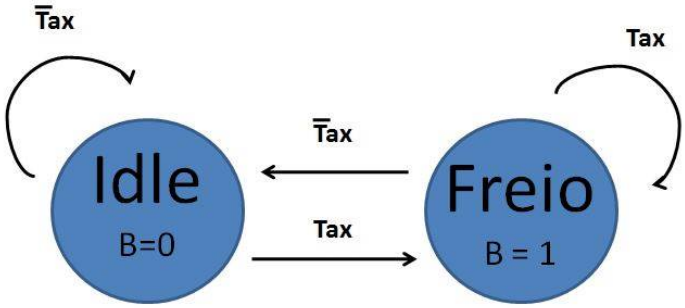


Figura 3.2. Configuração 1: detecção de frenagem

Nesta configuração existem apenas dois estados. No estado Idle, ocorrem leituras do acelerômetro do eixo x a intervalos de 33 ms. Nesse estado, a saída B que indica frenagem é 0. Deve-se permanecer neste estado até que o valor lido para aceleração no eixo x seja maior que um determinado limiar. Quando isso ocorre, muda-se para o segundo estado.

O estado Freio é o de frenagem identificada, indicado pela saída B ser 1. Quando o valor lido for menor que o limiar, retorna-se para o primeiro estado.

A variável que rege a mudança de estados é Tax, que será 0 quando a leitura de aceleração no eixo x for menor que o limiar estipulado. Caso a leitura for maior ou igual ao limiar, Tax será 1.

Para a segunda configuração, os estados Idle e Freio são basicamente idênticos e Tax se comporta da mesma forma. A diferença ocorre na adição de um terceiro estado, o Alerta. Mudanças para esse estado independem de Tax, sendo apenas necessário que Tg seja verdadeiro. Este estado representa alerta, quando ocorre mudança brusca de posição, indicando um acidente como uma queda. O objetivo deste estado é, portanto, gerar uma saída que indique um possível acidente. A saída deste estado ocorre quando a variável U é falsa.

A entrada U depende de ações externas, como por exemplo o aperto de um botão pelo usuário. A entrada Tg requer alguns testes para estipular como será implementada. Tg pode

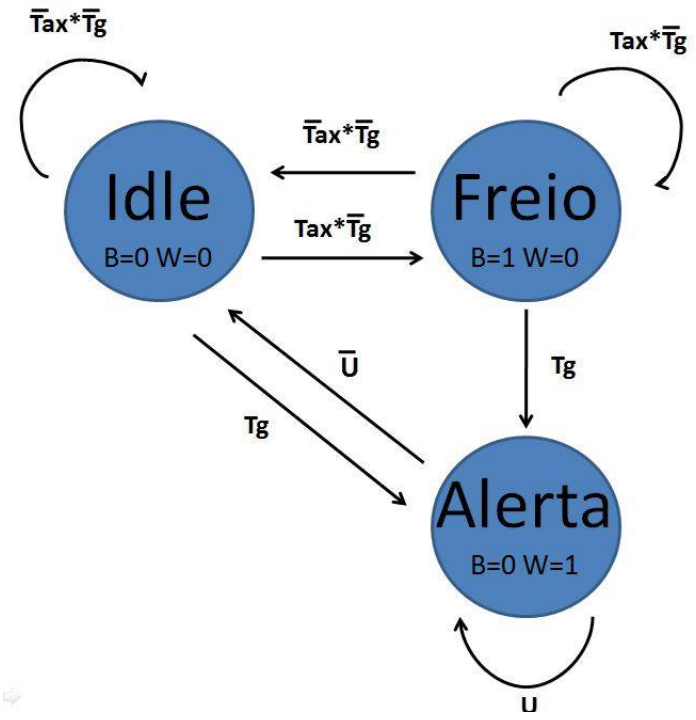


Figura 3.3. Configuração 2: detecção de frenagem com estado de alerta

requerer análise dos dados coletados por todos os acelerômetros e giroscópios em vários cenários de simulações de acidentes.

3.5 LIMIARES

O sucesso da máquina de estados depende da correta implementação dos limiares. Aqui, deve-se estabelecer uma boa configuração para Tax. Para isso é necessário realizar alguns testes em uma bicicleta, o veículo alvo.

Para os testes, a máquina de estados é substituída por um acumulador de medições em um vetor para realizar amostras de campo. A partir dos resultados encontrados estratégias podem ser estipuladas para o tratamento da entrada Tax.

Uma forma de reduzir mudança de estados é a utilização de dois limiares para o tratamento de Tax. Um primeiro para a transição de estado inicial para de frenagem, e um segundo para a transição de retorno para o inicial. O primeiro teria valor de corte maior, impedindo que a interferência de baixa amplitude cause constantes mudanças de estado. A diferença entre os limiares teria de ser definida por coleta de dados.

CAPÍTULO 4

MEDIDAS E ANÁLISE

4.1 PROCEDIMENTOS DE INICIALIZAÇÃO

Os procedimentos de inicialização, descritos na seção 3.3, são necessários após toda inicialização para garantir o bom funcionamento do dispositivo. É, portanto, imprescindível realizar os ajustes para que os testes sejam bem sucedidos.

A primeira dificuldade encontrada foi garantir que a comunicação serial funcionava corretamente. Inicialmente, o MPU não parecia comportar-se de forma apropriada. Após melhoria na rotina de comunicação e na retirada do modo *sleep* foi possível prosseguir.

Em seguida foi realizada a rotina de *self-test*, que após ser aplicada, verificava-se que alguns sensores falhavam no teste consistentemente. O MPU foi trocado, mas as falhas continuaram. Após alguns testes notou-se que os dados coletados dos registradores internos apresentavam leituras adversas ao esperado. Até este ponto, o teste de comunicação serial utilizando o registrador WHO_AM_I não foi aplicado.

No teste da comunicação serial, notou-se que com a função para leitura em sequência (*burst*), os dados lidos não se mostraram coerentes. Esse empecilho fez com que fosse possível ler apenas um registrador por vez, sendo necessário realizar diversas leituras unitárias ao invés de uma única em *burst*.

Após esse ajuste foi realizado novamente o *self-test* resultando na aprovação dos sensores.

Para a calibração, inicialmente foram utilizados valores *float* para que o resultado não dependesse de escala. No entanto, o CCS acusou que a utilização de operações com este tipo de variável deve ser evitada. O motivo seria o custo em termos de memória e tempo. Como consequência foram utilizados valores inteiros apenas.

4.2 MEDIDAS PARA OS LIMIARES

Para ajustar a máquina de estados, deve-se realizar testes com o objetivo de definir o limiar da entrada Tax. Os testes ocorrem supondo que o veículo transita em plano reto, sem inclinação. Este requisito visa eliminar a influência da gravidade sobre o acelerômetro do eixo de movimento da bicicleta. Para este estudo, foi escolhido o eixo x por facilitar a instalação na bicicleta.

A coleta de dados foi realizada com leituras do valor em ax a uma taxa de 30 Hz, organizados em um vetor. Este vetor é escrito na memória RAM, sendo necessário manter o suprimento de energia. Para isso, o computador deve estar conectado ao MSP durante os testes.

Para observar os dados, o próprio CCS apresenta uma ferramenta gráfica. Porém, para poder ser realizada a manipulação, os valores foram exportados para o programa Matlab.

A Figura 4.1 apresenta o gráfico com o resultado da coleta.

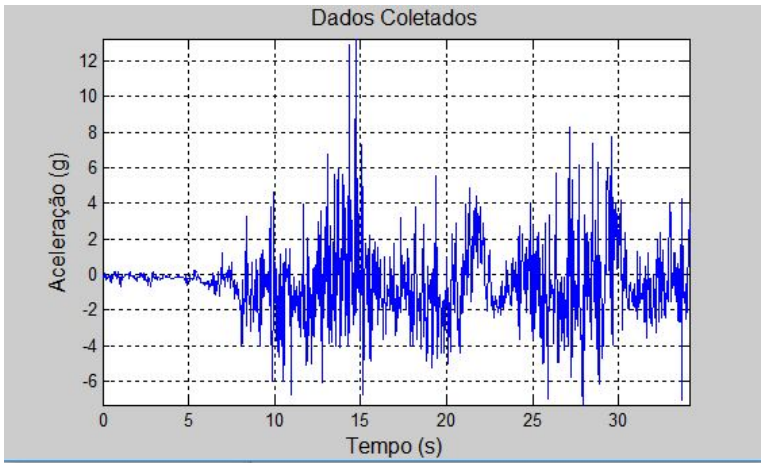


Figura 4.1. Dados coletados em teste de campo

Ao analisar o gráfico, nota-se que o sinal gerado é bastante inconstante. A partir disso, observa-se que um filtro seria útil para o tratamento do sinal. As alterações de aceleração buscadas são de baixa frequência devido a sua origem. Sendo assim, não é interessante mostrar as rápidas trepidações. Um filtro passa-baixa apresenta-se como forte candidato para a melhoria do sinal.

Para testar essa hipótese, foi utilizado o programa Matlab para aplicar filtros. Foram utilizados filtros digitais passa-baixa IIR (resposta a impulso infinito) de segunda ordem e taxa de amostragem de 30 Hz. Os filtros apresentados aqui variam apenas em sua frequência de

corte, sendo elas 1 Hz, 2 Hz e 5 Hz.

Um importante fator que contribui à utilização do filtro passa-baixa com frequência de corte de 5 Hz, é o fato de este ser a menor frequência de corte do filtro interno programável do MPU, conforme mostrado na tabela 4.1. Utilizar o filtro interno tem como vantagens economia de necessidade de processamento pelo MSP. O *delay* gerado por este filtro para esta configuração é de no máximo 19 ms. Como é coletada uma amostra a cada 33 ms, o atraso é negligenciável.

DLPF_CFG	Accelerometer (F_s = 1kHz)		Gyroscope		
	Bandwidth (Hz)	Delay (ms)	Bandwidth (Hz)	Delay (ms)	Fs (kHz)
0	260	0	256	0.98	8
1	184	2.0	188	1.9	1
2	94	3.0	98	2.8	1
3	44	4.9	42	4.8	1
4	21	8.5	20	8.3	1
5	10	13.8	10	13.4	1
6	5	19.0	5	18.6	1
7	RESERVED		RESERVED		8

Tabela 4.1. Filtro passa-baixa do MPU (MPU-6000..., b).

O filtro de 5 Hz com resposta apresentada na Figura 4.2 mostra redução da interferência não desejável. Para visualizar a melhoria do sinal, as figuras 4.3 e 4.4 apresentam em quais momentos ocorreram leituras acima de um limiar de 2g, com e sem o filtro.

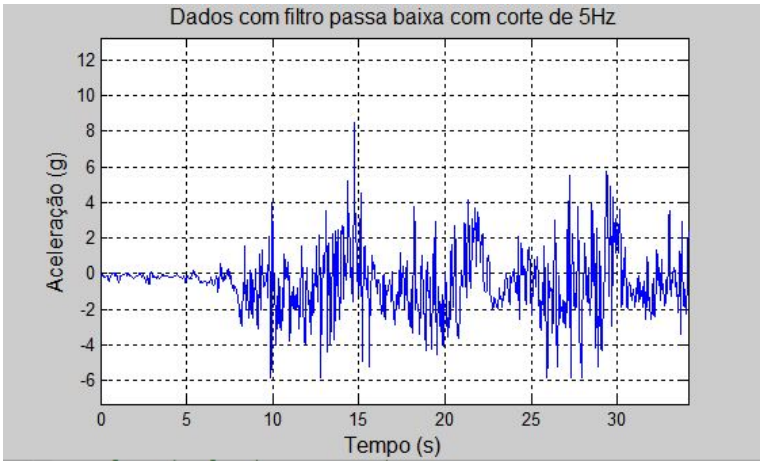


Figura 4.2. Dados tratados com filtro passa-baixa de 5 Hz

Supondo que o filtro simulado e o do MPU se comportam de forma similar, podemos analisar a Figura 4.2 e estipular uma solução para Tax. Aplicando um limiar de 2g, identificam-se os pontos em que houve reduções significantes de velocidade. Como desconhece-se seu comportamento em termos de atenuação, esta solução requer que sejam realizados testes com o filtro interno do MPU. Para visualizar a melhoria do sinal, as figuras 4.3 e 4.4 apresentam em quais

momentos ocorreram leituras acima do limiar de 2g, sem e com o filtro respectivamente.

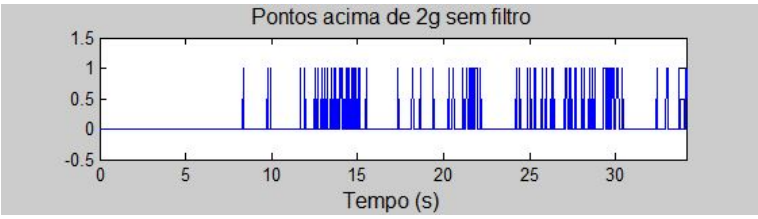


Figura 4.3. Respostas acima de 2g para os dados coletados

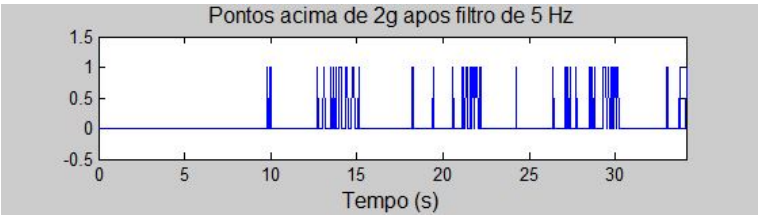


Figura 4.4. Respostas acima de 2g para os dados coletados com filtro de 5Hz

A comparação das duas imagens acima mostra que houve redução na quantidade de pontos isolados e insignificantes para a sinalização, sendo mais notável os pontos de freio entorno de 14, 22 e 30 segundos. O filtro de 5Hz com limiar de 2 g apresenta-se como uma solução para testes futuros, sendo ainda necessários testes com o filtro do MPU.

Pode-se melhorar o tratamento do sinal aplicando filtros com frequência de corte menores. Para isso, será necessária a utilização de filtragem por código no MSP.

Observando a Figura 4.5, filtro com corte de 2 Hz, nota-se grande redução do ruído. É notável os pontos de freio, porém permanecem curtos picos informando reduções de velocidade pontuais. A Figura 4.6 apresenta os momentos acima de 2g e nota-se melhoria em relação à Figura 4.4.

O filtro de corte em 1 Hz, com resposta apresentada na Figura 4.7, apresenta os mesmos intervalos apresentados pelo filtro de 2 Hz. No entanto, possui muito menos picos pontuais de desaceleração. Ele parece apresentar o melhor resultado para a estipulação de um limiar de freio. A Figura 4.8 apresenta os momentos acima de 1g e, novamente, houve melhoria em relação aos resultados com os filtros anteriores.

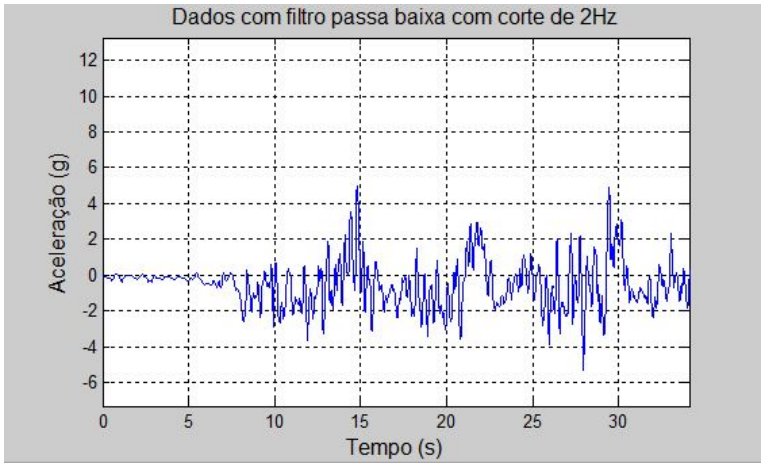


Figura 4.5. Dados tratados com filtro passa-baixa de 2 Hz

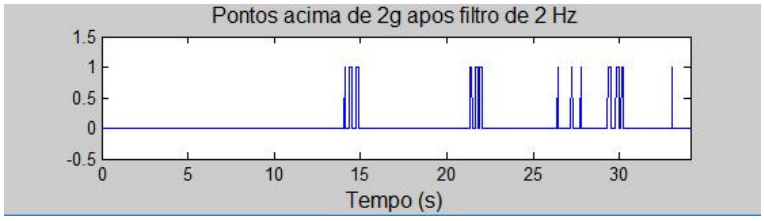


Figura 4.6. Respostas acima de 2g para os dados coletados com filtro de 2Hz

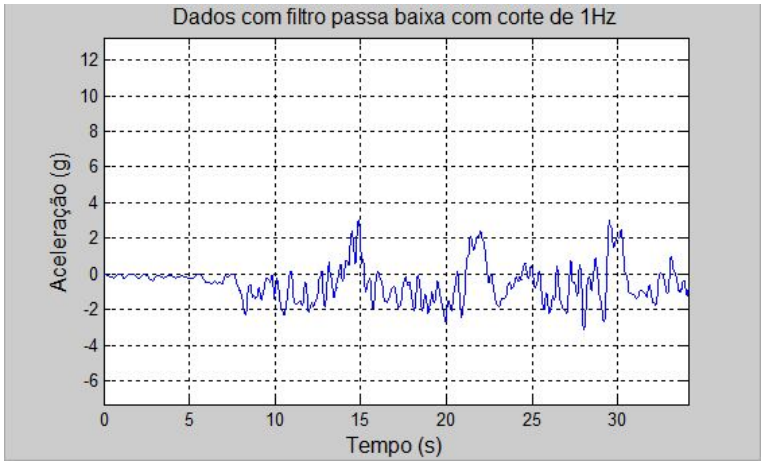


Figura 4.7. Dados tratados com filtro passa-baixa de 1 Hz

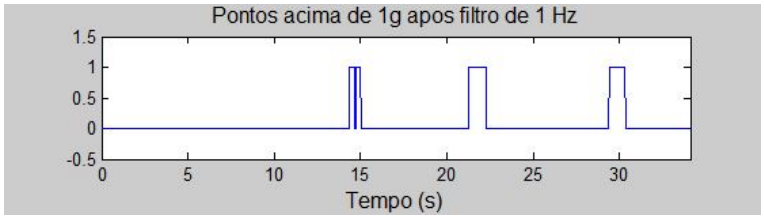


Figura 4.8. Respostas acima de 1g para os dados coletados com filtro de 1Hz

CAPÍTULO 5

CONCLUSÃO E PROPOSTAS

Neste trabalho, se buscou implementar uma solução para a aplicação de uma luz de sinalização de frenagem para bicicleta contido em apenas uma caixa. Para isso, foi feito o projeto de um sistema embarcado utilizando o microcontrolador para tratar os dados coletados do sensor.

Primeiro, no Capítulo 2, foi feita a apresentação das ferramentas utilizadas no projeto. Nele, foi detalhada a tecnologia de hardware, do software, dos sensores e da estrutura de código.

Em seguida, foi abordado, no Capítulo 3, a implementação dos recursos apresentados. Para isso, foram expostos o esquemático das ligações, uma breve apresentação sobre as livrarias utilizadas, os procedimentos para configuração do MPU-6050 e a máquina de estados. Neste capítulo, foi apresentada a demanda de dados para a calibração do sistema embarcado.

No Capítulo 4, foi apresentada a coleta de resultados do teste de campo realizado. Com os dados, foram realizadas análises para tratamento do sinal, incluindo a aplicação de filtros e a simulação de respostas. Com os resultados encontrados, foi apresentada uma solução para uma entrada da máquina de estados.

Como resultado do projeto, foi realizada a interface entre o microcontrolador e o sensor externo MPU-6050, bem como a apresentação de soluções ao problema e análise de coleta de dados. Ao longo do processo foram encontradas limitações inesperadas que requerem melhor análise para otimização. O modo de leitura em sequência não está se comportando da maneira esperada, apresentando leituras adversas à realidade. A influência das componentes de alta frequência do sinal gerado pela saída dos sensores necessitam ser filtradas, sendo ainda necessário testar o filtro da plataforma MPU.

O projeto ainda não tem suas funções plenamente aplicadas , abrindo espaço para a continuação do projeto. Como trabalhos futuros propõe-se:

- Aplicar o filtro passa-baixa digital da plataforma MPU-6050;

- Eliminar a componente da gravidade para permitir a utilização em plano inclinado. Para isso, recomenda-se utilizar o processador digital de movimento do próprio MPU-6050;
- Explorar a possibilidade de utilizar o limiar de detecção de movimento do MPU-6050;
- Explorar a utilização de decaimento temporizado. Forçar que a saída para o LED seja verdadeira por um tempo mínimo;
- Disparador de Schmitt como alternativa para limpar o sinal tornado a entrada Tax mais estável;
- Medir consumo de energia. Sugere-se utilizar a ferramenta EnergyTrace^{TM} do Code Composer Studio;
- Estudar a influência da temperatura sobre os sensores;
- Configurar a detecção de acidentes;
- Aplicar comunicação sem fio com aparelho celular para informar que houve acidente.

Apesar das limitações citadas, neste projeto foi possível realizar a interface entre o microcontrolador MSP430 e a plataforma MPU-6050. Além disso, foram realizados os procedimentos de inicialização e a coleta de dados dos sensores internos do MPU. Com os dados foi possível apresentar algumas soluções, que precisam passar por testes e implementações.

REFERÊNCIAS BIBLIOGRÁFICAS

CODE Composer Studio (CCS) Integrated Development Environment (IDE). Disponível em: <https://http://www.ti.com/tool/ccstudio>. 8

DAVIES, J. H. *MSP430 microcontroller basics*. [S.l.]: Elsevier, 2008. 5, 6

EZ-FET lite. Disponível em: <http://processors.wiki.ti.com/index.php/EZ-FET_lite>. 11

INSTRUMENTS, T. Msp430 family user guide. *Texas Instruments, 2015a. Disponivel em:< http://www. ti. com/lit/ug/slau144j/slau144j. pdf>. Acesso em*, v. 21, 2015. 6

INTRODUCTION to MEMS Accelerometers. Disponível em: <https://www.pcb.com/Resources/Technical-information/mems-accelerometers>. 12

MPU-6000 and MPU-6050 Product Specification. 14

MPU-6000 and MPU-6050 Register Map and Descriptions. v, 15, 16, 20, 21, 22, 29

ZELENOVSKY, R.; MENDONCA, A. Apendice h mpu 6050. iii, 12, 13

.1 APÊNDICE A - CÓDIGO PARA LEITURA DOS ACELERÔMETROS E GIROSCÓPIOS DO MPU-6050 UTILIZANDO MSP430

O código abaixo utiliza o MSP430F5529 para coletar os resultados do acelerômetro, giroscópio e sensor de temperatura. Os dados coletados são armazenados nas variáveis ACCEL em g, GYRO em graus/segundo e TEMP em graus Celsius.

O código pode ser obtido em <https://github.com/GuilhermeGilCampbell/Mpu-6050_with_Msp-430>.

```
1
2 #include <msp430.h>
3
4 ////////////////////////////////////////////////////////////////////////////////////////
5
6 #define MPU6050_DEFAULT_ADDRESS         0x68
7
8 // Mpu Registers
9 #define MPU6050_RA_PWR_MGMT_1           0x6B
10 #define MPU6050_RA_SMPLRT_DIV          0x19
11 #define MPU6050_RA_CONFIG              0x1A
12 #define MPU6050_RA_GYRO_CONFIG         0x1B
13 #define MPU6050_RA_ACCEL_CONFIG        0x1C
14 #define MPU6050_RA_ACCEL_XOUT_H        0x3B
15 #define MPU6050_RA_WHO_AM_I            0x75
16
17 // Scales
18 // Output(degrees/second) = Output(signed integer) * max_range/32767
19 #define MPU6050_GYRO_FS_250            0x00
20 #define MPU6050_GYRO_FS_500            0x01
21 #define MPU6050_GYRO_FS_1000           0x02
22 #define MPU6050_GYRO_FS_2000           0x03
23
24 // Output(g) = Output(signed integer) * max_range/32767
25 #define MPU6050_ACCEL_FS_2             0x00
26 #define MPU6050_ACCEL_FS_4             0x01
27 #define MPU6050_ACCEL_FS_8             0x02
28 #define MPU6050_ACCEL_FS_16            0x03
29
30
31 ////////////////////////////////////////////////////////////////////////////////////////
32
33 #define SLAVE           00
34 #define MASTER          01
35
36 #define TRANSMITTER     00
37 #define RECEIVER        01
38
39 struct {
40     unsigned char * data;
41     unsigned char count;
42     unsigned char index;
43     enum {sendBurst,sendAndRestart} mode;
44 } TX;
45
46 struct {
47     unsigned char * data;
48     unsigned char count;
49     unsigned char index;
50 } RX;
51
```

```
52 //Functions ////////////////////////////////////////////////////////////////////////////////////////////////////
53 void     setupPorts          (void);
54 void     setupClock          (void);
55 void     setupTimerA0        (void);
56 void     waitFor             (unsigned int time_ms);
57 void     setupSerial         (void);
58 void     serialSetMode       (unsigned char mode, unsigned char direction);
59 inline void serialStart      (unsigned char addr);
60 void     serialSendData      (unsigned char deviceAddr, unsigned char * dataPtr, unsigned char count);
61 void     serialGetData       (unsigned char deviceAddr, unsigned char from,unsigned char * to, unsigned char count);
62 unsigned char serialGetByte (unsigned char deviceAddr, unsigned char from);
63 void     mpuSetByte          (unsigned char addr, unsigned char data);
64 unsigned char mpuGetByte     (unsigned char addr);
65 void     mpuRead_nb          (unsigned char addr, unsigned char * data, unsigned char length);
66
67 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
68
69 float           ACCEL[3],GYRO[3],TEMP;      //converted value
70
71 int main(void) {
72
73     unsigned char   reply[14];                  //Vetor com as últimas medidas da função mpuRead
74     int             ax,ay,az,gx,gy,gz,temp;     //signed int value
75     unsigned char a_scale, g_scale;
76     __enable_interrupt();
77         WDTCTL = WDTPW | WDTHOLD;       // Stop watchdog timer
78         setupPorts();                   // Setup ports
79         setupClock();                   // Setup clock
80         setupTimerA0();                 // Setup timer
81         setupSerial();                  // Reset Serial Interface
82
83         //Desired Scale
84         a_scale = MPU6050_ACCEL_FS_16;      //2, 4, 8 or 16 g
85         g_scale = MPU6050_GYRO_FS_250;      //250, 500, 1000 or 2000 degrees/second
86
87         //Wake up MPU
88         mpuSetByte(MPU6050_RA_PWR_MGMT_1, 0x01);
89         waitFor(250);
90
91         //Test Comunication
92         mpuRead_nb(MPU6050_RA_WHO_AM_I,reply,1);
93         if(reply[0]==MPU6050_DEFAULT_ADDRESS){}
94         else while(1){P4OUT |= BIT7;};          //If com test fails, green led on
95
96         //Set scales
97         mpuSetByte(MPU6050_RA_GYRO_CONFIG , g_scale<<3);
98         mpuSetByte(MPU6050_RA_ACCEL_CONFIG , a_scale<<3);
99
100        //Get one full reading from accelerometers, giroscope and temperature
101        mpuRead_nb(MPU6050_RA_ACCEL_XOUT_H, reply, 14);
102        ax = (int) ((reply[0] << 8) | reply[1]) ;
103        ay = (int) ((reply[2] << 8) | reply[3]) ;
104        az = (int) ((reply[4] << 8) | reply[5]) ;
105        temp = (int) ((reply[6] << 8) | reply[7]) ;
106        gx = (int) ((reply[8] << 8) | reply[9]) ;
107        gy = (int) ((reply[10] << 8) | reply[11]) ;
108        gz = (int) ((reply[12] << 8) | reply[13]) ;
109
110        //Data conversion to it's apropriate unit
111        ACCEL[0] = (float) ax*(2<<a_scale)/32767;
112        ACCEL[1] = (float) ay*(2<<a_scale)/32767;
113        ACCEL[2] = (float) az*(2<<a_scale)/32767;
114        TEMP = (float) temp/340 + 36.53;
115        GYRO[0] = (float) gx*(250<<g_scale)/32767;
116        GYRO[1] = (float) gy*(250<<g_scale)/32767;
117        GYRO[2] = (float) gz*(250<<g_scale)/32767;
118
119
120        P1OUT = BIT0;   //Red led on after readings
121    return 0;
122 }
123 ;
124 // Ports Setup ////////////////////////////////////////////////////////////////////////////////////////////////
125 void setupPorts (void){
126
127     // Remove warning of uninitialized ports
128     // This will set all ports to input with a pull-down resistor
129         PADIR = 0x0000; PAREN = 0xFFFF; PAOUT = 0x0000; PASEL = 0x0000;
130         PBDIR = 0x0000; PBREN = 0xFFFF; PBOUT = 0x0000; PBSEL = 0x0000;
131         PCDIR = 0x0000; PCREN = 0xFFFF; PCOUT = 0x0000; PCSEL = 0x0000;
132         PDDIR = 0x0000; PDREN = 0xFFFF; PDOUT = 0x0000; PDSEL = 0x0000;
```

```
133
134     // Serial communication
135         P3SEL |=  BIT0 | BIT1;                    // Use dedicated module
136         P3REN &= ~BIT0;                     // Resistor enable
137         P3REN &= ~BIT1;
138         P3OUT |=  BIT0 | BIT1;                    // Pull-up
139
140     // LEDs
141         P1DIR   |=  BIT0;                 // Red   LED : output
142         P1OUT   &= ~BIT0;                 // Red   LED : off
143
144         P4DIR |=  BIT7;               // Green LED : output
145         P4OUT &= ~BIT7;               // Green LED : off
146 }
147
148
149 // Clock Setup ////////////////////////////////////////////////////////////////////////////////////////////////////
150 void setupClock (void) {
151     // Unified clock system
152     UCSCTL0 = 0x00;                           // Let FLL manage DCO and MOD
153     UCSCTL1 = DCORSEL_5;                      // Select DCO range to 16MHz
154     UCSCTL2 = FLLD__16        |               // Set FLL dividers
155               31;
156     UCSCTL3 = SELREF__XT1CLK  |               // Use Crystal 1 Oscillator for better precision
157               FLLREFDIV__1;                   // FLL input divider to 1
158     UCSCTL4 = SELA__XT1CLK    |               // ACLK  = Crystal 1, =>     32.768 Hz
159               SELS__DCOCLKDIV |               // SMCLK = DCO/FLLD   =>  1.048.576 Hz
160               SELM__DCOCLK;                   // MCLK  = DCO        => 16.777.216 Hz
161     UCSCTL5 = DIVPA__1        |               // Output dividers to 1
162               DIVA__1         |               // ACLK  divider 1
163               DIVS__1         |               // SMCLK divider 1
164               DIVM__1;                        // MCLK  divider 1
165
166     UCSCTL7 = 0;                              // Clear XT2,XT1,DCO fault flags
167
168 }
169
170 //Timer Setup ////////////////////////////////////////////////////////////////////////////////////////////////////
171 void setupTimerA0(void) {
172   // Timer configuration
173   TA0CTL = TASSEL__ACLK       |               // Select ACLK as clock source
174            ID__1              |               // Set clock divider to 1.
175            MC__STOP           |               // Setup but do not count
176            TACLR              |               // Clear timer
177            TAIE               |               // Enable Interrupts from timer
178            0;
179
180   // Timer A0 - CCR0 configuration
181   TA0CCR0  = 32767;                            // Count to 1s @32kHz
182
183 }
184
185 void waitFor(unsigned int time_ms)
186 {
187   // Configure timer A0 and start it.
188   TA0CCR0 = time_ms << 5;                     // 1 sec = 1024 ms (rounded up)
189   TA0CTL  |= MC__UP | TACLR;                  // Count up and clear timer
190
191   // Locks, waiting for the timer.
192   __low_power_mode_0();
193 }
194
195 #pragma vector=TIMER0_A1_VECTOR
196 __interrupt void ISR_TIMER0 (void) {
197   switch (__even_in_range(TA0IV,14)) {
198     case 0x0 : break;   // No interruption
199     case 0x2 : break;   // CCR1
200     case 0x4 : break;   // CCR2
200     case 0x4 : break;   // CCR2
201     case 0x6 : break;   // CCR3
202     case 0x8 : break;   // CCR4
203     case 0xA : break;   // CCR5
204     case 0xC : break;   // CCR6
205     case 0xE :          // Overflow
206       // Stop the timer
207       TA0CTL &= ~0x30;
208       // And exit low power mode
209       __low_power_mode_off_on_exit();
210       break;
211     default  : break;
212   }
213 }
```

```
214
215 //Serial Setup ////////////////////////////////////////////////////////////////////////////////////////////////////
216 void setupSerial(void)
217 {
218     // UCS0 Module configuration
219     // First Reset module
220     UCB0CTL1 |= UCSWRST;
221
222     // Protocol Synchronous I2C (only one supported at this time)
223     UCB0CTL0 = UCSYNC       |       // Syncronous mode enabled
224                UCMODE_3     |       // I2C mode
225                UCMST; // Master is the default (should remove this)
226
227     // Set Own Address (and reset UCGCEN, i.e., don't respond to a general call)
228     UCB0I2COA = 0x42;
229
230     // Set prescaler SMCLK @1M/10 = 100k
231     UCB0BRW = 10;
232
233     // Select SMCLK as clock source and clear reset
234     UCB0CTL1 = UCSSEL__SMCLK    |   // CLK source SMCLK
235                UCTR ;               // Transmiter mode
236                                     // UCSWRST cleared
237     // Transmitter is the default (should remove this)
238
239     // Enable module interrupts
240     UCB0IE =  UCNACKIE          | // Not-acknowledge
241               UCALIE            | // Arbitration lost
242               UCSTPIE           | // STOP condition
243               UCSTTIE           | // START condition
244               UCTXIE            | // TX buffer is ready
245               UCRXIE            | // Received something (RX buffer is ready)
246               0;                  // TX and RX flags are automatically disabled
247
248 }
249
250 //Functions for serial comunication ////////////////////////////////////////////////////////////////////////////////
251
252 // Transmiter or receiver
253 void serialSetMode( unsigned char mode, unsigned char direction)
254 {
255     UCB0CTL0 |= (mode      == MASTER     )? UCMST : 0;
256     UCB0CTL1 |= (direction == TRANSMITTER)? UCTR  : 0;
257 }
258
259 // Start
260 inline void serialStart(unsigned char addr)
261 {
262     // Set slave address
263     UCB0I2CSA = addr;
264
265     // Start Transmission
266     UCB0CTL1 |= UCTXSTT;
267 }
268
269 // Send "count" bytes of "dataPtr" to device with adress "device addr"
270 void serialSendData(unsigned char deviceAddr, unsigned char * dataPtr, unsigned char count)
271 {
272     // Setup serial interface (without reseting)
273     serialSetMode(MASTER, TRANSMITTER);
274     UCB0IE = 0x3F; // Enable interrupts
275
276     UCB0IE = 0x3F;
277     TX.data  = dataPtr;
278     TX.index = 0;
279     TX.count = count;
280     TX.mode  = sendBurst;
281
282     serialStart(deviceAddr);
283     __low_power_mode_0();
284 }
285
286 // Get "count" bytes from device ("deviceAddr") starting at "from" and write it on "to"
287 void serialGetData(unsigned char deviceAddr, unsigned char from,
288                    unsigned char * to, unsigned char count)
289 {
290     // Setup serial interface (without reseting)
291     serialSetMode(MASTER, TRANSMITTER);
292     UCB0IE = 0x3F; // Enable interrupts
293
```

```
294     // Assemble TX object
295     TX.data  = &from;
296     TX.count = 1;
297     TX.index = 0;
298     TX.mode  = sendAndRestart;
299
300     // Assemble RX object
301     RX.data  = to;
302     RX.count = count;
303     RX.index = 0;
304
305     // Start communication
306     serialStart(deviceAddr);
307     __low_power_mode_0();
308 }
309
310 //Return data  from "from" of the devise "deviceAddr"
311 unsigned char serialGetByte(unsigned char deviceAddr, unsigned char from)
312 {
313     unsigned char reply;
314     serialGetData(deviceAddr, from, &reply, 1);
315     return reply;
316 }
317
318 //Write "data" at the internal register "addr" of mpu
319 void mpuSetByte(unsigned char addr, unsigned char data)
320 {
321     unsigned char dataPtr[2];
322     dataPtr[0] = addr;
323     dataPtr[1] = data;
324
325     serialSendData(MPU6050_DEFAULT_ADDRESS, dataPtr , 2);
326 }
327
328 //Return data from internal register "addr" of mpu
329 unsigned char mpuGetByte(unsigned char addr)
330 {
331     return serialGetByte(MPU6050_DEFAULT_ADDRESS, addr);
332 }
333
334 //Read "length" times staring at "addr" from mpu and write it on "data"
335 void mpuRead_nb(unsigned char addr, unsigned char * data, unsigned char length)
336 {
337     unsigned char i;
338     for (i=length;i>0;i--){
339         data[length-i]=mpuGetByte(addr+(length-i));
340     }
341 }
342
343 #pragma vector=USCI_B0_VECTOR
344 __interrupt void ISR_USCI_B0 (void) {
345     switch (__even_in_range(UCB0IV,12)) {
346         case 0x0 : break;                   // -> No interrupt pending
347         case 0x2 : break;                   // -> Arbitration lost
348         case 0x4 :                          // -> Not acknowledgment
349             UCB0CTL1 |= UCTXSTP;            // Stop transmission because
350             __low_power_mode_off_on_exit(); // there is no one listening
351             break;
352         case 0x6 : break;                   // -> Start condition received
353         case 0x8 : break;                   // -> Stop condition received
354         case 0xA :                          // -> Data received (RXBUF full)
355             RX.data[RX.index++] = UCB0RXBUF;
356             if (RX.index == (RX.count-1)) {
357                 UCB0CTL1 = UCTXSTP;         // Request end of transmission while
358             }                               // receiving the last byte
359             if (RX.index == RX.count) {
360                 __low_power_mode_off_on_exit();
361             }
362             break;
363
364         case 0xC :                          // -> Ready to send (TXBUF empty)
365             if (TX.index != TX.count) {
366                 UCB0TXBUF =                 // Load TX buffer while
367                       TX.data[TX.index++];  // there is data to be transfered
368             } else {
369                 if (TX.mode == sendBurst) { // If previous transmission was the last
370                     UCB0CTL1 |= UCTXSTP;    // Simply stop transmission and exit
371                     __low_power_mode_off_on_exit();
372                 }
373                 if (TX.mode == sendAndRestart) {
374                     UCB0CTL1 &= ~UCTR;      // Set master receiver
```

```
375                     UCB0CTL1 |= UCTXSTT;     // Restart communication
376                     if (RX.count == 1) {     // If we expect to receive a single
377                         while (UCB0CTL1 & UCTXSTT);   // byte, then wait for acknowledge
378                         UCB0CTL1 |=UCTXSTP; // and send a stop request while
379                     }                       // receiving first byte
380                 }
381             }
382             break;
383         default  : break;
384     }
385 }
386 //
```