

G MPU 6050: Acelerômetro e Giroscópio

O MPU 6050 é um dispositivo integrado que combina um acelerômetro de 3 eixos e um giroscópio também de 3 eixos. Sua principal finalidade é a detecção e acompanhamento de movimentos. Citam-se o emprego na detecção da posição de celulares e *tablets*, nos *videogames*, na detecção de movimentos humanos e até na eliminação da trepidação em câmeras fotográficas e filmadoras.

O estudo disponível neste apêndice é limitado. Não se pretende exaurir todas as possibilidades deste componente, mas sim, apenas oferecer os primeiros passos para seu emprego. Ao final são apresentados programas ilustrativos que podem ser usados como partida para outros mais complexos. Ao longo deste capítulo, será usado o termo MPU para designar este componente.

A fabricante InvenSense (www.invensense.com) oferece dois manuais sobre seu *chip*:

- MPU-6000/MPU-6050 Product Specification e
- MPU-6000/MPU-6050 Register Map and Descriptions.

Entretanto, nestes manuais não se encontra uma descrição completa dos recursos do MPU e o fabricante parece omitir alguns detalhes. Por exemplo, as informações para o acesso direto ao Processador de Movimentos DMP (*Digital Motion Processor*) não estão disponíveis. Isso acontece porque a InvenSense espera que se faça uso de um programa proprietário. Entretanto, muitos usuários, empregando engenharia reversa, desvendaram diversos recursos deste *chip*.

Por isso, o que é apresentado neste apêndice reúne as experiências pessoais dos autores e também o resultado de pesquisas na Internet. Dentre os muitos programas disponíveis, os recomenda-se a consulta aos trabalhos elaborados por:

- Kris Winer (<https://github.com/kriswiner/MPU-6050>) e
- Jeff Rowberg que disponibiliza uma biblioteca em (<https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050>).

Apesar dessas dificuldades, o MPU-6050, ou simplesmente MPU, é um dispositivo muito confiável e barato. Seu uso é disseminado entre os hobistas e existe uma grande quantidade de informação e bibliotecas para o Arduino.

G.0. Quero Usar o MPU-6050 e Não Pretendo Ler Todo Este Apêndice

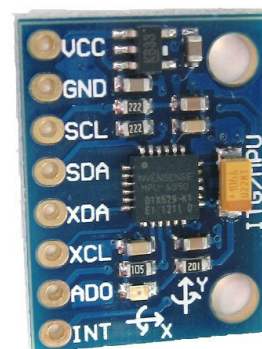
O MPU-6050 é um *chip* fabricado pela InvenSense e disponibilizado num encapsulamento para montagem em superfície. Devido às dificuldades para se trabalhar com tal encapsulamento, é comum encontrarmos pequenas placas que já o trazem soldado, o que facilita muito seu emprego. Este *chip* (MPU-6050) disponibiliza ao usuário um acelerômetro de 3 eixos (X, Y e Z) e um giroscópio de 3 eixos (X, Y e Z). Ele pertence à classe Unidade de Medida Inercial (IMU – *Inertial Motion Unit*) de 6 eixos.

O acelerômetro trabalha nas escalas ± 2 g, ± 4 g, ± 8 g e ± 16 g. O giroscópio trabalha nas escalas de ± 250 gr/s, ± 500 gr/s, ± 1000 gr/s e ± 2000 gr/s (gr/s = graus/segundo). Os valores entregues são inteiros de 16 *bits* com sinal. Isto significa que varrem a faixa de -32.768 até 32.767 (65.536 passos). Por exemplo, supondo a escala de ± 2 g, a resolução é dada por $4 \text{ g} / 65.536$ (ou então, $2 \text{ g} / 32.767$).

A placa mais comum no Brasil é a GY-521, apresentada na figura ao lado e que será objeto deste estudo. A Figura G.6 (adiante) traz duas sugestões para sua conexão com o Arduino. Essa placa possui um regulador interno de 3,3 V, assim, pode ser alimentada com 5 V. Seus pinos são:

- VCC = 5V;
- GND = Terra;
- SCL e SDA = porta TWI (I²C) do Arduino;
- AD0 = endereço alternativo e
- INT = pino de interrupção.

Os pinos XDA e XCL não são usados neste estudo.



O dispositivo GY-521 será aqui chamado de MPU-6050 ou apenas MPU. O pino AD0 permite que se usem dois desses dispositivos num único barramento I²C. Quando solto, este pino AD0 vai para zero graças a um resistor de *pull-down* interno. Com um resistor de *pull-up* externo é possível colocá-lo em nível alto e assim configurar a placa para operar no endereço alternativo.

A comunicação (SDA e SCL) é feita com o barramento I²C, também denominado de TWI. Existe um segundo barramento I²C (XDA e XDL), para outro sensor externo que não será abordado. No tópico G.4.2 (adiante) está a listagem de um conjunto de rotinas denominado Rot_i2c.ino que faz uso da biblioteca *Wire* e ajuda o leitor a acessar deste dispositivo. São 3 as funções importantes, descritas a seguir.

- `void i2c_wr(uint8_t adr, uint8_t reg, uint8_t dado)`, que escreve o *byte* `dado` no registrador `reg` do dispositivo cujo endereço é `adr`;

- `uint8_t char i2c_rd(uint8_t adr, uint8_t reg)`, que retorna o *byte* lido no registrador `reg` do dispositivo cujo endereço é `adr` e
- `void i2c_rd_rep(uint8_t adr, uint8_t reg, uint8_t qtd, uint8_t *vet)`, que retorna no vetor `vet` a leitura de `qtd` registradores a partir do registrador `reg` do dispositivo cujo endereço é `adr`.

A operação do MPU é feita através de uma grande quantidade de registradores. Os principais estão listados na Tabela G.2, mostrada adiante. Sua preparação para operação envolve uma série de etapas, que são descritas logo a seguir:

1. Retirar o MPU do modo *sleep* e definir relógio;
2. Testar comunicação;
3. Opcional: Realizar o auto teste (*self-test*);
4. Realizar a calibração ;
5. Configuração final do MPU com seleção das escalas e
6. Ficar em um laço (ou interrupção), lendo valores medidos pelo MPU.

A seguir são apresentados os registradores, com os valores sugeridos para programação de cada etapa e alguns comentários.

ETAPA 1: Retirar do modo *sleep*.

PWR_MGMT_1 (0x6B)							
7	6	5	4	3	2	1	0
DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		
0	0	0	0	0	0	0	1

Retirar o MPU do modo sleep e selecionar como fonte de relógio o PLL do eixo X do giroscópio.

ETAPA 2: Testar comunicação.

WHO_AM_I (0x75)							
7	6	5	4	3	2	1	0
-	1	1	0	1	0	0	-

A leitura deste registrador sempre retorna 0x68 e por isso é usado para testar a comunicação.

ETAPA 3: (opcional) Realizar o auto teste → Habilitar o modo auto teste do giroscópio e do acelerômetro e depois realizar auto teste (ver programa *ERGp2*).

GYRO_CONFIG (0x1B)							
7	6	5	4	3	2	1	0
XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]		-	-	-
1	1	1	1	0	0	0	0

ACCEL_CONFIG (0x1C)							
7	6	5	4	3	2	1	0
XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]		-	-	-
1	1	1	0	0	0	0	0

Colocar o giroscópio e o acelerômetro no modo de auto teste (self-test); note que foram selecionadas as escalas FS_SEL = 0 → +/- 250 gr/s e AFS_SEL = 0 → +/- 8 g.

ETAPA 4: Calibrar → Obter a média de várias leituras para conhecer o erro intrínseco de cada sensor (ver programa *ERGp2*).

PWR_MGMT_1 (0x6B)							
7	6	5	4	3	2	1	0
DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		
1	0	0	0	0	0	0	0

Ressetar, o MPU zera automaticamente o bit de reset.

PWR_MGMT_1 (0x6B)							
7	6	5	4	3	2	1	0
DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		
0	0	0	0	0	0	0	1

Retirar o MPU do modo sleep e selecionar como fonte de relógio o PLL do eixo X do giroscópio.

GYRO_CONFIG (0x1B)							
7	6	5	4	3	2	1	0
XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]		-	-	-
0	0	0	x	x	0	0	0

ACCEL_CONFIG (0x1C)							
7	6	5	4	3	2	1	0

XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]		-	-	-
0	0	0	x	x	0	0	0

Escolher as escalas para o giroscópio e o acelerômetro
(giroscópio FS_SEL: 00= +/-250gr/s, 01= +/-500gr/s, 10= +/-1000gr/s e 11= +/-2000gr/s)
(acelerômetro AFS_SEL: 00 = +/-2 g, 01 = +/- 4 g, 10 = +/- 8 g e 11 = +/- 16 g)

CONFIG (0x1A)							
7	6	5	4	3	2	1	0
-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		
0	0	0	0	0	0	0	1

Especificar a banda passante do filtro digital passa baixo, no caso foram selecionados 184 Hz de BW para o acelerômetro e 188 Hz de BW para o giroscópio, com taxa de 1 kHz.

SMPRT_DIV (0x19)							
7	6	5	4	3	2	1	0
SMPLRT_DIV[7:0]							
0	0	0	0	0	0	0	0

Colocar o divisor em 0, isto significa que a Taxa de Amostragem será de 1 kHz, dada pelo registrador acima (lembrar de somar 1 ao valor programado neste divisor).

INT_ENABLE (0x38)							
7	6	5	4	3	2	1	0
-	MOT_EN	-	FIFO_OFLOW_EN	I2C_MST_INT_EN	-	-	DATA_RDY_EN
0	0	0	0	0	0	0	1

Habilitar a interrupção por dado pronto. Na verdade não será usada interrupção, mas a consulta (polling) de dado pronto.

INT_STATUS (0x3A)							
7	6	5	4	3	2	1	0
-	MOT_INT	-	FIFO_OFLOW_INT	I2C_MST_INT	-	-	DATA_RDY_INT

Consultando o bit DATA_RDY_INT, calcular a média das leituras dos diversos eixos e assim calcular o erro intrínseco (off set ou bias) de cada sensor.

ETAPA 5: Configuração para operação com a seleção das escalas desejadas

PWR_MGMT_1 (0x6B)

7	6	5	4	3	2	1	0
DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		
0	0	0	0	0	0	0	1

Retirar o MPU do modo sleep e selecionar como fonte de relógio o PLL do eixo X do giroscópio.

CONFIG (0x1A)							
7	6	5	4	3	2	1	0
-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		
0	0	0	0	0	0	1	1

Especificar a banda passante do filtro digital passa baixo, para o caso foram selecionados 44 Hz de BW do acelerômetro e 42 Hz de BW para o giroscópio, com taxa de 1 kHz.

SMPRT_DIV (0x19)							
7	6	5	4	3	2	1	0
SMPLRT_DIV[7:0]							
0	0	0	0	0	1	0	0

Por exemplo, foi programado 4, o que significa que o divisor é 5 (somar 1 ao valor programado), logo a Taxa de Amostragem é 200 Hz.

GYRO_CONFIG (0x1B)							
7	6	5	4	3	2	1	0
XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]		-	-	-
0	0	0	x	x	0	0	0

ACCEL_CONFIG (0x1C)							
7	6	5	4	3	2	1	0
XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]		-	-	-
0	0	0	x	x	0	0	0

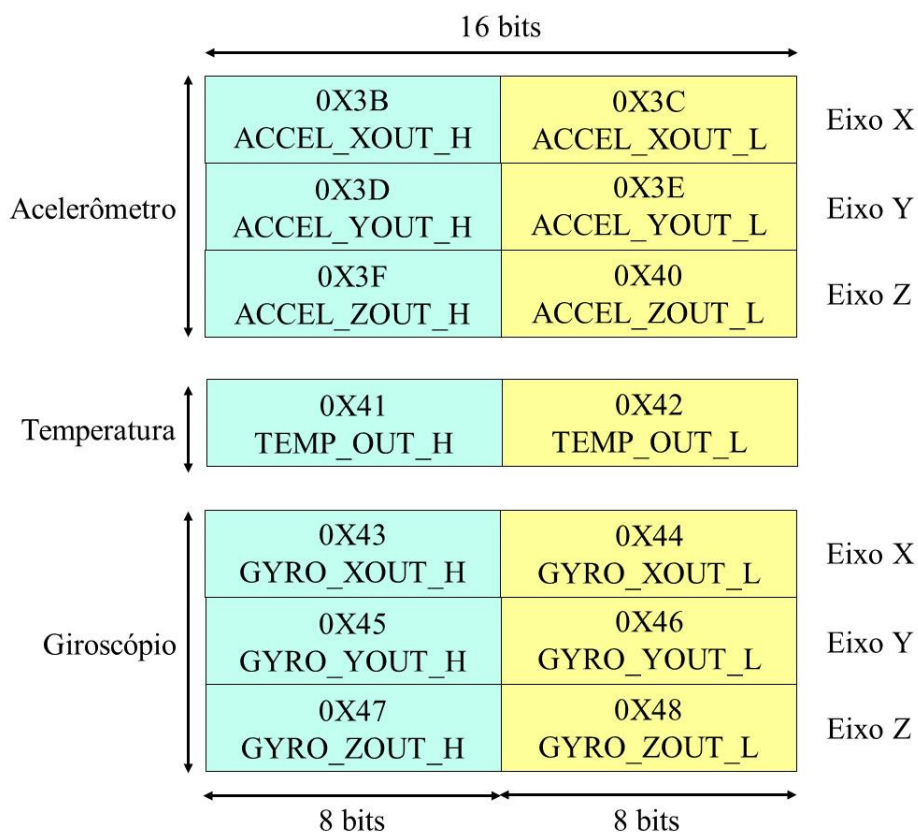
*Escolher as escalas para o giroscópio e o acelerômetro
(giroscópio FS_SEL: 00= +/-250gr/s, 01= +/-500gr/s, 10= +/-1000gr/s e 11= +/-2000gr/s)
(acelerômetro AFS_SEL: 00 = +/-2 g, 01 = +/- 4 g, 10 = +/- 8 g e 11 = +/- 16 g)*

ETAPA 6: Empregar o MPU lendo diretamente os registradores, fazendo o *polling* de dado pronto (DATA_RDY_INT) ou então usar interrupção.

A figura abaixo mostra os 16 registradores que fornecem as leituras dos 3 eixos do acelerômetro, dos 3 eixos do giroscópio e da temperatura. Como eles estão em sequência, é fácil lê-los com a função `void i2c_rd_rep(uint8_t adr, uint8_t reg, uint8_t qtd, uint8_t *vet)`. Os valores de aceleração e rotação são fornecidos na representação de 16 *bits* com sinal. Isto significa que sua faixa vai desde -32.768 até 32.767. A interpretação desses valores depende da escala que está sendo usada. Por exemplo, com o acelerômetro na escala +/- 2 g, cada *bit* vale 2/32.767. Se for lido o valor 12.345, significa uma aceleração de 0,75 g (12.345 x 2/32.767).

A temperatura tem uma interpretação diferente. Deve ser usada a fórmula abaixo, sugerida pelo fabricante, onde TEMP_OUT é o valor em 16 *bits* lido a partir dos registradores de temperatura:

$$T_{Celsius} = \frac{TEMP_OUT}{340} + 36,53.$$



Os exercícios resolvidos apresentam diversas sugestões de programas. Os autores recomendam a leitura de todo esse apêndice.

G.1. Conceitos sobre Acelerômetros e Giroscópios

O acelerômetro, como é óbvio, serve para medir a aceleração a que o dispositivo está submetido. A forma mais simples de se entender o funcionamento de tal dispositivo é analisar uma barra flexível com uma massa presa em sua extremidade, como mostrado na Figura G.1. Quando em repouso, a haste deve estar estendida, porém, quando submetida a uma aceleração, essa haste deve fletir em função da inércia da massa colocada em sua extremidade. O uso de 3 sensores desse tipo, um em cada direção nos permite criar um acelerômetro de 3 eixos. Como estamos na superfície terrestre, o dispositivo estará sempre sujeito à aceleração da gravidade ($g = 9,81 \text{ m/s}^2$), ou seja, um dos eixos deverá indicar 1 g (a não ser que se posicione o acelerômetro de forma inclinada).

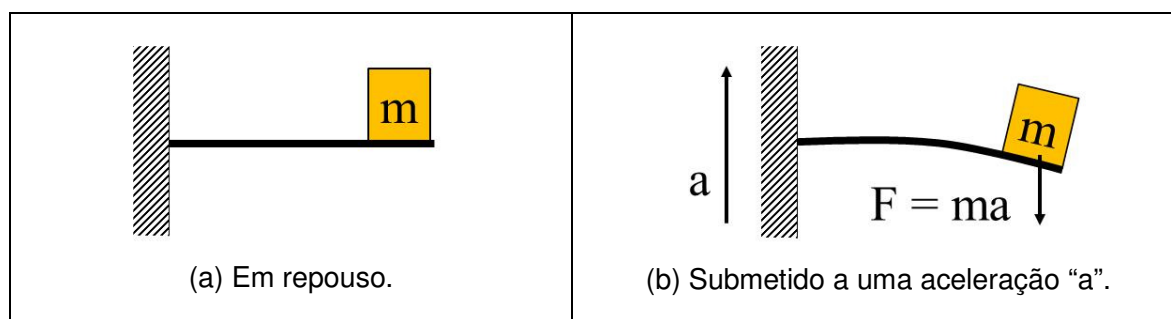


Figura G.1. Exemplo ilustrativo de um acelerômetro usando uma haste flexível com uma extremidade fixa e uma massa “m” na outra extremidade.

Outra forma de se medir a aceleração é com o emprego de cristal piezoelétrico. Tal cristal apresenta deformação quando submetido a uma tensão e, por outro lado, apresenta uma tensão quando sofre uma deformação. A Figura G.2 apresenta um acelerômetro que explora esta propriedade. Quando em repouso, uma mola mantém o cristal na posição e com tensão de saída igual a zero. Já quando submetido a uma aceleração, devido a inércia da massa, surge uma força que vai deformar o cristal e gerar uma tensão diferente de zero. Quando maior a força, maior a tensão.

--	--

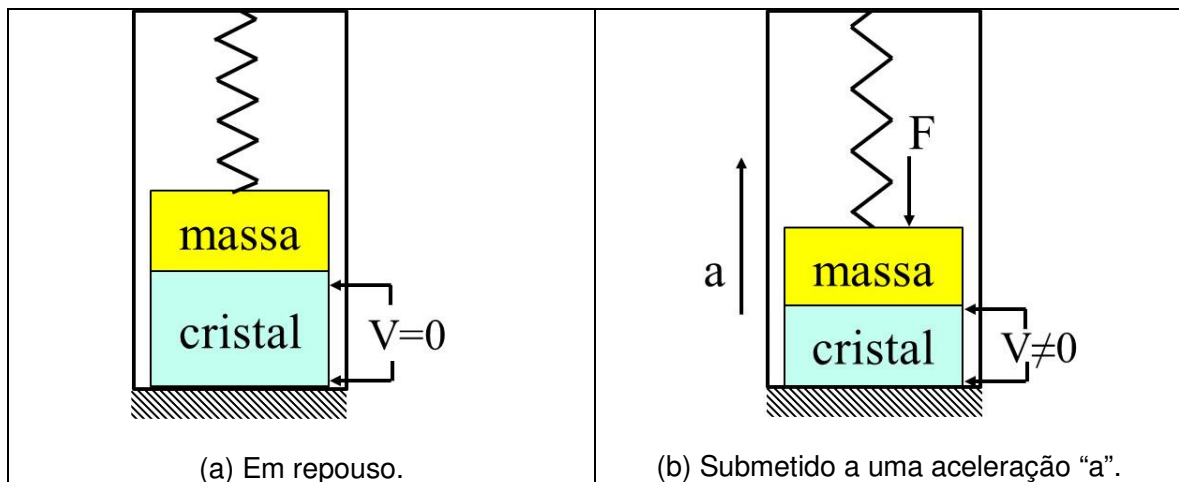


Figura G.2. Exemplo ilustrativo de um acelerômetro usando um cristal piezoelétrico e uma massa que o deforma quando submetida a uma aceleração.

O MPU-6050 usa um princípio um pouco diferente, construído com a tecnologia denominada Sistema Micro Eletromecânico, MEMS, (*Micro-Electro-Mechanical System*). Esta tecnologia permite construir sistemas mecânicos miniaturizados. No caso do acelerômetro, o tamanho é da ordem de 0,1 mm. A Figura G.3 apresenta uma representação simplificada de um acelerômetro. Note a massa e a presença de duas “molas”. As placas da porção fixa e da porção móvel formam diversos pares de capacitores. Esses capacitores são denominados de C1 e C2 (a figura apresenta apenas um par de C1 e C2). Quando em repouso essas capacitâncias devem ser idênticas. Porém, quando a peça é submetida a uma aceleração, da esquerda para a direita no caso do exemplo da figura, a inércia faz a parte móvel se movimentar (relativamente) na direção oposta e com isso a capacitância de C2 fica maior que a de C1. Pela relação entre os esses dois capacitores, é possível estimar a aceleração a que a peça está submetida.

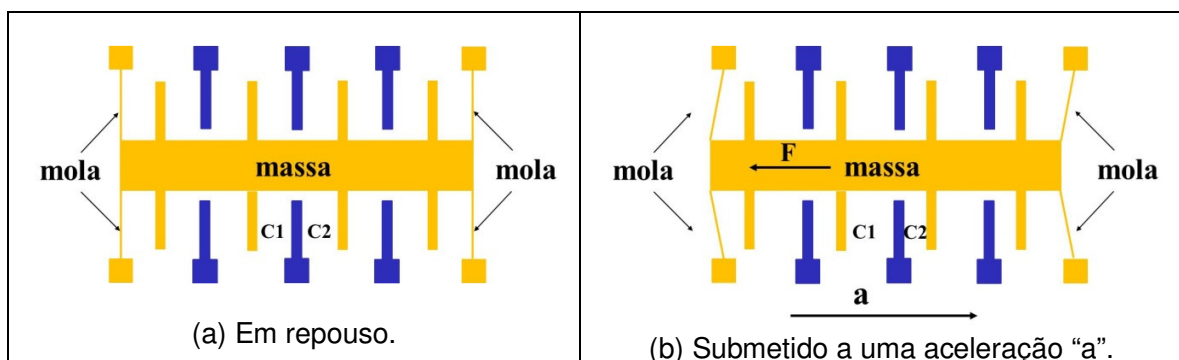


Figura G.3. Exemplo ilustrativo de um acelerômetro MEMS, usando uma peça fixa e uma peça móvel cuja posição relativa, quando submetida a uma aceleração, provoca variação das capacitâncias C1 e C2.

A explicação do giroscópio é um pouco mais complexa, pois envolve o conceito da Aceleração de Coriolis. Iniciamos imaginando uma massa que se desloca numa determinada direção, como mostrado na Figura G.4. Se a essa massa aplicarmos um giro ou uma rotação, surge uma força perpendicular à direção do movimento, resultado da Aceleração de Coriolis. Se girarmos no sentido oposto, a Força de Coriolis é invertida. Quanto maior for a velocidade angular deste giro, maior será o módulo da força.

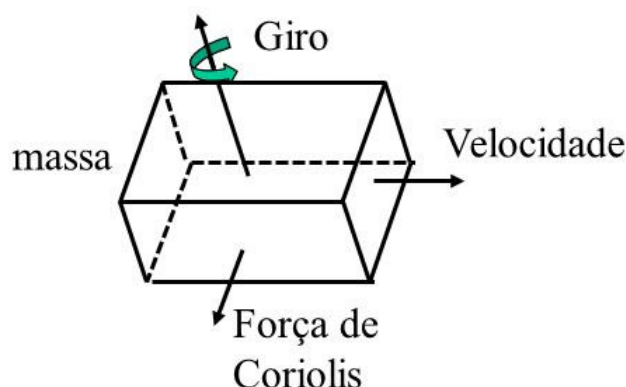


Figura G.4. Ilustração da Força de Coriolis agindo sobre uma massa que foi submetida a um giro enquanto estava em movimento.

O giroscópio que vamos descrever é na verdade um acelerômetro construído para medir a Aceleração de Coriolis. Na Figura G.5 vemos o acelerômetro MEMS, descrito na Figura G.3, mas agora submetido a um movimento oscilatório na horizontal. Esta oscilação garante que a massa esteja sempre em movimento. Quando for submetido a um giro, o acelerômetro irá indicar o valor da Aceleração de Coriolis, que permite estimar a velocidade angular aplicada à peça.

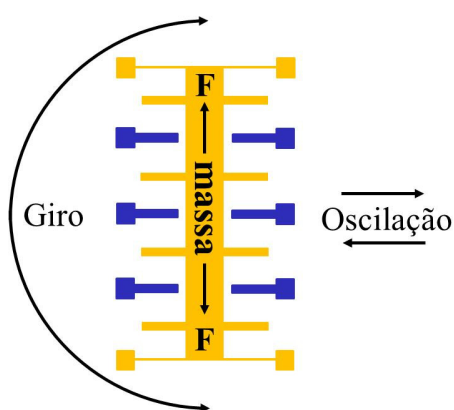


Figura G.5. Ilustração de um giroscópio que estima a velocidade angular a partir da Aceleração de Coriolis medida por um acelerômetro MEMS que é mantido em permanente oscilação.

De uma forma um pouco diferente, a Aceleração de Coriolis pode ser experimentada com um HD USB (não acontece com o HD de estado sólido). Após ser conectado ao USB, o disco que está dentro do HD começa a girar. Se pegarmos esse HD com a mão e o movimentarmos com um giro, vamos sentir um comportamento um pouco inesperado, como se uma força estivesse atuando em uma de suas extremidades.

G.2. Características do MPU-6050

Abaixo é apresentada uma lista das principais características do MPU-6050.

- Giroscópio de 3 eixos:
 - Escalas de +/- 250, +/- 500, +/- 1.000, +/- 2.000 graus/segundo;
 - Pino FSYNC para sincronismo externo;
 - ADCs de 16 *bits*;
 - Corrente de operação = 3,6 mA;
 - Corrente *standby* = 5 μ A.
- Acelerômetro de 3 eixos:
 - Escalas de +/- 2 g, +/- 4 g, +/- 8 g, +/- 16 g;
 - ADCs de 16 *bits*;
 - Corrente de operação = 500 μ A;
 - Corrente *standby* inferior a 110 μ A;
 - Interrupção que pode ser disparado por uma aceleração elevada;
- Recursos adicionais:
 - Processador Digital de Movimento (DMP – *Digital Motion Processor*);
 - Capacidade para 9 eixos usando o DMP;
 - Barramento I²C (TWI) auxiliar para sensor externo;
 - Consumo máximo de 3,9 mA (com 6 eixos e DMP habilitados);
 - *Buffer* para uma fila interna de 1024 *bytes*;
 - Sensor de temperatura;
 - Suporta choques de até 10.000 g;
 - I²C (TWI) operando em 400 kHz;

G.3. Como Conectar o MPU-6050

É preciso alertar que existem diversos fabricantes disponibilizando placas (*breakout boards*) com o *chip* MPU-6050. Não há padronização nas pinagens e algumas placas nem têm nome de identificação. Por isso, para se orientar, recomendamos uma consulta à página <http://playground.arduino.cc/Main/MPU-6050>, que traz explicações sobre as placas mais comuns.

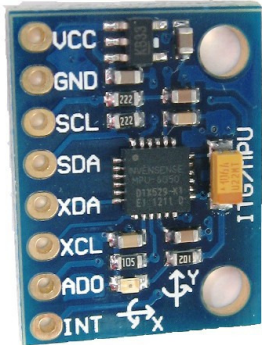
Dentre os principais fabricantes de placas com o MPU-6050, citamos:

- Sparkfun (SEN-11028): <https://www.sparkfun.com>
- Drotek (IMU 10DOF e IMU 6DOF): <http://www.drotek.com>

- Flyduino: <http://flyduino.net>

Uma placa que é muito comum no Brasil é a GY-521, cuja foto é apresentada na tabela abaixo. Ela é interessante por trazer um regulador de 3,3V. Isto permite que seja alimentada com faixa de tensão de 3,3 até 5 V. Existem diversas recomendações para que se use 5 V para beneficiar a precisão. A Tabela G.1 lista a pinagem desta placa. As linhas SCL e SDA já têm resistores de *pull-up* (4,7 k Ω) e a linha AD0 tem um resistor de *pull-down* (4,7 k Ω). Se a linha de endereço AD0 for deixada solta, o endereço da placa é 0x68. Porém, se ela for conectada a VCC, o endereço passa a ser 0x69. Isto permite que se usem dois *chips* MPU-6050 em um mesmo sistema. As linhas XCL e XDA formam o barramento I²C auxiliar para conectar outro dispositivo, como um magnetômetro. Esse barramento auxiliar não será abordado neste estudo.

Tabela G.1. Pinagem da placa GY-521 (MPU-6050)

Pino	Descrição	Observações	Foto
VCC	3,3 até 5,0 Volts	-	
GND	Terra	-	
SCL	Bus I ² C (TWI)	<i>pull-up</i> de 4,7 k Ω	
SDA	Bus I ² C (TWI)	<i>pull-up</i> de 4,7 k Ω	
XDA	Bus I ² C (TWI) aux.	-	
XCL	Bus I ² C (TWI) aux.	-	
AD0	Endereço	<i>pull-down</i> de 4,7 k Ω	
INT	Interrupção	-	

(imagem obtida na página <http://playground.arduino.cc/Main/MPU-6050>).

G.3.1. Conexão Elétrica do MPU-6050 (GY-521)

A conexão com uma placa Arduino é muito simples. É recomendado que se remova a alimentação antes de se fazer as conexões. A Figura G.6 traz duas sugestões: uma para o Arduino Mega e outra para o Uno. O VCC e o Terra devem ser ligados aos respectivos pinos do Arduino. O barramento I²C (TWI) deve ser ligado aos pinos SDA e SCL. O pino INT, que é o de interrupção (se for usada) deve ser ligado a alguma interrupção do Arduino. No caso do Uno foi escolhida a INT0 (PD2) e no Mega a INT4 (PE4), porque a INT0 do Mega é compartilhada com a linha SCL. As linhas do barramento I²C auxiliar devem ser deixadas soltas. Para terminar, o pino de endereço AD0 deve estar solto para se usar o endereço 0x68 ou conectado a VCC, para se usar o endereço 0x69.

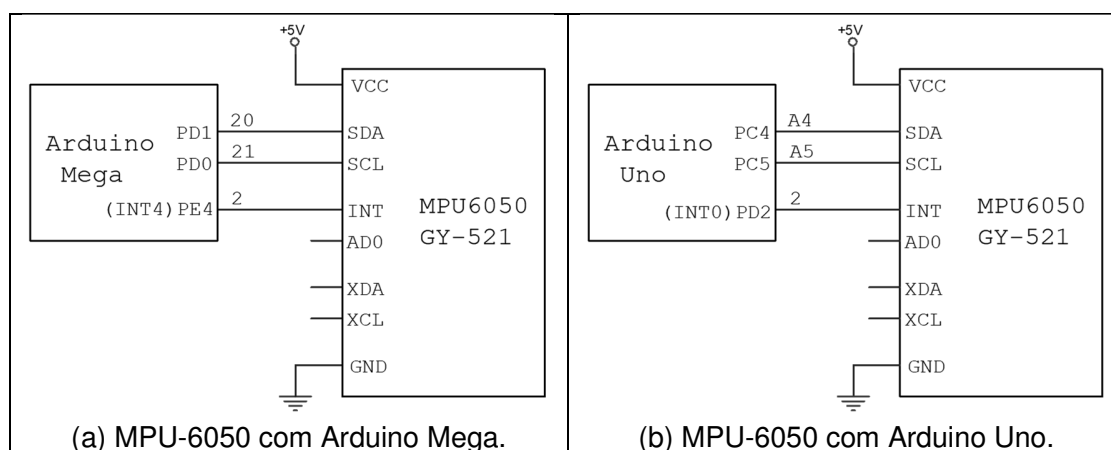


Figura G.6. Conexão do MPU-6050 com o Arduino Mega ou Uno.

Uma vez feitas as conexões mostradas na Figura G.6, o dispositivo já pode ser usado. Para aqueles que estiverem com pressa, recomenda-se uma consulta aos programas apresentados no Exercício Resolvido ER G.1. Antes de se usar o MPU-6050 é necessário fazer o auto teste para ver o dispositivo está dentro das especificações e depois fazer a calibração.

G.4. Como Operar o MPU-6050

O MPU-6050 é acessado e controlado através de seus registradores. A Tabela G.2 apresenta apenas os registradores que serão usados neste estudo. A quantidade disponível é bem maior. Para uma lista completa, favor consultar o manual. É importante citar que existem registradores que não estão listados nem no manual.

Tabela G.2. Lista dos registradores que serão usados

End	Registradores	7	6	5	4	3	2	1	0
0x0D	SELF_TEST_X	XA_TEST[4-2]			XG_TEST[4-0]				
0x0E	SELF_TEST_Y	YA_TEST[4-2]			YG_TEST[4-0]				
0x0F	SELF_TEST_Z	ZA_TEST[4-2]			ZG_TEST[4-0]				
0x10	SELF_TEST_A	-	-	XA_TEST[1-0]		YA_TEST[1-0]		ZA_TEST[1-0]	
0x19	SMPLRT_DIV	SMPLRT_DIV [7-0]							
0x1A	CONFIG	-	-	EXT_SYNC_SET[2-0]			DLPF_CFG[2-0]		
0x1B	GYRO_CONFIG	-	-	-	FS_SEL[1-0]		-	-	-
0x1C	ACCEL_CONFIG	XA_ST	YA_ST	ZA_ST	AFS_SEL[1-0]		-	-	-
0x1F	MOT_THR	MOT_THR[7-0]							
0x23	FIFO_EN	TEMP_FIFO_EN	XG_FIFO_EN	YG_FIFO_EN	ZG_FIFO_EN	ACCEL_FIFO_EN	SLV2	SLV1	SLV0
0x37	INT_PIN_CFG	INT_LEVEL	INT_OPEN	LATCH_INT_EN	INT_RD_CLEAR	FSYNC_INT_LEVEL	FSYNC_INT_EN	I2C_BY_PASS_EN	-
0x38	INT_ENABLE	-	MOT_EN	-	FIFO_OFLOW_EN	I2C_MST_INT_EN	-	-	DATA_RDY_EN
0x3A	INT_STATUS	-	MOT_INT	-	FIFO_OFLOW_INT	I2C_MST_INT	-	-	DATA_RDY_INT
0x3B	ACCEL_XOUT_H	ACCEL_XOUT[15-8]							
0x3C	ACCEL_XOUT_L	ACCEL_XOUT[7-0]							
0x3D	ACCEL_YOUT_H	ACCEL_YOUT[15-8]							

0x3E	ACCEL_YOUT_L	ACCEL_YOUT[7-0]							
0x3F	ACCEL_ZOUT_H	ACCEL_ZOUT[15-8]							
0x40	ACCEL_ZOUT_L	ACCEL_ZOUT[7-0]							
0x41	TEMP_OUT_H	TEMP_OUT[15-8]							
0x42	TEMP_OUT_L	TEMP_OUT[7-0]							
0x43	GYRO_XOUT_H	GYRO_XOUT[15-8]							
0x44	GYRO_XOUT_L	GYRO_XOUT[7-0]							
0x45	GYRO_YOUT_H	GYRO_YOUT[15-8]							
0x46	GYRO_YOUT_L	GYRO_YOUT[7-0]							
0x47	GYRO_ZOUT_H	GYRO_ZOUT[15-8]							
0x48	GYRO_ZOUT_L	GYRO_ZOUT[7-0]							
0x68	SIGNAL_PATH_RESET	-	-	-	-	-	GYRO_RESET	ACCEL_RESET	TEMP_RESET
0x69	MOT_DETECT_CTRL	-	-	ACCEL_ON_DELAY[1-0]		-	-	-	-
0x6A	USER_CTRL	-	FIFO_EN	I2C_MST_EN	I2C_IF_DIS	-	FIFO_RESET	I2C_MST_RESET	SIG_COND_RESET
0x6B	PWR_MGNT_1	DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2-0]		
0x6C	PWR_MGNT_2	LP_WAKE_CTRL[1-0]		STBY_XA	STBY_YA	STBY_YA	STBY_XG	STBY_YG	STBY_ZG
0x72	FIFO_COUNTH	FIFO_COUNT[15-8]							
0x73	FIFO_COOUNTL	FIFO_COUNT[7-0]							
0x74	FIFO_RW	FIFO_DATA[7-0]							
0x75	WHO_AM_I	-	WHO_AM_I[6-1]						-

G.4.1. Interpretação das Leituras do MPU-6050 (GY-521)

Para se usar o MPU-6050 é preciso saber interpretar suas leituras. A Tabela G.2 apresenta uma lista parcial de seus registradores. Os dados de aceleração, temperatura e giroscópio têm o tamanho de 16 *bits* e estão disponíveis em 14 registradores, cujos endereços vão desde 0x3B até 0x48. Note que esses registradores são de 8 *bits*, assim, o resultado em 16 *bits* é formado pela concatenação de dois registradores. A Figura G.7 apresenta essa concatenação.

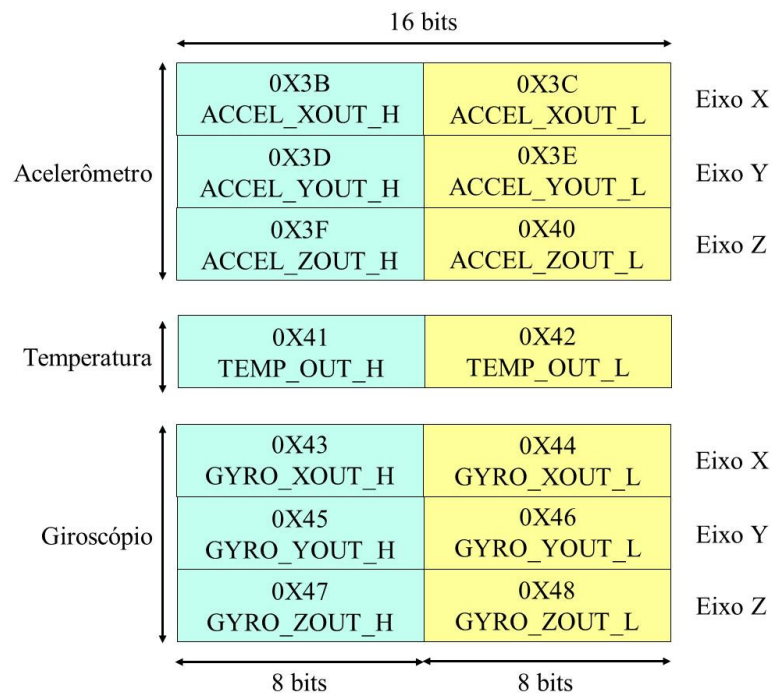


Figura G.7. Concatenação dos registradores para se obter as leituras do MPU-6050.

O trecho de programa a seguir apresenta uma sugestão de como fazer esta concatenação. É considerado que:

- axh = contenha leitura do registrador ACCEL_XOUT_H e
- axl = contenha leitura do registrador ACCEL_XOUT_L

```
// Concatenação de dois registradores do MPU-6050

int16_t ax;           //Receber leitura da aceleração eixo X
uint8_t axh, axl;     //Receber leituras parciais da aceleração eixo X

// axh recebe conteúdo do registrador ACCEL_XOUT_H
// axl recebe conteúdo do registrador ACCEL_XOUT_L

ax = (int16_t) ( ((int16_t)axh << 8) | axl ); //aceleração eixo X
```


Note que o *byte* mais significativo (*axh*) foi deslocado 8 *bits* para a esquerda e depois concatenado com o *byte* menos significativo (*axl*). A declaração *cast* (*int16_t*) foi usada para converter o *byte* mais significativo (*axh*) para 16 *bits* antes de fazer o deslocamento para a esquerda. Vale lembrar a equivalência entre as declarações (na maioria das vezes):

- `uint8_t` equivale a `unsigned char` e
- `int16_t` equivale a `int`.

Em caso de dúvidas, consulte o Apêndice C.

Os valores de aceleração, rotação são dados em 16 *bits* com sinal. Isto significa que sua faixa vai desde -32.768 até 32.767. A interpretação desses valores depende da escala que está sendo usada. O MPU-6050 oferece as seguintes escalas, que são especificadas nos registradores GYRO_CONFIG (0x1B) e ACCEL_CONFIG (0x1C), como mostrado na Tabela G.3.

Tabela G.3. Escalas e resoluções para o Acelerômetro e o Giroscópio

Acelerômetro			Giroscópio		
AFS_SEL	Escala	Resolução	FS_SEL	Escala	Resolução
0	+/- 2 g	2/32.767	0	+/- 250 gr/s	250/32.767
1	+/- 4 g	4/32.767	1	+/- 500 gr/s	500/32.767
2	+/- 8 g	8/ 32.767	2	+/- 1000 gr/s	1.000/32.767
3	+/- 16 g	16/32.767	3	+/- 2000 gr/s	2.000/32.767

gr = graus

A Figura G.8 apresenta a função que relaciona os dados lidos nos registradores e os valores de aceleração e rotação. Note as escalas indicadas na figura. A conta a ser feita é muito simples. Os exemplos a seguir auxiliam na compreensão.

Exemplo G.1: calcular a aceleração para o caso de ser lido 15.000, quando estava em uso a escala +/- 4 g.

$$aceleração = 15.000 \times \frac{4}{32.767} = 1,83 \text{ g.}$$

Exemplo G.2: calcular a aceleração para o caso de ser lido -20.000, quando estava em uso a escala +/- 8 g.

$$aceleração = -20.000 \times \frac{8}{32.767} = -4,88 \text{ g.}$$

Exemplo G.3: calcular a velocidade angular (giroscópio) para o caso de ser lido 12.345, quando estava em uso a escala +/- 250 graus/s

$$velocidade\ angular = 12.345 \times \frac{250}{32.767} = 94,19\ graus/s.$$

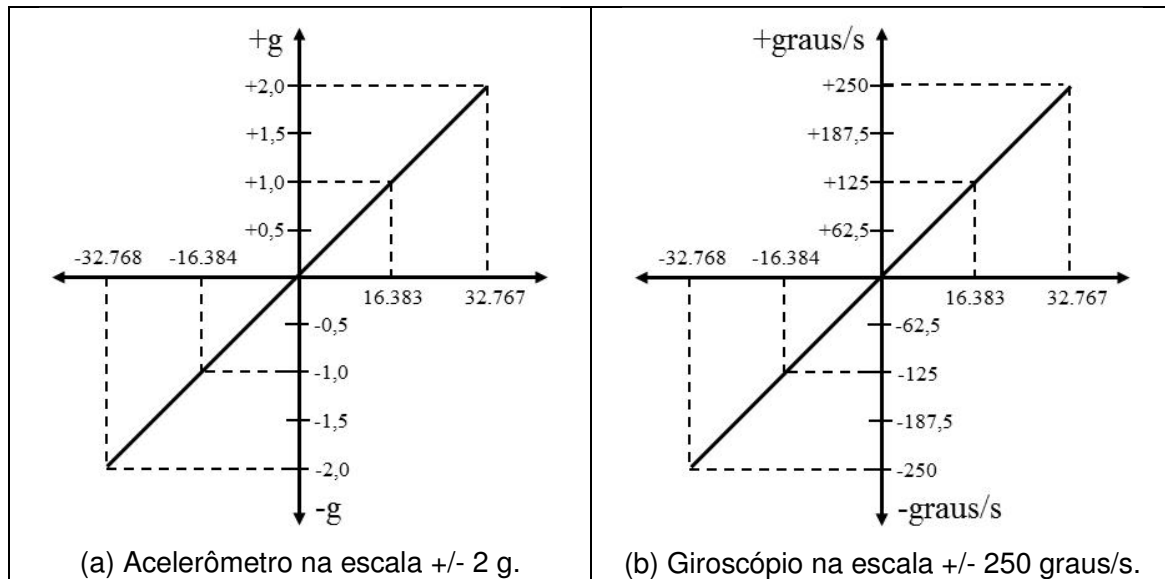


Figura G.8. Funções para interpretar as leituras dos valores do acelerômetro e do giroscópio, nas escalas indicadas.

A temperatura tem uma interpretação diferente. Para esse caso, deve ser usada a seguinte fórmula, sugerida pelo fabricante, onde TEMP_OUT é o valor em 16 *bits* lido a partir dos registradores de temperatura:

$$T_{Celsius} = \frac{TEMP_OUT}{340} + 36,53.$$

Exemplo G.4: calcular a temperatura para o caso de ser lido -5.678.

$$T_{Celsius} = \frac{-5.678}{340} + 36,53 = 19,83\ graus\ Celsius.$$

G.4.2. Preparação do MPU-6050 (GY-521) e Barramento i²C (TWI)

É preciso tomar uma série de procedimentos antes de se empregar o MPU-6050. Ao ser energizado (ligado) o MPU-6050 inicia no modo de baixo consumo (*sleep*). Assim, o primeiro procedimento é retirar o MPU do modo *sleep*. Além disso, uma série de outros procedimentos precisa ser executada. A lista abaixo apresenta uma sugestão.

1. Retirar o MPU do modo *sleep* e testar comunicação;
2. Realizar o auto teste (*self-test*);
3. Realizar a calibração;
4. Configurar o MPU e
5. Selecionar as escalas;
6. MPU está operacional e pronto para uso.

A comunicação com o MPU-6050 é feita via barramento I²C, que a família AVR chama de TWI. O Capítulo 11 apresenta um estudo completo desse recurso TWI, onde são apresentadas as ferramentas para o usuário criar sua própria biblioteca I²C. Entretanto, já que o objetivo deste anexo é apenas o estudo do MPU-6050, será usada a biblioteca *Wire* para evitar um excesso de informações. Essa biblioteca é de amplo uso entre os usuários do Arduino. Usando a biblioteca, foram construídas 3 funções, descritas abaixo e listadas a seguir.

- void **i2c_wr**(uint8_t adr, uint8_t reg, uint8_t dado), escreve o *byte* dado no registrador *reg* do dispositivo cujo endereço é *adr*.
- uint8_t char **i2c_rd**(uint8_t adr, uint8_t reg), retorna o *byte* lido no registrador *reg* do dispositivo cujo endereço é *adr*.
- void **i2c_rd_rep**(uint8_t adr, uint8_t reg, uint8_t qtd, uint8_t *vet), que retorna no vetor *vet* a leitura de *qtd* registradores a partir do registrador *reg* do dispositivo cujo endereço é *adr*.

*Rot_i2c.ino: conjunto de funções para facilitar o acesso ao MPU. Note que essas rotinas fazem uso do biblioteca WIRE.h
(este arquivo deve estar no mesmo diretório em que está o programa em elaboração)*

```
// Escrever um dado num registrador do dispositivo adr
void i2c_wr(uint8_t adr, uint8_t reg, uint8_t dado){
    Wire.beginTransmission(adr); //Inicializar buffer TX
    Wire.write(reg);              //buffer=registrador
    Wire.write(dado);             //buffer=dado
    Wire.endTransmission();       //Enviar buffer e finalizar
}

// Ler um dado de um registrador do dispositivo adr
uint8_t char i2c_rd(uint8_t adr, uint8_t reg){
    Wire.beginTransmission(adr); //Inicializar buffer TX
    Wire.write(reg);              //buffer=registrador
    Wire.endTransmission(false); //Enviar buffer e gerar restart
    Wire.requestFrom(endereco, (uint8_t) 1); //Ler um byte
    return Wire.read();           //Receber um dado
}
```

```
// Ler qtd dados a partir do de um registrador do dispositivo adr
void i2c_rd_rep(uint8_t adr, uint8_t reg, uint8_t qtd, uint8_t *vet){
    uint8_t cont;
    Wire.beginTransaction(adr);          //Inicializar buffer TX
    Wire.write(registrador);            //buffer=registrador
    Wire.endTransmission(false);        //Enviar buffer e gerar restart
    Wire.requestFrom(endereco, qtd);    //Ler bytes
    for (cont=0; cont<qtd; cont++){
        vet[cont]=Wire.read();          //Receber um dado por vez
    }
}
```

G.4.3. Retirar MPU do Modo *Sleep* e Testar a Comunicação

Como já foi dito, ao ser energizado, o MPU parte no modo *sleep*. Para retirá-lo deste modo, basta zerar o *bit* SLEEP do registrador PWR_MGMT_1 (0x6B). Além disso, o MPU inicia usando o oscilador interno de 8 MHz. O manual recomenda fortemente que, ao invés desse oscilador interno, se use o relógio de um dos giroscópios, pois eles oferecem melhor estabilidade. O mais usual é selecionar o relógio do eixo X (CLKSEL = 1).

PWR_MGMT_1 (0x6B)							
7	6	5	4	3	2	1	0
DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		
0	0	0	0	0	0	0	1

Figura G.9. Atualização do registrador PWR_MGMT_1 para retirar o MPU do modo *sleep* e selecionar como fonte de relógio o PLL do eixo X do giroscópio.

A forma mais simples de se testar a comunicação com o MPU é lendo o registrador WHO_AM_I (0x75) que sempre retorna o valor 0x68. Esse é o endereço do MPU no barramento I²C (TWI). O valor do pino AD0 não interfere na leitura deste registrador.

O ER G.2 apresenta um programa que retira o MPU do modo *sleep*, seleciona o relógio e testa a comunicação.

G.4.4. Realizar o Self_Test do MPU

Para garantir o correto funcionamento do MPU, o fabricante previu um procedimento para testar as porções elétricas e mecânicas do dispositivo, denominado de *Self-Test*. Esse teste deve ser realizado com cada um dos 6 eixos. Quando o *Self-Test* está ativado, a porção eletrônica provoca uma atuação pré-determinada nos sensores, gerando assim uma saída que deve ser confrontada com os limites indicados pelo fabricante.

A Invensense (fabricante) indica o programa MotionApps para realizar o *Self-Test*. Por outro lado, existem na Internet diversos programas com essa finalidade. Porém, há certa

disparidade entre eles. Aqui, para cumprir com nossa finalidade vamos seguir o protocolo de teste especificado no manual.

Para que possamos explicar o teste, usaremos os símbolos listados abaixo. Note que no *self-test*, são usados dois valores para cada eixo: um que corresponde à leitura do eixo (GX ou GX_{ST}, de 16 *bits*) e outro que é a resposta ao modo teste (GXT, de 5 *bits*).

- GX = leitura do eixo X do giroscópio com o *self-test* desabilitado (idem para GY, GZ, AX, AY e AZ, em 16 *bits* com sinal)
- GX_{ST} = leitura do eixo X do giroscópio com o *self-test* habilitado (idem para GY_{ST}, GZ_{ST}, AX_{ST}, AY_{ST} e AZ_{ST}, em 16 *bits* com sinal);
- STR_GX = resposta ao *self-test* para o eixo X do giroscópio (idem para STR_GY, STR_GZ, STR_AX, STR_AY e STR_AZ);
- FT_GX = “corte” de fábrica (*factory trim*) para o eixo X do giroscópio (idem para FT_GY, FT_GZ, FT_AX, FT_AY e FT_AZ);
- GXT = valor do registrador de teste para o eixo X do giroscópio (idem para GYT, GZT, AXT, AYT e AZT, em 5 *bits* com sinal);
- P_GX = divergência entre o “corte” de fábrica e a resposta ao *self-test* para o eixo X do giroscópio em percentagem (idem para P_GY, P_GZ, P_AX, P_AY e P_AZ);

Tabela G.4. Fórmulas para o cálculo do Corte de Fábrica (Factory Trim) de acordo com o manual do fabricante

Giroscópio	Acelerômetro
$FT_{GX} = 25 \times 131 \times 1,046^{(GXT-1)}$	$FT_{AX} = 4096 \times 0,34 \times \left(\frac{0,92}{0,34}\right)^{\frac{AXT-1}{30}}$
$FT_{GY} = -25 \times 131 \times 1,046^{(GYT-1)}$	$FT_{AY} = 4096 \times 0,34 \times \left(\frac{0,92}{0,34}\right)^{\frac{AYT-1}{30}}$
$FT_{GZ} = 25 \times 131 \times 1,046^{(GZT-1)}$	$FT_{AZ} = 4096 \times 0,34 \times \left(\frac{0,92}{0,34}\right)^{\frac{AZT-1}{30}}$

Observação: se o valor do registrador de teste (GXT, GYT, ...) for zero, o “corte” de fábrica correspondente deverá ser zero.

A seguir é descrito o procedimento para realizar o *self-test* segundo o que está descrito no manual do fabricante. Será usado como exemplo o eixo X do giroscópio. O mesmo procedimento deve ser repetido para os outros 5 eixos do MPU.

1. Selecionar no registrador de configuração (ACCEL_CONFIG e GYRO_CONFIG) a escala +/-250 graus/s para o giroscópio e a escala +/- 8 g para o acelerômetro.
2. Fazer a leitura de GX (GYRO_XOUT_H e GYRO_XOUT_L);
3. Selecionar no registrador de configuração (ACCEL_CONFIG e GYRO_CONFIG) o modo teste, as escalas +/-250 graus/s e +/- 8 g devem ser mantidas.
4. Fazer a leitura de GX_{ST} (GYRO_XOUT_H e GYRO_XOUT_L);

5. Fazer a leitura do registrador de teste (SELF_TEST_X)
6. Separar o valor de GXT;
7. Calcular $STR_GX = GX_{ST} - GX$;
8. Calcular FT_GX de acordo com a Tabela G.4 (verificar se GXT \neq 0);
9. Calcular a percentagem do desvio, segundo a fórmula abaixo; para passar no *self-test*, essa percentagem do desvio (P_GX) deve ser menor que 14%.

$$P_GX = 100 \times \left(\frac{STR_GX - FT_GX}{FT_GX} \right)$$

O programa do ER G.2 apresenta a rotina `uint8_t MPU_st(void)`, que realiza o *self-test* como foi descrito e retorna 1 (TRUE) se o MPU passar ou retorna 0 (FALSE) se o MPU falhar. Neste programa, o leitor pode remover os comentários para imprimir no Monitor Serial todos os valores obtidos durante este teste.

G.4.4. Calibração do MPU

Como era de se esperar, com o MPU perfeitamente parado sobre uma superfície, as leituras dos sensores não são zero, mas sim ligeiramente diferentes de zero. Isso acontece porque sempre há imprecisão (erro intrínseco) na fabricação do *chip*. Em inglês esse erro é chamado de *bias* ou *offset*. Em português usaremos simplesmente o termo “erro”. A calibração consiste em determinar o erro médio em cada um dos 6 eixos para depois poder descontá-lo, de tal forma que as leituras fiquem o mais próximas de zero possível quando o dispositivo está parado. É claro que um dos eixos do acelerômetro deve indicar algo próximo da aceleração da gravidade (1 g), que será incluído dentro do erro calculado.

Para determinar esse erro, basta colocar o MPU sobre uma superfície estável e realizar uma grande quantidade de medidas em cada eixo. O erro (*bias* ou *offset*) será dado pela média das medidas em cada eixo. De posse desse valor, ele deve ser descontado das leituras futuras. É importante comentar que a temperatura tem influência no comportamento do MPU, assim, após ligar, é interessante esperar algum tempo (2 a 5 minutos) para que o MPU estabilize sua temperatura. Neste estudo, não será levado em conta o efeito da alteração da temperatura. Recomenda-se, portanto, cuidado quando o MPU for usado em ambientes com grandes diferenças de temperatura. Neste caso, seria conveniente fazer uma nova calibração a cada grande alteração de temperatura.

Existem duas formas de se trabalhar com os resultados da calibração. A primeira, mais simples, consiste em guardar o valor numérico (inteiro, 16 *bits*) do erro dos sensores e descontá-lo das próximas leituras. Esta é mais simples porque, a cada leitura, se faz apenas uma conta com inteiros. A segunda forma consiste em calcular o erro em “g” e em “graus/s” (ponto flutuante), que será subtraído do valor calculado a partir das leituras futuras. Esta última forma é mais adequada quando a aplicação demanda pela troca das escalas do MPU.

O Exercício Resolvido G.2 apresenta um programa onde são realizadas todas as etapas necessárias para se colocar o MPU no modo operacional. Em destaque está a rotina `void MPU_calibra(int16_t *bias, float *valor)` que faz 256 leituras de cada sensor para calcular o erro médio. No vetor `bias` ela retorna o valor inteiro do erro de cada sensor, na sequência acelerômetro (X, Y e Z) e giroscópio (X, Y e Z). Já no vetor `valor`, a função retorna o erro em “g” do acelerômetro e o erro em “graus/s” do giroscópio, na mesma sequência.

G.4.5. Selecionando a Banda Passante do Filtro Digital Passa Baixa (LPDF)

O MPU oferece o recurso de um Filtro Digital Passa Baixa (DLPF, em inglês *Digital Low Pass Filter*). Este filtro digital permite que se eliminem ruídos, que podem interferir com o sinal que se está medindo. Para explicar melhor, vamos imaginar que o MPU esteja sendo usado para acompanhar os movimentos de uma plataforma móvel, acionada por um motor. Consideremos ainda que esse motor, quando acionado, produza vibrações em torno de 200 Hz. Esse ruído vibratório de 200 Hz certamente irá corromper todas as medições de aceleração e rotação. Com o uso do filtro passa baixo, por exemplo, selecionado para frequências de corte em 94 Hz (acelerômetro) e em 98 Hz (giroscópio), esse ruído de 200 Hz será bastante atenuado. O preço a ser pago por usar tal filtro é um atraso nas leituras. Por exemplo, para o acelerômetro, esse filtro digital (corte 94 Hz) introduz um atraso de 3,0 ms (vide Tabela G.5).

A Tabela G.5 apresenta as possíveis configurações de Banda Passante (BW) do Filtro Digital Passa Baixa (DLPF) e do atraso que resulta em função de seu uso. É preciso notar que a frequência de amostragem (f_s) do Acelerômetro é sempre de 1 kHz, ou seja, uma amostra a cada 1 ms. Já o Giroscópio pode trabalhar com a frequência de amostragem (f_s) de 1 kHz ou de 8 kHz. Quando o giroscópio for configurado para trabalhar em 8 kHz (uma amostra a cada 0,125 ms) diversas leituras do acelerômetro serão iguais, pois ele continua a operar em 1 kHz.

Tabela G.5. Configuração da Frequência de Amostragem (f_s) e da Banda Passante (BW)

DLPF _CFG	Acelerômetro			Giroscópio		
	BW (Hz)	Atraso (ms)	F_s (kHz)	BW (Hz)	Atraso (ms)	F_s (kHz)
0	260	0	1	256	0,98	8
1	184	2,0	1	188	1,9	1
2	94	3,0	1	98	2,8	1
3	44	4,9	1	42	4,8	1
4	21	8,5	1	20	8,3	1
5	10	13,8	1	10	13,4	1
6	5	19,0	1	5	18,6	1
7	Reservado		1	Reservado		8

A escolha desses parâmetros é feita no registrador CONFIG (0x1A), que além da configuração DLPF_CFG, controla a sincronização externa. Em suma o registrador CONFIG permite configurar dois parâmetros do MPU:

- DLPF_CFG[2:0] → filtro digital passa baixo
- EXT_SYNC_SET[2:0] → sincronização externa

A sincronização externa permite que se relacionem as medidas do MPU com um sinal externo. Isso pode ser muito útil, por exemplo, quando se pretende remover a trepidação de câmeras fotográficas ou filmadoras. A informação de sincronização pode ser configurada para ocupar o *bit* menos significativo de um dos 6 eixos. Como o pino FSYNC não está disponível na placa GY-521, esse o recurso de sincronização não será abordado.

G.4.6. Taxa de Amostragem do MPU-6050

Existem 4 recursos do MPU que dependem do que o fabricante chama de Taxa de Amostragem. São eles:

- Registradores de saída dos sensores;
- Saída da FIFO (FIFO será estudada no tópico seguinte);
- Amostragem do Processador de Movimento (DMP) e
- Detecção de Movimento.

Essa Taxa de Amostragem está baseada na frequência de amostragem (f_s) do giroscópio, que foi descrita no item anterior (ver Tabela G.5). Ela é configurada pelo usuário com o uso do parâmetro de 8 *bits* denominado SMPLR_DIV (registrador SMPRT_DIV, 0x19), de acordo com a seguinte equação.

$$Taxa\ de\ Amostragem = \frac{f_s\ do\ Giroscópio}{SMPLR_DIV + 1}$$

Apresentam-se abaixo o cálculo feito com 2 casos:

- DLPF_CFG = 3 e SMPLR_DIV = 4 → Taxa de amostragem = 1.000 / 5 = 200 Hz.
- DLPF_CFG = 0 e SMPLR_DIV = 4 → Taxa de amostragem = 8.000 / 5 = 1.600 Hz.

O Exercício Resolvido G.3 apresenta um exemplo do uso da taxa de amostragem.

G.4.7. Emprego da FIFO do MPU-6050

O termo FIFO vem do inglês *First In, First Out* e é usado para designar uma fila comum, onde o primeiro a chegar é o primeiro a ser atendido. O MPU disponibiliza para o usuário uma FIFO de 1024 *bytes*, ou seja, uma fila com 1024 posições de um *byte*. Lembre-se de que a informação de cada eixo tem 16 *bits*, ou seja, ocupa duas posições na fila. Se considerarmos, por exemplo, o caso do uso dos 6 eixos, a fila será suficiente para guardar

85 leituras ($1024/12 = 85,33$). A atualização da FIFO é feita segunda a Taxa de Amostragem, cujo cálculo foi explicado no item anterior.

Esta FIFO pode ser útil, especialmente quando se tem restrição de consumo de energia. O processador pode ser colocado no modo baixo consumo enquanto o MPU adquire os dados e os armazena na FIFO. Outra possibilidade é a dela ser usada apenas para simplificar o programa de tratamento de dados, que pode trabalhar com o processamento em pacotes.

Com o emprego do registrador FIFO_EN (0x23) o usuário seleciona quais sensores devem ser armazenados na FIFO, como mostrado na Figura G.10. Note que para o giroscópio é possível escolher cada eixo individualmente, enquanto que para o acelerômetro os 3 eixos compõem um único bloco. As posições SLV2, SLV1 e SLV0 são para um sensor externo e não serão aqui abordados.

FIFO_EN (0x23)							
7	6	5	4	3	2	1	0
TEMP_ FIFO_EN	XG_ FIFO_EN	YG_ FIFO_EN	ZG_ FIFO_EN	ACCEL_ FIFO_EN	SLV2	SLV1	SLV0

Figura G.9. Descrição do Registrador FIFO_EN usado para indicar quais leituras deverão ser guardadas na FIFO.

O Contador da FIFO indica a quantidade de *bytes* que foi armazenada e, portanto, a quantidade que pode ser lida. Ele é formado pelos registradores FIFO_COUNT_H (0x72) e FIFO_COUNT_L (0x73). Esses dois registradores somente são atualizados com o valor correto por ocasião da leitura do *byte* mais significativo (FIFO_COUNT_H). Em outras palavras, o FIFO_COUNT_H deve ser lido primeiro. A quantidade de *bytes* é diretamente proporcional à quantidade de amostras disponíveis.

A leitura dos dados armazenados na FIFO é realizada com o registrador FIFO_R_W (0x74). Todas as leituras são feitas neste mesmo endereço, sendo que o controle interno da FIFO se preocupa em entregar os dados na sequência correta. Os dados são recebidos no sentido crescente dos endereços dos registradores. Por exemplo, se o usuário habilitar a FIFO para receber apenas os dados dos 6 eixos (mas não a temperatura), sua leitura trará o conteúdo dos registradores de 0x3B a 0x40 e de 0x43 a 0x48.

O transbordamento da FIFO (*overflow*) é indicado pelo *bit* FIFO_OFLOW_INT do registrador INT_STATUS (0x3A). Se este *bit* estiver em 1, significa que aconteceu o transbordamento da fila. Neste caso, os dados mais antigos são sobrescritos. É possível programar para que o transbordamento provoque uma interrupção.

No caso da FIFO vazia, as leituras retornarão o último dado lido. Isto acontece enquanto não chegar um novo dado. O usuário deve sempre checar o valor do registrador FIFO_COUNT (0x72), antes de iniciar a leitura.

O Exercício Resolvido G.3 apresenta um exemplo de uso da FIFO.

G.4.8. Interrupção com o MPU-6050

A placa GY-521, base para este estudo, disponibiliza o pino INT do MPU, que pode ser conectado a uma entrada de interrupção do Arduino. Para o caso do Arduino Uno, a sugestão é conectá-lo ao pino 2, que é o da interrupção 0 (INT0, PD2). Já para o Arduino Mega, a sugestão é também usar o pino 2, que neste caso corresponde à interrupção 4 (INT4, PE4). Vide a Figura G.6.

O MPU pode provocar interrupção a partir dos seguintes eventos:

- Dado disponível (DATA_RDY) → ocorre toda vez que os registradores dos sensores são atualizados;
- Transbordamento da FIFO (FIFO_OFLOW) → ocorre quando o MPU escreve um dado na FIFO que já está cheia;
- Detecção de movimento (MOT) → quando é ultrapassado o limiar de aceleração (*motion*) especificado pelo usuário e
- Interrupção do I²C master (I2C_MST) → não usado neste estudo.

A habilitação das opções de interrupção é feita com o registrador INT_ENABLE (0x38), que é descrito na Figura G.10. O usuário, ao colocar o *bit* correspondente em 1, indica quais eventos podem gerar interrupção. Uma vez que existem diversas possibilidades de evento, a leitura do registrador INT_STATUS (0x3A) permite ao usuário determinar qual foi o evento causador da interrupção, tal como mostrado na Figura G.11. Todos os *flags* desse registrador de *status* são automaticamente zerados quando lidos.

INT_ENABLE (0x38)							
7	6	5	4	3	2	1	0
-	MOT_EN	-	FIFO_OFLOW_EN	I2C_MST_EN	-	-	DATA_RDY_EN

Figura G.10. Descrição do Registrador INT_ENABLE que permite escolher as interrupções que serão usadas.

INT_STATUS (0x3A)							
7	6	5	4	3	2	1	0
-	MOT_INT	-	FIFO_OFLOW_INT	I2C_MST_INT	-	-	DATA_RDY_INT

Figura G.11. Descrição do Registrador INT_STATUS que indica qual evento provocou a interrupção.

A configuração do comportamento do pino INT é feita com o registrador INT_PIN_CFG (0x37), mostrado na Figura G.12. Os *bits* 3, 2 e 1 desse registrador serão ignorados neste estudo. A Tabela G.6 indica a função de cada *bit*. É recomendado que o *bit* INT_OPEN, no caso do Arduino, seja usado na configuração *push-pull*. O *bit* LATCH_INT_EN indica como se comporta o pulso de interrupção. O mais prático é usar o modo de pulso de 50

μs. No outro modo, é necessário definir no *bit* INT_RD_CLEAR qual ação de leitura vai apagar o pedido de interrupção.

INT_PIN_CFG (0x37)							
7	6	5	4	3	2	1	0
INT_LEVEL	INT_OPEN	LATCH_INT_EN	INT_RD_CLEAR	X	X	X	-

Figura G.12. Descrição do Registrador INT_PIN_CFG que permite configurar o comportamento do pino de interrupção (INT).

Tabela G.6. Descrição dos bits do registrador INT_PIN_CFG (0x37), que configura o pino de interrupção (INT)

Bit	Lógico	Função
INT_LEVEL	0	Pino INT ativo em nível alto
	1	Pino INT ativo em nível baixo
INT_OPEN	0	Configurado com <i>push-pull</i> (<i>pull-up</i> e <i>pull-down</i>)
	1	Configurado com dreno aberto
LATCH_INT_EN	0	Pino INT gera pulso de 50 μs
	1	Pino INT fica em High até interrupção ser apagada
INT_RD_CLEAR	0	Leitura em INT_STATUS apaga <i>bits</i> de <i>status</i>
	1	Qualquer operação de leitura apaga <i>bits</i> de <i>status</i>

O Exercício Resolvido G.4 apresenta um programa que faz as leituras do MPU com a ajuda de interrupções.

G.4.9. Limiar para Detecção de Movimento

Como o nome diz, este recurso permite especificar um limiar para indicar presença de movimento. Esse limiar é, na verdade, um número de 8 *bits* (0 até 255) sem sinal. Se o valor absoluto de um dos acelerômetros ultrapassar este número, é indicado o evento de Detecção de Movimento. O limiar é especificado no registrador MOT_THR (0x1F). É claro que esse limiar deve ser usado de forma coerente com a escala em uso.

Por exemplo, se a escala de +/- 2g estiver em uso e o limiar (MOT_THR) for igual a 164, a Detecção de Movimento acontecerá quando a aceleração ultrapassar 0,01 g. Veja o cálculo abaixo.

$$\text{Limiar (em g)} = 164 \times \frac{2}{32767} = 0,01 \text{ g}$$

G.4.10. Todos os Registradores do MPU-60

O manual do fabricante não apresenta todos os registradores disponíveis no MPU. A Tabela G.7 apresenta a lista dos registradores que foram identificados.

Tabela G.7. Lista completa dos registradores do MPU

Hexa	Registrador	Hexa	Registrador
0x00	XGOWFS_TC	0x3C	ACCEL_XOUT_L
0x01	YGOWFS_TC	0x3D	ACCEL_YOUT_H
0x02	ZGOWFS_TC	0x3E	ACCEL_YOUT_L
0x03	X_FINE_GAIN	0x3F	ACCEL_ZOUT_H
0x04	Y_FINE_GAIN	0x40	ACCEL_ZOUT_L
0x05	Z_FINE_GAIN	0x41	TEMP_OUT_H
0x06	XA_OFFSET_H	0x42	TEMP_OUT_L
0x07	XA_OFFSET_L_TC	0x43	GYRO_XOUT_H
0x08	YA_OFFSET_H	0x44	GYRO_XOUT_L
0x09	YA_OFFSET_L_TC	0x45	GYRO_YOUT_H
0x0A	ZA_OFFSET_H	0x46	GYRO_YOUT_L
0x0B	ZA_OFFSET_L_TC	0x47	GYRO_ZOUT_H
0x0D	SELF_TEST_X	0x48	GYRO_ZOUT_L
0x0E	SELF_TEST_Y	0x49	EXT_SENS_DATA_00
0x0F	SELF_TEST_Z	0x4A	EXT_SENS_DATA_01
0x10	SELF_TEST_A	0x4B	EXT_SENS_DATA_02
0x13	XG_OOWFS_USRH	0x4C	EXT_SENS_DATA_03
0x14	XG_OOWFS_USRL	0x4D	EXT_SENS_DATA_04
0x15	YG_OOWFS_USRH	0x4E	EXT_SENS_DATA_05
0x16	YG_OOWFS_USRL	0x4F	EXT_SENS_DATA_06
0x17	ZG_OOWFS_USRH	0x50	EXT_SENS_DATA_07
0x18	ZG_OOWFS_USRL	0x51	EXT_SENS_DATA_08
0x19	SMPLRT_DIV	0x52	EXT_SENS_DATA_09
0x1A	CONFIG	0x53	EXT_SENS_DATA_10
0x1B	GYRO_CONFIG	0x54	EXT_SENS_DATA_11
0x1C	ACCEL_CONFIG	0x55	EXT_SENS_DATA_12
0x1D	FF_THR	0x56	EXT_SENS_DATA_13

0x1E	FF_DUR
0x1F	MOT_THR
0x20	MOT_DUR
0x21	ZMOT_THR
0x22	ZRMOT_DUR
0x23	FIFO_EN
0x24	I2C_MST_CTRL
0x25	I2C_SLV0_ADDR
0x26	I2C_SLV0_REG
0x27	I2C_SLV0_CTRL
0x28	I2C_SLV1_ADDR
0x29	I2C_SLV1_REG
0x2A	I2C_SLV1_CTRL
0x2B	I2C_SLV2_ADDR
0x2C	I2C_SLV2_REG
0x2D	I2C_SLV2_CTRL
0x2E	I2C_SLV3_ADDR
0x2F	I2C_SLV3_REG
0x30	I2C_SLV3_CTRL
0x31	I2C_SLV4_ADDR
0x32	I2C_SLV4_REG
0x33	I2C_SLV4_DO
0x34	I2C_SLV4_CTRL
0x35	I2C_SLV4_DI
0x36	I2C_MST_STATUS
0x37	INT_PIN_CFG
0x38	INT_ENABLE
0x39	DMP_INT_STATUS
0x3A	INT_STATUS
0x3B	ACCEL_XOUT_H

0x57	EXT_SENS_DATA_14
0x58	EXT_SENS_DATA_15
0x59	EXT_SENS_DATA_16
0x5A	EXT_SENS_DATA_17
0x5B	EXT_SENS_DATA_18
0x5C	EXT_SENS_DATA_19
0x5D	EXT_SENS_DATA_20
0x5E	EXT_SENS_DATA_21
0x5F	EXT_SENS_DATA_22
0x60	EXT_SENS_DATA_23
0x61	MOT_DETECT_STATUS
0x63	I2C_SLV0_DO
0x64	I2C_SLV1_DO
0x65	I2C_SLV2_DO
0x66	I2C_SLV3_DO
0x67	I2C_MST_DELAY_CTRL
0x68	SIGNAL_PATH_RESET
0x69	MOT_DETECT_CTRL
0x6A	USER_CTRL
0x6B	PWR_MGMT_1
0x6C	PWR_MGMT_2
0x6D	DMP_BANK
0x6E	DMP_RW_PNT
0x6F	DMP_REG
0x70	DMP_REG_1
0x71	DMP_REG_2
0x72	FIFO_COUNTH
0x73	FIFO_COUNTL
0x74	FIFO_R_W
0x75	WHO_AM_I_MPU6050

O leitor deve usar esses registradores não documentados por sua própria conta e risco. Para se obter alguma orientação, é recomendado a consulta aos sítios (já citados anteriormente):

- Kris Winer (<https://github.com/kriswiner/MPU-6050>) e
- Jeff Rowberg que disponibiliza uma biblioteca em (<https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050>).

G.5. Exercícios Resolvidos

Aqui, sob a forma de exercícios, iremos apresentar algumas rotinas que são úteis para o emprego do MPU-6050. Esses exemplos tomarão como base o Arduino Mega. Para os demais modelos, será necessária uma pequena adaptação.

ER G.1. Como primeiro exercício, vamos escrever uma rotina que simplesmente faça a leitura dos dados do acelerômetro, do giroscópio e da temperatura do MPU-6050.

Solução:

Os dados do MPU são palavras de 16 *bits* com sinal. Como o barramento I²C (TWI) é orientado a *bytes*, são necessários dois *bytes* (*High* e *Low*) para cada dado. Para o acelerômetro (eixos X, Y e Z) são 6 *bytes*, para o termômetro são 2 *bytes* e para o giroscópio (eixos X, Y e Z) são necessários mais 6 *bytes*. Assim, são 14 *bytes*. O interessante é que esses registradores estão em sequência, a partir do endereço 0x3B (registrador ACCEL_XOUT_H).

O programa ERH1p1a está listado abaixo e faz uso da biblioteca “Wire.h”. Ele desperta o MPU-6050 e em seguida entra em um laço que faz a leitura sequencial dos 14 *bytes*, monta as palavras no formato 16 *bits* com sinal e as imprime no monitor serial.

ERG p1a: Fazer leitura do MPU-6050 com a biblioteca Wire.h

```
// ERGp1a
// Imprimir os dados do MPU no monitor serial

//Carregar a biblioteca I2C (TWI)
#include<Wire.h>

#define MPU 0x68 //Endereço do MPU (AD0 = GND)

//Variaveis para armazenar valores dos sensores
int ax,ay,az,tp,gx,gy,gz;

void setup(){
  Serial.begin(9600);

  // Despertar o MPU
  Wire.begin();
  Wire.beginTransmission(MPU);
```

```

    Wire.write(0x6B);
    Wire.write(0);
    Wire.endTransmission(true);
}

void loop() {
    Wire.beginTransaction(MPU);
    Wire.write(0x3B); //Inciar com ACCEL_XOUT_H
    Wire.endTransmission(false);
    Wire.requestFrom(MPU,14,true); //pedir 14 leituras

    ax=Wire.read()<<8|Wire.read(); //0x3B e 0x3C (ACCEL_XOUT)
    ay=Wire.read()<<8|Wire.read(); //0x3D e 0x3E (ACCEL_YOUT)
    az=Wire.read()<<8|Wire.read(); //0x3F e 0x40 (ACCEL_ZOUT)
    tp=Wire.read()<<8|Wire.read(); //0x41 e 0x42 (TEMP_OUT)
    gx=Wire.read()<<8|Wire.read(); //0x43 e 0x44 (GYRO_XOUT)
    gy=Wire.read()<<8|Wire.read(); //0x45 e 0x46 (GYRO_YOUT)
    gz=Wire.read()<<8|Wire.read(); //0x47 e 0x48 (GYRO_ZOUT)

    // Imprimir os dados do Acelerômetro
    Serial.print("AX = "); Serial.print(ax);
    Serial.print(" | AY = "); Serial.print(ay);
    Serial.print(" | AZ = "); Serial.print(az);

    // Imprimir a Temperatura em Celcius
    Serial.print(" | Tp = "); Serial.print(tp/340.00+36.53);

    // Imprimir os dados do Giroscópio
    Serial.print(" | GX = "); Serial.print(gx);
    Serial.print(" | GY = "); Serial.print(gy);
    Serial.print(" | GZ = "); Serial.println(gz);

    delay(300); //Aguardar 0,3 seg e repetir
}

```

O programa ERH1p1b, que está listado abaixo, é um pouco mais sofisticado e não faz uso de qualquer biblioteca. Para melhor compreensão deste programa, é recomendado um estudo do Capítulo 11, onde se faz a descrição do recurso TWI. O programa inicia com um acesso ao registrador PWR_MGMT_1 (endereço 0x6B), desperta o MPU e programa uma referência de relógio. Em seguida prepara o Temporizador/Contador 1 do AVR para fazer uma leitura do MPU a cada 10 ms e guardar os dados na variável `vetor[14]`. A qualquer momento o programa principal lê os dados dessa variável e os imprime no monitor serial. Para evitar que esse `vetor` seja lido enquanto está sendo atualizado pela interrupção, foi usada a variável `tranca`. Enquanto `tranca = 1`, a interrupção não pode atualizar o `vetor`.

ER Gp1b: Fazer leitura do MPU-6050 sem usar a biblioteca Wire.h, ou seja, controlando diretamente a interface TWI do Arduino

```
//ERGp1b
// Exemplo de acesso ao MPU6050
// Controla diretamente o recurso TWI do AVR, com interrupções
// Apresenta dados do Acelerômetro, Temperatura e Giroscópio

// Endereços do MPU, veja que 0xD0 = 0x68<<1
#define MPU_WR 0xD0 //MPU para escrita
#define MPU_RD 0xD1 //MPU para leitura

// Registradores da MPU
#define PWR_MGMT_1 0x6B //registrador p/ gerenciar potencia
#define ACCEL_XOUT_H 0x3B //registrador XH do acelerômetro

// Códigos de Status do TWI
#define COD_START_OK 8 //Start OK
#define COD_RSTART_OK 0x10 //ReStart OK
#define COD_SLA_WR_ACK 0x18 //EET enviado e ACK recebido
#define COD_TX_DATA_ACK 0x28 //Dado enviado e ACK recebido
#define COD_SLA_RD_ACK 0x40 //EER enviado e ACK recebido
#define COD_RX_DATA_NACK 0x58 //Dado recebido e NACK gerado
#define COD_RX_DATA_ACK 0x50 //Dado recebido e ACK gerado

// Variáveis acessadas pela interrupção
volatile unsigned int estado; //Indicar a etapa da interrupção
volatile unsigned char vet[14]; //Receber os dados do MPU
volatile unsigned char vetor[14]; //Passar dados para P Principal
volatile unsigned char ix; //Indexador do vetor
volatile unsigned char tranca; //Permitir compartilhamento dos dados

// Indexador para identificar o erro
// Para as rotinas TWI que não usam interrupção
int idx = 1;

void setup() {
    Serial.begin(9600);
    // Arduino: Programar frequência do SCL
    TWBR = 72; //SCL = 100 kHz
    TWSR = 0; //TWPS1 = TWPS0 = 0 Pré-escalador

    TWI_start(idx++); //START
    TWI_ER(MPU_WR, idx++); //Endereçar MPU para escrita
    TWI_dado_ER(PWR_MGMT_1, idx++); //Informar acesso ao PWR_MGMT_1 (0x6B)
    TWI_dado_ER(1, idx++); //Selecionar PLL eixo X como referência
    TWI_stop(); //Gerar STOP para finalizar
    delay(100); //Delay para gerar condição de STOP

    //Timer1 usando OCR1A para interromper a cada 10 mseg
```



```

    TCCR1A = 0;
    TCCR1B = (1<<WGM12) | (1<<CS12);
    OCR1A = 623;           //Contagem para 10 mseg
    TIMSK1=(1<<OCIE1A);   //Habilitar interrupção
    tranca=0;             //Sincronização com interrupção
}

void loop() {
    int i,ax, ay, az, tp, gx, gy, gz;

    delay(1000); // pausa de 1 segundo
    tranca=1;    //Bloquear para usar vetor
    ax=(vetor[0]<<8) | vetor[1];
    ay=(vetor[2]<<8) | vetor[3];
    az=(vetor[4]<<8) | vetor[5];
    tp=(vetor[6]<<8) | vetor[7];
    gx=(vetor[8]<<8) | vetor[9];
    gy=(vetor[10]<<8) | vetor[11];
    gz=(vetor[12]<<8) | vetor[13];
    tranca=0;    //Desbloquear

    Serial.print("AX = ");    Serial.print(ax);
    Serial.print(" | AY = "); Serial.print(ay);
    Serial.print(" | AZ = "); Serial.print(az);
    Serial.print(" | Tp = "); Serial.print(tp / 340.00 + 36.53);
    Serial.print(" | GX = "); Serial.print(gx);
    Serial.print(" | GY = "); Serial.print(gy);
    Serial.print(" | GZ = "); Serial.print(gz);
    Serial.print("\n");
}

// Gerar um START no TWI
void TWI_start (int ix) {
    TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN); //Enviar START
    while ( !(TWCR & (1 << TWINT))) ;                //Esperar TWINT = 1
    if ( (TWSR & 0xF8) != COD_START_OK) TWI_erro(1, ix);
}

// Enviar STOP
void TWI_stop (void) {
    TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWSTO); //Enviar STOP
    delay (50); //atraso para que STOP seja percebido
}

// Enviar endereço de Escrita do Escravo (ER) e esperar ACK
// eer = endereço do escravo receptor
void TWI_ER (unsigned char eer, int ix) {
    TWDR = eer; //Endereço de escrita no escravo
    TWCR = (1 << TWINT) | (1 << TWEN); //Enviar endereço
    while ( !(TWCR & (1 << TWINT))); //Esperar TWINT = 1
}

```

```

    if ( (TWSR & 0xF8) != COD_SLA_WR_ACK) TWI_erro(2, ix);
}

//Enviar dado para escravo previamente endereçado
void TWI_dado_ER (unsigned char dado, int ix) {
    TWDR = dado; //dado a ser enviado
    TWCR = (1 << TWINT) | (1 << TWEN); //Enviar dado
    while ( !(TWCR & (1 << TWINT))); //Esperar TWINT = 1
    if ( (TWSR & 0xF8) != COD_TX_DATA_ACK) TWI_erro(4, ix);
}

////////////////////////////////////////
// ISR TWI - Rotina para atender interrupção TWI
ISR(TWI_vect){

    if (estado == 1){
        if ( (TWSR&0xF8)!=COD_START_OK ) TWI_erro(7,estado);
        TWDR=MPU_WR; //Endereço de escrita no MPU
        // Enviar endereço
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    }

    if (estado == 2){
        if ( (TWSR & 0xF8) != COD_SLA_WR_ACK) TWI_erro(7,estado);
        TWDR=ACCEL_XOUT_H; //Endereço do registrador XH do Acelerômetro
        // Enviar dado
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    }

    if (estado == 3){
        if ( (TWSR & 0xF8) != COD_TX_DATA_ACK) TWI_erro(7,estado);
        //Enviar ReSTART
        TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN) | (1<<TWIE);
    }

    if (estado == 4){
        if ( (TWSR & 0xF8) != COD_RSTART_OK) TWI_erro(7,estado);
        TWDR=MPU_RD; //Endereço de leitura
        //Enviar endereço
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    }

    if (estado == 5){
        if ( (TWSR & 0xF8) != COD_SLA_RD_ACK) TWI_erro(7,estado);
        // Iniciar a recepção do dados do MPU
        TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    }

    if (estado>5 && estado<18){
        if ( (TWSR & 0xF8) != COD_RX_DATA_ACK) TWI_erro(7,estado);
    }
}

```

```

    vet[ix++]=TWDR;    //Receber um dado
    // Preparar para receber próximo dado do MPU
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE); //
}

if (estado==18){
    if ( (TWSR & 0xF8) != COD_RX_DATA_ACK) TWI_erro(7,estado);
    vet[ix++]=TWDR;    //Receber penúltimo dado
    // Preparar para receber o último dado e gerar NACK
    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE); //NACK
}

if (estado == 19){
    if ( (TWSR & 0xF8) != COD_RX_DATA_NACK) TWI_erro(7,estado);
    vet[ix++]=TWDR;    //Receber último dado
    // Se não bloqueado, fazer a cópia para o vetor
    if (tranca == 0) for (ix=0;ix<14; ix++)  vetor[ix]=vet[ix];
    TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWSTO); // Enviar STOP
}
estado++;    //Incrementar estado
}

//Imprimir mensagens de erro
void TWI_erro(int cod, int ix) {
    char msg[50];
    switch (cod) {
        case 1:
            sprintf(msg, "(ix=%d) Erro no START: TWSR = %02X\n",ix,TWSR&0xF8);
            Serial.print(msg); break;
        case 2:
            sprintf(msg, "ix=%d) Erro Enderecar ER: TWSR=%02X\n",ix,TWSR&0xF8);
            Serial.print(msg); break;
        case 3:
            sprintf(msg, "ix=%d) Erro Enderecar ET: TWSR=%02X\n",ix,TWSR&0xF8);
            Serial.print(msg); break;
        case 4:
            sprintf(msg, "ix=%d) Erro Envio Dado: TWSR=%02X\n",ix,TWSR&0xF8);
            Serial.print(msg); break;
        case 5:
            sprintf(msg, "ix=%d) Erro Rec. e ACK: TWSR=%02X\n",ix,TWSR&0xF8);
            Serial.print(msg); break;
        case 6:
            sprintf(msg, "ix=%d) Erro Rec. e NACK: TWSR=%02X\n",ix,TWSR&0xF8);
            Serial.print(msg); break;
        case 7:
            sprintf(msg, "estado=%d) Interrupcao: TWSR = %02X\n",ix,TWSR&0xF8);
            Serial.print(msg); break;
    }
}
}

```

```
// ISR Timer 1 - Interrupção OCR1A (a cada 10 mseg)
ISR(TIMER1_COMPA_vect){
    estado=1; //Inicializar estado
    ix=0;      //Zerar indexador do vetor
    // START para iniciar nova leitura do MPU (14 bytes)
    TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN) | (1<<TWIE);
}
```

ER G.2. Escrever uma rotina que realize as etapas listadas abaixo e depois mostre no Monitor Serial, a cada 0,5 seg, os valores do acelerômetro em unidades de g, do giroscópio em graus/s e da temperatura em °C. Use as escalas +/- 2 g e +/- 250 graus/s:

1. Retirar o MPU do modo *sleep* e testar a comunicação;
2. Realizar o auto teste (*self-test*);
3. Realizar a calibração;
4. Configurar o MPU
5. Selecionar a escala +/- 2 g e +/- 250 graus/s;
6. Mostrar os dados dos sensores a cada 0,5 s.

Solução:

O programa apresentado abaixo realiza as etapas solicitadas. Na mesma pasta onde está o este programa, é preciso colocar também a rotina Rot_i2c.ino (item G.4.2), que contém as funções básicas para acessar o barramento I²C.

Atendendo ao pedido, o programa solução retira o MPU do modo *sleep* e logo em seguida testa a comunicação através da interface I²C. Na eventualidade do MPU não responder à comunicação, é recomendado que se desligue e em seguida religue a alimentação. Não é raro acontecer que o MPU, ao ser interrompido, fique travado no meio de uma operação. A seguir são descritas as duas funções que atendem a esses dois pedidos.

- `void MPU_acorda(void)`, retira o MPU do modo *sleep* e programa como fonte de relógio o PLL do eixo X do giroscópio e
- `uint8_t MPU_whoami(void)`, retorna a leitura do registrador WHO_AM_I.

Em seguida, o programa chama a função descrita abaixo, que realiza o auto teste (*self-test*) do MPU, seguindo o que foi descrito no tópico G.4.4. Para ver os resultados do auto teste, basta reativar as linhas que estão comentadas.

- `uint8_t MPU_st(void)` → função que realiza o auto teste e retorna TRUE (1) se teve sucesso ou FALSE (0) se o MPU falhar no teste.

Se o MPU passar no auto teste faz-se em seguida sua calibração, cuja finalidade é determinar o erro intrínseco de cada sensor para que se possa descontá-lo das leituras futuras. A função descrita abaixo está encarregada dessa tarefa. Note que ela retorna dois parâmetros.

- `void MPU_calibra(int16_t *bias, float *valor)`
 - `int16_t bias[6]` → vetor com os valores inteiros do erro, na sequência: acelerômetro X, Y e Z e depois giroscópio X, Y e Z e

- `float valor[6]` → vetor com os erros em g e em graus/s, na sequência: acelerômetro X, Y e Z e depois giroscópio X, Y e Z.

É interessante estudar com um pouco de detalhes essa função `MPU_calibra (*bias, *valor)`. Os registradores são configurados da seguinte forma:

- `PWR_MGMT_1 = 0x80` → faz o *reset* do MPU, note que o *bit* de *reset* é zerado automaticamente;
- `PWR_MGMT_1 = 0x01` → cancela o modo *sleep* e seleciona como relógio o PLL do eixo X do giroscópio;
- `MPU_escalas(GIRO_FS_250, ACCEL_FS_2G)` → seleciona as escalas mais baixas, para oferecer a melhor precisão;
- `CONFIG = 0x01` → seleciona a maior banda possível para os filtros passa baixo e programa o giroscópio para operar em 1 kHz;
- `SMPLRT_DIV = 0x00` → faz taxa de amostragem igual a 1 kHz;
- `INT_ENABLE = 0x01` → habilita interrupção por dado pronto, note que o Arduino não fará uso da interrupção, mas sim da consulta (*polling*) do *bit* `DATA_RDY_INT` do registrador `INT_STATUS` (0x3A);
- Leitura do registrador `INT_STATUS` (0x3A) → para garantir que o *bit* `DATA_RDY_INT` inicie zerado (este *bit* é zerado a cada leitura em 1);

Após toda essa inicialização, a função entra num laço em que acumula (faz o somatório) em variáveis de 32 *bits* os resultados de 256 leituras consecutivas. Assim, para se obter a média, basta dividir por 256 os diversos somatórios. A função constrói essa divisão com 8 deslocamentos (`>>8`) para a direita. O resultado é a média do erro de cada sensor em um formato inteiro (16 *bits*). Depois, esse erro é transformado em “g” e em “gr/s”.

Para ver os resultados da calibração, basta descomentar as linhas que realizam a impressão. Elas estão presentes no final desta função.

Antes de entrar no laço final, o programa ainda faz uso de mais duas funções, descritas abaixo. A função `MPU_inicializar (void)` tem a finalidade de colocar o MPU num estado conhecido.

- `void MPU_inicializar(void)` → retira o MPU do modo *sleep*, seleciona como relógio o PLL do eixo X do giroscópio, define as bandas dos filtros para 44 Hz (acelerômetro) e 42 Hz (giroscópio) e programa a taxa de amostragem para 200 Hz;
- `void MPU_escalas(uint8_t gfs, uint8_t afs)` → seleciona as escalas desejadas pelo usuário.

Após isso tudo, o programa entra em um laço infinito onde, lê os valores do MPU, desconta os erros intrínsecos, calcula a aceleração e a velocidade angular e mostra os resultados no Monitor Serial. Para que o laço seja repetido a cada 0,5 s, é usada a função `delay(500)`. O leitor atento, vai perceber que o laço de repetição demora um pouco mais de 500 ms, mas esse erro foi considerado aceitável.

ER Gp2: Realizar todas as etapas para colocar o MPU-6050 no modo operacional e mostrar as leituras a cada 0,5 segundo

```
// ERGp2
// Colocar no mesmo diretório o arquivo Rot_i2c.ino

// 1. Retirar o MPU do modo sleep e testar comunicação;
// 2. Realizar o auto teste (self-test);
// 3. Realizar a calibração;
// 4. Configurar o MPU e
// 5. Selecionar a escala +/- 2 g e +/- 250 graus/s;
// 6. Mostrar dados a cada 0,5 seg

#include <Wire.h>

#define MPU_ADR 0x68 //Endereço MPU-6050

//Escala para Giroscópio
#define GIRO_FS_250 0 // +/- 250 graus/seg
#define GIRO_FS_500 1 // +/- 500 graus/seg
#define GIRO_FS_1000 2 // +/- 1000 graus/seg
#define GIRO_FS_2000 3 // +/- 2000 graus/seg

//Escala para Acelerômetro
#define ACCEL_FS_2G 0 // +/- 2g
#define ACCEL_FS_4G 1 // +/- 4g
#define ACCEL_FS_8G 2 // +/- 8g
#define ACCEL_FS_16G 3 // +/- 16g

// Registradores do MPU-6050 que foram usados
#define SELF_TEST_X 0x0D
#define SELF_TEST_Y 0x0E
#define SELF_TEST_Z 0x0F
#define SELF_TEST_A 0x10
#define SMPLRT_DIV 0x19
#define CONFIG 0x1A
#define GYRO_CONFIG 0x1B
#define ACCEL_CONFIG 0x1C
#define INT_PIN_CFG 0x37
#define INT_ENABLE 0x38
#define INT_STATUS 0x3A
#define ACCEL_XOUT_H 0x3B
#define PWR_MGMT_1 0x6B
#define WHO_AM_I 0x75

float giro_res, acel_res; //Resolução, depende da escala
int16_t bias_int[6]; //Guardar erro (bias) acel:X,Y,Z e giro:X,Y,Z
float bias_float[6]; //Guardar erro (bias) em g e graus/s

void setup() {
```

```

Serial.begin(9600);
MPU_acorda();    //Acordar MPU

// Testar comunicação com MPU
if (MPU_whoami() == 0x68) Serial.println("MPU Respondendo!");
else
    MPU_erro(1);

// Realizar Self-Test com MPU
if (MPU_st()) Serial.println("MPU passou no Self-Test!");
else
    MPU_erro(2);

// Calibração depende das escalas
// Para esse exemplo foram selecionadas as mais baixas
// MPU_escalas(GIRO_FS_250, ACCEL_FS_2G); //Selecionar Escalas
// Calibrar o MPU
MPU_calibra(bias_int, bias_float);

// Configuração básica do MPU
MPU_inicializar();
MPU_escalas(GIRO_FS_250, ACCEL_FS_2G);
}

void loop() {
    uint8_t aux[14];
    int16_t axi, ayi, azi, gxi, gyi, gzi, tpi; //Valores inteiros
    float ax, ay, az, gx, gy, gz, tp;

    i2c_rd_rep(MPU_ADR, ACCEL_XOUT_H, 14, aux);
    axi = (int16_t)((aux[0] << 8) | aux[1]);    //Montar Acel X
    ayi = (int16_t)((aux[2] << 8) | aux[3]);    //Montar Acel Y
    azi = (int16_t)((aux[4] << 8) | aux[5]);    //Montar Acel Z
    tpi = (int16_t)((aux[6] << 8) | aux[7]);    //Montar Temp
    gxi = (int16_t)((aux[8] << 8) | aux[9]);    //Montar Giro X
    gyi = (int16_t)((aux[10] << 8) | aux[11]); //Montar Giro Y
    gzi = (int16_t)((aux[12] << 8) | aux[13]); //Montar Giro Z

    ax = (float)(axi - bias_int[0]) * acel_res; // Calcular Acel X
    ay = (float)(ayi - bias_int[1]) * acel_res; // Calcular Acel Y
    az = (float)(azi - bias_int[2]) * acel_res; // Calcular Acel Z
    tp = (float) tpi / 340. + 36.53;           // Calcular Temp
    gx = (float)(gxi - bias_int[3]) * giro_res; // Calcular Giro X
    gy = (float)(gyi - bias_int[4]) * giro_res; // Calcular Giro Y
    gz = (float)(gzi - bias_int[5]) * giro_res; // Calcular Giro Z

    Serial.print("Acel g: ");
    Serial.print(ax, 3);    Serial.print(" ");
    Serial.print(ay, 3);    Serial.print(" ");
    Serial.print(az, 3);    Serial.print(" ");

    Serial.print("Giro graus/s: ");

```

```
Serial.print(gx, 3);    Serial.print(" ");
Serial.print(gy, 3);    Serial.print(" ");
Serial.print(gz, 3);    Serial.print(" ");

Serial.print("Temp C: ");
Serial.print(tp, 2);    Serial.print("\n");

delay(500); //Esperar 0,5 seg
}

// Acordar o MPU e programar para usar relógio Giro X
void MPU_acorda(void) {
    i2c_wr(MPU_ADR, PWR_MGMT_1, 1); //Acorda MPU e seleciona relógio
}

// Ler o registrador WHO_AM_I
uint8_t MPU_whoami(void) {
    return i2c_rd(MPU_ADR, WHO_AM_I); //Ler registrador WHO_AM_I
}

// Colocar o MPU num estado conhecido
// Algumas operações são redundantes dependendo de
// quando esta função é chamada
void MPU_inicializar(void) {

    // Despertar MPU, Relógio = PLL do Giro-x
    i2c_wr(MPU_ADR, PWR_MGMT_1, 0x01);
    delay(200); //Esperar PLL estabilizar

    // Taxa = 1 kHz, Banda: Acel=44 Hz e Giro=42 Hz
    i2c_wr(MPU_ADR, CONFIG, 0x03);

    // Taxa de amostragem = taxa/(1+SMPLRT_DIV)
    i2c_wr(MPU_ADR, SMPLRT_DIV, 0x04); //Taxa = 200 Hz
}

// Selecionar escalas para o MPU e calcula resolução
// Atualiza giro_res e acel_res (globais)
void MPU_escalas(uint8_t gfs, uint8_t afs) {
    i2c_wr(MPU_ADR, GYRO_CONFIG, gfs << 3); //FS do Giro
    i2c_wr(MPU_ADR, ACCEL_CONFIG, afs << 3); //FS do Acel

    // Calcular resolucao do Giroscópio
    switch (gfs) {
        case GIRO_FS_250: giro_res = 250.0 / 32768.0; break;
        case GIRO_FS_500: giro_res = 500.0 / 32768.0; break;
        case GIRO_FS_1000: giro_res = 1000.0 / 32768.0; break;
        case GIRO_FS_2000: giro_res = 2000.0 / 32768.0; break;
    }
}
```



```

// Calcular resolução do Acelerômetro
switch (afs) {
    case ACEL_FS_2G:  acel_res = 2.0 / 32768.0; break;
    case ACEL_FS_4G:  acel_res = 4.0 / 32768.0; break;
    case ACEL_FS_8G:  acel_res = 8.0 / 32768.0; break;
    case ACEL_FS_16G: acel_res = 16.0 / 32768.0; break;
}
}

// Realizar Self-Test (ST)
uint8_t MPU_st(void) {
    char msg[100];
    uint8_t aux[14]; //Auxiliar na leitura dos registradores
    int16_t gx1, gy1, gz1, ax1, ay1, az1; //Valores ST desabilitado
    int16_t gx2, gy2, gz2, ax2, ay2, az2; //Valores ST habilitado
    uint8_t gx3, gy3, gz3, ax3, ay3, az3; //Valores reg de Self-Test
    float   gxf, gyf, gzf, axf, ayf, azf; //Factory Trim
    float   gxr, gyr, gxr, axr, ayr, azr; //Alteração em %

    // Desabilitar Self_Test
    i2c_wr(MPU_ADR, ACCEL_CONFIG, ACEL_FS_8G << 3); //Self-test desab
    i2c_wr(MPU_ADR, GYRO_CONFIG, GIRO_FS_250 << 3); //Self-test desab
    delay(250); //Aguardar configuração estabilizar

    // Ler valores com Self-Test desabilitado.
    i2c_rd_rep(MPU_ADR, ACCEL_XOUT_H, 14, aux);
    ax1 = (int16_t)((aux[0] << 8) | aux[1]) ;
    ay1 = (int16_t)((aux[2] << 8) | aux[3]) ;
    az1 = (int16_t)((aux[4] << 8) | aux[5]) ;
    gx1 = (int16_t)((aux[8] << 8) | aux[9]) ;
    gy1 = (int16_t)((aux[10] << 8) | aux[11]) ;
    gz1 = (int16_t)((aux[12] << 8) | aux[13]) ;

    // Habilitar Self_Test
    i2c_wr(MPU_ADR, ACCEL_CONFIG, (0xE0 | ACEL_FS_8G << 3)); //Acel hab.
    i2c_wr(MPU_ADR, GYRO_CONFIG, (0xE0 | GIRO_FS_250 << 3)); //Giro hab.
    delay(250); //Aguardar configuração estabilizar

    // Ler valores com Self-Test fabilitado.
    i2c_rd_rep(MPU_ADR, ACCEL_XOUT_H, 14, aux);
    ax2 = (int16_t)((aux[0] << 8) | aux[1]) ;
    ay2 = (int16_t)((aux[2] << 8) | aux[3]) ;
    az2 = (int16_t)((aux[4] << 8) | aux[5]) ;
    gx2 = (int16_t)((aux[8] << 8) | aux[9]) ;
    gy2 = (int16_t)((aux[10] << 8) | aux[11]) ;
    gz2 = (int16_t)((aux[12] << 8) | aux[13]) ;

    // Leitura dos resultados do self-test
    aux[0] = i2c_rd(MPU_ADR, SELF_TEST_X); //Eixo X: resultado self-test
    aux[1] = i2c_rd(MPU_ADR, SELF_TEST_Y); //Eixo Y: resultado self-test

```

```

aux[2] = i2c_rd(MPU_ADR, SELF_TEST_Z); //Eixo Z: resultado self-test
aux[3] = i2c_rd(MPU_ADR, SELF_TEST_A); //Restante dos resultados

// Extrair dados do registradores de self-test
ax3 = (aux[0] >> 3) | ((aux[3] >> 4) & 3); // XA_TEST
ay3 = (aux[1] >> 3) | ((aux[3] >> 2) & 3); // YA_TEST
az3 = (aux[2] >> 3) | (aux[3] & 3);          // ZA_TEST
gx3 = aux[0] & 0x1F; // XG_TEST
gy3 = aux[1] & 0x1F; // YG_TEST
gz3 = aux[2] & 0x1F; // ZG_TEST

// Calcular os Factory Trim
axf = (4096.0*0.34) * (pow((0.92/0.34), (((float)ax3 - 1.0) / 30.0)));
ayf = (4096.0*0.34) * (pow((0.92/0.34), (((float)ay3 - 1.0) / 30.0)));
azf = (4096.0*0.34) * (pow((0.92/0.34), (((float)az3 - 1.0) / 30.0)));
gxf = ( 25.0 * 131.0) * (pow( 1.046, ((float)gx3 - 1.0) ));
gyf = (-25.0 * 131.0) * (pow( 1.046, ((float)gy3 - 1.0) ));
gzf = ( 25.0 * 131.0) * (pow( 1.046, ((float)gz3 - 1.0) ));

// Se registrador = 0 --> Factory Trim = 0
if (ax3 == 0) axf = 0;
if (ay3 == 0) ayf = 0;
if (az3 == 0) azf = 0;
if (gx3 == 0) gxf = 0;
if (gy3 == 0) gyf = 0;
if (gz3 == 0) gzf = 0;

// Calcular as Percentagens de Alteração
axr = 100.0 * ((float)(ax2 - ax1) - axf) / axf;
ayr = 100.0 * ((float)(ay2 - ay1) - ayf) / ayf;
azr = 100.0 * ((float)(az2 - az1) - azf) / azf;
gxr = 100.0 * ((float)(gx2 - gx1) - gxf) / gxf;
gyr = 100.0 * ((float)(gy2 - gy1) - gyf) / gyf;
gxr = 100.0 * ((float)(gz2 - gz1) - gzf) / gzf;

// Remova os comentários para imprimir os resultados
/* Serial.println("Self Test:  OFF      ON      TEST      FT      %");
sprintf(msg,"Acel X   :  %+06d  %+06d  %+04d   ",ax1, ax2, ax3);
Serial.print(msg); Serial.print(axf,2); Serial.print(" ");
Serial.println(axr,2);
sprintf(msg,"Acel Y   :  %+06d  %+06d  %+04d   ",ay1, ay2, ay3);
Serial.print(msg); Serial.print(ayf,2); Serial.print(" ");
Serial.println(ayr,2);
sprintf(msg,"Acel Z   :  %+06d  %+06d  %+04d   ",az1, az2, az3);
Serial.print(msg); Serial.print(azf,2); Serial.print(" ");
Serial.println(azr,2);
sprintf(msg,"Giro X   :  %+06d  %+06d  %+04d   ",gx1, gx2, gx3);
Serial.print(msg); Serial.print(gxf,2); Serial.print(" ");
Serial.println(gxr,2);
sprintf(msg,"Giro Y   :  %+06d  %+06d  %+04d   ",gy1, gy2, gy3);

```

```

Serial.print(msg); Serial.print(gyf,2); Serial.print(" ");
Serial.println(gyr,2);
sprintf(msg,"Giro Z : %+06d %+06d %+04d ",gz1, gz2, gz3);
Serial.print(msg); Serial.print(gzf,2); Serial.print(" ");
Serial.println(gzr,2); */

// Tolerância de +/- 14%
if (axr<14 && ayr<14 && azr<14 && gxr<14 && gyr<14 && gxr<14)
    return 1; //Passou no Self-Test
else return 0; //Falhou no Self-Test
}

// Calibar (bias) o MPU
// Calcular o erro em 256 leituras consecutivas
void MPU_calibra(int16_t *bias, float *valor) {
    char msg[100];
    uint16_t j;
    uint8_t data[14]; //Receber dados do acel e giro
    int32_t abx,aby,abz,gbx,gby,gbz; //Somatório erro acel e giro

    // Rresetar MPU, o Reset se apaga sozinho
    // Reset ==> registradores=0, exceto PWR_MGMT=0x40
    // Seleciona automaticamente esacalas 2g e 250gr/s
    i2c_wr(MPU_ADR, PWR_MGMT_1, 0x80);
    delay(100); //Esperar terminar o Reset

    // Sair do modo sleep e selecionar Relógio = PLL do Giro-x
    i2c_wr(MPU_ADR, PWR_MGMT_1, 0x01);
    delay(200); //Esperar estabilizar

    // Selecionar a escala desejada para a calibração
    // Função atualiza as variáveis acel_res e giro_res
    MPU_escalas(GIRO_FS_250, ACEL_FS_2G);

    // Preparar MPU para cálculo do erro do acelerômetro e do giroscópio
    i2c_wr(MPU_ADR, CONFIG, 0x01); //BW_Acel=184Hz e BW_Giro=188 Hz
    i2c_wr(MPU_ADR, SMPLRT_DIV, 0x00); //Taxa de amostragem em 1 kHz

    // Habilitar bit DATA_RDY_INT para indicar dado pronto
    i2c_wr(MPU_ADR, INT_ENABLE, 0x01); //Será usado polling
    i2c_rd(MPU_ADR, INT_STATUS); //Leitura para apagar o bit

    // Laço principal: somatório de 256 leituras
    abx=aby=abz=gbx=gby=gbz=0;
    for (j=0; j<256; j++){
        while ( (i2c_rd(MPU_ADR,INT_STATUS)&1) == 0) ;
        i2c_rd_rep(MPU_ADR, ACCEL_XOUT_H, 14, data);
        abx += (int32_t) (((int16_t)data[0] << 8) | data[1] );
        aby += (int32_t) (((int16_t)data[2] << 8) | data[3] );
        abz += (int32_t) (((int16_t)data[4] << 8) | data[5] );
    }
}

```

```

    gbx += (int32_t) (((int16_t)data[8] << 8) | data[9] );
    gby += (int32_t) (((int16_t)data[10] << 8) | data[11] );
    gbz += (int32_t) (((int16_t)data[12] << 8) | data[13] );
}

// Calcular média (>>8 corresponde a dividir por 256)
// e transferir para a variável de saída
bias[0] = (int16_t) (abx>>8); //acel X
bias[1] = (int16_t) (aby>>8); //acel Y
bias[2] = (int16_t) (abz>>8); //acel Z
bias[3] = (int16_t) (gbx>>8); //giro X
bias[4] = (int16_t) (gby>>8); //giro Y
bias[5] = (int16_t) (gbz>>8); //giro Z

// Calcular o erro (bias) em valores de g e graus/s
valor[0] = bias[0] * acel_res; //acel X
valor[1] = bias[1] * acel_res; //acel Y
valor[2] = bias[2] * acel_res; //acel Z
valor[3] = bias[3] * giro_res; //giro X
valor[4] = bias[4] * giro_res; //giro Y
valor[5] = bias[5] * giro_res; //giro Z

// Remova os comentário para imprimir os valores de offset
/*Serial.println("Resultados da Calibracao:");
sprintf(msg,"Acel X: %+06d --> ",bias[0]); Serial.print(msg);
Serial.print(valor[0],2); Serial.println(" g");
sprintf(msg,"Acel Y: %+06d --> ",bias[1]); Serial.print(msg);
Serial.print(valor[1],2); Serial.println(" g");
sprintf(msg,"Acel Z: %+06d --> ",bias[2]); Serial.print(msg);
Serial.print(valor[2],2); Serial.println(" g");
sprintf(msg,"Giro X: %+06d --> ",bias[3]); Serial.print(msg);
Serial.print(valor[3],2); Serial.println(" graus/s");
sprintf(msg,"Giro Y: %+06d --> ",bias[4]); Serial.print(msg);
Serial.print(valor[4],2); Serial.println(" graus/s");
sprintf(msg,"Giro Z: %+06d --> ",bias[5]); Serial.print(msg);
Serial.print(valor[5],2); Serial.println(" graus/s"); */
}

// Imprimir mensagens de erro travar execução
void MPU_erro(int erro) {
    switch (erro) {
        case 1: Serial.println("MPU nao responde ao I2C!"); break;
        case 2: Serial.println("MPU falhou no Self-Test!"); break;
    }
    // Travar execução
    for (;;) ;
}

```

ER G.3. Empregar a FIFO para receber dados do MPU. Programar a frequência de amostragem para 100 Hz e usar as escalas de +/- 2 g e +/- 250 graus/s.

Solução:

O programa solução (ERGp3.ino) faz a inicialização usual do MPU, programa as escalas de acordo com o solicitado e faz o registrador SMPLRT_DIV = 9 para que a taxa de amostragem seja de 100 Hz. No registrador FIFO_EN (0x23) é indicado para se colocar na FIFO os valores dos 6 eixos, mas não o da temperatura. Assim, a cada vez é colocado na FIFO um pacote de 12 *bytes* (6 valores, cada um com 2 *bytes*). A Figura G.13 indica a configuração do registrador que controla a FIFO.

FIFO_EN (0x23)							
7	6	5	4	3	2	1	0
TEMP_ FIFO_EN	XG_ FIFO_EN	YG_ FIFO_EN	ZG_ FIFO_EN	ACCEL_ FIFO_EN	SLV2	SLV1	SLV0
0	1	1	1	1	0	0	0

Figura G.13. Configuração do registrador para que as 3 leituras do giroscópio e as 3 leituras do acelerômetro sejam armazenadas na FIFO.

É claro que devemos ler a FIFO antes que ela fique completamente cheia e transborde. Por isso, vamos considerar a relação entre o tamanho da fila e a taxa de amostragem. A pergunta que se faz é: de quanto em quanto tempo devemos ler a fila para evitar o transbordamento? Estamos recebendo leituras com os valores dos 6 eixos, sendo que cada valor tem 2 *bytes*. Logo, a cada período de amostragem, é escrito um pacote de 12 *bytes* na FIFO. Como a FIFO tem capacidade de 1 KB, sua capacidade é de 85,33 pacotes (1024/12). Foi programada a frequência de amostragem de 100 Hz, então o período é de 10 ms. Assim, temos de esvaziar a FIFO, no máximo, a cada 850 ms.

Ao examinar o programa abaixo o leitor vai notar no laço principal um `delay(250)` e não um `delay(850)`, como seria o esperado. Foi usado o valor de 250 ms porque a porção que faz a impressão na porta serial consome tempo. O programa indica a quantidade de pacotes que estavam disponíveis na FIFO no momento da leitura. O leitor pode tentar variar o parâmetro da função `delay()` de forma a tentar chegar o mais próximo possível de 85.

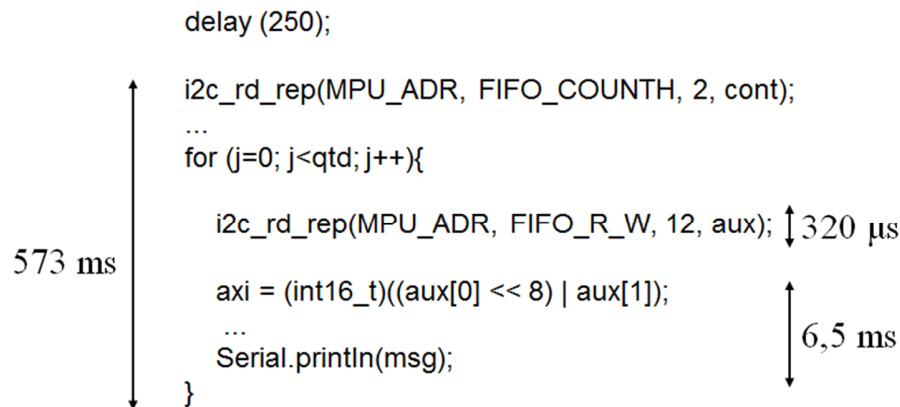


Figura G.14. Esquemático para indicar o tempo gasto pelo laço principal do programa ERGp3.ino. Foram deixadas apenas algumas instruções para o leitor localizar as posições principais. As medições foram realizadas com o auxílio de um osciloscópio.

Antes de apresentar a listagem do programa, é importante comentar que a intenção foi apenas a de exemplificar o uso da FIFO. Por isso, não foi feito o auto teste e nem a calibração. Além disso, os dados são impressos sem qualquer interpretação.

ER Gp3: Exemplificar o uso da FIFO

```

// ERGp3
// Usar a FIFO para acessar o MPU
// Colocar no mesmo diretório o arquivo Rot_i2c.ino

#include <Wire.h>

#define TRUE 1
#define FALSE 0

#define MPU_ADR 0x68 //Endereço MPU-6050

//Escala para Giroscópio
#define GIRO_FS_250 0 // +/- 250 graus/seg
#define GIRO_FS_500 1 // +/- 500 graus/seg
#define GIRO_FS_1000 2 // +/- 1000 graus/seg
#define GIRO_FS_2000 3 // +/- 2000 graus/seg

//Escala para Acelerômetro
#define ACEL_FS_2G 0 // +/- 2g
#define ACEL_FS_4G 1 // +/- 4g
#define ACEL_FS_8G 2 // +/- 8g
#define ACEL_FS_16G 3 // +/- 16g

// Registradores do MPU-6050 que foram usados

```

```
#define SMPLRT_DIV      0x19
#define CONFIG          0x1A
#define GYRO_CONFIG     0x1B
#define ACCEL_CONFIG    0x1C
#define FIFO_EN         0x23
#define INT_PIN_CFG     0x37
#define USER_CTRL      0x6A
#define PWR_MGMT_1      0x6B
#define FIFO_COUNTH     0x72
#define FIFO_COUNTL     0x73
#define FIFO_R_W        0x74
#define WHO_AM_I        0x75

float giro_res, acel_res; //Resolução, depende da escala

void setup() {
    Serial.begin(115200);

    MPU_acorda();    //Acordar MPU e selecionar CLK=Giro-X

    // Testar comunicação com MPU
    if (MPU_whoami() == 0x68) Serial.println("MPU Respondendo!");
    else {Serial.print("MPU nao responde!");  for (;;) ; }

    MPU_escalas(GIRO_FS_250, ACCEL_FS_2G);

    // Taxa = 1 kHz, Banda: Accl=44 Hz e Giro=42 Hz
    i2c_wr(MPU_ADR, CONFIG, 0x03);

    // Taxa de amostragem = taxa/(1+SMPLRT_DIV)
    i2c_wr(MPU_ADR, SMPLRT_DIV, 9);    //Taxa = 100 Hz

    // Inicializar FIFO
    i2c_wr(MPU_ADR, USER_CTRL, 0x40); //RESET FIFO
    delay(100);                        //Aguardar Reset FIFO
    i2c_wr(MPU_ADR, USER_CTRL, 0x40); //Habilitar FIFO
    i2c_wr(MPU_ADR, FIFO_EN, 0x78);   //FIFO = Accl e giro
}

void loop() {
    uint8_t aux[12], cont[2];
    char msg[100];
    uint16_t qtd, j;
    int16_t axi, ayi, azi, gxi, gyi, gzi; //Valores inteiros

    delay (250);

    i2c_rd_rep(MPU_ADR, FIFO_COUNTH, 2, cont); //Ler contador da FIFO
    qtd = ((uint16_t)cont[0] << 8) | cont[1];  //Montar total de dados
    qtd = qtd/12;                             //6 valores X 2 bytes
}
```

```

Serial.print("Ler da FIFO:"); Serial.println(qtd);

for (j=0; j<qtd; j++){
    i2c_rd_rep(MPU_ADR, FIFO_R_W, 12, aux);    //ler 12 dados
    axi = (int16_t)((aux[0] << 8) | aux[1]);    //Montar Acel X
    ayi = (int16_t)((aux[2] << 8) | aux[3]);    //Montar Acel Y
    azi = (int16_t)((aux[4] << 8) | aux[5]);    //Montar Acel Z
    gxi = (int16_t)((aux[6] << 8) | aux[7]);    //Montar Giro X
    gyi = (int16_t)((aux[8] << 8) | aux[9]);    //Montar Giro Y
    gzi = (int16_t)((aux[10] << 8) | aux[11]); //Montar Giro Z
    sprintf(msg,"qtd=%06u>  Acel  g:  %+06d    %+06d    %+06d    ",qtd-
j,axi,ayi,azi);
    Serial.print(msg);
    sprintf(msg,"Giro g/s:  %+06d  %+06d  %+06d  ",gxi,gyi,gzi);
    Serial.println(msg);
}
}

// Acordar o MPU e programar para usar relógio Giro X
void MPU_acorda(void) {
    i2c_wr(MPU_ADR, PWR_MGMT_1, 1); //Acorda MPU e seleciona relógio
}

// Ler o registrador WHO_AM_I
uint8_t MPU_whoami(void) {
    return i2c_rd(MPU_ADR, WHO_AM_I); //Ler registrador WHO_AM_I
}

// Selecionar escalas para o MPU e calcula resolução
// Atualiza giro_res e acel_res (globais)
void MPU_escalas(uint8_t gfs, uint8_t afs) {
    i2c_wr(MPU_ADR, GYRO_CONFIG, gfs << 3); //FS do Giro
    i2c_wr(MPU_ADR, ACCEL_CONFIG, afs << 3); //FS do Acel

    // Calcular resolução do Giroscópio
    switch (gfs) {
        case GIRO_FS_250: giro_res = 250.0 / 32768.0; break;
        case GIRO_FS_500: giro_res = 500.0 / 32768.0; break;
        case GIRO_FS_1000: giro_res = 1000.0 / 32768.0; break;
        case GIRO_FS_2000: giro_res = 2000.0 / 32768.0; break;
    }

    // Calcular resolução do Acelerômetro
    switch (afs) {
        case ACEL_FS_2G: acel_res = 2.0 / 32768.0; break;
        case ACEL_FS_4G: acel_res = 4.0 / 32768.0; break;
        case ACEL_FS_8G: acel_res = 8.0 / 32768.0; break;
        case ACEL_FS_16G: acel_res = 16.0 / 32768.0; break;
    }
}

```



```
}

```

ER G.4. Empregar a interrupção de dado pronto para receber dados do MPU-6050 na taxa de 200 Hz.

Solução:

O programa apresentado, de nome ER Gp4, tomou como exemplo o Arduino Mega, por isso, fez uso da interrupção INT4, que está disponível no pino 2 (PE4), como mostrado na Figura G.6. Ele pode ser facilmente modificado para as outras versões do Arduino. Para tornar o exemplo o mais simples possível, foram usadas as funções do Arduino para configurar o pino 2 e a interrupção, como descrito logo abaixo. Estão comentadas as linhas que fazem a configuração da interrupção diretamente nos registradores do AVR.

- `pinMode(2, INPUT_PULLUP)` → define pino 2 como entrada com *pull-up*;
- `attachInterrupt(4, ISR_MPU, FALLING)` → habilita INT4 para trabalhar por flanco de descida e especifica que a rotina de interrupção é a função `ISR_MPU`;

As funções para auto teste (*self-test*) e calibragem foram omitidas para simplificar o programa. É claro que elas devem ser incluídas numa versão definitiva.

Os registradores do MPU a serem usados neste exercício estão listados abaixo. Note que rotina `void MPU_acorda(void)`, faz o registrador `PWR_MGMT_1 = 1`. Após essa configuração, irá surgir no pino INT pulsos de 50 µs espaçados de 5 ms (frequência de amostragem de 200 Hz).

- `PWR_MGMT_1 = 1` → *sleep* = 0 e indica para usar relógio do giroscópio (eixo X);
- `CONFIG = 3` → Taxa de 1 kHz e bandas de 44 Hz e 42 Hz;
- `SMPLRT_DIV = 4` → amostragem de 200 Hz ($200 = 1 \text{ kHz} / (4+1)$);
- `INT_PIN_CFG = 0x80` → pulso de 50 µs em nível baixo com *push-pull* e
- `INT_ENABLE = 1` → interromper quando dado estiver pronto.

Foi usada a variável `flag` para fazer a sincronização entre o programa principal e a interrupção. O programa principal fica preso em um laço enquanto `flag = FALSE` (0). A rotina que trata a interrupção 4, `ISR(INT4_vect)`, quando executada faz `flag = TRUE` (1). O programa principal ao perceber isso, faz `flag = FALSE`, lê os dados do MPU e os imprime. Em seguida volta a ficar preso em um laço enquanto `flag = FALSE`.

Resumindo, cada vez que `flag = TRUE`, é porque existe um conjunto de dados a ser lido. A variável `cont` é incrementada a cada interrupção. O leitor vai notar que a rotina de impressão é muito lenta, pois a cada impressão essa variável `cont` salta diversos valores. São as interrupções que não foram aproveitadas enquanto se fazia a impressão dos dados.

ER Gp4: Emprego do MPU com recepção de dados via interrupção

```
// ERGp4
// Usar a interrupção para receber dados do MPU
// Colocar no mesmo diretório o arquivo Rot_i2c.ino

```

```

#include <Wire.h>

#define TRUE 1
#define FALSE 0

#define MPU_ADR 0x68 //Endereço MPU-6050

//Escala para Giroscópio
#define GIRO_FS_250 0 // +/- 250 graus/seg
#define GIRO_FS_500 1 // +/- 500 graus/seg
#define GIRO_FS_1000 2 // +/- 1000 graus/seg
#define GIRO_FS_2000 3 // +/- 2000 graus/seg

//Escala para Acelerômetro
#define ACCEL_FS_2G 0 // +/- 2g
#define ACCEL_FS_4G 1 // +/- 4g
#define ACCEL_FS_8G 2 // +/- 8g
#define ACCEL_FS_16G 3 // +/- 16g

// Registradores do MPU-6050 que foram usados
#define SMPLRT_DIV 0x19
#define CONFIG 0x1A
#define GYRO_CONFIG 0x1B
#define ACCEL_CONFIG 0x1C
#define INT_PIN_CFG 0x37
#define INT_ENABLE 0x38
#define ACCEL_XOUT_H 0x3B
#define PWR_MGMT_1 0x6B
#define WHO_AM_I 0x75

volatile uint8_t flag; //Indicar que a interrupção aconteceu
volatile uint16_t cont;
float giro_res, accel_res; //Resolução, depende da escala

void setup() {
    Serial.begin(9600);

    // Preparar Pino 2 (PE4) como entrada com pull-up
    //DDRE = DDRE & ~(1 << DDE4); //DDD4=0, entrada
    //PORTE = PORTE | (1 << PE4); //Pull-up ligado (PORTE4=1)
    pinMode(2, INPUT_PULLUP);

    // Preparar interrupção INT4 por flanco de descida
    EICRB = (EICRB | (1<<ISC41)) & ~(1 << ISC40); //INT4 = flanco descida
    EIMSK = EIMSK | (1 << INT4); //INT4 habilitada

    MPU_acorda(); //Acordar MPU e selecionar CLK=Giro-X

    // Testar comunicação com MPU

```

```

    if (MPU_whoami() == 0x68) Serial.println("MPU Respondendo!");
    else {Serial.print("MPU nao responde!");  for (;;) ; }

    MPU_escalas(GIRO_FS_250, ACEL_FS_2G);

    // Taxa = 1 kHz, Banda: Acel=44 Hz e Giro=42 Hz
    i2c_wr(MPU_ADR, CONFIG, 0x03);

    // Taxa de amostragem = taxa/(1+SMPLRT_DIV)
    i2c_wr(MPU_ADR, SMPLRT_DIV, 0x04); //Taxa = 200 Hz

    //Interrupção ativa em baixo, push-pull, pulso de 50 µseg
    i2c_wr(MPU_ADR, INT_PIN_CFG, 0x80);
    i2c_wr(MPU_ADR, INT_ENABLE, 0x01); //Hab interrup dado pronto
    flag=FALSE;
}

void loop() {
    uint8_t aux[14];
    char msg[100];
    int16_t axi, ayi, azi, gxi, gyi, gzi, tpi; //Valores inteiros

    while (flag == FALSE) ;
    flag=FALSE;

    i2c_rd_rep(MPU_ADR, ACCEL_XOUT_H, 14, aux);
    axi = (int16_t)((aux[0] << 8) | aux[1]); //Montar Acel X
    ayi = (int16_t)((aux[2] << 8) | aux[3]); //Montar Acel Y
    azi = (int16_t)((aux[4] << 8) | aux[5]); //Montar Acel Z
    tpi = (int16_t)((aux[6] << 8) | aux[7]); //Montar Temp
    gxi = (int16_t)((aux[8] << 8) | aux[9]); //Montar Giro X
    gyi = (int16_t)((aux[10] << 8) | aux[11]); //Montar Giro Y
    gzi = (int16_t)((aux[12] << 8) | aux[13]); //Montar Giro Z

    sprintf(msg,"cont=%06u>      Acel      g:      %+06d      %+06d      %+06d",
    cont,axi,ayi,azi);
    Serial.print(msg);
    sprintf(msg,"Giro g/s: %+06d  %+06d  %+06d  ",gxi,gyi,gzi);
    Serial.println(msg);
}

// Acordar o MPU e programar para usar relógio Giro X
void MPU_acorda(void) {
    i2c_wr(MPU_ADR, PWR_MGMT_1, 1); //Acorda MPU e seleciona relógio
}

// Ler o registrador WHO_AM_I
uint8_t MPU_whoami(void) {
    return i2c_rd(MPU_ADR, WHO_AM_I); //Ler registrador WHO_AM_I
}

```

```
// Selecionar escalas para o MPU e calcula resolução
// Atualiza giro_res e acel_res (globais)
void MPU_escalas(uint8_t gfs, uint8_t afs) {
    i2c_wr(MPU_ADR, GYRO_CONFIG, gfs << 3); //FS do Giro
    i2c_wr(MPU_ADR, ACCEL_CONFIG, afs << 3); //FS do Acel

    // Calcular resolução do Giroscópio
    switch (gfs) {
        case GIRO_FS_250: giro_res = 250.0 / 32768.0; break;
        case GIRO_FS_500: giro_res = 500.0 / 32768.0; break;
        case GIRO_FS_1000: giro_res = 1000.0 / 32768.0; break;
        case GIRO_FS_2000: giro_res = 2000.0 / 32768.0; break;
    }

    // Calcular resolução do Acelerômetro
    switch (afs) {
        case ACEL_FS_2G: acel_res = 2.0 / 32768.0; break;
        case ACEL_FS_4G: acel_res = 4.0 / 32768.0; break;
        case ACEL_FS_8G: acel_res = 8.0 / 32768.0; break;
        case ACEL_FS_16G: acel_res = 16.0 / 32768.0; break;
    }
}

// ISR para a interrupção INT4
ISR(INT4_vect) {
    flag=TRUE;
    cont++;
}
```