Denilson Figueiredo de Sá

Dispositivo apontador com interface USB usando magnetômetro

Rio de Janeiro - RJ, Brasil

16 de novembro de 2011

Denilson Figueiredo de Sá

Dispositivo apontador com interface USB usando magnetômetro

Orientador:
Nelson Quilula Vasconcelos

Departamento de Ciência da Computação
Instituto de Matemática
Universidade Federal do Rio de Janeiro

Rio de Janeiro - RJ, Brasil

16 de novembro de 2011

Dispositivo apontador com interface USB usando magnetômetro

Denilson Figueiredo de Sá

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Apresentado por:

_____

Denilson Figueiredo de Sá

Aprovado por:

_____

Prof. Nelson Quilula Vasconcelos
Orientador

_____

Prof. Adriano Joaquim de Oliveira Cruz

_____

Prof[a]. Silvana Rossetto

Rio de Janeiro - RJ, Brasil

16 de novembro de 2011

Agradecimentos

Agradeço aos professores do DCC/UFRJ, em especial aos professores Nelson Quilula Vasconcelos, Adriano Joaquim de Oliveira Cruz, e Silvana Rossetto por participarem deste projeto, e aos professores João Carlos Pereira da Silva, Márcia Rosana Cerioli e Monique Moura Carmona pelo apoio durante esses anos de faculdade.

Agradeço aos amigos de faculdade, em especial Alexandre Araújo Moreira, Luana Pinto Araújo e Priscila Neves Bilangieri por toda a paciência e apoio nos momentos em que mais precisei de ajuda.

Agradeço também ao amigo de faculdade Bruno Bottino Ferreira pela paciência durante as semanas finais deste projeto.

Agradeço aos amigos de trabalho, em especial Claudio Sá de Abreu e Marcelo Salhab Brogliato pelo apoio e pelo crescimento pessoal e profissional que me proporcionaram.

Resumo

Este projeto tem como objetivo implementar um dispositivo apontador compatível com um *mouse* USB utilizando um microcontrolador AVR ATmega8 de 8 bits e um magnetômetro. O dispositivo fruto deste projeto permite o usuário controlar o ponteiro do mouse movendo um sensor no ar, simplesmente apontando-o para a posição desejada na tela. É uma forma bastante intuitiva de controlar o ponteiro em situações onde um *mouse* não é adequado. Pode também ser usado para fins de acessibilidade, controlando o ponteiro através de movimentos da cabeça ou de qualquer outra parte do corpo.

Abstract

This project implements a USB HID absolute pointing device using an ATmega8 AVR 8-bit microcontroller and a magnetometer. This device allows the user to control the mouse pointer by just moving a sensor in the air, pointing it to the desired screen position. It is a very intuitive way to control the pointer whenever it is not appropriate to use a mouse. It can also be used for accessibility, controlling the pointer by head movements, or movements of any other part of the body.

Lista de Figuras

Lista de Siglas

ADC Analog-to-Digital Converter 8

API Application Programming Interface 27

DIP Dual In-line Package 7

EEPROM Electrically Erasable Programmable Read-Only Memory 7

HID Human Interface Device 12

I^2C Inter-Integrated Circuit 8

ISP In-System Programmer 16

LCC Leaded Chip Carrier 8

LED Light-Emitting Diode 16

LKML Linux Kernel Mailing List 33

NRZI Non-Return-to-Zero Inverted 9

PCB Printed Circuit Board 8

PDIP Plastic Dual In-line Package 7

QFN Quad-Flat No-leads Package 7

QFP Quad Flat Package 7

SCL Serial Clock 13

SDA Serial Data 13

SRAM Static Random-Access Memory 7

TWI Two-Wire Interface 8

USB-IF USB Implementers' Forum 28

USB Universal Serial Bus 9

Sumário

1 Introdução

"*The world is moving so fast these days that the man who says it can't be done is generally interrupted by someone doing it.*"

Elbert Green Hubbard

Neste capítulo são apresentados a motivação e os objetivos deste projeto, uma lista de trabalhos relacionados e a estrutura da monografia.

1.1 Motivação e objetivos

Este projeto apresenta uma implementação de um dispositivo USB do tipo *absolute pointing device* que pode ser utilizado para controlar o ponteiro do *mouse* na tela do computador seguindo os movimentos de um sensor. Dentre outras aplicações, este dispositivo pode ser muito conveniente em apresentações que usem um computador ligado a um projetor.

O sensor empregado neste projeto é um magnetômetro, também conhecido como bússola digital, o qual permite medir a direção e a intensidade do campo magnético. Desta forma, é possível saber a direção (em relação à Terra) para onde o sensor está sendo apontado. A partir dessa informação é calculada a posição desejada para o ponteiro do *mouse*.

Todos os cálculos são realizados por um microcontrolador ATmega8, no qual também foi implementado o protocolo USB HID. Assim, o dispositivo fruto deste projeto pode ser usado em qualquer computador sem a necessidade da instalação de *drivers* específicos no sistema operacional.

A ideia para este projeto surgiu da apresentação "Google I/O 2011: The Secrets of Google Pac-Man: A Game Show", na qual o palestrante Marcin Wichary utiliza um *iPod touch*[1] preso ao seu pulso para controlar um objeto no telão através dos movimentos de seu braço [1]. O software que rodava no *iPod touch* durante a palestra era uma página escrita em HTML5 e JavaScript, a qual se comunicava com um servidor NodeJS através de uma conexão Wi-Fi. Conforme ele movia seu braço, os dados do acelerômetro eram enviados para o servidor, que por sua vez os repassava para o computador que estava ligado ao projetor.

Um *iPod touch* é um equipamento bastante poderoso, mas também é relativamente grande e caro[2]. Não é muito prático deixar um dispositivo desse tamanho preso ao braço ou a qualquer outra parte do corpo. Além disso, não é possível controlar um computador diretamente através dele, é necessário instalar tanto no *iPod* como no computador algum *software* específico para essa funcionalidade.

Diante dessas observações este projeto foi iniciado, objetivando ser mais barato, menor e mais simples de usar.

Uma vez concluído e posteriormente refinado, este projeto pode ser aplicado para diversos fins. Estas são algumas das aplicações possíveis:

[1] *iPod touch* é um reprodutor multimídia portátil bastante similar ao telefone celular *iPhone*, porém sem todos os recursos deste. O *iPod touch* possui um acelerômetro e um giroscópio embutidos, mas não possui um magnetômetro.

[2] R$ 729,00 no site do fabricante [2].

- Pode ser utilizado durante apresentações, permitindo ao palestrante apontar regiões do telão através do gesto natural de se apontar com a mão.
- Pode ser utilizado em situações onde um *mouse* não é viável, situações onde não há um apoio para um *mouse* convencional.
- Pode ser utilizado em exposições interativas, entretenimento e jogos.
- Pode ser utilizado por pessoas que não podem ou não conseguem movimentar um *mouse*. O pequeno sensor pode ser preso à cabeça ou qualquer outra parte do corpo.

Todo o material desenvolvido neste projeto está também disponível *online* em repositórios Mercurial e Git:
`https://bitbucket.org/denilsonsa/atmega8-magnetometer-usb-mouse`
`https://github.com/denilsonsa/atmega8-magnetometer-usb-mouse`

1.2 Trabalhos relacionados

1.2.1 Wiimote

Wiimote é o nome dado aos controles do vídeo-game Wii, fabricado pela Nintendo. É um controle sem fio com acelerômetro embutido, capaz de detectar movimentos, e também funciona como um apontador dentro da interface do vídeo-game, bastando apontá-lo para a tela. Para isso, é necessário colocar acima ou abaixo da tela uma *sensor bar*, que nada mais é do que um conjunto de LEDs infravermelhos [3].

Cada Wiimote possui uma câmera infravermelha na sua região frontal. A posição do ponteiro é calculada através da posição dos LEDs, capturada pela câmera do controle.

O funcionamento esperado do dispositivo implementado neste projeto – controlar o ponteiro na tela simplesmente apontando um dispositivo de *hardware* para a posição desejada – é muito similar ao Wiimote, embora com tecnologias distintas.

1.2.2 SmartNav, TrackIR, HeadMouse

A empresa NaturalPoint criou um produto chamado SmartNav [4] para controlar o ponteiro do *mouse* através de gestos. O produto é composto de uma câmera colocada acima do monitor e um software proprietário para Windows ou Mac OS X. A câmera possui LEDs infravermelhos

embutidos e consegue captar os pontos onde essa luz foi refletida. O usuário deve colar um pequeno pedaço de papel reflexivo em sua cabeça, boné, ou qualquer outra parte do corpo. O software no computador, então, processa a imagem da câmera e a transforma em movimentos do ponteiro do *mouse*.

A mesma empresa também lançou um produto chamado TrackIR [5] que funciona de maneira similar, porém é voltado para o mercado de jogos. Neste caso, em vez de controlar o ponteiro do *mouse*, os movimentos da cabeça são enviados diretamente para o jogo, onde são traduzidos para movimentos da câmera virtual dentro do ambiente simulado. Para obter seis graus de liberdade nos movimentos, o usuário precisa colocar na cabeça uma armação contendo três pontos reflexivos ou três LEDs infravermelhos.

Ambos os produtos exigem que o usuário fique na frente de uma câmera e exigem um software proprietário instalado e configurado na máquina. O produto TrackIR está à venda a partir de US$ 99,95, enquanto o SmartNav está à venda a partir de US$ 399,00

A empresa Origin Instruments tem um produto chamado HeadMouse [6] cujo funcionamento é muito similar ao SmartNav, mas não requer nenhum software instalado. Esse produto se identifica como um *mouse* USB e portanto funciona em qualquer sistema. Está à venda por US$ 995,00.

1.2.3 Projetos baseados em acelerômetro

WiSHABI [7] é um projeto de dispositivo sem fio USB que pode funcionar como teclado ou *mouse* e faz leituras a partir de um acelerômetro. Foi implementado usando dois microcontroladores ATmega8.

"USB Wireless Tilt Mouse + Minesweeper" [8] é um outro projeto de *mouse* sem fio baseado num acelerômetro. Utiliza microcontroladores ATmega32 ou ATmega644.

TiltStick [9] é um projeto que implementa um *joystick* USB baseado em leituras de um acelerômetro.

Em comum, todos esses projetos usam microcontroladores de 8 bits da família AVR e implementam um dispositivo USB HID através do *driver* V-USB.

1.3 Estrutura da monografia

O capítulo 2 descreve os componentes e protocolos usados para a realização deste trabalho.

O capítulo 3 descreve o *hardware* do dispositivo.

O capítulo 4 descreve o *firmware* (*software*) do dispositivo.

O capítulo 5 descreve o problema de transformação de coordenadas, apresentando as abordagens experimentadas e seus resultados.

O capítulo 6 apresenta as conclusões deste trabalho, com os resultados alcançados e as limitações encontradas, assim como sugestões para trabalhos futuros.

2 Componentes e protocolos

"*I loved music, and in my ninth year at MIT, I decided to buy a hi-fi set. I figured that all I needed to do was look at the specifications. So I bought what looked like the best one, turned it on, and turned it off in five minutes, the sound was so poor.*"

Amar Gopal Bose

Neste capítulo são apresentados o microcontrolador ATmega8 e o sensor HMC5883L, assim como os protocolos USB e USB HID, usados na comunicação do microcontrolador com o computador, e o protocolo I^2C, usado na comunicação do microcontrolador com o sensor.

2.1 Microcontrolador ATmega8

ATmega8 é um microcontrolador de 8 bits da família AVR, fabricado pela *Atmel Corporation*. Suas características principais são [10]:

- Arquitetura Harvard, com espaços de endereçamento distintos para instruções e para variáveis.
- 8192 bytes de memória *Flash* para guardar o programa.
- 512 bytes de memória Electrically Erasable Programmable Read-Only Memory (EEPROM) para guardar parâmetros de configuração do programa.
- 1024 bytes de memória Static Random-Access Memory (SRAM) volátil para as variáveis.
- Voltagem de operação de 4,5 V a 5,5 V.
- Clock máximo de 16 MHz usando um cristal externo.
- Interface de comunicação serial I^2C/TWI.
- Disponível no encapsulamento PDIP[1] de 28 pinos, assim como QFP e QFN[2] de 32 pinos.

A arquitetura dos processadores AVR foi projetada em conjunto com os desenvolvedores do compilador de C da *IAR Systems*. Como consequência, o conjunto de instruções do AVR foi pensado de modo a minimizar o *overhead* durante a execução de programas que tenham sido escritos em linguagens de alto nível [11].

Além disso, as instruções são executadas num pipeline de dois estágios, permitindo um desempenho máximo de uma instrução por ciclo de *clock*. Nem sempre esse desempenho é alcançado, pois algumas instruções (como as que acessam a memória SRAM e os desvios) demoram pelo menos dois ciclos [10].

Todas as instruções da arquitetura AVR ocupam 16 ou 32 bits (2 ou 4 bytes). Por esse motivo, a memória *Flash* é endereçada por *words* de 16 bits. Podemos dizer que o ATmega8 possui 8192 bytes de memória *Flash*, ou de maneira equivalente, 4096 *words*. A memória SRAM e a memória EEPROM são endereçadas por bytes [10].

[1] O encapsulamento do tipo Plastic Dual In-line Package (PDIP), também chamado de Dual In-line Package (DIP), é o ideal para se trabalhar numa protoboard, pois o circuito integrado pode ser diretamente encaixado nela.

[2] Para o produto final, em ambiente de produção industrial, o ideal é usar um encapsulamento mais compacto, como Quad Flat Package (QFP) ou Quad-Flat No-leads Package (QFN)

O microcontrolador ATmega8 inclui um módulo de comunicação serial Inter-Integrated Circuit (I^2C), porém, para evitar problemas de patentes e licenças, a *Atmel Corporation* usa o nome Two-Wire Interface (TWI) para sua implementação dessa interface [12].

Embora existam alguns modelos de microcontrolador AVR com controlador USB embutido [13], o microcontrolador ATmega8 usado neste projeto não possui nenhum tipo de *hardware* dedicado para essa função. Para tal, foi usado um *driver* que implementa o protocolo USB via software, diretamente no *firmware* do microcontrolador. Essa solução será descrita em mais detalhes na seção 4.6.

2.2 Sensor HMC5883L

HMC5883L é um magnetômetro fabricado pela *Honeywell*. Um magnetômetro é também conhecido como "bússola digital" ou "bússola eletrônica".

Esse sensor trabalha de 2,16 V a 3,6 V e é capaz de medir a intensidade do campo magnético usando o efeito magnetorresistivo em três eixos perpendiculares (X, Y, Z) e converte essas medidas para um formato digital através de um Analog-to-Digital Converter (ADC) de 12 bits, chegando a uma precisão de 1° a 2°. Sua interface de comunicação é I^2C [14].

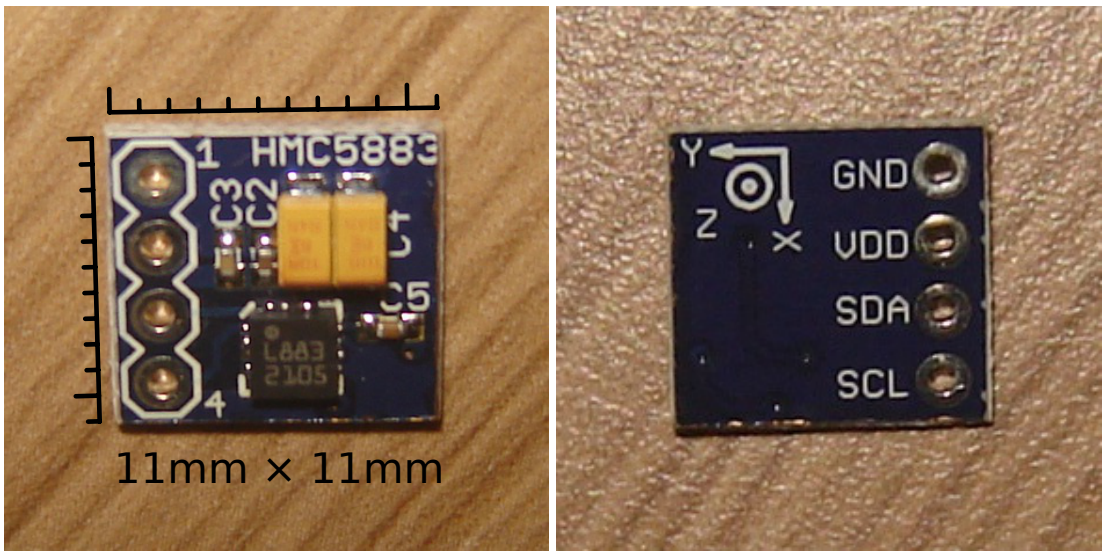


Figura 2.1: Fotos da PCB contendo o sensor

O circuito integrado possui encapsulamento Leaded Chip Carrier (LCC) e tem apenas $3{,}0 \times 3{,}0 \times 0{,}9$ mm de tamanho. É impossível trabalhar manualmente com algo tão minúsculo, por isso foi adquirida uma Printed Circuit Board (PCB) já contendo o sensor e alguns capacitores [15]. Essa placa possui quatro contatos, sendo metade deles para alimentação (GND e V_{DD}) e a outra metade para comunicação I^2C (SDA e SCL). Essa PCB pode ser vista na figura 2.1.

2.3 USB

A partir do ano de 1994, as empresas *Compaq*, *Hewlett-Packard*, *Intel*, *Lucent*, *Microsoft*, *NEC* e *Philips* desenvolveram um protocolo chamado Universal Serial Bus (USB). Essa iniciativa foi motivada pela inexistência de um barramento bidirecional de baixo custo para periféricos de baixa e média velocidade. Além disso, a falta de flexibilidade dos barramentos até então existentes não permitia reutilizá-los para outros periféricos, pois eram projetados para usos bastante específicos [16].

A primeira versão do USB, conhecida como "USB 1.0", foi oficialmente lançada em janeiro de 1996 e definiu duas velocidades: *low speed* (1,5 Mbit/s) e *full speed* (12 Mbit/s). Alguns anos depois, em setembro de 1998, foi lançada sua primeira revisão, "USB 1.1", que resolveu alguns problemas da versão anterior mas não introduziu nenhuma mudança significativa.

A primeira grande revisão do protocolo foi lançada em abril de 2000, com o nome de "USB 2.0". Esta versão introduziu uma terceira velocidade – *high speed* (480 Mbit/s) – mas manteve total compatibilidade com a revisão anterior do protocolo: dispositivos USB 1.x funcionam em *hosts* USB 2.0, e dispositivos USB 2.0 funcionam em *hosts* USB 1.x (porém limitados a *full speed*).

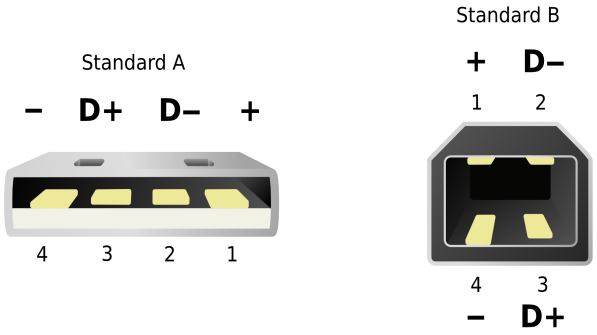


Figura 2.2: Conectores do tipo A e do tipo B para USB 1.x/2.0

Uma conexão USB é formada por quatro vias (conforme pode ser visto na figura 2.2), sendo duas para alimentação (V_{BUS} = 5 V e GND) e duas para comunicação (D+ e D-). Os dados são transmitidos usando a codificação Non-Return-to-Zero Inverted (NRZI) com *bit stuffing* (um "zero" é inserido após seis bits "um" consecutivos) [16] [17].

A velocidade de um dispositivo é determinada por *hardware*, de acordo com a posição do resistor de *pull-up*[3] nas vias de dados. Dispositivos *low speed* possuem um resistor de *pull-up*

[3] Resistores de *pull-up* ou de *pull-down* servem para manter uma via de dados num nível lógico conhecido enquanto nenhuma transferência ocorre. Normalmente possuem um valor relativamente alto (de 1,5 kΩ a 10 kΩ) para drenar pouca corrente. Um resistor de *pull-up* conecta a via ao V_{DD} e a mantém no nível lógico 1, enquanto

ligado ao D-, enquanto dispositivos *full speed* possuem um resistor de *pull-up* ligado ao D+. Se não há nenhum resistor de *pull-up*, assume-se que não há nenhum dispositivo conectado [16] [17].

Dispositivos *high speed*, introduzidos no USB 2.0, inicialmente se identificam como *full speed* (com um resistor de *pull-up* ligado ao D+), mas removem o resistor após uma negociação realizada durante o USB *reset*, caso o *host* também suporte *high speed*. Caso contrário, o resistor será mantido e o dispositivo funcionará em *full speed*. Essa negociação incial permite a compatibilidade entre dispositivos USB 2.0 *high speed* e *hosts* USB 1.x [16] [17].

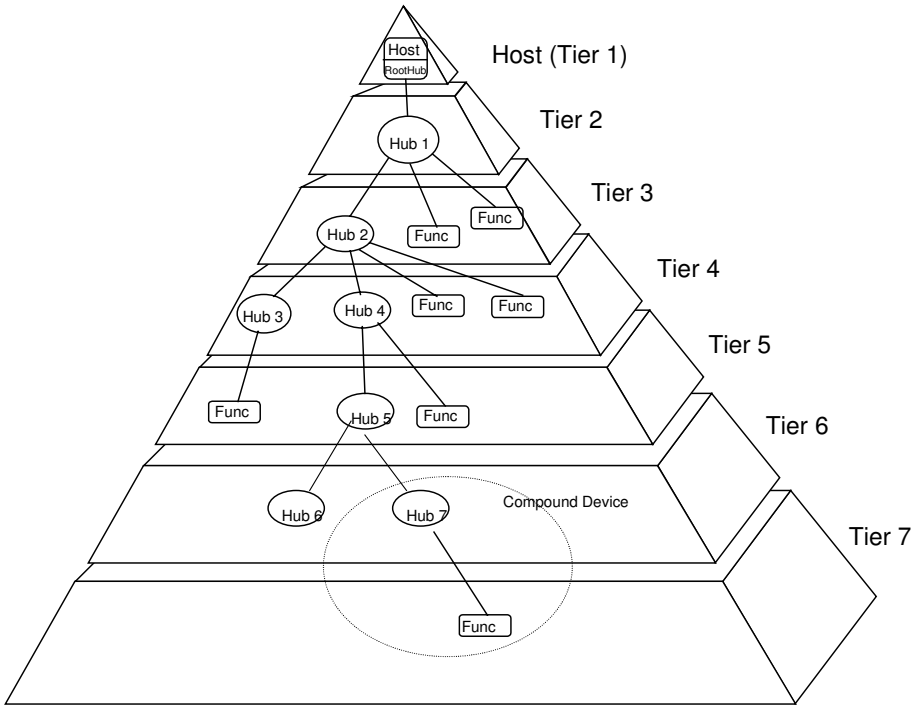


Figura 2.3: Topologia do barramento USB

A conexão de equipamentos USB segue uma topologia de estrela em camadas, que também pode ser entendida como uma árvore (conforme figura 2.3). No centro da estrela (ou na raiz da árvore) temos obrigatoriamente o *host*, que normalmente é um computador. Por definição, o *host* contém um *root hub*. Cada cabo USB é uma conexão ponto-a-ponto que liga um *hub* da camada imediatamente acima a um dispositivo na camada imediatamente abaixo. Um dispositivo pode ser um outro *hub* ou então uma função. Por limitações de tempo de propagação, o número máximo de camadas é sete, incluindo a camada que contém o *host* [16].

A especificação USB define dois tipos de conectores (ilustrados na figura 2.2). Um cabo USB possui uma ponta de cada tipo, sendo que o plugue do tipo A é conectado ao *hub* da camada acima, e o plugue do tipo B é conectado ao dispositivo da camada abaixo. Posteriormente, foram

um resistor de *pull-down* conecta a via ao GND e a mantém no nível lógico 0.

também especificados conectores de tamanho mais reduzido (Mini e Micro), mas os conceitos continuam os mesmos.

O protocolo de comunicação do USB pode ser entendido como uma arquitetura *master/slave*, na qual o *host* é o *master* e os dispositivos têm o papel de *slave*. Só há um *host* por barramento USB e ele é responsável por iniciar e gerenciar cada transação. Como consequência, a maior parte da lógica está centralizada no *host*, simplificando bastante a implementação de dispositivos que se conectam ao barramento USB [17].

Em novembro de 2008, foi lançada a especificação do USB 3.0, que introduziu uma quarta velocidade: *super speed* (5 Gbit/s). A quantidade de mudanças para suportar essa nova velocidade é grande e foge do escopo deste trabalho.

2.3.1 *Endpoints*

Um *endpoint* pode ser visto como um endereço lógico que pode receber dados do *host* ou enviar dados para ele. Todo dispositivo USB possui o *endpoint 0*, usado pelo próprio protocolo para mensagens de controle e de *status* [17].

Além do *endpoint 0*, um dispositivo USB normalmente possui pelo menos mais um *endpoint*, usado para o objetivo fim do dispositivo.

Há quatro tipos de *endpoints*: [17]

Control
: Bidirecional, usado durante a inicialização do dispositivo para comandos de controle do próprio protocolo USB.

Interrupt
: Unidirecional, latência garantida.

Isochronous
: Unidirecional, latência e largura de banda garantidas, sem garantia de entrega, detecção de erros sem tentativa de reenvio. Apenas para dispositivos *full speed* e *high speed*.

Bulk
: Unidirecional, sem garantia de latência ou de banda, porém com entrega garantida e recuperação de erros. Apenas para dispositivos *full speed* e *high speed*.

Endpoints do tipo *bulk* são usados quando é importante transmitir dados corretamente, sem

preocupação com o tempo necessário para concluir a transmissão. Dispositivos de armazenamento (como *pen drives*) e *scanners* de fotos fazem uso deste tipo de *endpoint*.

Endpoints do tipo *isochronous* são usados principalmente por dispositivos de *streaming* multimídia (audio e vídeo), nos quais há um fluxo constante de dados e o tempo de entrega é crítico.

Endpoints do tipo *interrupt* são usados para notificar o *host* de eventos do dispositivo, ou para notificar o dispositivo a respeito de eventos do *host*. Normalmente são mensagens curtas. Dispositivos de interface com o usuário, descritos na seção 2.4, fazem uso deste tipo de *endpoint*.

Quando as mensagens são enviadas do *host* para o dispositivo, o *endpoint* é chamado de *OUT*. Quando são do dispositivo para o *host*, o *endpoint* é chamado de *IN*.

É importante observar que um mesmo dispositivo pode ter múltiplos *endpoints* de tipos diferentes. Por exemplo, um *scanner* pode ter um *endpoint* do tipo *bulk-in* para transferir as imagens e um *endpoint* do tipo *interrupt-in* para notificar o *host* quando o botão do dispositivo é apertado.

2.4 USB HID

De modo a permitir a existência de dispositivos *plug-and-play* – que funcionam assim que são ligados ao *host*, sem necessidade de configurações feitas pelo usuário ou de instalação de drivers específicos – foram definidas algumas classes de dispositivos. Por exemplo, os populares *pen drives*, que substituíram os disquetes e CDs regraváveis para transferência de arquivos, pertencem à classe *USB Mass Storage*, e os sistemas operacionais modernos já incluem suporte nativo a dispositivos dessa classe.

Dispositivos de interface com seres humanos fazem parte da classe *USB Human Interface Device (HID)* e abrangem principalmente teclados, *mouses* e *joysticks*. No entanto, essa classe foi projetada de forma genérica e inclui outros tipos de dispositivos com necessidades similares, tais como medidores de temperatura, controles de um painel (botões, alavancas, ajustes), volantes, pedais, leitores de códigos de barra, ou ainda novos dispositivos que não foram previstos inicialmente na especificação [18].

A classe USB HID tem como objetivos ser compacta (para reduzir o espaço necessário no *firmware* do dispositivo), ser flexível e extensível, ser genérica e auto-descritiva (de modo que cada dispositivo descreva suas características de forma padronizada para o *host*). O driver da classe USB HID, presente no sistema operacional, é capaz de se comunicar com qualquer

dispositivo dessa classe [18].

Eventos do dispositivo (e.g., um botão foi pressionado ou o *mouse* foi movimentado) são enviados ao *host* através de um *interrupt-in endpoint*. Mensagens geradas pelo *host* e que devem ser enviadas para o dispositivo (e.g., acender ou apagar um dos indicadores do teclado) podem utilizar um *interrupt-out endpoint* ou o *control endpoint* [18].

Essas mensagens enviadas ou recebidas pelo dispositivo são chamadas de *reports*. O formato delas é definido pelo próprio dispositivo, através do *report descriptor*, o qual é enviado ao *host* durante a negociação inicial do protocolo USB [18].

A primeira versão da especificação do USB HID foi lançada em janeiro de 1996 (versão 1.0). Sua primeira revisão ficou disponível em abril de 1999 (versão 1.1). Sua segunda revisão, que também é a mais recente, foi lançada em junho de 2001 (versão 1.11). Apesar desta versão ter sido lançada após o USB 2.0, a especificação do USB HID menciona apenas as duas velocidades de dispositivos presentes no USB 1.x, e incorretamente chama dispositivos *full speed* de *high-speed* [18].

2.5 I^2C

Inter-Integrated Circuit (I^2C) é um protocolo de comunicação serial bidirecional desenvolvido em 1982 pela *Philips Semiconductors* (atual *NXP Semiconductors*). Foi projetado como um protocolo simples e eficiente para comunicação entre circuitos integrados. Utiliza apenas duas vias: Serial Clock (SCL) e Serial Data (SDA) [19].

A arquitetura do I^2C é baseada no modelo *master/slave*, sendo que qualquer um dos dispositivos do barramento pode assumir o papel de *master*, e inclusive o papel pode mudar ao longo do tempo. O protocolo permite também múltiplos *masters* no mesmo barramento [19] [10]. No entanto, para este trabalho foi necessário apenas ligar dois dispositivos: o microcontrolador ATmega8 (funcionando sempre como *master*) e o sensor HMC5883L (funcionando sempre como *slave*).

Tanto a via de dados SDA como a via de *clock* SCL são bidirecionais. O *master* é responsável por gerar o sinal de *clock*, mas o dispositivo *slave* pode esticar o período de *clock* em determinadas situações, indicando que ainda não está pronto para responder [19].

A comunicação no I^2C é baseada em bytes. O *master* envia um sinal de *START*, seguido de um byte de endereço. Nesse byte, sete bits, indicam o endereço do *slave* e o oitavo bit indica o tipo de comunicação (escrita ou leitura). Após a transmissão desse byte, o *master* libera a linha

de dados e o *slave* envia um bit zero durante próximo pulso de *clock*, sinalizando que recebeu o byte (*acknowledge*) [20].

Logo após o byte de endereço, inicia-se a transmissão dos bytes de dados. Caso seja uma escrita, o *master* envia os bytes de dados exatamente da mesma forma como enviou o byte de endereço, esperando pelo bit de *acknowledge* após cada byte enviado. Caso seja uma leitura, o *master* é responsável por gerar o *clock* (conforme já mencionado anteriormente), e o *slave* envia um bit a cada pulso do *clock*. Após cada oito bits recebidos (ou seja, após cada byte), o *master* deve enviar um bit de *acknowledge* para o *slave*. Em outras palavras, cada byte transmitido numa direção é sempre seguido de um bit de *acknowledge* transmitido na direção contrária [20].

O *master* finaliza uma transmissão enviando um sinal *STOP*, ou então enviando um novo sinal *START* para iniciar uma nova transmissão imediatamente (condição conhecida como *REPEATED START*) [10].

Pode-se também observar que a quantidade total de bytes transferidos é uma escolha do *master*, e o *slave* não tem como saber inicialmente quantos bytes serão requisitados.

3 *Descrição do hardware*

"*It's hardware that makes a machine fast. It's software that makes a fast machine slow.*"

Craig Bruce

Neste capítulo são detalhados todos os componentes eletrônicos usados neste projeto.

3.1 Visão geral do *hardware*

O núcleo do circuito é o microcontrolador ATmega8, o qual é responsável por ler os dados do sensor, fazer cálculos para converter os valores lidos, e enviar as coordenadas via USB.

A seção 3.2 descreve como todos os componentes são ligados ao microcontrolador.

A seção 3.3 detalha a interface entre o microcontrolador e a porta USB.

A seção 3.4 detalha a interface entre o microcontrolador e o sensor.

Um diagrama do circuito completo pode ser visto na figura 3.1.

3.2 Componentes ligados ao microcontrolador

Um cristal de 12 MHz, ligado aos pinos 9 e 10, define o *clock* do microcontrolador. O *driver* usado para a implementação do protocolo USB, descrito na seção 4.6, permite o uso de cristais de 12 MHz, 15 MHz, 16 MHz ou 20 MHz. A escolha pelo cristal de 12 MHz foi arbitrária.

O pino 1 ($\overline{\text{RESET}}$) possui um resistor de *pull-up* (ligado ao V_{CC}) e um botão ligado ao GND. Quando o botão é pressionado, o nível lógico do pino cai para zero e o microcontrolador é reiniciado. Esse botão de *reset* se mostrou bastante útil durante o desenvolvimento, mas não é necessário para o funcionamento deste projeto.

Há três Light-Emitting Diodes (LEDs) ligados aos pinos 11, 12 e 13 (PORTD5, PORTD6, PORTD7), montados de modo a acender quando o valor lógico 1 é escrito. Foram usados para fins de depuração, mas não são essenciais para o funcionamento deste projeto.

Há três botões ligados aos pinos 23, 24 e 25 (PORTC0, PORTC1, PORTC2), e ainda uma chave ligada ao pino 26 (PORTC3). Quando pressionados, o nível lógico de cada pino cai para zero. Cada pino possui internamente um resistor de *pull-up*, o qual foi habilitado pelo *firmware*, dispensando o uso de resistores externos.

Os pinos 17, 18 e 19 (MOSI, MISO, SCK), juntamente com o pino 1 ($\overline{\text{RESET}}$) são usados para In-System Programmer (ISP). Por simplicidade e por falta de necessidade, decidiu-se não usá-los para nenhuma outra função.

Por fim, os pinos 2 e 4 (PD0 e PD2) estão ligados às vias USB D- e USB D+, e serão detalhados na seção 3.3; e os pinos 27 e 28 (SDA e SCL) estão ligados ao sensor através de um barramento I^2C, e serão detalhados na seção 3.4.2.

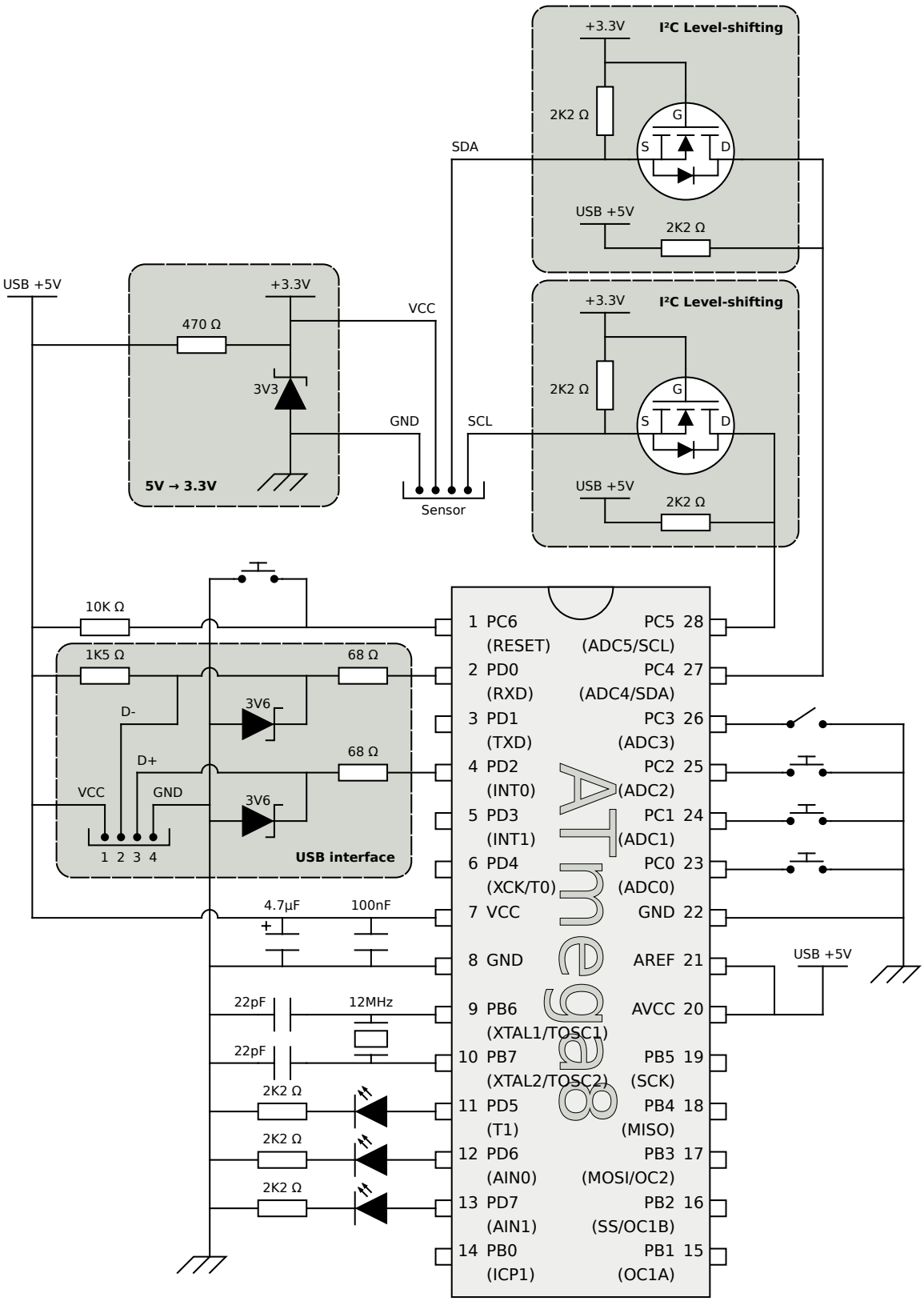


Figura 3.1: Diagrama completo do circuito

3.3 Interface USB

Uma porta USB fornece 5 V na via V_{BUS}, porém a especificação limita as vias de dados D+ e D- ao intervalo de 2,8 V a 3,6 V [16].

Uma solução seria usar um microcontrolador que possa funcionar a tensões mais baixas e colocar um circuito para reduzir a alimentação para dentro do intervalo desejado. Modelos mais recentes da linha AVR permitem o uso dessa solução [21]. No entanto, o microcontrolador ATmega8 usado neste projeto não trabalha com tensões inferiores a 4,5 V [10].

Uma outra solução é manter o microcontrolador alimentado com 5 V e adicionar um circuito para reduzir a saída dos pinos do AVR para a tensão desejada. Esta foi a solução empregada neste projeto.

O circuito usado para reduzir a tensão nas linhas de dados segue o mesmo esquema usado no projeto USBasp [22], também disponível no arquivo `circuits/with-zener.png` do *driver* V-USB [23]. É composto de um diodo Zener de 3,6 V ligando a linha de dados ao GND e um resistor de 68 Ω entre o microcontrolador e o diodo Zener.

Quando o microcontrolador escreve o nível lógico 1, seu pino sobe para 5 V. Como essa tensão é maior que a tensão de ruptura do Zener, este começa a conduzir, limitando a tensão da linha de dados. A diferença de $5 - 3{,}6 = 1{,}4$ V fica no resistor, resultando numa corrente de $1{,}4\,\text{V}/68\,\Omega = 20{,}6$ mA, que é abaixo do limite de 40 mA para cada pino do microcontrolador [10].

Além disso, conforme mencionado na seção 2.3, é preciso adicionar um resistor de *pull-up* à linha D-, indicando que este dispositivo é *low speed*. A especificação do USB 2.0 define que esse resistor deve ter 1,5 kΩ [16]. O projeto USBasp, no entanto, usa um resistor de 2,2 kΩ [22], que, mesmo fora da especificação, também funcionou.

Por fim, para o funcionamento correto do *driver* V-USB, é necessário que a linha USB D+ esteja ligada ao pino INT0 do microcontrolador. Além disso, é preciso que as duas vias de dados estejam ligadas a pinos que pertençam à mesma porta do microcontrolador [23]. Portanto, a linha D+ foi ligada ao pino 4 (PD2/INT0), e a linha D- foi ligada ao pino 2 (PD0).

3.4 Interface com o sensor

3.4.1 Alimentação

O sensor HMC5883L trabalha com alimentação de 2,16 V a 3,6 V [14]. Como a USB fornece 5 V, é necessário adicionar um circuito para reduzir a tensão até um nível adequado.

A solução aqui empregada utiliza um diodo Zener de 3,3 V e um resistor de 470 Ω [24]. É bastante similar àquela descrita na seção 3.3, mudando apenas os valores dos componentes.

O diodo Zener e o resistor estão ligados em série entre V_{CC} e GND, drenando constantemente uma corrente de $(5\,V - 3{,}3\,V)/470\,\Omega = 3{,}6\,mA$ e mantendo a tensão de 3,3 V em relação ao GND entre esses dois componentes.

Considerando que o sensor drena apenas 0,1 mA quando em uso [14], a queda de tensão na alimentação quando em carga é desprezível, e portanto essa solução é bastante adequada para este propósito.

3.4.2 Barramento I^2C

Se o microcontrolador e o sensor trabalhassem na mesma tensão, o barramento I^2C seria apenas um par de fios ligando os pinos correspondentes dos dois componentes, e dois resistores de *pull-up*, um em cada fio.

Todavia, os dois componentes não trabalham na mesma tensão. A primeira abordagem para este problema é verificar em suas especificações elétricas se as tensões de cada um estão dentro da faixa aceitável para o outro [25]. A tensão máxima de alimentação para o sensor é de 3,6 V [14], e tensão máxima tolerada no pino de alimentação é de 4,8 V [14], que é abaixo da tensão de 5 V do microcontrolador. Isso significa que não é prudente ligar os dois componentes diretamente.

Neste caso, faz-se necessário adicionar um circuito para converter o nível lógico 1 entre 5 V e 3,3 V. Essa conversão precisa funcionar nas duas direções, pois o barramento é bidirecional.

A solução adotada utiliza dois resistores de *pull-up* e um transistor MOSFET de canal N para cada uma das vias do barramento. É uma solução proposta pela própria *Philips/NXP*, desenvolvedora do padrão I^2C [25] [26]. Vários modelos diferentes de transistor MOSFET podem ser usados para este fim. Neste projeto, foi usado o modelo 2N7000. Os resistores de *pull-up* usados foram de 2,2 kΩ.

Para entender como essa solução funciona, é preciso analisá-la em três estados: [25] [26]

Nenhum dos dois dispositivos está puxando o barramento para zero

O lado de baixa tensão está em 3,3 V devido ao resistor de *pull-up*. A diferença de potencial entre os terminais *gate* e *source* do transistor é zero, e portanto ele não está conduzindo. Desta forma, o lado de tensão mais alta está em 5 V devido ao resistor de *pull-up*. Neste estado, os dois lados do barramento estão no nível lógico 1, apesar das tensões diferentes.

O dispositivo de menor tensão está puxando o barramento para zero

A diferença de potencial entre os terminais *gate* e *source* é 3,3 V, e o transistor entra em condução. O lado de tensão mais alta é também puxado para zero, através do transistor em condução.

O dispositivo de maior tensão está puxando o barramento para zero

O lado de menor tensão é também puxado para baixo através do diodo interno do transistor MOSFET. Isso ocorre até que a diferença de potencial entre *gate* e *source* fique grande o suficiente, quando então o transistor entra em condução e o lado de menor tensão é puxado para zero através do transistor.

4 Descrição do software

"*Premature optimization is the root of all evil.*"
Donald Ervin Knuth

Neste capítulo é detalhado o *firmware* deste projeto, assim como o ambiente de desenvolvimento e algumas das dificuldades encontradas.

4.1 Visão geral do *software*

Essencialmente, o trabalho do microcontrolador deste projeto é receber medições do sensor, aplicar um algoritmo de transformação de coordenadas e enviar para o computador as novas coordenadas do ponteiro. Para que isso pudesse ser alcançado, o *firmware* foi dividido em três grandes partes.

A primeira parte é a comunicação com o sensor, descrita nas seções 4.4 e 4.5.

A segunda parte é transformar os medições do sensor em coordenadas na tela do computador. Um estudo das diferentes abordagens para esse problema e seus resultados é apresentado no capítulo 5.

A terceira parte é a comunicação com o computador, alcançada através da implementação de um USB HID e descrita nas seções de 4.6 a 4.9.

4.2 Ambiente de desenvolvimento

O *firmware* deste projeto foi desenvolvido em ambiente Linux x86_64, utilizando o compilador AVR-GCC versão 4.5.3, a biblioteca AVR-Libc versão 1.7.0 e as ferramentas do projeto Binutils versão 2.21.1. O código-fonte foi escrito na linguagem C e é facilmente portável para outros microcontroladores da família AVR. Para gravação do *firmware* no microcontrolador, foi usado o programa AVRDUDE.

Todo o ambiente de desenvolvimento usado (AVR-GCC, Binutils, AVR-Libc, AVRDUDE) é composto por *software* livre e está disponível para os principais sistemas operacionais (FreeBSD, Linux, Mac OS X e Windows) [27].

O *firmware* deste projeto pode ser compilado em qualquer sistema operacional que tenha as ferramentas citadas, mesmo em outras versões, embora alguns ajustes no Makefile possam ser necessários.

4.3 *Boot loader*

Durante o início do projeto, o ciclo de desenvolvimento podia ser resumido nestas etapas:

1. Compilar o código-fonte.
2. Desconectar a *protoboard* da USB.

3. Conectar o gravador de AVR na USB.
4. Gravar a nova versão do *firmware*.
5. Desconectar o gravador de AVR.
6. Reconectar a *protoboard* na USB.

Rapidamente esse processo se mostrou bastante demorado e ineficiente, atrasando os testes e a depuração do código recém-escrito, e portanto atrasando o desenvolvimento. Além disso, também causava um desgaste mecânico desnecessário na porta USB. Embora esse desgaste seja desprezível a curto prazo, poderia se tornar um problema real para projetos de longo prazo.

Visando agilizar e simplificar o ciclo de desenvolvimento, foi gravado um *boot loader* na região de *boot* do microcontrolador.

Foi usado o USBaspLoader [28] como *boot loader*. Ele foi escolhido por ser *open source* e por também usar o *driver* V-USB, o qual será descrito em mais detalhes na seção 4.6.

Um *boot loader* é gravado nos endereços mais altos da memória *Flash*. O microcontrolador ATmega8 tem suporte a *boot loaders* de tamanhos 256, 512, 1024 e 2048 bytes [10]. Após compilado, o USBaspLoader possui 2028 bytes, e portanto ocupa os 2048 bytes superiores da memória *Flash*.

Logo após um *reset*, o ATmega8 normalmente começaria a execução do programa a partir do endereço mais baixo da memória. No entanto, quando se ativa o *boot loader* (através dos *fuse bits* do microcontrolador [10]), a execução começa no endereço da região de *boot*.

Portanto, ao iniciar o microcontrolador, o USBaspLoader é executado antes do *firmware* principal. Se uma determinada condição (configurável) for verdadeira, então o USBaspLoader assume o controle e se identifica como um gravador USBasp[1]. Caso essa condição seja falsa (antes ou durante a execução do USBaspLoader), então o *firmware* principal é executado.

Com o uso deste *boot loader*, o ciclo de desenvolvimento foi reduzido para:

1. Compilar o código-fonte.
2. Deixar ligada a chave conectada ao pino 26 (PORTC3) do microcontrolador.
3. Apertar o botão de *reset*.

[1] USBasp é um gravador de AVR com conexão USB que é implementado usando um microcontrolador ATmega8 ou similar [22].

4. Gravar a nova versão do *firmware*.

Após receber uma nova versão do *firmware* principal, o USBaspLoader automaticamente inicia a execução desse *firmware* recém-gravado.

Com o uso de um *boot loader*, o ciclo de desenvolvimento tornou-se notavelmente mais rápido e dinâmico. Testar mudanças no *firmware* passou a ser fácil, e os testes se tornaram mais frequentes.

Como desvantagem, o espaço disponível na memória *Flash* do microcontrolador foi reduzido de 8192 para 6144 bytes.

4.4 Comunicação I²C/TWI

O microcontrolador ATmega8 possui um *hardware* dedicado para comunicação I²C (chamada de TWI, conforme citado na seção 2.1). No entanto, seu uso é relativamente burocrático, tendo que implementar uma máquina de estados para gerenciar os registradores do módulo de TWI e a transmissão de dados [10]. O exemplo de código disponível na página 170 do *datasheet* do ATmega8 utiliza a técnica de *busy-wait* e não é adequado para o firmware deste projeto.

De maneira geral, a técnica de *busy-wait* (espera ocupada) deve ser evitada porque bloqueia a execução do restante do código por tempo indeterminado, durante o qual nenhum tipo de trabalho útil pode ser realizado. Uma das possíveis consequências é causar um *watchdog reset*, reiniciando a execução do código do *firmware*. Além disso, a espera ocupada não garantiria que a função `usbPoll()` fosse chamada em intervalos menores que 50 ms, conforme descrito na seção 4.6.1.

Reconhecendo a dificuldade de usar esse módulo de maneira eficiente, a própria *Atmel* publicou um documento e o código-fonte de um *driver*[2] que utiliza o módulo TWI de maneira mais eficiente [20].

Para este projeto, foi usado o *driver* disponível na *application note* "AVR315: Using the TWI module as I²C master". Seu funcionamento é baseado em interrupções, e portanto pode realizar a transmissão de vários bytes de maneira assíncrona (não-bloqueante), uma característica altamente desejável. A rotina de tratamento da interrupção basicamente implementa uma máquina de estados [20].

[2] Neste contexto, "*driver*" significa um pedaço de *software* dentro do *firmware* que controla um módulo de hardware do microcontrolador. Não confundir com *drivers* do sistema operacional.

O *driver* possui um *buffer* interno para armazenar os dados a serem enviados ou os dados sendo recebidos. O tamanho desse *buffer* é configurável em tempo de compilação, através da linha:

```
#define TWI_BUFFER_SIZE     7
```

O código-fonte estava originalmente escrito para o compilador IAR. Adaptá-lo para o compilador GCC foi uma tarefa bastante simples, bastando apenas trocar os cabeçalhos e trocar a sintaxe da declaração da rotina de interrupção [29].

```
// IAR:
#include "ioavr.h"
#include "inavr.h"


// IAR syntax:
#pragma vector=TWI_vect
__interrupt void TWI_ISR(void)
{
  ...
}
```

```
// GCC:
#include <avr/io.h>
#include <avr/interrupt.h>


// GCC syntax:
ISR(TWI_vect)
{
  ...
}
```

4.5 Comunicação com o sensor

O sensor HMC5883L possui treze registradores de 8 bits que podem ser acessados pela comunicação serial, mais um registrador interno chamado de *address pointer*, o qual aponta para algum dos outros registradores e cujo valor não pode ser lido [14].

Quando o *master* inicia uma escrita, o primeiro byte recebido pelo sensor é considerado o endereço de um registrador, e esse endereço é salvo no *address pointer*. Caso um segundo byte seja recebido, esse valor é salvo no registrador apontado pelo *address pointer*.

Assim, para escrever num dos registradores de configuração do sensor, o *firmware* do microcontrolador prepara um vetor de três bytes: o endereço do sensor (juntamente com o bit que define uma operação de escrita), o número do registrador de configuração que será modificado, e o valor a ser escrito no registrador. Esse vetor é repassado ao *driver* de TWI (descrito na seção 4.4), o qual copia esses bytes para um *buffer* interno e inicia a comunicação assincronamente.

Quando o *master* inicia uma leitura, o sensor envia um byte com o valor contido no registrador apontado pelo *address pointer* e incrementa o *address pointer* para apontar para o próximo registrador. Caso outro byte seja requisitado (ainda na mesma transmissão, ou em outra transmissão), o processo se repete, enviado o valor contido no registrador apontado pelo *address*

pointer e incrementando-o. Desta forma, é possível ler o conteúdo de vários registradores adjacentes de maneira bem eficiente, com *overhead* de comunicação mínimo.

No entanto, a implementação do sensor não é perfeita, e em certos casos o auto-incremento do *address pointer* não funciona como esperado. Por exemplo, uma tentativa de ler todos os registradores (usando o auto-incremento para iterar automaticamente por todos eles) retorna valores diferentes (e inválidos) daqueles retornados por leituras individuais. De maneira geral, o auto-incremento funciona corretamente apenas para a leitura dos três registradores de identificação do sensor e para a leitura dos seis registradores de dados, não atrapalhando o uso normal do sensor.

Para ler os valores X, Y, Z do sensor, o microcontrolador começa realizando uma escrita no sensor, enviando o valor `0x03` como endereço do registrador. A seguir, realiza uma leitura de seis bytes consecutivos. Por fim, esses seis bytes lidos são convertidos numa `struct` de três variáveis inteiras de 16 bits.

4.5.1 Configuração do sensor

Durante a inicialização do *firmware* do microcontrolador, os registradores de configuração do sensor são escritos.

O sensor foi configurado para atualizar os registradores de dados 75 vezes por segundo, sendo que cada valor é a média de oito amostras. O ganho do sensor foi configurado para ±1,3 G, o que corresponde a 1090 bits por gauss[3], ou 0,92 mG por bit. Este ganho foi escolhido por permitir a maior precisão possível das medidas sem causar *overflow* no sensor.

O sensor foi configurado para o modo de medição contínua, i.e., está continuamente medindo o campo magnético e atualizando os registradores de dados (75 vezes por segundo), mesmo que o microcontrolador não faça a leitura dessas medidas.

Por limitação do sensor, o modo de medição contínua permite obter medições a uma taxa máxima de 75 Hz. Apesar disso, é possível obter medições a até 160 Hz se for configurado para o modo de medições individuais e o pino DRDY for monitorado [14]. Este pino normalmente fica no nível lógico 1, porém cai para 0 por 250 µs assim que uma nova medida é colocada nos registradores de dados [14]. Portanto, seria possível ligar esse pino DRDY a algum dos pinos de interrupção do microcontrolador e escrever uma rotina que leia os dados e inicie a próxima medição.

[3] O símbolo G corresponde à unidade de densidade de fluxo magnético "gauss" [30]. O *datasheet* do sensor, no entanto, utiliza Ga como símbolo para essa unidade.

Todavia, a PCB usada neste projeto (figura 2.1) não possui um contato para o pino DRDY, e portanto não foi possível se aproveitar dessa técnica.

4.6 *Driver* V-USB

O *driver* V-USB foi desenvolvido pela *Objective Development Software GmbH.* e é uma implementação em *software* do protocolo USB, funcionando em diversos microcontroladores da família AVR que não possuem um controlador USB embutido [23].

Por ser implementado em *software*, o tempo de execução das rotinas do *driver* é crucial, e essas rotinas foram implementadas (pelos autores do *driver*) em *assembly* para *clocks* de 12 MHz, 15 MHz, 16 MHz ou 20 MHz usando cristal, ou 12,8 MHz ou 16,5 MHz usando o oscilador interno do microcontrolador. Não é possível usar o *driver* com frequências diferentes destas [23].

Apesar das rotinas internas estarem escritas em *assembly*, o *driver* disponibiliza uma Application Programming Interface (API) bastante simples em linguagem C.

O *driver* implementa um dispositivo USB 1.1 *low speed*, mas possui algumas limitações. Uma delas é assumir que não há erros durante a comunicação, pois não há tempo de CPU suficiente para implementar checagem de erros via *software*. Além disso, as características elétricas dos pinos do microcontrolador não atendem estritamente a especificação USB [23].

A especificação USB também define que os dispositivos devem implementar um modo *suspend*, no qual não devem consumir mais que 500 µA [17]. O *driver* não implementa esse modo, mas explica em linhas gerais como seria possível implementá-lo [23]. Para este projeto, não foi implementado o suporte ao modo *suspend*, pois não é uma tarefa trivial e está fora do escopo deste trabalho.

Considerando as limitações citadas nos parágrafos anteriores, o dispositivo fruto deste projeto não é totalmente conformante com a especificação USB, mas isso não causou problemas nos computadores onde o dispositivo foi testado, conforme resultados no capítulo 6. Além disso, essas limitações não impediram que mais de cem projetos tenham sido criados com sucesso [31].

4.6.1 Configuração do *driver* V-USB

Toda a configuração do *driver* V-USB é feita no arquivo `usbconfig.h`. Como a maior parte da configuração é compartilhada com o *boot loader*, foi criado um arquivo `hardwareconfig.h`

que contém as configurações compartilhadas entre o *firmware* principal e o *boot loader* (tais como em quais pinos estão ligadas as linhas D- e D+), enquanto as configurações específicas foram mantidas no arquivo `usbconfig.h`.

Para o *firmware* principal, foi habilitado um *interrupt-in endpoint* com intervalo *polling* de 10 ms (o menor valor possível para dispositivos *low speed*). Também foram definidos outros parâmetros, como o nome do dispositivo, o nome do fabricante, o *vendor-id* e o *product-id*.

Para um produto USB ser comercializado, o fabricante precisa adquirir um *vendor-id* juntamente com o USB Implementers' Forum (USB-IF), e pode escolher o *product-id* que quiser. Para projetos criados por *hobby* ou para fins acadêmicos, o custo e a burocracia para se adquirir tal número não é viável.[4] Portanto, para este projeto foi usado um par de *vendor-id*:*product-id* compartilhado pela *Objective Development Software GmbH.*[5].

Uma vez configurados os parâmetros necessários nos cabeçalhos do *driver*, usar sua API na linguagem C foi bastante simples. É necessário chamar a função `usbInit()` na inicialização do *firmware*, antes de habilitar as interrupções do microcontrolador, e é preciso chamar a função `usbPoll()` dentro do *loop* principal do *firmware*. É importante chamar essa função em intervalos menores que 50 ms para que o *driver* funcione corretamente. Por fim, também foi necessário implementar a função `usbFunctionSetup()`, responsável por responder a algumas das mensagens recebidas no *control endpoint 0* (as outras mensagens são respondidas diretamente pelo *driver*) [34].

4.7 Teclado USB

Logo no início do projeto foi necessário testar se a comunicação USB estava funcionando corretamente, validando o *software* e o *hardware*. Além disso, embora alguns LEDs fossem suficientes para os testes iniciais de comunicação com o sensor, era preciso ter alguma forma de *output* para exibir os dados lidos.

A solução foi implementar um teclado USB no microcontrolador. Essa implementação simula teclas sendo pressionadas de modo a escrever *strings* na tela do computador. Em outras palavras, foi uma forma de se implementar o comando *print* dentro do *firmware*.

[4] O logotipo USB, é uma marca registrada do USB-IF, e só pode ser usado em produtos que passaram nos testes dessa organização. [32] O custo para se adquirir um *vendor-id* é de pelo menos US$ 2.000,00 [33].

[5] Este projeto se identifica como um dispositivo híbrido, funcionando ao mesmo tempo como *mouse* e teclado. Não há nenhum par compartilhado de *vendor-id*:*product-id* que seja adequado a este projeto [23]. Após uma breve troca de e-mails com a *Objective Development Software GmbH.*, e sabendo que o uso era para fins acadêmicos, foi sugerido usar valores quaisquer, aleatórios para esses campos.

Teclados e *mouses* USB podem ser implementados como dispositivos de *boot*. Tais dispositivos possuem um *report descriptor* predeterminado, definido no padrão USB HID. Isso foi criado para permitir o uso de teclados e *mouses* mesmo nos estágios iniciais do *boot* do computador, quando pode ser inviável ter uma implementação completa da especificação HID [18].

O protocolo *boot* obriga que o dispositivo se identifique ou como um teclado ou como um *mouse*, mas não como os dois [18]. Além disso, ele define um *report descriptor* que suporta até seis teclas pressionadas simultaneamente, além das teclas modificadoras [18], o que é desnecessário para um teclado que vai apenas escrever *strings*. Por todos esses motivos, foi decidido implementar um teclado que não suporta o protocolo *boot*.

O teclado implementado neste projeto utiliza o *report descriptor* mostrado na listagem 4.1, o qual foi baseado no projeto HIDKeys [35]. O código do *report descriptor* foi escrito utilizando a ferramenta HID Descriptor Tool [36], disponível no próprio site do USB-IF.

```
0x05, 0x01,         // USAGE_PAGE (Generic Desktop)
0x09, 0x06,         // USAGE (Keyboard)
0xa1, 0x01,         // COLLECTION (Application)
0x85, 0x01,         //   REPORT_ID (1)
// Modifier keys
0x05, 0x07,         //   USAGE_PAGE (Keyboard)
0x19, 0xe0,         //   USAGE_MINIMUM (Keyboard LeftControl)
0x29, 0xe7,         //   USAGE_MAXIMUM (Keyboard Right GUI)
0x15, 0x00,         //   LOGICAL_MINIMUM (0)
0x25, 0x01,         //   LOGICAL_MAXIMUM (1)
0x75, 0x01,         //   REPORT_SIZE (1)
0x95, 0x08,         //   REPORT_COUNT (8)
0x81, 0x02,         //   INPUT (Data,Var,Abs)
// Normal keys
0x05, 0x07,         //   USAGE_PAGE (Keyboard)
0x19, 0x00,         //   USAGE_MINIMUM (Reserved (no event indicated))
0x29, 0x65,         //   USAGE_MAXIMUM (Keyboard Application)
0x15, 0x00,         //   LOGICAL_MINIMUM (0)
0x25, 0x65,         //   LOGICAL_MAXIMUM (101)
0x75, 0x08,         //   REPORT_SIZE (8)
0x95, 0x01,         //   REPORT_COUNT (1)
0x81, 0x00,         //   INPUT (Data,Ary,Abs)
0xc0,               // END_COLLECTION
```

Listagem 4.1: *Report descriptor* do teclado USB

O *report* usado por este teclado possui três bytes. O primeiro deles é o *Report ID* e tem valor 1, identificando que este *report* corresponde ao *report descriptor* do teclado[6]. No segundo byte, cada um dos 8 bits corresponde a uma tecla modificadora, embora o *firmware* utilize apenas a tecla *left shift*. O terceiro byte é a tecla sendo pressionada.

Internamente, o teclado virtual do microcontrolador possui um *buffer* de tamanho fixo que

[6] Na seção 4.9, o *Report ID* do *mouse* é definido com o valor 2.

armazena a *string* a ser digitada, e um ponteiro indicando o próximo caractere do *buffer* que deve ser digitado. No *loop* principal do *firmware*, é checado se o *interrupt-in endpoint* está pronto para enviar dados. Se estiver, e houver algum caractere ainda não digitado, o *firmware* converte o caractere para o código da tecla correspondente e envia o *report*.

Letras maiúsculas e alguns símbolos ligam o bit correspondente à tecla *left shift*, enquanto os outros caracteres desligam esse bit. Além disso, foi necessário detectar caracteres consecutivos que são convertidos para a mesma tecla (e.g., letras repetidas), e neste caso é enviado um *report* indicando nenhuma tecla pressionada antes de enviar o *report* correspondente ao caractere.

A especificação USB HID Usage Tables define que o código de uma tecla mantém-se o mesmo, independente do que estiver impresso na tecla. Isto significa que a tecla "Z" de um teclado QWERTZ envia o mesmo código que a tecla "Y" de um teclado QWERTY [37]. Em outras palavras, a interpretação do código da tecla é de inteira responsabilidade do sistema operacional, através da configuração do *layout* do teclado. O teclado implementado neste projeto supõe estar configurado com *layout* US ou BR-ABNT2.

Por fim, foram implementadas no *firmware* algumas funções para converter números inteiros em *strings*, seja na base decimal ou na base hexadecimal. Algumas dessas rotinas utilizam outras funções já implementadas na AVR-Libc.

As rotinas do teclado USB foram usadas para fins de depuração e também como base para o menu de configuração do dispositivo, descrito na seção 4.8.

4.8 Menu de configuração

Conforme mais funcionalidades estavam sendo implementadas no *firmware* (algumas delas para fins de depuração), a quantidade delas superou rapidamente o número de botões presentes no circuito. Como consequência, ficou cada vez mais difícil de adicionar outra funcionalidade ou mesmo usar as já existentes. Foi sentida a necessidade de uma interface melhor, mais prática e mais intuitiva para o uso do *firmware*.

Essa foi a motivação para implementar um sistema de menus interativos. Esse sistema combina o teclado virtual USB previamente implementado (descrito na seção 4.7) com os três botões presentes na *protoboard* (descritos na seção 3.1).

Um dos botões foi escolhido para funcionar como "confirma", enquanto os outros dois funcionam como "próximo item" e "item anterior".

O sistema de menus é iniciado apertando o botão "confirma", o item atualmente selecionado

é digitado na tela usando a funcionalidade do teclado USB (já descrito na seção 4.7). Todas as principais funcionalidades do *firmware* podem ser acessadas através dos menus, simplificando o uso do dispositivo (tanto para o usuário final quanto para o desenvolvedor).

4.9 *Mouse* USB

O objetivo final deste projeto é controlar o ponteiro do *mouse* na tela do computador, e para isso foi preciso implementar um *mouse* USB no microcontrolador de maneira similar ao teclado USB descrito na seção 4.7.

Todavia, o *mouse* deste projeto tem uma diferença fundamental em relação aos *mouses* convencionais. A transformação de coordenadas, descrita no capítulo 5, obtém como resultado um par de coordenadas (x, y) em relação à tela. Essas coordenadas são absolutas, no sentido de indicar uma posição específica da tela. Um *mouse* convencional, no entanto, detecta apenas movimentos e obtém coordenadas relativas $(\Delta x, \Delta y)$, indicando a nova posição em relação à posição atual do ponteiro. Devido a essa diferença, o dispositivo implementado neste projeto é na verdade considerado um *absolute pointing device*[7].

A especificação USB HID define um *report descriptor* para *mouses* que implementem o protocolo *boot* [18]. Assim como a implementação do teclado (seção 4.7), este dispositivo não tem suporte ao protocolo *boot* e utiliza o *report descriptor* mostrado na listagem 4.2.

O *report* usado por este *mouse* possui seis bytes. O primeiro deles é o *Report ID* e tem valor 2, identificando que este *report* corresponde ao *report descriptor* do *mouse*[8]. Os quatro bytes seguintes correspondem à posição (x, y), onde cada uma das duas coordenadas é representada por um número inteiro de 16 bits dentro do intervalo de 0 a 32 767. Valores fora desse intervalo são considerados *null values* e devem ser ignorados pelo *host* [18]. O último byte representa os botões do *mouse*, sendo que os três bits menos significativos correspondem aos três botões, e os cinco bits restantes são ignorados.

O algoritmo de transformação de coordenadas implementado no microcontrolador foi a abordagem usando um sistema de três equações lineares (seção 5.3). Essa solução foi escolhida por ser a mais simples e portanto ocupar pouco espaço na memória bastante limitada do microcontrolador[9]. Os cálculos utilizam variáveis `float` de 32 bits [38], e a resolução do sistema

[7] Apesar de na verdade ser um *absolute pointing device*, o nome *mouse*, por ser mais simples, mais curto e mais fácil de entender, é usado em todo o projeto.

[8] Na seção 4.7, o *Report ID* do teclado é definido com o valor 1.

[9] Outras abordagens mais complexas podem ser implementadas caso microcontrolador ATmega8 seja substituído por outro com mais memória *Flash*.

```
0x05, 0x01,         // USAGE_PAGE (Generic Desktop)
0x09, 0x02,         // USAGE (Mouse)
0xa1, 0x01,         // COLLECTION (Application)
0x85, 0x02,         //   REPORT_ID (2)
0x09, 0x01,         //   USAGE (Pointer)
0xa1, 0x00,         //   COLLECTION (Physical)
// X, Y movement
0x09, 0x30,         //     USAGE (X)
0x09, 0x31,         //     USAGE (Y)
0x15, 0x00,         //     LOGICAL_MINIMUM (0)
0x26, 0xff, 0x7f,   //     LOGICAL_MAXIMUM (32767)
0x75, 0x10,         //     REPORT_SIZE (16)
0x95, 0x02,         //     REPORT_COUNT (2)
0x81, 0x42,         //     INPUT (Data,Var,Abs,Null)
0xc0,               //   END_COLLECTION
// Buttons
0x05, 0x09,         //   USAGE_PAGE (Button)
0x19, 0x01,         //   USAGE_MINIMUM (Button 1)
0x29, 0x03,         //   USAGE_MAXIMUM (Button 3)
0x15, 0x00,         //   LOGICAL_MINIMUM (0)
0x25, 0x01,         //   LOGICAL_MAXIMUM (1)
0x75, 0x01,         //   REPORT_SIZE (1)
0x95, 0x03,         //   REPORT_COUNT (3)
0x81, 0x02,         //   INPUT (Data,Var,Abs)
// Padding for the buttons
0x75, 0x01,         //   REPORT_SIZE (1)
0x95, 0x05,         //   REPORT_COUNT (5)
0x81, 0x03,         //   INPUT (Cnst,Var,Abs)
0xc0                // END_COLLECTION
```

Listagem 4.2: *Report descriptor* do *mouse* USB

linear utiliza o método de *Gauss-Jordan* [39].

Para o funcionamento desse algoritmo de transformação de coordenadas é necessário primeiramente calibrar os cantos da tela. Foi implementado um item no menu (seção 4.8) para essa finalidade.

Do ponto de vista do usuário final, a calibração dos cantos é bastante simples. O usuário aponta o sensor na direção de um dos cantos e ativa o item do menu correspondente a esse canto. Nesse momento, é feita uma leitura do sensor, e o vetor lido é salvo na EEPROM e está imediatamente disponível para uso do algoritmo.

4.9.1 Suavização do movimento

O resultado do primeiro teste após a implementação do *mouse* USB e da transformação de coordenadas foi insatisfatório. Embora o algoritmo funcionasse corretamente e o ponteiro acompanhasse os movimentos do sensor, a posição do ponteiro não era estável. Mesmo com o

sensor parado, o ponteiro oscilava significativamente na tela, indicando um ruído nas medições.

Esse ruído pode ser consequência da interferência magnética de aparelhos elétricos e eletrônicos, ou apenas uma limitação da precisão do sensor (que, conforme descrito na seção 2.2, possui um ADC de 12 bits). As razões exatas para tal ruído não são conhecidas.

Foi então implementado um filtro de suavização de segunda ordem. O filtro é aplicado após a transformação de coordenadas, mas antes de preencher o *report*. O valor final das coordenadas na tela é uma média ponderada calculada desta forma:

$$x'_t = (1-\gamma)x'_{t-1} + \gamma x_t$$
$$x''_t = (1-\gamma)x''_{t-1} + \gamma x'_t$$

Onde x_t é o resultado da transformação de coordenadas, x'_t e x''_t são o resultado dos filtros de primeira e segunda ordem, e x'_{t-1} e x''_{t-1} são os resultados dos filtros após a medição anterior. O valor final da coordenada x do ponteiro na tela é x''_t. Um cálculo análogo é efetuado para a coordenada y.

Experimentalmente foi escolhido $\gamma = 0.125$, considerando um balanço entre o tempo de resposta e a intensidade do filtro. Foi também observado que valores menores para γ, como $\gamma = 0.0625$, introduziram um atraso perceptível para o usuário.

4.9.2 Bug no kernel do Linux

Durante os testes do *mouse* USB, foi descoberto um bug na implementação do suporte a USB HID no kernel do Linux.

A especificação USB HID define que valores fora dos limites lógicos para um campo de um *report* devem ser ignorados [18]. Neste projeto, conforme descrito na seção 4.9, cada coordenada é representada por um inteiro de 16 bits dentro do intervalo de 0 a 32 767. Valores fora desse intervalo, porém ainda representáveis em 16 bits, deveriam ser ignorados pelo sistema operacional. No entanto, foi observado que esses valores eram interpretados pelo sistema e causavam movimento indevido do ponteiro do *mouse*.

O código do *firmware* foi então adaptado para não enviar valores fora desse intervalo. Não houve nenhuma perda de funcionalidade causada por essa adaptação.

O bug foi reportado na Linux Kernel Mailing List (LKML) [40] e possivelmente será

corrigido na versão 3.3 ou posterior [41].

4.10 Outras funcionalidades

4.10.1 *Debouncing* dos botões

Um problema bem conhecido na eletrônica é o fenômeno de *bouncing*. Botões são inerentemente mecânicos, e, ao serem pressionados, demora uma fração de segundo até que o contato seja estabilizado. Isso ocorre devido às características elásticas dos materiais que os compõem. Nesse curto intervalo de tempo, o botão oscila rapidamente entre os estados de aberto e fechado.

Conhecendo esse problema, faz-se necessário aplicar alguma técnica de *debouncing*. A solução empregada neste projeto foi inspirada na solução adotada pelo projeto TiltStick [9]. Consiste em considerar a mudança de estado do botão apenas caso tenham-se passadas oito leituras consecutivas no novo estado. Isso foi implementado usando um byte para cada botão. A cada leitura, o valor do byte é deslocado um bit para a esquerda e a leitura atual é guardada no bit menos significativo. Caso o valor do byte seja igual a `0x00`, considera-se que o botão não está pressionado. Caso o valor seja igual a `0xFF`, considera-se que o botão está pressionado[10]. Qualquer outro valor é ignorado, e o botão mantém o estado anterior.

4.10.2 Gravar configurações na EEPROM

A biblioteca AVR-Libc já inclui funções para leitura e gravação de dados na EEPROM. Essas funções são implementadas usando a técnica de *busy-wait*, e portanto podem bloquear a execução do firmware por tanto tempo quanto demorar a leitura ou escrita dos bytes requisitados [42].

O tempo total para a leitura de um byte da EEPROM no ATmega8 é de quatro ou cinco ciclos de *clock*[11], enquanto a escrita de cada byte demora aproximadamente 8,5 ms [10]. Portanto, a leitura é rápida o suficiente para não causar nenhum tipo de problema, mas a escrita é bastante demorada, e realizar escritas através da técnica de *busy-wait* não é adequado pelos mesmos motivos já citados na seção 4.4.

Foi então necessário implementar um tipo de escrita não bloqueante. A própria *Atmel*

[10] No *hardware*, o nível lógico 0 representa um botão pressionado. Como é mais intuitivo pensar que 1 representa o botão pressionado e 0 botão solto, o valor lido pelo *hardware* é invertido pela rotina de leitura dos botões.

[11] O *datasheet* não é claro, informando que o acesso à EEPROM demora apenas um ciclo, mas também que a CPU fica parada por quatro ciclos antes que a instrução seguinte seja executada.

publicou uma solução para esse problema na *application note* "AVR104: Buffered Interrupt Controlled EEPROM Writes" [43]. Sua implementação é bastante completa e inclui um *buffer* de escritas bastante poderoso, permitindo que múltiplas escritas para endereços arbitrários na EEPROM sejam empilhadas simultaneamente. Além disso, também tem suporte aos diferentes *sleep modes* do microcontrolador [10].

Embora seja uma excelente solução para o caso geral, essa implementação é grande demais para as necessidades deste projeto. Sabendo que *firmware* não usa *sleep modes* e que não ocorrerá o caso de mais de uma gravação simultânea, foi possível simplificar bastante o código dessa implementação.

Apenas um trecho da rotina de tratamento da interrupção foi aproveitada dessa *application note*, todo o restante foi reescrito ou simplesmente apagado. O código final empregado no *firmware* utiliza um *buffer* simples que permite gravar uma sequência de bytes numa posição contígua da EEPROM.

4.10.3 Calibração do "zero" do sensor

Durante o desenvolvimento do projeto foi observado que os vetores correspondentes às medições do campo magnético estavam enviesados, i.e., não eram centralizados ao redor das coordenadas $(0,0,0)$.

Não se sabe se esse efeito é uma característica intrínseca do sensor, ou se é causado por fatores externos. O *datasheet* do sensor tem algumas recomendações sobre o projeto da PCB na qual o sensor será montado. Em especial, recomenda afastar componentes que contenham materiais ferrosos, assim como evitar trilhas condutoras embaixo ou próximo do sensor [14]. Não foi possível descobrir se o fabricante da PCB adquirida (figura 2.1 na seção 2.2) tomou tais precauções.

Para resolver esse problema, foi implementado um item no menu para a calibração do "zero" do sensor. Durante a calibração, o sensor deve ser movido em todas as direções, de modo a descobrir os valores máximo e mínimo para cada um dos eixos. Uma vez concluída a calibração, o viés é calculado como a média entre os valores máximo e o mínimo, e é então gravado na EEPROM. Após calibrado, todas as leituras do sensor são compensadas via *software*, usando o valor calibrado. O resultado pode ser visto na figura 4.1.

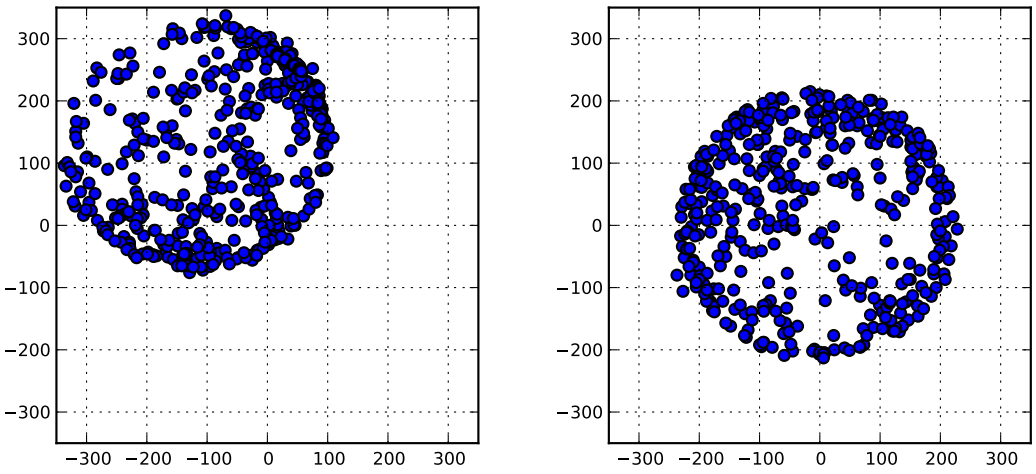


Figura 4.1: Medições do sensor antes e depois da calibração do “zero”

5 *Transformação de coordenadas*

"*Models are a bunch of equations; physics is what happens when you drop a hammer on your big toe. They are not the same. Just ask your big toe.*"

Ken Matusow

Neste capítulo é apresentado o problema de transformação de coordenadas, as abordagens experimentadas e os resultados observados em cada abordagem.

5.1 Ferramentas auxiliares

O principal desafio teórico deste projeto foi transformar os vetores tridimensionais lidos do sensor em coordenadas bidimensionais na tela do computador. Não existe uma única solução para este problema e foi preciso explorar algumas abordagens antes de decidir qual delas seria implementada no microcontrolador.

Com o objetivo de experimentar e avaliar as diferentes abordagens para a transformação de coordenadas, foram desenvolvidas algumas ferramentas na linguagem Python. A escolha da linguagem foi motivada pela facilidade e rapidez de se escrever protótipos e pela extensa e abrangente biblioteca disponível.

A primeira ferramenta foi chamada de `generate_sphere_vectors.py` e serve para simular leituras do sensor. Esta ferramenta imprime na saída padrão uma lista de vetores 3D que cobrem pouco menos da metade da superfície de uma esfera. Também imprime quatro vetores correspondentes aos cantos da tela, os quais são usados nos cálculos explicados nas seções 5.2 e 5.3. A listagem 5.1 mostra um trecho da saída desta ferramenta.

```
topleft
172     69      75
topright
172     -69     75
bottomright
172     -69     -75
bottomleft
172     69      -75
0       141     141
2       141     141
5       141     141
7       141     141
10      141     141
```

Listagem 5.1: Primeiras linhas da saída do programa `generate_sphere_vectors.py`

A figura 5.1 mostra um gráfico dos vetores 3D impressos por esta ferramenta. Esse gráfico corresponde à região de $-90°$ a $90°$ de longitude e $-45°$ a $45°$ de latitude. Há um espaçamento de $1°$ de longitude e $2°$ de latitude entre os vetores.

A segunda ferramenta foi chamada de `convert_coordinates.py` e implementa os algoritmos de transformação de coordenadas explicados nas seções 5.2 e 5.3. Esta ferramenta lê vetores 3D a partir da entrada padrão e imprime na saída padrão as coordenadas 2D correspondentes, calculadas de acordo com o algoritmo selecionado. Um exemplo de entrada e saída para esta ferramenta pode ser visto na listagem 5.2

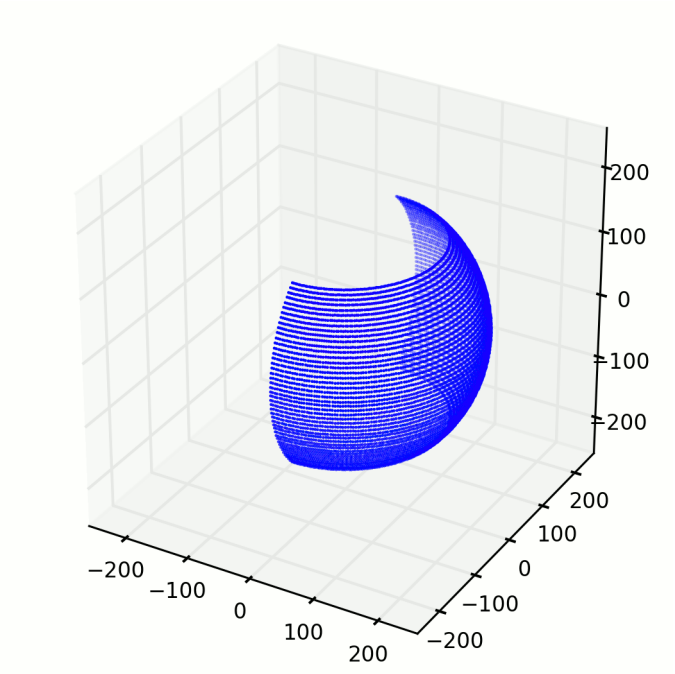


Figura 5.1: Visualização gráfica da saída do programa `generate_sphere_vectors.py`

```
topleft
172     69      75
topright
172     -69     75
bottomright
172     -69     -75
bottomleft
172     69      -75
198     24      -3                      0.348924 0.517374
198     -3      24                      0.518884 0.36101
190     -44     45                      0.788635 0.228421
```

Listagem 5.2: Entrada para o programa `convert_coordinates.py` e a saída correspondente

A terceira ferramenta foi chamada de `draw_points.py` e serve para visualizar graficamente as coordenadas 2D. Esta ferramenta desenha um ponto na tela para cada par de coordenadas lido da entrada padrão. A figura 5.2 mostra a saída desta ferramenta para os três pontos da listagem 5.2.

Essas três ferramentas podem ser usadas separadamente ou encadeadas através de um *pipeline*[1].

[1] *Pipeline* significa ligar a saída padrão de um programa à entrada padrão do programa seguinte.

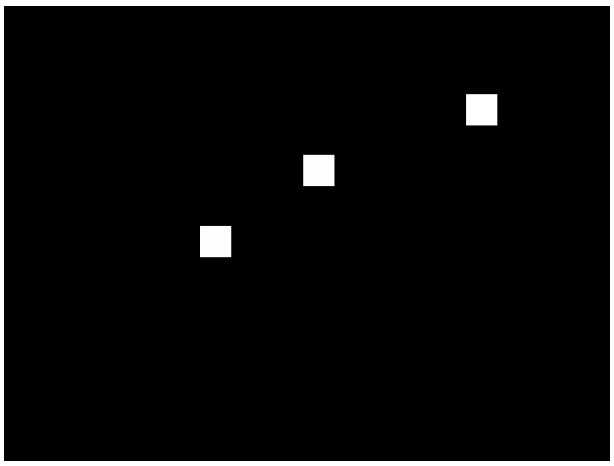


Figura 5.2: Exemplo de saída do programa `draw_points.py`

5.2 Transformação usando geometria

Para esta abordagem é necessário calibrar os quatro cantos da tela, i.e., definir quatro vetores no espaço tais que:

- O vetor A aponta na direção do canto superior esquerdo da tela.
- O vetor B aponta na direção do canto superior direito da tela.
- O vetor C aponta na direção do canto inferior direito da tela.
- O vetor D aponta na direção do canto inferior esquerdo da tela.

Também é definido o vetor P como sendo a direção apontada pelo usuário. O objetivo da transformação de coordenadas é mapear o vetor P em coordenadas (x, y) contidas no plano da tela do computador. As magnitudes dos vetores não são importantes, apenas a direção é considerada. Todos esses vetores podem ser vistos na figura 5.3

Uma vez definido o problema, a etapa seguinte foi quebrá-lo em subproblemas. Em vez de tentar calcular diretamente as coordenadas 2D relativas ao vetor P, calcula-se a projeção de P em cada uma das bordas da tela, para a seguir interpolar as coordenadas (x, y) a partir das bordas, conforme a figura 5.4. Essa decisão foi tomada porque o quadrilátero $ABCD$ não é necessariamente um retângulo.

Para calcular essa projeção de P, primeiramente calcula-se o vetor unitário N (5.1), normal ao plano de A e B. São então calculados P_N (5.2), a componente de P paralela ao vetor N; e P' (5.3), a componente de P contida no plano de A e B. Isso é ilustrado na figura 5.5.

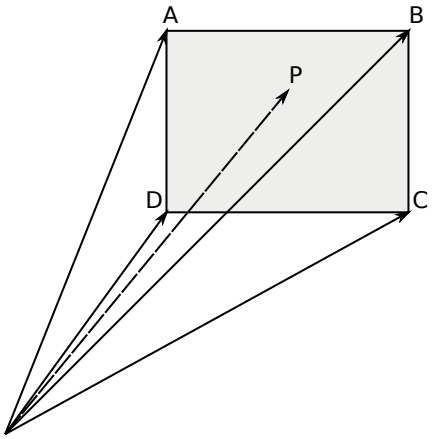


Figura 5.3: Definição dos vetores 3D

$$N = \frac{A \times B}{||A \times B||} \tag{5.1}$$

$$P_N = (N \cdot P)N \tag{5.2}$$

$$P' = P - P_N \tag{5.3}$$

Uma vez tendo P', a etapa final é calcular sua posição na borda delimitada por A e B. Essa posição é definida como um escalar α entre 0 e 1, sendo que 0 indica que P' é coincidente com o vetor A, e 1 indica que P' é coincidente com o vetor B. Valores fora desse intervalo indicam que P' está fora do setor angular delimitado por A e B.

Como A, B e P' estão no mesmo plano, o valor exato de α pode ser calculado através de um sistema linear de duas variáveis α e β, o qual pode ser representado com a equação vetorial (5.4).

$$A + \alpha(B - A) = \beta P' \tag{5.4}$$

As bases do espaço bidimensional correspondente ao plano dos vetores A, B e P' podem ser definidas como os vetores X e Y ortonormais (5.5). Todos os pontos desse plano podem ser representados por uma combinação linear entre os vetores X e Y.

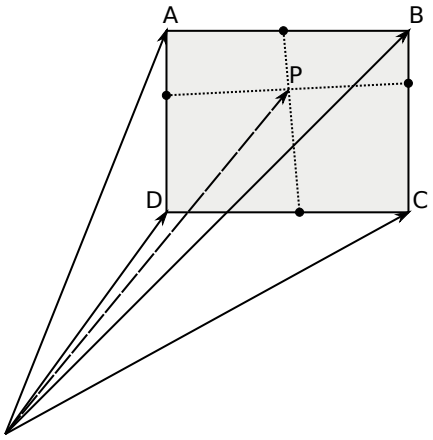


Figura 5.4: As coordenadas 2D podem ser interpoladas a partir da projeção de P em cada borda

$$X = \frac{A}{||A||} \qquad Y = \frac{A \times N}{||A \times N||} \tag{5.5}$$

Definidas as bases do espaço bidimensional, as componentes de A, $(B-A)$ e P' nesse espaço podem ser calculadas usando produto interno. (5.6)

$$\begin{aligned} A_X &= A \cdot X & A_Y &= A \cdot Y \\ (B-A)_X &= (B-A) \cdot X & (B-A)_Y &= (B-A) \cdot Y \\ P'_X &= P' \cdot X & P'_Y &= P' \cdot Y \end{aligned} \tag{5.6}$$

A equação vetorial (5.4) pode ser reescrita como duas equações escalares (5.7), utilizando as componentes definidas em (5.6).

$$\begin{aligned} A_X + \alpha(B-A)_X &= \beta P'_X \\ A_Y + \alpha(B-A)_Y &= \beta P'_Y \end{aligned} \tag{5.7}$$

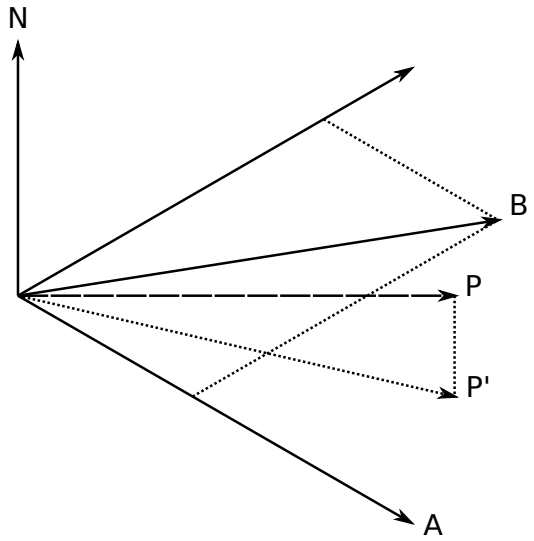


Figura 5.5: O vetor P' é a componente de P contida no plano de A e B

Por fim, resolvendo o sistema linear definido em (5.4) e (5.7), obtemos o valor exato de α correspondente a uma das bordas da tela.

Repetindo os cálculos para as outras bordas, podemos calcular (x, y) através de uma interpolação linear (5.8).

$$
\begin{aligned}
x(y) &= y(DC - AB) + AB \\
y(x) &= x(BC - AD) + AD \\
x &= \frac{AD(DC - AB) + AB}{1 - (BC - AD)(DC - AB)}
\end{aligned}
\tag{5.8}
$$

onde

$$AB, BC, DC, AD = \text{valor de } \alpha \text{ para cada borda}$$

Além do cálculo do valor exato de α, também foram experimentados cálculos aproximados para α, usando razão entre ângulos (5.9), razão entre distâncias (5.10), razão entre senos (5.11), razão entre cossenos (5.12), razão entre tangentes (5.13). Os resultados de cada abordagem estão listados na seção 5.4.

$$\alpha = \frac{\text{ang}(A, P')}{\text{ang}(A, B)} \tag{5.9}$$

$$\alpha = \frac{\left|\frac{A}{|A|} - \frac{P'}{|P'|}\right|}{\left|\frac{A}{|A|} - \frac{B}{|B|}\right|} \tag{5.10}$$

$$\alpha = \frac{\sin(\text{ang}(A, P'))}{\sin(\text{ang}(A, B))} \tag{5.11}$$

$$\alpha = \frac{\cos(\text{ang}(A, P'))}{\cos(\text{ang}(A, B))} \tag{5.12}$$

$$\alpha = \frac{\tan(\text{ang}(A, P'))}{\tan(\text{ang}(A, B))} \tag{5.13}$$

O ângulo entre dois vetores pode ser calculado como mostrado em (5.14).

$$\begin{aligned} \text{ang}(A, B) &= \arccos(\cos(\theta)) \\ &= \arccos\left(\frac{A \cdot B}{|A||B|}\right) \end{aligned} \tag{5.14}$$

5.3 Transformação usando sistema de equações lineares

Para esta abordagem é necessário calibrar três dos quatro cantos da tela. São definidos três vetores no espaço tais que:

- O vetor A aponta na direção do canto superior esquerdo da tela.
- O vetor B aponta na direção do canto superior direito da tela.
- O vetor D aponta na direção do canto inferior esquerdo da tela.

Também é definido o vetor P como sendo a direção apontada pelo usuário. Esta abordagem transforma o vetor P diretamente em coordenadas (x, y) contidas no plano da tela do computador através da solução de um sistema linear de três variáveis.

Esse sistema decorre da observação de que qualquer ponto da tela pode ser representado como ponto A somado de um deslocamento horizontal na direção $B - A$ e um deslocamento vertical na direção $D - A$, conforme figura 5.6.

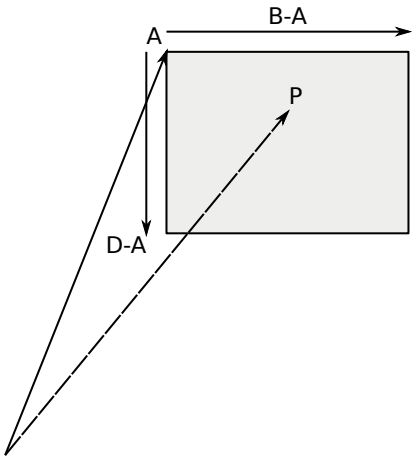


Figura 5.6: O vetor P pode ser representado como $A + x(B - A) + y(D - A)$

O sistema pode ser representado na forma vetorial conforme equações (5.15) e (5.16). As coordenadas (x, y), já no plano da tela do computador, são a solução desse sistema. A componente z é descartada.

$$A + x(B - A) + y(D - A) = zP \tag{5.15}$$

$$x(B - A) + y(D - A) - zP = -A \tag{5.16}$$

5.4 Resultados

As abordagens descritas nas seções 5.2 e 5.3 foram testadas experimentalmente usando o conjunto de ferramentas da seção 5.1.

Cada uma das abordagens foi testada cinco vezes usando o conjunto de vetores gerados pela ferramenta `generate_sphere_vectors.py` – ferramenta descrita na seção 5.1, cujos vetores estão representados na figura 5.1 –, variando apenas a calibração do cantos da tela entre cada teste. Isso permitiu analisar o comportamento de cada algoritmo sob diferentes condições.

Os vetores de calibração escolhidos para cada teste possuem aberturas de 30°, 45°, 60°, 75° e 85°, tanto na latitude como na longitude. Essa abertura é ilustrada na figura 5.7.

As figuras de 5.8 a 5.14 mostram o resultado de cada abordagem sendo testada com cada uma das cinco aberturas citadas. Foram produzidas encadeando as três ferramentas auxiliares

descritas na seção 5.1 e mostram as deformações introduzidas por cada abordagem, lembrando que o espaço entre os pontos é de 1° horizontalmente e 2° verticalmente.

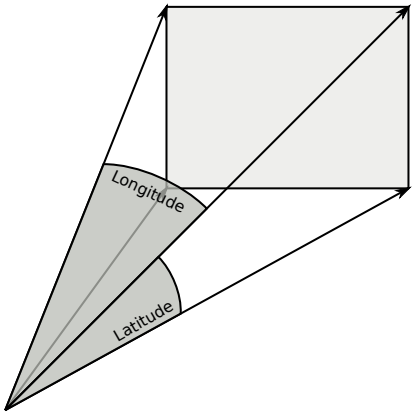


Figura 5.7: Os dois ângulos de abertura para os vetores de calibração

É possível observar que, com exceção da razão entre cossenos (5.12), todas as abordagens tiveram um bom resultado para aberturas pequenas, de até 30° ou 45°. As distorções somente são acentuadas para aberturas maiores, a partir de 30° ou 60°.

A razão entre ângulos (5.9) e a razão entre distâncias (5.10) apresentaram comportamentos muito parecidos. A região central da tela sofreu pouca deformação, enquanto as regiões próximas das bordas apresentaram distorções bastante acentuadas.

A razão entre senos (5.11) apresentou um comportamento próximo da razão entre ângulos e da razão entre distâncias, porém com deformações mais acentuadas tanto na região central como na região periférica.

As coordenadas transformadas usando a razão entre cossenos (5.12) apresentaram uma rotação de 15° a 25° em todas as aberturas testadas. Por outro lado, foi a abordagem com menor

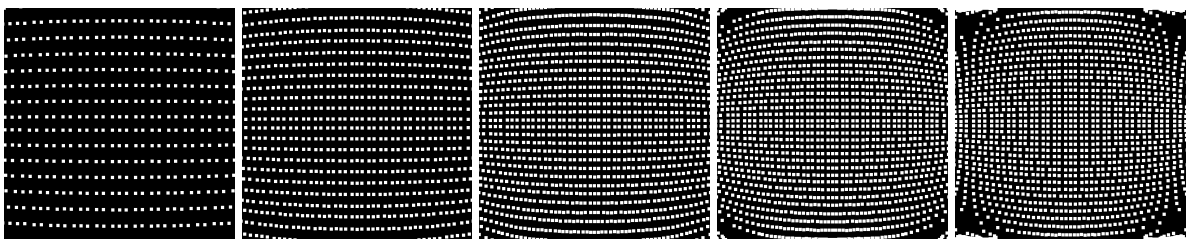


Figura 5.8: Cálculo exato de α

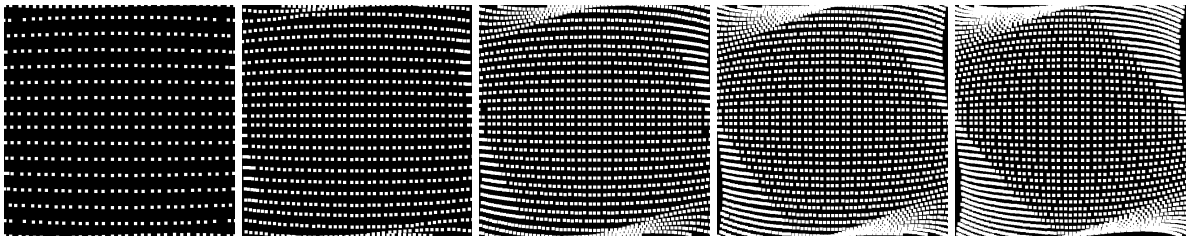


Figura 5.9: Razão entre ângulos

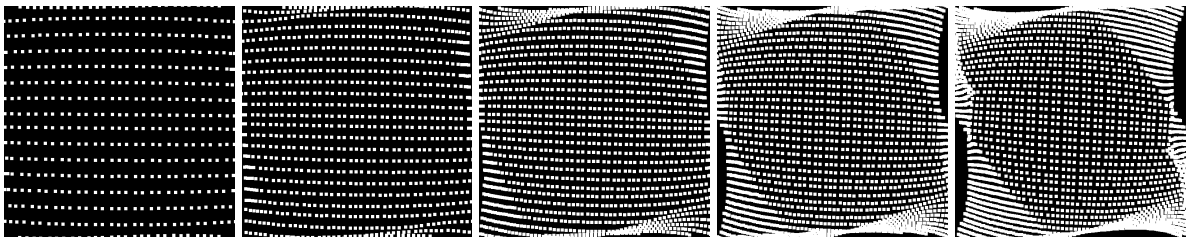


Figura 5.10: Razão entre distâncias

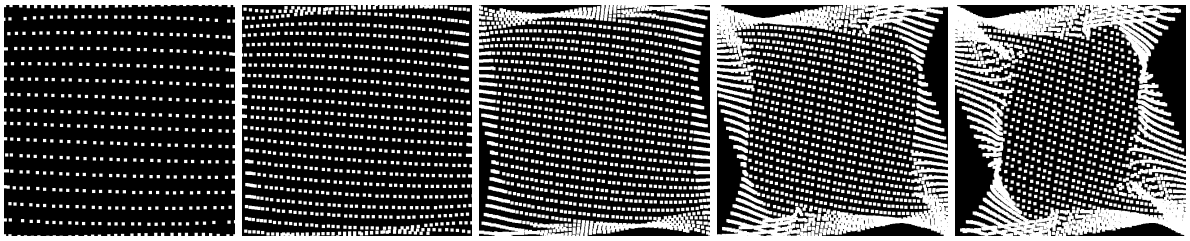


Figura 5.11: Razão entre senos

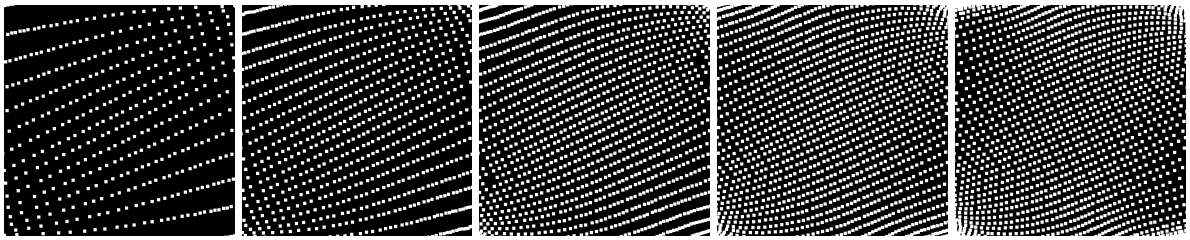


Figura 5.12: Razão entre cossenos

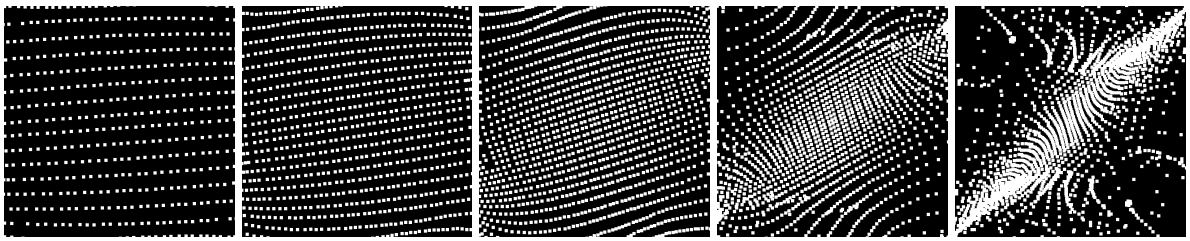


Figura 5.13: Razão entre tangentes

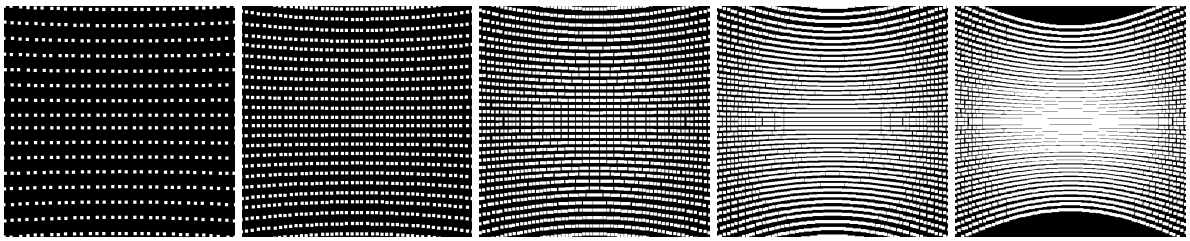


Figura 5.14: Sistema linear de 3 variáveis

distorção nas bordas.

A razão entre tangentes (5.13) só obteve bons resultados para aberturas pequenas. Para aberturas maiores, as distorções são extremamente acentuadas. Isso não é surpresa, visto que a função tangente tende para o infinito quando a abertura tende para 90°.

O cálculo exato do escalar α obteve o melhor resultado geral, apresentando muito pouca deformação na região central assim como uma área de distorções periféricas relativamente pequena, significativamente menor que a área das outras abordagens.

Por fim, a abordagem usando um sistema linear de três equações (5.14) apresentou distorções cada vez mais acentuadas conforme os ângulos de abertura ficavam maiores. Essas distorções ocorrem por se mapear uma região da esfera num plano, e portanto são relativamente pequenas quando a região da esfera é suficientemente reduzida.

6 Conclusões

" *This was a triumph.*
I'm making a note here:
HUGE SUCCESS.
It's hard to overstate
my satisfaction. "
Jonathan Coulton
Música final do jogo *Portal*

Neste capítulo são apresentados os resultados alcançados, as limitações encontradas e sugestões para trabalhos futuros.

6.1 Resultados alcançados

De maneira geral, o projeto obteve grande sucesso em seus objetivos. O dispositivo funciona como esperado e é fácil de usar.

O *boot loader* (seção 4.3) se mostrou extremamente prático durante o desenvolvimento, e é uma excelente ferramenta para ser usada em projetos futuros.

O *driver* V-USB (seção 4.6) se confirmou como uma opção bastante prática para desenvolver dispositivos USB *low speed*. A implementação de um USB HID neste projeto funcionou como esperado: o comportamento foi *plug-and-play*, funcionando imediatamente nos sistemas operacionais testados – Linux, Mac OS X e Windows – sem precisar de qualquer intervenção manual.

A escolha de um microcontrolador AVR também foi bem sucedida, considerando a quantidade e a qualidade das ferramentas disponíveis para essa arquitetura.

Por outro lado, a escolha do microcontrolador ATmega8 foi um pouco restritiva devido à sua memória *Flash* relativamente pequena. Foi necessário desabilitar partes do *firmware* durante o desenvolvimento para que ele pudesse coexistir com o *boot loader* na memória. Ao final do projeto, alguns itens menos importantes do menu de configurações foram desabilitados para que o *firmware* pudesse caber inteiro na memória.

Apesar da implementação de um menu (seção 4.8) ter aumentado substancialmente o tamanho do *firmware*, ele foi fundamental para tornar intuitiva a configuração dos parâmetros do dispositivo. Dado que se tenha espaço suficiente, pode-se até incluir instruções de uso no sistema de menus, dispensando quase que totalmente a necessidade de um manual de instruções para o usuário.

Limitações nas medidas do sensor puderam ser observadas como oscilações na posição do ponteiro do *mouse*. Mesmo depois de implementado um algoritmo de suavização (seção 4.9.1), o ponteiro ainda oscilava de maneira perceptível, embora com amplitude bastante reduzida.

Por fim, o uso individual de um magnetômetro para controlar o ponteiro do *mouse* possui suas limitações. Como o campo magnético da Terra é aproximadamente paralelo à superfície (exceto nos pólos), o dispositivo deste projeto só funciona satisfatoriamente se o usuário estiver de frente para o Norte ou para o Sul. Caso o usuário esteja de frente para o Leste ou Oeste, os movimentos verticais serão altamente comprometidos. Isso acontece porque esse tipo de movimento será provocado por uma rotação do sensor num eixo paralelo ao campo magnético, e esse tipo de movimento não causa variação na leitura do sensor (ou causa uma variação muito

pequena).

6.2 Trabalhos futuros

Apesar deste projeto funcionar como esperado, ainda pode ser refinado de várias formas.

A comunicação com o sensor foi implementada usando um conjunto de fios e o protocolo I^2C. Substituir essa interface por uma comunicação sem fio tornaria o dispositivo muito mais prático para o usuário.

O sensor usado neste projeto, conforme descrito na seção 4.5.1, permite no máximo 75 medições por segundo no modo de medição contínua. Seria possível alcançar até 160 medições por segundo caso a PCB onde o sensor foi colocado tivesse um contato para o pino DRDY, como mostra a figura 6.1.

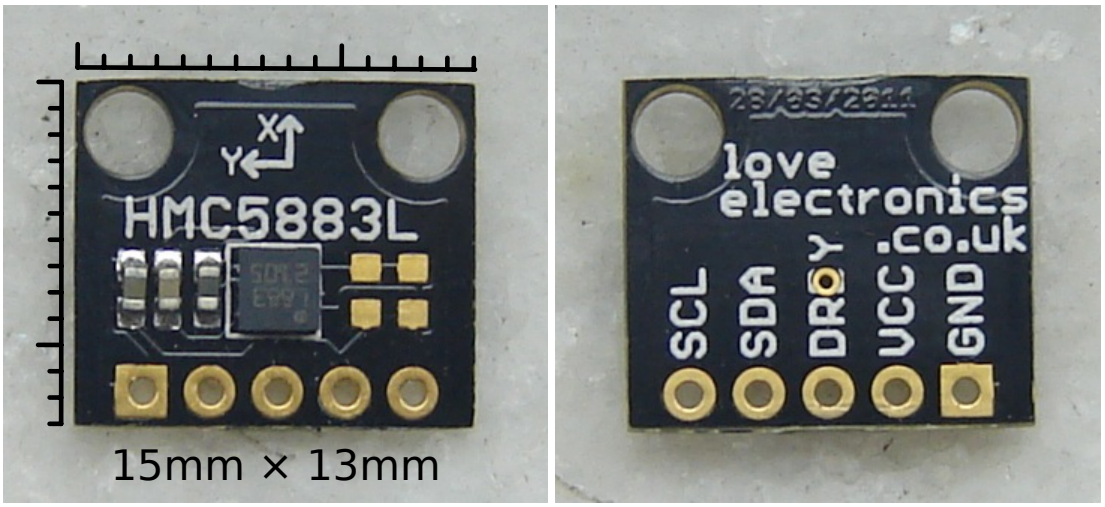


Figura 6.1: Fotos de uma PCB com cinco contatos disponíveis

Outra opção de trabalho futuro é trocar o sensor HMC5883L por algum outro com medições mais precisas. Isso tem potencial de diminuir as oscilações do ponteiro, aumentando a precisão durante o uso do dispositivo.

Podem ainda ser feitos estudos teóricos e experimentais sobre diferentes algoritmos de transformação de coordenadas. Com um microcontrolador com mais memória, seria possível implementar algoritmos mais complexos.

Finalmente, a adição de um segundo sensor do tipo acelerômetro ou giroscópio eliminaria a restrição de estar virado para o Norte ou para o Sul. Seria necessário estudar novas abordagens para a transformação de coordenadas, visto que as abordagens investigadas neste projeto utilizam dados vindos de apenas um sensor.

Referências Bibliográficas

[1] Marcin Wichary; Ryan Germick. *Google I/O 2011: The Secrets of Google Pac-Man: A Game Show*. 11 maio 2011. Trecho do vídeo entre 2min46s e 4min40s. Disponível em: <http://www.youtube.com/watch?v=ttavBa4giPc>. Acesso em: 2011-10-27.

[2] IPOD touch - Apple Store (Brasil). Disponível em: <http://store.apple.com/br/browse/home/shop_ipod/family/ipod_touch>. Acesso em: 2011-11-02.

[3] Wikipedia. *Wiimote*. Disponível em: <http://en.wikipedia.org/wiki/Wiimote>. Acesso em: 2011-11-02.

[4] NaturalPoint. *SmartNav*. Disponível em: <http://www.naturalpoint.com/smartnav/>. Acesso em: 2011-11-02.

[5] NaturalPoint. *TrackIR*. Disponível em: <http://www.naturalpoint.com/trackir/>. Acesso em: 2011-11-02.

[6] Origin Instruments Corporation. *HeadMouse*. Disponível em: <http://www.orin.com/access/headmouse/>. Acesso em: 2011-11-02.

[7] Jadon Clews. *WiSHABI - Wireless, Single Handed, Accelerometer-Based, USB-HID Compliant, PC Interface*. 31 out. 2008. Disponível em: <http://vusb.wikidot.com/project:wishabi>. Acesso em: 2011-11-02.

[8] Skyler Schneider, J. *USB Wireless Tilt Mouse + Minesweeper*. 2010. Disponível em: <http://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2010/ss868_jfe5/ss868_jfe5/>. Acesso em: 2011-11-02.

[9] Till Harbaum. *USB TiltStick*. 2008. Versão 20080207. Disponível em: <http://www.harbaum.org/till/tiltstick/index.shtml>. Acesso em: 2011-10-30.

[10] Atmel Corporation. *ATmega8 datasheet*. 2001–2011. 9, 17–19, 21, 33, 158, 161, 168, 213, 216, 235, 282 p. Rev. 2486Z. Disponível em: <http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf>. Acesso em: 2011-10-27.

[11] Gaute Myklebust. *The AVR Microcontroller and C Compiler Co-Design*. Disponível em: <http://www.atmel.com/dyn/resources/prod_documents/COMPILER.pdf>. Acesso em: 2011-10-28.

[12] AVR Libc. *Example using the two-wire interface (TWI)*. 16 fev. 2011. Versão 1.7.1. Disponível em: <http://www.nongnu.org/avr-libc/user-manual/group__twi__demo.html>. Acesso em: 2011-10-27.

[13] ATMEL AVR 8- and 32-bit Microcontrollers > megaAVR > Overview. Disponível em: <http://www.atmel.com/dyn/products/devices.asp?category_id=163&family_id=607&subfamily_id=760>. Acesso em: 2011-10-28.

[14] Honeywell. *HMC5883L datasheet*. 2010. 3-Axis Digital Compass IC. Disponível em: <http://c48754.r54.cf3.rackcdn.com/HMC5883L.pdf>. Acesso em: 2011-10-27.

[15] HMC5883L HMC5883 Triple Axis Magnetometer Sensor board. Disponível em: <http://cgi.ebay.com/HMC5883L-HMC5883-Triple-Axis-Magnetometer-Sensor-board-/260770948278>. Acesso em: 2011-05-29.

[16] COMPAQ et al. *Universal Serial Bus Specification Revision 2.0*. 27 abr. 2000. 16, 141–142, 157, 178–180 p. Disponível em: <http://www.usb.org/developers/docs/usb_20.zip>. Acesso em: 2011-10-27.

[17] Beyond Logic. *USB in a NutShell*. 2001–2010. Disponível em: <http://www.beyondlogic.org/usbnutshell/usb1.shtml>. Acesso em: 2011-10-27.

[18] USB Implementers' Forum. *Device Class Definition for Human Interface Devices (HID)*. 27 jun. 2001. 1–2, 9–10, 20, 59–61 p. Versão 1.11. Disponível em: <http://www.usb.org/developers/devclass_docs/HID1_11.pdf>. Acesso em: 2011-10-27.

[19] Philips Semiconductors. *UM10204: I2C-bus specification and user manual*. 19 jun. 2007. 6, 13 p. Rev. 03. Disponível em: <http://www.nxp.com/documents/user_manual/UM10204.pdf>. Acesso em: 2011-10-28.

[20] Atmel Corporation. *AVR315: Using the TWI module as I2C master*. 2010. 3, 13 p. Disponível em: <http://www.atmel.com/dyn/resources/prod_documents/doc2564.pdf>. Acesso em: 2011-10-27.

[21] Atmel Corporation. *ATmega48A/PA/88A/PA/168A/PA/328/P datasheet*. 2009–2011. 322 p. Rev. 8271D. Disponível em: <http://www.atmel.com/dyn/resources/prod_documents/doc8271.pdf>. Acesso em: 2011-10-28.

[22] Thomas Fischl. *USBasp - USB programmer for Atmel AVR controllers*. 2005–2011. Disponível em: <http://www.fischl.de/usbasp/>. Acesso em: 2011-10-27.

[23] Objective Development Software GmbH. *V-USB - A Firmware-Only USB Driver for Atmel AVR Microcontrollers*. 2008–2010. Versão 20100715. Disponível em: <http://www.obdev.at/products/vusb/index.html>. Acesso em: 2011-10-27.

[24] Microchip Technology Inc. *3V Tips 'n Tricks*. 2008. Disponível em: <http://www.microchip.com/stellent/groups/techpub_sg/documents/devicedoc/en026368.pdf>. Acesso em: 2011-10-27.

[25] Herman Schutte. *Bi-directional level shifter for I2C-bus and other systems*. 4 ago. 1997. 9–10 p. Disponível em: <http://www.kip.uni-heidelberg.de/lhcb/Publications/external/AN97055.pdf>. Acesso em: 2011-10-27.

[26] NXP Semiconductors. *Level shifting techniques in I2C-bus design*. 18 jun. 2007. 4 p. Disponível em: <http://www.nxp.com/documents/application_note/AN10441.pdf>. Acesso em: 2011-10-27.

[27] AVR Libc. *Toolchain Overview*. 16 fev. 2011. Versão 1.7.1. Disponível em: <http://www.nongnu.org/avr-libc/user-manual/overview.html>. Acesso em: 2011-10-29.

[28] Objective Development Software GmbH. *USBaspLoader - Bootloader Emulating USBasp*. 2008–2010. Versão 20100727. Disponível em: <http://www.obdev.at/products/vusb/usbasploader.html>. Acesso em: 2011-10-27.

[29] AVR Libc. *Porting From IAR to AVR GCC*. 16 fev. 2011. Versão 1.7.1. Disponível em: <http://www.nongnu.org/avr-libc/user-manual/porting.html>. Acesso em: 2011-10-27.

[30] Bureau International des Poids et Mesures. *The International System of Units (SI)*. 2006. 128 p. 8ª ed. Disponível em: <http://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf>. Acesso em: 2011-10-30.

[31] Objective Development Software GmbH. *All Projects Based on V-USB*. Disponível em: <http://www.obdev.at/products/vusb/prjall.html>. Acesso em: 2011-10-30.

[32] USB Implementers' Forum. *USB-IF Logo Trademark License Agreement and Usage Guidelines*. Disponível em: <http://www.usb.org/developers/logo_license/>. Acesso em: 2011-10-30.

[33] USB Implementers' Forum. *Getting a Vendor ID*. Disponível em: <http://www.usb.org/developers/vendor/>. Acesso em: 2011-10-30.

[34] DRIVER API - V-USB. 21 abr. 2009. Disponível em: <http://vusb.wikidot.com/driver-api>. Acesso em: 2011-10-30.

[35] Objective Development Software GmbH. *HIDKeys - An Example USB HID*. 2006–2007. Versão 20070329. Disponível em: <http://www.obdev.at/products/vusb/hidkeys.html>. Acesso em: 2011-10-30.

[36] HID Descriptor Tool. Disponível em: <http://www.usb.org/developers/hidpage/dt2_4.zip>. Acesso em: 2011-10-27.

[37] USB Implementers' Forum. *HID Usage Tables*. 28 out. 2004. 53 p. Versão 1.12. Disponível em: <http://www.usb.org/developers/devclass_docs/Hut1_12v2.pdf>. Acesso em: 2011-10-27.

[38] AVR Libc. *Frequently Asked Questions*. 16 fev. 2011. Versão 1.7.1. Disponível em: <http://www.nongnu.org/avr-libc/user-manual/FAQ.html>. Acesso em: 2011-10-27.

[39] J. Elonen. *Simple Gauss-Jordan elimination in Python*. abr. 2005. Reimplementado em C. Disponível em: <http://elonen.iki.fi/code/misc-notes/python-gaussj/index.html>. Acesso em: 2011-11-02.

[40] Denilson Figueiredo de Sá. *Linux USB HID should ignore values outside Logical Minimum/Maximum range*. 22 out. 2011. Disponível em: <https://lkml.org/lkml/2011/10/22/142>. Acesso em: 2011-11-02.

[41] Jiri Kosina. *Re: Linux USB HID should ignore values outside Logical Minimum/Maximum range*. 31 out. 2011. Disponível em: <https://lkml.org/lkml/2011/10/31/125>. Acesso em: 2011-11-02.

[42] AVR Libc. *<avr/eeprom.h>: EEPROM handling*. 16 fev. 2011. Versão 1.7.1. Disponível em: <http://www.nongnu.org/avr-libc/user-manual/group__avr__eeprom.html>. Acesso em: 2011-11-02.

[43] Atmel Corporation. *AVR104: Buffered Interrupt Controlled EEPROM Writes*. 2003. Disponível em: <http://www.atmel.com/dyn/resources/prod_documents/doc2540.pdf>. Acesso em: 2011-10-27.

APÊNDICE A – main.c

```
1  /* Name: main.c
2   * Project: atmega8-magnetometer-usb-mouse
3   * Author: Denilson Figueiredo de Sa
4   * Creation Date: 2011-08-29
5   * Tabsize: 4
6   * License: GNU GPL v2 or GNU GPL v3
7   *
8   * Includes third-party code:
9   *
10  * - V-USB from OBJECTIVE DEVELOPMENT Software GmbH
11  *   http://www.obdev.at/products/vusb/index.html
12  *
13  * - USBaspLoader from OBJECTIVE DEVELOPMENT Software GmbH
14  *   http://www.obdev.at/products/vusb/usbasploader.html
15  *
16  * - AVR315 TWI Master Implementation from Atmel
17  *
       http://www.atmel.com/dyn/products/documents.asp?category_id=163&family_id=607&sub
18  */
19
20 // Headers from AVR-Libc
21 #include <avr/io.h>
22 #include <avr/interrupt.h>
23 #include <avr/eeprom.h>
24 #include <avr/pgmspace.h>
25 #include <avr/wdt.h>
26 #include <util/delay.h>
27
28 // V-USB driver from http://www.obdev.at/products/vusb/
29 #include "usbdrv.h"
30
31 // It's also possible to include "usbdrv.c" directly, if we also add
32 // this definition at the top of this file:
33 // #define USB_PUBLIC static
34 // However, this only saved 10 bytes.
35
36 // I'm not using serial-line debugging
37 //#include "oddebug.h"
38
39 // AVR315 Using the TWI module as I2C master
40 #include "avr315/TWI_Master.h"
41
42 // Non-blocking interrupt-based EEPROM writing.
43 #include "int_eeprom.h"
44
45 // Sensor communication over I2C (TWI)
46 #include "sensor.h"
47
48 // Button handling code
49 #include "buttons.h"
50
51
52 #if ENABLE_KEYBOARD
53
54 // Keyboard emulation code
55 #include "keyemu.h"
```

```
56 // Menu user interface for configuring the device
57 #include "menu.h"
58
59 #endif
60
61
62 #if ENABLE_MOUSE
63
64 // Mouse emulation code
65 #include "mouseemu.h"
66
67 #endif
68
69
70 ////////////////////////////////////////////////////
71 // Hardware description                        {{{
72
73 /* ATmega8 pin assignments:
74  * PB0: (not used)
75  * PB1: (not used)
76  * PB2: (not used)
77  * PB3: (not used - MOSI)
78  * PB4: (not used - MISO)
79  * PB5: (not used - SCK)
80  * PB6: 12MHz crystal
81  * PB7: 12MHz crystal
82  *
83  * PC0: Button 1
84  * PC1: Button 2
85  * PC2: Button 3
86  * PC3: Switch button (if held to GND during power-on, starts the
      bootloader)
87  * PC4: I2C - SDA
88  * PC5: I2C - SCL
89  * PC6: Reset pin (with an external 10K pull-up to VCC)
90  *
91  * PD0: USB-
92  * PD1: (not used - debug tx)
93  * PD2: USB+ (int0)
94  * PD3: (not used)
95  * PD4: (not used)
96  * PD5: red debug LED
97  * PD6: yellow debug LED
98  * PD7: green debug LED
99  *
100 * If you change the ports, remember to update:
101 * - main.c: hardware_init()
102 * - buttons.c: update_button_state()
103 * - buttons.h and menu.c: BUTTON_* definitions
104 * - mouseemu.c: mouse_update_buttons()
105 *
106 * If the Switch is ON, runs the "mouseemu" code. In this mode, buttons 1,
107 * 2 and 3 emulate mouse click.
108 * If the Switch is OFF, runs the "keyemu" code. In this mode, there is an
109 * interactive menu system to configure and debug the sensor. Buttons 1 and
110 * 2 are next/prev item, and button 3 is "confirm".
111 */
112
113 #define LED_TURN_ON(led)  do { PORTD |=  (led); } while(0)
114 #define LED_TURN_OFF(led) do { PORTD &= ~(led); } while(0)
115 #define LED_TOGGLE(led)   do { PORTD ^=  (led); } while(0)
116
117 // Bit masks for each LED (in PORTD)
118 #define RED_LED    (1 << 5)
119 #define YELLOW_LED (1 << 6)
120 #define GREEN_LED  (1 << 7)
121 #define ALL_LEDS   (GREEN_LED | YELLOW_LED | RED_LED)
122
123 // }}}
124
125 ////////////////////////////////////////////////////
```

```
126 // USB HID Report Descriptor                                {{{
127
128 // If this HID report descriptor is changed, remember to update
129 // USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH from usbconfig.h
130 PROGMEM char usbHidReportDescriptor[USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH]
131 __attribute__((externally_visible))
132 = {
133     // Keyboard
134     0x05, 0x01,             // USAGE_PAGE (Generic Desktop)
135     0x09, 0x06,             // USAGE (Keyboard)
136     0xa1, 0x01,             // COLLECTION (Application)
137     0x85, 0x01,             //   REPORT_ID (1)
138     // Modifier keys (they must come BEFORE the real keys)
139     0x05, 0x07,             //   USAGE_PAGE (Keyboard)
140     0x19, 0xe0,             //   USAGE_MINIMUM (Keyboard LeftControl)
141     0x29, 0xe7,             //   USAGE_MAXIMUM (Keyboard Right GUI)
142     0x15, 0x00,             //   LOGICAL_MINIMUM (0)
143     0x25, 0x01,             //   LOGICAL_MAXIMUM (1)
144     0x75, 0x01,             //   REPORT_SIZE (1)
145     0x95, 0x08,             //   REPORT_COUNT (8)
146     0x81, 0x02,             //   INPUT (Data,Var,Abs)
147     // Normal keys
148 //  0x05, 0x07,             //   USAGE_PAGE (Keyboard)
149     0x19, 0x00,             //   USAGE_MINIMUM (Reserved (no event
        indicated))
150     0x29, 0x65,             //   USAGE_MAXIMUM (Keyboard Application)
151 //  0x15, 0x00,             //   LOGICAL_MINIMUM (0)
152     0x25, 0x65,             //   LOGICAL_MAXIMUM (101)
153     0x75, 0x08,             //   REPORT_SIZE (8)
154     0x95, 0x01,             //   REPORT_COUNT (1)
155     0x81, 0x00,             //   INPUT (Data,Ary,Abs)
156     0xc0,                   // END_COLLECTION
157
158     // Mouse
159     0x05, 0x01,             // USAGE_PAGE (Generic Desktop)
160     0x09, 0x02,             // USAGE (Mouse)
161     0xa1, 0x01,             // COLLECTION (Application)
162     0x85, 0x02,             //   REPORT_ID (2)
163 //  0x05, 0x01,             //   USAGE_PAGE (Generic Desktop)
164     0x09, 0x01,             //   USAGE (Pointer)
165     0xa1, 0x00,             //   COLLECTION (Physical)
166     // X, Y movement
167 //  0x05, 0x01,             //     USAGE_PAGE (Generic Desktop)
168     0x09, 0x30,             //     USAGE (X)
169     0x09, 0x31,             //     USAGE (Y)
170 //  0x15, 0x00,             //     LOGICAL_MINIMUM (0)
171     0x26, 0xff, 0x7f,       //     LOGICAL_MAXIMUM (32767)
172 //  0x35, 0x00,             //     PHYSICAL_MINIMUM (0)
173 //  0x46, 0xff, 0x7f,       //     PHYSICAL_MAXIMUM (32767)
174     0x75, 0x10,             //     REPORT_SIZE (16)
175     0x95, 0x02,             //     REPORT_COUNT (2)
176     0x81, 0x42,             //     INPUT (Data,Var,Abs,Null)
177     0xc0,                   //   END_COLLECTION
178     // Buttons
179     0x05, 0x09,             //   USAGE_PAGE (Button)
180     0x19, 0x01,             //   USAGE_MINIMUM (Button 1)
181     0x29, 0x03,             //   USAGE_MAXIMUM (Button 3)
182 //  0x15, 0x00,             //   LOGICAL_MINIMUM (0)
183     0x25, 0x01,             //   LOGICAL_MAXIMUM (1)
184     0x75, 0x01,             //   REPORT_SIZE (1)
185     0x95, 0x03,             //   REPORT_COUNT (3)
186     0x81, 0x02,             //   INPUT (Data,Var,Abs)
187     // Padding for the buttons
188 //  0x75, 0x01,             //   REPORT_SIZE (1)
189     0x95, 0x05,             //   REPORT_COUNT (5)
190     0x81, 0x03,             //   INPUT (Cnst,Var,Abs)
191     0xc0                    // END_COLLECTION
192 };
193
194 // This device does not support BOOT protocol from HID specification.
```

```
195 //
196 // The keyboard portion is very limited, when compared to actual keyboards,
197 // but it's perfect for a simple communication from the firmware to the
198 // user.  It supports the common keyboard modifiers (although the firmware
199 // only uses the left shift), and supports only one key at time. This is
200 // enough for writing the "menu" interface, and uses only 2 bytes (plus the
201 // report ID).
202 //
203 // The mouse portion is actually an absolute pointing device, and not a
204 // standard mouse (that instead sends relative movements). It supports 2
205 // axes (X and Y) with 16-bit for each one, although it doesn't use the
206 // full 16-bit range. It also has 3 buttons. That means 2+2+1=5 bytes for
207 // the report (plus 1 byte for the report ID).
208 //
209 // Redundant entries (such as LOGICAL_MINIMUM and USAGE_PAGE) have been
210 // commented out where possible, in order to save a few bytes.
211 //
212 // PHYSICAL_MINIMUM and PHYSICAL_MAXIMUM, when undefined, assume the same
213 // values as LOGICAL_MINIMUM and LOGICAL_MAXIMUM.
214 //
215 // Note about where the buttons are located in the Report Descriptor:
216 // The buttons are placed outside the "Physical" collection just because in
217 // this project the buttons are on the breadboard, while the sensor is more
218 // than one meter away from the buttons.
219 // However, putting these buttons inside or outside that collection makes
220 // no difference at all for the software, feel free to move them around.
221 //
222 // Also note that ENABLE_KEYBOARD and ENABLE_MOUSE options don't change the
223 // HID Descriptor. Instead, they only enable/disable the code that
224 // implements the keyboard or the mouse.
225
226 // }}}
227
228 ////////////////////////////////////////////////////
229 // Main code                                      {{{
230
231 // Disabling idle rate because it is useless here. And because it saves
        quite
232 // a few bytes.
233 #define ENABLE_IDLE_RATE 0
234
235 #if ENABLE_IDLE_RATE
236 // As defined in section 7.2.4 Set_Idle Request
237 // of Device Class Definition for Human Interface Devices (HID) version
238 // 1.11 pages 52 and 53 (or 62 and 63) of HID1_11.pdf
239 //
240 // Set/Get IDLE defines how long the device should keep "quiet" if the
241 // state has not changed.
242 // Recommended default value for keyboard is 500ms, and infinity for
243 // joystick and mice.
244 //
245 // This value is measured in multiples of 4ms.
246 // A value of zero means indefinite/infinity.
247 static uchar idle_rate;
248 #endif
249
250 static void hardware_init(void) {  // {{{
251     // Configuring Watchdog to about 2 seconds
252     // See pages 43 and 44 from ATmega8 datasheet
253     // See also
            http://www.nongnu.org/avr-libc/user-manual/group__avr__watchdog.html
254     wdt_enable(WDTO_2S);
255
256     PORTB = 0xff;  // activate all pull-ups
257     DDRB = 0;      // all pins input
258     PORTC = 0xff;  // activate all pull-ups
259     DDRC = 0;      // all pins input
260
261     // From usbdrv.h:
262     //#define USBMASK ((1<<USB_CFG_DPLUS_BIT) | (1<<USB_CFG_DMINUS_BIT))
263
```

```
264     // activate pull-ups, except on USB lines and LED pins
265     PORTD = 0xFF ^ (USBMASK | ALL_LEDS);
266     // LED pins as output, the other pins as input
267     DDRD = 0 | ALL_LEDS;
268
269     // Doing a USB reset
270     // This is done here because the device might have been reset
271     // by the watchdog or some condition other than power-up.
272     //
273     // A reset is done by holding both D+ and D- low (setting the
274     // pins as output with value zero) for longer than 10ms.
275     //
276     // See page 145 of usb_20.pdf
277     // See also http://www.beyondlogic.org/usbnutshell/usb2.shtml
278
279     DDRD |= USBMASK;    // Setting as output
280     PORTD &= ~USBMASK;  // Setting as zero
281
282     _delay_ms(15);  // Holding this state for at least 10ms
283
284     DDRD &= ~USBMASK;   // Setting as input
285     //PORTD &= ~USBMASK;  // Pull-ups are already disabled
286
287     // End of USB reset
288
289     // Disabling Timer0 Interrupt
290     // It's disabled by default, anyway, so this shouldn't be needed
291     TIMSK &= ~(TOIE0);
292
293     // Configuring Timer0 (with main clock at 12MHz)
294     // 0 = No clock (timer stopped)
295     // 1 = Prescaler = 1     =>   0.0213333ms
296     // 2 = Prescaler = 8     =>   0.1706666ms
297     // 3 = Prescaler = 64    =>   1.3653333ms
298     // 4 = Prescaler = 256   =>   5.4613333ms
299     // 5 = Prescaler = 1024  =>  21.8453333ms
300     // 6 = External clock source on T0 pin (falling edge)
301     // 7 = External clock source on T0 pin (rising edge)
302     // See page 72 from ATmega8 datasheet.
303     // Also thanks to
            http://frank.circleofcurrent.com/cache/avrtimercalc.htm
304     TCCR0 = 3;
305
306     // I'm using Timer0 as a 1.365ms ticker. Every time it overflows, the
            TOV0
307     // flag in TIFR is set.
308
309     // I'm not using serial-line debugging
310     //odDebugInit();
311
312     LED_TURN_ON(YELLOW_LED);
313 }  // }}}
314
315
316 uchar
317 __attribute__((externally_visible))
318 usbFunctionSetup(uchar data[8]) {  // {{{
319     usbRequest_t *rq = (void *)data;
320
321     if ((rq->bmRequestType & USBRQ_TYPE_MASK) == USBRQ_TYPE_CLASS) {
322         // class request type
323
324         if (rq->bRequest == USBRQ_HID_GET_REPORT){
325             // wValue: ReportType (highbyte), ReportID (lowbyte)
326             // we only have one report type, so don't look at wValue
327
328             // This seems to be called as one of the final initialization
329             // steps of the device, after the ReportDescriptor has been
                    sent.
330             // Returns the initial state of the device.
331             LED_TURN_OFF(GREEN_LED);
332
```

```
333 #if ENABLE_KEYBOARD
334             if (rq->wValue.bytes[0] == 1) {
335                 // Keyboard report
336 
337                 // Not needed as I the struct already has sane values
338                 // build_report_from_char('\0');
339 
340                 usbMsgPtr = (void*) &keyboard_report;
341                 return sizeof(keyboard_report);
342             }
343 #endif
344 
345 #if ENABLE_MOUSE
346             if (rq->wValue.bytes[0] == 2) {
347                 // Mouse report
348                 usbMsgPtr = (void*) &mouse_report;
349                 return sizeof(mouse_report);
350             }
351 #endif
352 
353 #if ENABLE_IDLE_RATE
354         } else if (rq->bRequest == USBRQ_HID_GET_IDLE) {
355             usbMsgPtr = &idle_rate;
356             return 1;
357 
358         } else if (rq->bRequest == USBRQ_HID_SET_IDLE) {
359             idle_rate = rq->wValue.bytes[1];
360 #endif
361         }
362 
363     } else {
364         /* no vendor specific requests implemented */
365     }
366     return 0;
367 }  // }}}
368 
369 
370 void
371 __attribute__ ((noreturn))
372 main(void) {  // {{{
373     uchar sensor_probe_counter = 0;
374     uchar timer_overflow = 0;
375 
376 #if ENABLE_IDLE_RATE
377     int idle_counter = 0;
378 #endif
379 
380     cli();
381 
382     hardware_init();
383 
384 #if ENABLE_KEYBOARD
385     init_keyboard_emulation();
386     init_ui_system();
387 #endif
388 #if ENABLE_MOUSE
389     init_mouse_emulation();
390 #endif
391 
392     TWI_Master_Initialise();
393     usbInit();
394     init_int_eeprom();
395     init_button_state();
396 
397     wdt_reset();
398     sei();
399 
400     // Sensor initialization must be done with interrupts enabled!
401     // It uses I2C (TWI) to configure the sensor.
402     sensor_init_configuration();
403 
```

```
404     LED_TURN_ON(GREEN_LED);
405
406     for (;;) {  // main event loop
407         wdt_reset();
408         usbPoll();
409
410         if (TIFR & (1<<TOV0)) {
411             timer_overflow = 1;
412
413             // Resetting the Timer0
414             // Setting this bit to one will clear it.
415             TIFR = 1<<TOV0;
416         } else {
417             timer_overflow = 0;
418         }
419
420         update_button_state(timer_overflow);
421
422         // Red LED lights up if there is any kind of error in I2C
                communication
423         if ( TWI_statusReg.lastTransOK ) {
424             LED_TURN_OFF(RED_LED);
425         } else {
426             LED_TURN_ON(RED_LED);
427         }
428
429         // Handling the state change of the main switch
430         if (ON_KEY_UP(BUTTON_SWITCH)) {
431             // Upon releasing the switch, stop the continuous reading.
432             sensor_stop_continuous_reading();
433
434 #if ENABLE_KEYBOARD
435             // And also reset the menu system.
436             init_ui_system();
437 #endif
438         } else if (ON_KEY_DOWN(BUTTON_SWITCH)) {
439             // Upon pressing the switch, start the continuous reading for
440             // mouse emulation code.
441             sensor_start_continuous_reading();
442         }
443
444         // Continuous reading of sensor data
445         if (sensor.continuous_reading) {  // {{{
446             // Timer is set to 1.365ms
447             if (timer_overflow) {
448                 // The sensor is configured for 75Hz measurements.
449                 // I'm using this timer to read the values twice that rate.
450                 // 5 * 1.365ms = 6.827ms ~= 146Hz
451                 if (sensor_probe_counter > 0){
452                     // Waiting...
453                     sensor_probe_counter--;
454                 }
455             }
456             if (sensor_probe_counter == 0) {
457                 // Time for reading new data!
458                 uchar return_code;
459
460                 return_code = sensor_read_data_registers();
461                 if (return_code == SENSOR_FUNC_DONE || return_code ==
                        SENSOR_FUNC_ERROR) {
462                     // Restart the counter+timer
463                     sensor_probe_counter = 5;
464                 }
465             }
466         }  // }}}
467
468 #if ENABLE_IDLE_RATE
469         // Timer is set to 1.365ms
470         if (timer_overflow) {  // {{{
471             // Implementing the idle rate...
472             if (idle_rate != 0) {
```

```
473                 if (idle_counter > 0){
474                     idle_counter--;
475                 } else {
476                     // idle_counter counts how many Timer0 overflows are
477                     // required before sending another report.
478                     // The exact formula is:
479                     // idle_counter = (idle_rate * 4)/1.365;
480                     // But it's better to avoid floating point math.
481                     // 4/1.365 = 2.93, so let's just multiply it by 3.
482                     idle_counter = idle_rate * 3;
483 
484                     //keyDidChange = 1;
485                     LED_TOGGLE(YELLOW_LED);
486                     // TODO: Actually implement idle rate... Should re-send
487                     // the current status.
488                 }
489             }
490         }  // }}}
491 #endif
492 
493         // MAIN code. Code that emulates the mouse or implements the menu
494         // system.
495         if (button.state & BUTTON_SWITCH) {
496             // Code for when the switch is held down
497             // Should read data and do things
498 
499 #if ENABLE_MOUSE
500             // nothing here
501 #endif
502         } else {
503             // Code for when the switch is "off"
504             // Basically, this is the menu system (implemented as keyboard)
505 
506 #if ENABLE_KEYBOARD
507             ui_main_code();
508 #endif
509         }
510 
511         // Sending USB Interrupt-in report
512         if(usbInterruptIsReady()) {
513             if (0) {
514                 // This useless "if" is here to make all the following
515                 // conditionals an "else if", and thus making it a lot
516                 // easier to add/remove them using preprocessor directives.
517             }
518 #if ENABLE_KEYBOARD
519             else if(string_output_pointer != NULL){
520                 // Automatically send keyboard report if there is something
521                 // in the buffer
522                 send_next_char();
523                 usbSetInterrupt((void*) &keyboard_report,
                        sizeof(keyboard_report));
524             }
525 #endif
526 #if ENABLE_MOUSE
527             else if (button.state & BUTTON_SWITCH) {
528                 if (mouse_prepare_next_report()) {
529                     usbSetInterrupt((void*) &mouse_report,
                            sizeof(mouse_report));
530                 }
531             }
532 #endif
533         }
534     }
535 }  // }}}
536 
537 // }}}
538 
539 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
    foldmarker={{{,}}}
```

APÊNDICE B – buttons.h e buttons.c

```
1  /* Name: buttons.h
2   *
3   * See the .c file for more information
4   */
5
6  #ifndef __buttons_h_included__
7  #define __buttons_h_included__
8
9  #include "common.h"
10
11
12 typedef struct ButtonState {
13     // "Public" vars, already filtered for debouncing
14     uchar state;
15     uchar changed;
16
17     // This is used to "freeze" the pointer movement for a short while,
           right
18     // after a click, in order to avoid accidentally dragging the clicked
19     // object. This is useful because the sensor captures a lot of noise.
20     uchar recent_state_change;
21
22     // "Private" button debouncing state
23     uchar debouncing[4];  // We have 4 buttons/switches
24 } ButtonState;
25
26 extern ButtonState button;
27
28
29 // Bit masks for each button (in PORTC and PINC)
30 // (also used in button.state and button.changed vars)
31 #define BUTTON_1      (1 << 0)
32 #define BUTTON_2      (1 << 1)
33 #define BUTTON_3      (1 << 2)
34 #define BUTTON_SWITCH (1 << 3)
35 #define ALL_BUTTONS   (BUTTON_1 | BUTTON_2 | BUTTON_3 | BUTTON_SWITCH)
36
37
38 // Handy macros!
39 // These have the same name/meaning of JavaScript events
40 #define ON_KEY_DOWN(button_mask) ((button.changed & (button_mask)) &&
       (button.state & (button_mask)))
41 #define ON_KEY_UP(button_mask)   ((button.changed & (button_mask)) &&
       !(button.state & (button_mask)))
42
43
44 // Init does nothing
45 #define init_button_state() do{ }while(0)
46
47 void update_button_state(uchar timer_overflow);
48
49
50 #endif  // __buttons_h_included____
51
52 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
       foldmarker={{{,}}}
```

```
1  /* Name: buttons.c
2   * Project: atmega8-magnetometer-usb-mouse
3   * Author: Denilson Figueiredo de Sa
4   * Creation Date: 2011-10-18
5   * Tabsize: 4
6   * License: GNU GPL v2 or GNU GPL v3
7   */
8
9
10 #include <avr/io.h>
11
12 #include "buttons.h"
13
14
15 ButtonState button;
16
17
18 /*
19 void init_button_state() {  // {{{
20     // According to avr-libc FAQ, the compiler automatically initializes
21     // all variables with zero.
22     memset(&button, 0, sizeof(button));
23 }  // }}}
24 */
25
26
27 void update_button_state(uchar timer_overflow) { // {{{
28     // This function implements debouncing code.
29     // It should be called at every iteration of the main loop.
30
31     uchar filtered_state;
32
33     ButtonState *button_ptr = &button;
34     FIX_POINTER(button_ptr);
35
36     filtered_state = button_ptr->state;
37
38     // Timer is set to 1.365ms
39     if (timer_overflow) {
40         uchar raw_state;
41         uchar i;
42
43         // Buttons are on PC0, PC1, PC2, PC3
44         // Buttons are ON when connected to GND, and read as zero
45         // Buttons are OFF when open, internal pull-ups make them read as
               one
46
47         // The low nibble of PINC maps to the 4 buttons
48         raw_state = (~PINC) & 0x0F;
49         // "raw_state" has the button state, with 1 for pressed and 0 for
               released.
50         // Still needs debouncing...
51
52         // This debouncing solution is inspired by tiltstick-20080207
               firmware.
53         //
54         // Poll the buttons into a shift register for de-bouncing.
55         // A button is considered pressed when the register reaches 0xFF.
56         // A button is considered released when the register reaches 0x00.
57         // Any intermediate value does not change the button state.
58         //
59         // 8 * 1.365ms = ~11ms without interruption
60         for (i=0; i<4; i++) {
61             button_ptr->debouncing[i] =
62                 (button_ptr->debouncing[i]<<1)
63                 | ( ((raw_state & (1<<i)))? 1 : 0 );
64
65             if (button_ptr->debouncing[i] == 0) {
66                 // Releasing this button
67                 filtered_state &= ~(1<<i);
68             } else if (button_ptr->debouncing[i] == 0xFF) {
69                 // Pressing this button
```

```
70                  filtered_state |=  (1<<i);
71              }
72          }
73
74          if (button_ptr->recent_state_change) {
75              button_ptr->recent_state_change--;
76          }
77      }
78
79      // Storing the final, filtered, updated state
80      button_ptr->changed = button_ptr->state ^ filtered_state;
81      button_ptr->state = filtered_state;
82
83      // If any button has been pressed
84      if (button_ptr->changed & filtered_state) {
85          // This value was choosen empirically.
86          // 64 * 1.365ms = 87.36ms = 11.45Hz
87          button_ptr->recent_state_change = 64;
88      }
89  }  // }}}
90
91
92  // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
        foldmarker={{{,}}}
```

APÊNDICE C – common.h

```
1  /* Name: common.h
2   *
3   * Random useful things that I don't want to repeat everywhere.
4   */
5
6  #ifndef __common_h_included__
7  #define __common_h_included__
8
9
10 #ifndef uchar
11 #define uchar  unsigned char
12 #endif
13
14
15 // http://www.tty1.net/blog/2008-04-29-avr-gcc-optimisations_en.html
16 #define FIX_POINTER(_ptr) __asm__ __volatile__("" : "=b" (_ptr) : "0"
       (_ptr))
17
18
19 #endif  // __common_h_included__
20
21 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
       foldmarker={{{,}}}
```

APÊNDICE D – int_eeprom.h e int_eeprom.c

```
1  /* Name: int_eeprom.h
2   *
3   * See the .c file for more information
4   */
5
6  #ifndef __int_eeprom_h_included__
7  #define __int_eeprom_h_included__
8
9
10 // Maximum block that can be written (at once) to the EEPROM
11 // (measured in bytes)
12 // 1 boolean zero_compensation (1 bytes)
13 // 1 XYZVector for the zero value (6 bytes)
14 // 4 XYZVectors for the calibration corners (6 bytes each)
15 // Total of 31
16 #define INT_EEPROM_BUFFER_SIZE   32
17
18
19 // Init does nothing
20 #define init_int_eeprom() do{ }while(0)
21
22 void int_eeprom_write_block(
23         const void * src,
24         void* address,
25         unsigned char size);
26
27
28 #endif  // __int_eeprom_h_included____
29
30 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
     foldmarker={{{,}}}
```

```
1  /* Name: int_eeprom.c
2   * Project: atmega8-magnetometer-usb-mouse
3   * Author: Denilson Figueiredo de Sa
4   * Creation Date: 2011-10-05
5   * Tabsize: 4
6   * License: GNU GPL v2 or GNU GPL v3
7   *
8   * Non-blocking interrupt-based EEPROM writing code.
9   * For reading from the EEPROM, use the avr-libc functions.
10  *
11  * The interrupt handler code is loosely based on "AVR104 Buffered
        Interrupt
12  * Controlled EEPROM Writes on tinyAVR and megaAVR devices".
13  */
14
15
16 #include <avr/io.h>
17 #include <avr/interrupt.h>
18 #include <string.h>
19
20 #include "int_eeprom.h"
21
22
23 // EEPROM destination address
24 static void* eeprom_address;
25 // Buffer for writing the EEPROM
26 static unsigned char eeprom_buffer[INT_EEPROM_BUFFER_SIZE];
27 // The size of the block currently in the buffer
28 static unsigned char eeprom_block_size;
29 // The index of the next byte to be written
30 static unsigned char eeprom_next_byte;
31
32
33 #define ENABLE_EE_RDY_INTERRUPT()   do { EECR |=  (1 << EERIE); } while(0)
34 #define DISABLE_EE_RDY_INTERRUPT()  do { EECR &= ~(1 << EERIE); } while(0)
35
36
37 /*
38 static void init_int_eeprom() {  // {{{
39     // According to avr-libc FAQ, the compiler automatically initializes
          all
40     // variables with zero. Also, the EE_RDY interrupt is disabled by
          default
41     // in ATmega8.
42     // Conclusion: this function is redundant.
43
44     DISABLE_EE_RDY_INTERRUPT();
45     eeprom_block_size = 0;
46 }  // }}}
47 */
48
49
50 void int_eeprom_write_block(
51         const void * src,
52         void* address,
53         unsigned char size) {  // {{{
54
55     memcpy(eeprom_buffer, src, size);
56     eeprom_block_size = size;
57     eeprom_address = address;
58     eeprom_next_byte = 0;
59
60     ENABLE_EE_RDY_INTERRUPT();
61 }  // }}}
62
63
64 ISR(EE_RDY_vect) {  // {{{
65     //if ( SPMCR & (1 << SPMEN) ) // Is Self-Programming Currently Active?
66     //  return;                   // Yes, Return to main()
67
68     EEAR = (unsigned int) eeprom_address + eeprom_next_byte;
69     EEDR = eeprom_buffer[eeprom_next_byte];
```

```
70
71      EECR |= (1 << EEMWE);  // Assert EEPROM Master Write Enable
72      EECR |= (1 << EEWE);   // Assert EEPROM Write Enable
73
74      eeprom_next_byte++;
75
76      if (eeprom_next_byte == eeprom_block_size) {
77          // Buffer is now empty
78          eeprom_block_size = 0;
79
80          DISABLE_EE_RDY_INTERRUPT();
81      }
82  }  // }}}
83
84
85  // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
        foldmarker={{{,}}}
```

APÊNDICE E – keyemu.h e keyemu.c

```
1  /* Name: keyemu.h
2   *
3   * See the .c file for more information
4   */
5
6  #ifndef __keyemu_h_included__
7  #define __keyemu_h_included__
8
9  #include "common.h"
10 #include "sensor.h"
11
12
13 typedef struct KeyboardReport {
14     uchar report_id;
15     uchar modifier;
16     uchar key;
17 } KeyboardReport;
18
19
20 extern KeyboardReport keyboard_report;
21
22
23 // Copies a string from PGM to string_output_buffer and also sets
24 // string_output_pointer.
25 #define output_pgm_string(str) do { \
26         strcpy_P(string_output_buffer, str); \
27         string_output_pointer = string_output_buffer; \
28     } while(0)
29
30
31 #define STRING_OUTPUT_BUFFER_SIZE 100
32 extern uchar *string_output_pointer;
33 extern uchar string_output_buffer[STRING_OUTPUT_BUFFER_SIZE];
34
35
36 void init_keyboard_emulation();
37 void build_report_from_char(uchar c);
38 uchar send_next_char();
39
40 uchar nibble_to_hex(uchar n);
41 void uchar_to_hex(uchar v, uchar *str);
42 void int_to_hex(int v, uchar *str);
43 uchar* int_to_dec(int v, uchar *str);
44 uchar* append_newline_to_str(uchar *str);
45 uchar* array_to_hexdump(uchar *data, uchar len, uchar *str);
46 uchar* XYZVector_to_string(XYZVector* vector, uchar *str);
47
48
49 #endif  // __keyemu_h_included____
50
51 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
       foldmarker={{{,}}}
```

```
/* Name: keyemu.c
 * Project: atmega8-magnetometer-usb-mouse
 * Author: Denilson Figueiredo de Sa
 * Creation Date: 2011-10-18
 * Tabsize: 4
 * License: GNU GPL v2 or GNU GPL v3
 */


// For NULL definition
#include <stddef.h>

#include <avr/pgmspace.h>
#include <string.h>
#include <stdlib.h>

#include "common.h"
#include "keyemu.h"


// HID report
KeyboardReport keyboard_report;

// Pointer to RAM for the string being typed.
uchar *string_output_pointer = NULL;

// Shared output buffer, other functions are free to use this as needed.
uchar string_output_buffer[STRING_OUTPUT_BUFFER_SIZE];


// Keyboard usage values  {{{
// See chapter 10 (Keyboard/Keypad Page) from "USB HID Usage Tables"
// (page 53 of Hut1_12v2.pdf)

#define MOD_CONTROL_LEFT    (1<<0)
#define MOD_SHIFT_LEFT      (1<<1)
#define MOD_ALT_LEFT        (1<<2)
#define MOD_GUI_LEFT        (1<<3)
#define MOD_CONTROL_RIGHT   (1<<4)
#define MOD_SHIFT_RIGHT     (1<<5)
#define MOD_ALT_RIGHT       (1<<6)
#define MOD_GUI_RIGHT       (1<<7)

#define KEY_A           4
#define KEY_B           5
#define KEY_C           6
#define KEY_D           7
#define KEY_E           8
#define KEY_F           9
#define KEY_G           10
#define KEY_H           11
#define KEY_I           12
#define KEY_J           13
#define KEY_K           14
#define KEY_L           15
#define KEY_M           16
#define KEY_N           17
#define KEY_O           18
#define KEY_P           19
#define KEY_Q           20
#define KEY_R           21
#define KEY_S           22
#define KEY_T           23
#define KEY_U           24
#define KEY_V           25
#define KEY_W           26
#define KEY_X           27
#define KEY_Y           28
#define KEY_Z           29
#define KEY_1           30
```

```
71  #define KEY_2          31
72  #define KEY_3          32
73  #define KEY_4          33
74  #define KEY_5          34
75  #define KEY_6          35
76  #define KEY_7          36
77  #define KEY_8          37
78  #define KEY_9          38
79  #define KEY_0          39
80  #define KEY_ENTER      40
81  #define KEY_ESCAPE     41
82  #define KEY_TAB        43
83  #define KEY_SPACE      44
84  #define KEY_MINUS      45
85  #define KEY_EQUAL      46
86  #define KEY_SEMICOLON  51
87  #define KEY_COMMA      54
88  #define KEY_PERIOD     55
89  #define KEY_F1         58
90  #define KEY_F2         59
91  #define KEY_F3         60
92  #define KEY_F4         61
93  #define KEY_F5         62
94  #define KEY_F6         63
95  #define KEY_F7         64
96  #define KEY_F8         65
97  #define KEY_F9         66
98  #define KEY_F10        67
99  #define KEY_F11        68
100 #define KEY_F12        69
101
102 // }}}
103
104
105 // Mapping between chars and their keyboard codes  {{{
106
107 typedef struct KeyAndModifier {
108     uchar key;
109     uchar modifier;
110 } KeyAndModifier;
111
112
113 // Warning: ';:' key is c-cedilla in BR-ABNT2 layout
114 // Warning: '/?' key is ';:' in BR-ABNT2 layout
115 // Other characters not in this lookup table:
116 //   \t \n A-Z a-z [\] ^_` {|} ~
117 static KeyAndModifier char_to_key[] PROGMEM = {  // {{{
118     {KEY_SPACE    , 0              }, // SPACE
119     {KEY_1        , MOD_SHIFT_LEFT}, // !
120     {0            , 0              }, // "
121     {KEY_3        , MOD_SHIFT_LEFT}, // #
122     {KEY_4        , MOD_SHIFT_LEFT}, // $
123     {KEY_5        , MOD_SHIFT_LEFT}, // %
124     {KEY_7        , MOD_SHIFT_LEFT}, // &
125     {0            , 0              }, // '
126     {KEY_9        , MOD_SHIFT_LEFT}, // (
127     {KEY_0        , MOD_SHIFT_LEFT}, // )
128     {KEY_8        , MOD_SHIFT_LEFT}, // *
129     {KEY_EQUAL    , MOD_SHIFT_LEFT}, // +
130     {KEY_COMMA    , 0              }, // ,
131     {KEY_MINUS    , 0              }, // -
132     {KEY_PERIOD   , 0              }, // .
133     {0            , 0              }, // /
134     {KEY_0        , 0              }, // 0
135     {KEY_1        , 0              }, // 1
136     {KEY_2        , 0              }, // 2
137     {KEY_3        , 0              }, // 3
138     {KEY_4        , 0              }, // 4
139     {KEY_5        , 0              }, // 5
```

```
140     {KEY_6         , 0              }, // 6
141     {KEY_7         , 0              }, // 7
142     {KEY_8         , 0              }, // 8
143     {KEY_9         , 0              }, // 9
144     {KEY_SEMICOLON , MOD_SHIFT_LEFT}, // :
145     {KEY_SEMICOLON , 0              }, // ;
146     {KEY_COMMA     , MOD_SHIFT_LEFT}, // <
147     {KEY_EQUAL     , 0              }, // =
148     {KEY_PERIOD    , MOD_SHIFT_LEFT}, // >
149     {0             , 0              }, // ?
150     {KEY_2         , MOD_SHIFT_LEFT}  // @
151 };  // }}}
152
153 // }}}
154
155
156 void init_keyboard_emulation() {  // {{{
157     // According to avr-libc FAQ, the compiler automatically initializes
158     // all variables with zero.
159     keyboard_report.report_id = 1;
160     //keyboard_report.modifier = 0;
161     //keyboard_report.key = 0;
162 }  // }}}
163
164 void build_report_from_char(uchar c) {  // {{{
165     // Using a local pointer saves around 6 bytes
166     KeyboardReport *repptr = &keyboard_report;
167     FIX_POINTER(repptr);
168
169     // For most cases, modifier is zero
170     repptr->modifier = 0;
171
172     if (c >= ' ' && c <= '@') {
173         repptr->modifier = pgm_read_byte_near(&char_to_key[c - '
            '].modifier);
174         repptr->key = pgm_read_byte_near(&char_to_key[c - ' '].key);
175     } else if (c >= 'A' && c <= 'Z') {
176         repptr->modifier = MOD_SHIFT_LEFT;
177         repptr->key = KEY_A + c - 'A';
178     } else if (c >= 'a' && c <= 'z') {
179         //repptr->modifier = 0;
180         repptr->key = KEY_A + c - 'a';
181     } else {
182         switch (c) {
183             case '\n':
184                 //repptr->modifier = 0;
185                 repptr->key = KEY_ENTER;
186                 break;
187             case '\t':
188                 //repptr->modifier = 0;
189                 repptr->key = KEY_TAB;
190                 break;
191             case '_':
192                 repptr->modifier = MOD_SHIFT_LEFT;
193                 repptr->key = KEY_MINUS;
194                 break;
195
196             default:
197                 //repptr->modifier = 0;
198                 repptr->key = 0;
199         }
200     }
201 }  // }}}
202
203
204 uchar send_next_char() {  // {{{
205     // Builds a Report with the char pointed by 'string_output_pointer'.
206     //
207     // If a valid char is found, builds the report and returns 1.
208     // If the pointer is NULL or the char is '\0', builds a "no key being
209     // pressed" report and returns 0.
```

```
210     // (note: the return value is being ignored by main())
211     //
212     // If the next char uses the same key as the previous one, then sends a
213     // "no key" before sending the char.
214
215     // Using a local pointer saves around 2 bytes
216     KeyboardReport *repptr = &keyboard_report;
217     FIX_POINTER(repptr);
218
219     if (string_output_pointer != NULL && *string_output_pointer != '\0') {
220         uchar old_key;
221
222         old_key = repptr->key;
223         build_report_from_char(*string_output_pointer);
224
225         if (old_key == repptr->key && repptr->key != 0) {
226             // Inserting a key release if the next key would be the same as
227             // the previous one
228             repptr->modifier = 0;
229             repptr->key = 0;
230         } else {
231             string_output_pointer++;
232         }
233
234         return 1;
235     } else {
236         repptr->modifier = 0;
237         repptr->key = 0;
238         string_output_pointer = NULL;
239         return 0;
240     }
241 }  // }}}
242
243
244 ////////////////////////////////////////////////////////
245 // String utilities
246 // Not exactly part of keyboard emulation, but useless outside it.
247
248 uchar nibble_to_hex(uchar n) {  // {{{
249     // I'm supposing n is already in range 0x00..0x0F
250     if (n < 10)
251         return '0' + n;
252     else
253         return 'A' + n - 10;
254 }  // }}}
255
256 void uchar_to_hex(uchar v, uchar *str) {  // {{{
257     // NOTE: The NULL terminator is NOT added!
258     *str = nibble_to_hex(v >> 4);
259     str++;
260     *str = nibble_to_hex(v & 0x0F);
261 }  // }}}
262
263 void int_to_hex(int v, uchar *str) {  // {{{
264     // I'm supposing int is 16-bit
265     // NOTE: The NULL terminator is NOT added!
266     uchar_to_hex((uchar)(v >> 8), str);
267     uchar_to_hex((uchar) v      , str+2);
268 }  // }}}
269
270 uchar* int_to_dec(int v, uchar *str) {  // {{{
271     // Returns a pointer to the '\0' char
272
273     itoa(v, (char*)str, 10);
274     while (*str != '\0') {
275         str++;
276     }
277     return str;
278 }  // }}}
279
280 uchar* append_newline_to_str(uchar *str) {  // {{{
281     // Returns a pointer to the '\0' char
```

```
282
283     while (*str != '\0') {
284         str++;
285     }
286     *str     = '\n';
287     *(str+1) = '\0';
288
289     return str+1;
290 }  // }}}
291
292 uchar* array_to_hexdump(uchar *data, uchar len, uchar *str) {  // {{{
293     // Builds a string of this form:
294     // "DE AD F0 0D"
295     // One space between each byte, ending the string with '\0'
296     //
297     // Returns a pointer to the '\0' char
298
299     // I'm supposing that len > 0
300     uchar_to_hex(*data, str);
301
302     while (--len) {
303         data++;
304         // str[0] and str[1] are the hex digits
305         str[2] = ' ';
306         str += 3;
307         uchar_to_hex(*data, str);
308     }
309
310     str[2] = '\0';
311
312     return str+2;
313 }  // }}}
314
315 uchar* XYZVector_to_string(XYZVector* vector, uchar *str) {  // {{{
316     // "-1234\t1234\t-1234\n"
317
318     str = int_to_dec(vector->x, str);
319     *str = '\t';
320     str++;
321
322     str = int_to_dec(vector->y, str);
323     *str = '\t';
324     str++;
325
326     str = int_to_dec(vector->z, str);
327     *str = '\n';
328     str++;
329
330     *str = '\0';
331
332     return str;
333 }  // }}}
334
335
336 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
    foldmarker={{{,}}}
```

APÊNDICE F – menu.h e menu.c

```
1  /* Name: menu.h
2   *
3   * See the .c file for more information
4   */
5
6  #ifndef __menu_h_included__
7  #define __menu_h_included__
8
9
10 void init_ui_system();
11 void ui_main_code();
12
13
14 #endif  // __menu_h_included____
15
16 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
       foldmarker={{{,}}}
```

```
1  /* Name: menu.c
2   * Project: atmega8-magnetometer-usb-mouse
3   * Author: Denilson Figueiredo de Sa
4   * Creation Date: 2011-09-27
5   * Tabsize: 4
6   * License: GNU GPL v2 or GNU GPL v3
7   */
8
9
10 // For NULL definition
11 #include <stddef.h>
12
13 #include <avr/pgmspace.h>
14
15 #include "buttons.h"
16 #include "common.h"
17 #include "int_eeprom.h"
18 #include "keyemu.h"
19 #include "sensor.h"
20
21 #include "menu.h"
22
23
24 #define BUTTON_PREV    BUTTON_1
25 #define BUTTON_NEXT    BUTTON_2
26 #define BUTTON_CONFIRM BUTTON_3
27
28
29 ////////////////////////////////////////////////////////
30 // Menu definitions (constants in progmem)               {{{
31
32 // All available UI widgets  {{{
33
34 // Menus
35 // Note: the ROOT (empty) menu must be ZERO
36 #define UI_ROOT_MENU    0
37 #define UI_MAIN_MENU    1
38 #define UI_ZERO_MENU    2
39 #define UI_CORNERS_MENU 3
40 #define UI_SENSOR_MENU  4
41
42 // UI_MIN_MENU_ID must be ZERO
43 #define UI_MIN_MENU_ID UI_ROOT_MENU
44 #define UI_MAX_MENU_ID UI_SENSOR_MENU
45
46 // Checking if a widget is a menu
47 #define UI_IS_MENU(id) ((id) >= UI_MIN_MENU_ID && (id) <= UI_MAX_MENU_ID)
48
49 // Other widgets
50 #define UI_ZERO_PRINT_WIDGET              0x10
51 #define UI_ZERO_CAL_WIDGET                0x11
52 #define UI_ZERO_TOGGLE_WIDGET             0x12
53 #define UI_CORNERS_PRINT_WIDGET           0x13
54 #define UI_CORNERS_SET_TOPLEFT_WIDGET     0x14
55 #define UI_CORNERS_SET_TOPRIGHT_WIDGET    0x15
56 #define UI_CORNERS_SET_BOTTOMLEFT_WIDGET  0x16
57 #define UI_CORNERS_SET_BOTTOMRIGHT_WIDGET 0x17
58 #define UI_CORNERS_SET_ANYTHING_WIDGET    0x18
59 #define UI_SENSOR_ID_WIDGET               0x19
60 #define UI_SENSOR_XYZ_ONCE_WIDGET         0x1A
61 #define UI_SENSOR_XYZ_CONT_WIDGET         0x1B
62 #define UI_KEYBOARD_TEST_WIDGET           0x1C
63 // }}}
64
65 typedef struct MenuItem {  // {{{
66     // The menu text string pointer
67     PGM_P text;
68     // The widget that will be activated on this menu item
69     uchar action;
70 } MenuItem;  // }}}
71
```

```
72 // Root/empty menu  {{{
73 // Yeah, this is just a "fake" menu with only one empty item.
74
75 static const char     empty_menu_1[] PROGMEM = "";
76 #define               empty_menu_total_items 1
77 static const MenuItem empty_menu_items[] PROGMEM = {
78     {empty_menu_1, UI_MAIN_MENU}
79 };
80 // }}}
81
82 // Error menu, for when something goes wrong  {{{
83 static const char     error_menu_1[] PROGMEM = "Menu error\n";
84 #define               error_menu_total_items 1
85 static const MenuItem error_menu_items[] PROGMEM = {
86     {error_menu_1, 0}
87 };
88 // }}}
89
90 // Main menu, with all main options  {{{
91 #if ENABLE_FULL_MENU
92 static const char     main_menu_1[] PROGMEM = "1. Zero calibration >>\n";
93 static const char     main_menu_2[] PROGMEM = "2. Corner calibration >>\n";
94 static const char     main_menu_3[] PROGMEM = "3. Sensor data >>\n";
95 static const char     main_menu_4[] PROGMEM = "4. Keyboard test\n";
96 static const char     main_menu_5[] PROGMEM = "5. << quit menu\n";
97 #define               main_menu_total_items 5
98 #else
99 static const char     main_menu_1[] PROGMEM = "1. Zero >>\n";
100 static const char     main_menu_2[] PROGMEM = "2. Corner >>\n";
101 static const char     main_menu_3[] PROGMEM = "3. Sensor data >>\n";
102 static const char     main_menu_5[] PROGMEM = "4. << quit menu\n";
103 #define               main_menu_total_items 4
104 #endif
105
106 static const MenuItem main_menu_items[] PROGMEM = {
107     {main_menu_1, UI_ZERO_MENU},
108     {main_menu_2, UI_CORNERS_MENU},
109     {main_menu_3, UI_SENSOR_MENU},
110 #if ENABLE_FULL_MENU
111     {main_menu_4, UI_KEYBOARD_TEST_WIDGET},
112 #endif
113     {main_menu_5, 0}
114 };
115 // }}}
116
117 // Zero calibration menu  {{{
118 static const char     zero_menu_1[] PROGMEM = "1.1. Print zero\n";
119 static const char     zero_menu_2[] PROGMEM = "1.2. Recalibrate zero\n";
120 static const char     zero_menu_3[] PROGMEM = "1.3. Toggle zero
       compensation\n";
121 static const char     zero_menu_4[] PROGMEM = "1.4. << back\n";
122 #define               zero_menu_total_items 4
123 static const MenuItem zero_menu_items[] PROGMEM = {
124     {zero_menu_1, UI_ZERO_PRINT_WIDGET},
125     {zero_menu_2, UI_ZERO_CAL_WIDGET},
126     {zero_menu_3, UI_ZERO_TOGGLE_WIDGET},
127     {zero_menu_4, 0}
128 };
129
130 // Other zerocal messages:
131 #if ENABLE_FULL_MENU
132 static const char zero_calibration_instructions[] PROGMEM = "Move the
       sensor to get the maximum and minimum value for each axis. Press the
       button to finish.\n";
133 static const char zero_compensation_prefix[] PROGMEM = "Zero compensation
       is ";
134 static const char zero_compensation_suffix_on[] PROGMEM = "ENABLED\n";
135 static const char zero_compensation_suffix_off[] PROGMEM = "DISABLED\n";
136 #else
137 //static const char zero_calibration_instructions[] PROGMEM = "";
```

```
138 static const char zero_compensation_prefix[] PROGMEM = "Zero comp. is ";
139 static const char zero_compensation_suffix_on[] PROGMEM = "ON\n";
140 static const char zero_compensation_suffix_off[] PROGMEM = "OFF\n";
141 #endif
142
143 // }}}
144
145 // Corner calibration menu  {{{
146 static const char     corners_menu_1[] PROGMEM = "2.1. Print corners\n";
147 static const char     corners_menu_2[] PROGMEM = "2.2. Set topleft\n";
148 static const char     corners_menu_3[] PROGMEM = "2.3. Set topright\n";
149 static const char     corners_menu_4[] PROGMEM = "2.4. Set bottomleft\n";
150 static const char     corners_menu_5[] PROGMEM = "2.5. Set bottomright\n";
151 static const char     corners_menu_6[] PROGMEM = "2.6. << back\n";
152 #define               corners_menu_total_items 6
153 static const MenuItem corners_menu_items[] PROGMEM = {
154     {corners_menu_1, UI_CORNERS_PRINT_WIDGET},
155     {corners_menu_2, UI_CORNERS_SET_TOPLEFT_WIDGET},
156     {corners_menu_3, UI_CORNERS_SET_TOPRIGHT_WIDGET},
157     {corners_menu_4, UI_CORNERS_SET_BOTTOMLEFT_WIDGET},
158     {corners_menu_5, UI_CORNERS_SET_BOTTOMRIGHT_WIDGET},
159     {corners_menu_6, 0}
160 };
161
162 // Corner names:
163 // Slightly hacked from the menu strings. Saves about 40 bytes this way.
164 static const PGM_P corners_names[4] PROGMEM = {
165     // +offset = number of characters in "2.x. Set "
166     corners_menu_2 + 9,
167     corners_menu_3 + 9,
168     corners_menu_4 + 9,
169     corners_menu_5 + 9
170 };
171 // }}}
172
173 // Sensor data menu  {{{
174 #if ENABLE_FULL_MENU
175 static const char     sensor_menu_1[] PROGMEM = "3.1. Print sensor
        identification\n";
176 static const char     sensor_menu_2[] PROGMEM = "3.2. Print X,Y,Z once\n";
177 static const char     sensor_menu_3[] PROGMEM = "3.3. Print X,Y,Z
        continually\n";
178 static const char     sensor_menu_4[] PROGMEM = "3.4. << back\n";
179 #define               sensor_menu_total_items 4
180 #else
181 static const char     sensor_menu_2[] PROGMEM = "3.1. Print X,Y,Z once\n";
182 static const char     sensor_menu_3[] PROGMEM = "3.2. Print X,Y,Z
        continually\n";
183 static const char     sensor_menu_4[] PROGMEM = "3.3. << back\n";
184 #define               sensor_menu_total_items 3
185 #endif
186
187 static const MenuItem sensor_menu_items[] PROGMEM = {
188 #if ENABLE_FULL_MENU
189     {sensor_menu_1, UI_SENSOR_ID_WIDGET},
190 #endif
191     {sensor_menu_2, UI_SENSOR_XYZ_ONCE_WIDGET},
192     {sensor_menu_3, UI_SENSOR_XYZ_CONT_WIDGET},
193     {sensor_menu_4, 0}
194 };
195
196 // Error message:
197 static const char  error_sensor_string[] PROGMEM = "Sensor reading
        error\n";
198 // }}}
199
200 #if ENABLE_FULL_MENU
201 static const char keyboard_test_string[] PROGMEM = "AAaaAAaaZz 0123456789
        !@#$%&*() -_ =+ ,< .> :; /?\n";
202 #endif
```

```
203
204 // Data used by ui_load_menu_items()  {{{
205 typedef struct MenuLoadingInfo {
206     // Pointer to the array of MenuItems
207     PGM_VOID_P menu_items;
208     // Number of items in this menu
209     uchar total_items;
210 } MenuLoadingInfo;
211
212 #define MENU_LOADING(prefix) {prefix##_menu_items,
        prefix##_menu_total_items}
213 static const MenuLoadingInfo menu_loading[] PROGMEM = {
214     MENU_LOADING(error),  // The error menu is at element 0
215     MENU_LOADING(empty),  // And the UI with id ZERO starts at element 1
216     MENU_LOADING(main),
217     MENU_LOADING(zero),
218     MENU_LOADING(corners),
219     MENU_LOADING(sensor)
220 };
221 #undef MENU_LOADING
222 // }}}
223
224 // }}}
225
226
227 ////////////////////////////////////////////////////////
228 // UI and Menu variables (state)                   {{{
229
230 typedef struct UIState {
231     // Current active UI widget
232     uchar widget_id;
233     // Current menu item, or other state for non-menu widgets
234     uchar menu_item;
235 } UIState;
236
237 // Current UI state
238 UIState ui;
239
240 // At most 5 stacked widgets. This value is arbitrary.
241 UIState ui_stack[5];
242 uchar ui_stack_top;
243
244 // Each menu can have at most 7 menu items. This value is arbitrary.
245 static MenuItem ui_menu_items[7];
246 static uchar ui_menu_total_items;
247
248 // Should the current menu item be printed in the next ui_main_code() call?
249 // If the firmware is busy printing something else, this flag won't be
        reset,
250 // and the current menu item will be printed whenever it is appropriate.
251 uchar ui_should_print_menu_item;
252
253 // }}}
254
255
256 ////////////////////////////////////////////////////////
257 // UI and menu handling functions                  {{{
258
259 static void ui_load_menu_items() {  // {{{
260     // Loads the menu strings from PROGMEM to RAM
261
262     // Load from this id
263     uchar id;
264     // Pointer to the source address
265     PGM_VOID_P menu_items;
266
267     if (UI_IS_MENU(ui.widget_id)) {
268         id = ui.widget_id - UI_MIN_MENU_ID + 1;
269     } else {
270         id = 0;
271     }
272
```

```
273     // This version is more portable:
274     //memcpy_P(&menu_items, &menu_loading[id].menu_items, sizeof(PGM_P));
275     // But this version is shorter (40 bytes smaller):
276     // But this assertion must be true: assert(sizeof(PGM_VOID_P) == 2)
277     menu_items = (PGM_VOID_P)
            pgm_read_word_near(&menu_loading[id].menu_items);
278
279     ui_menu_total_items =
            pgm_read_byte_near(&menu_loading[id].total_items);
280
281     // Copying to RAM array ui_menu_items
282     memcpy_P(ui_menu_items, menu_items, ui_menu_total_items *
            sizeof(*ui_menu_items));
283 }  // }}}
284
285 static void ui_push_state() {  // {{{
286     ui_stack[ui_stack_top] = ui;
287     ui_stack_top++;
288 }  // }}}
289
290 static void ui_pop_state() {  // {{{
291     if (ui_stack_top > 0) {
292         ui_stack_top--;
293         ui = ui_stack[ui_stack_top];
294     } else {
295         // If the stack is empty, just reload the root menu
296         ui.widget_id = UI_ROOT_MENU;
297         ui.menu_item = 0;
298     }
299
300     // If the state is a menu, reload the items and print the current item.
301     if (UI_IS_MENU(ui.widget_id)) {
302         ui_load_menu_items();
303         ui_should_print_menu_item = 1;
304     }
305 }  // }}}
306
307 static void ui_prev_menu_item() {  // {{{
308     if (ui.menu_item == 0) {
309         ui.menu_item = ui_menu_total_items;
310     }
311     ui.menu_item--;
312     ui_should_print_menu_item = 1;
313 }  // }}}
314
315 static void ui_next_menu_item() {  // {{{
316     ui.menu_item++;
317     if (ui.menu_item == ui_menu_total_items) {
318         ui.menu_item = 0;
319     }
320     ui_should_print_menu_item = 1;
321 }  // }}}
322
323 static void ui_enter_widget(uchar new_widget) {  // {{{
324     // Pushes the current widget (usually a parent menu), and enters a
325     // (sub)menu or widget.
326
327     ui_push_state();
328
329     ui.widget_id = new_widget;
330     ui.menu_item = 0;
331
332     // If the new widget is a menu, load the items and print the current
            one.
333     if (UI_IS_MENU(ui.widget_id)) {
334         ui_load_menu_items();
335         ui_should_print_menu_item = 1;
336     }
337 }  // }}}
338
339 // }}}
340
```

```
341
342 ////////////////////////////////////////////////////////
343 // UI and menu public functions                    {{{
344
345 void init_ui_system() {   // {{{
346     // Must be called in the main initialization routine
347
348     // Emptying the stack
349     ui_stack_top = 0;
350
351     // Calling ui_pop_state() with an empty stack will reload the initial
352     // widget (the root/empty menu).
353     ui_pop_state();
354 }  // }}}
355
356 static void ui_menu_code() {  // {{{
357
358     // If the current menu item needs to be printed and the firmware is not
359     // busy printing something else
360     if (ui_should_print_menu_item && string_output_pointer == NULL) {
361         output_pgm_string(ui_menu_items[ui.menu_item].text);
362         ui_should_print_menu_item = 0;
363     }
364
365     // Handling button events
366     if (ON_KEY_DOWN(BUTTON_PREV)) {
367         ui_prev_menu_item();
368     } else if (ON_KEY_DOWN(BUTTON_NEXT)) {
369         ui_next_menu_item();
370     } else if (ON_KEY_DOWN(BUTTON_CONFIRM)) {
371         uchar action;
372         action = ui_menu_items[ui.menu_item].action;
373
374         if (action == 0) {
375             ui_pop_state();
376         } else {
377             ui_enter_widget(action);
378         }
379     }
380 }  // }}}
381
382 void ui_main_code() {  // {{{
383     // This must be called in the main loop.
384     //
385     // This function handles the actions of all UI widgets.
386
387     uchar return_code;
388
389     SensorData *sens = &sensor;
390     FIX_POINTER(sens);
391
392     if (UI_IS_MENU(ui.widget_id)) {
393         ui_menu_code();
394     } else {
395         switch (ui.widget_id) {
396             ////////////////////
397             case UI_ZERO_PRINT_WIDGET:  // {{{
398                 if (string_output_pointer != NULL) {
399                     // Do nothing, let's wait the previous output...
400                 } else {
401                     // Printing X,Y,Z zero...
402                     XYZVector_to_string(&sens->e.zero,
                            string_output_buffer);
403
404                     // ...and the boolean value
405                     strcat_P(string_output_buffer,
                            zero_compensation_prefix);
406                     if (sens->e.zero_compensation) {
407                         strcat_P(string_output_buffer,
                                zero_compensation_suffix_on);
408                     } else {
```

```
409                         strcat_P(string_output_buffer,
                                zero_compensation_suffix_off);
410                     }
411
412                     string_output_pointer = string_output_buffer;
413                     ui_pop_state();
414                 }
415                 break;  // }}}
416
417             ////////////////////
418             case UI_ZERO_CAL_WIDGET:  // {{{
419                 if (ui.menu_item == 0) {
420                     if (string_output_pointer != NULL) {
421                         // Do nothing, let's wait the previous output...
422                         break;
423                     }
424
425 #if ENABLE_FULL_MENU
426                     // Print instructions
427                     output_pgm_string(zero_calibration_instructions);
428 #endif
429
430                     // Must disable zero compensation before calibration
431                     sens->e.zero_compensation = 0;
432
433                     sensor_start_continuous_reading();
434                     ui.menu_item = 1;
435                 } else if(ui.menu_item == 1) {
436                     // The first reading
437                     if (sens->new_data_available) {
438                         sens->new_data_available = 0;
439
440                         if (!sens->overflow) {
441                             sens->zero_min = sens->data;
442                             sens->zero_max = sens->data;
443                             // Using memcpy costs a few more bytes than
                                    simple attribution
444                             //memcpy(&sens->zero_min, &sens->data,
                                    sizeof(sens->data));
445                             //memcpy(&sens->zero_max, &sens->data,
                                    sizeof(sens->data));
446
447                             ui.menu_item = 2;
448                         }
449                     }
450                 } else {
451                     if (sens->new_data_available) {
452                         sens->new_data_available = 0;
453
454                         if (!sens->overflow) {
455                             // The following 6 if statements cost 96 bytes
456                             if (sens->data.x < sens->zero_min.x)
                                    sens->zero_min.x = sens->data.x;
457                             if (sens->data.y < sens->zero_min.y)
                                    sens->zero_min.y = sens->data.y;
458                             if (sens->data.z < sens->zero_min.z)
                                    sens->zero_min.z = sens->data.z;
459
460                             if (sens->data.x > sens->zero_max.x)
                                    sens->zero_max.x = sens->data.x;
461                             if (sens->data.y > sens->zero_max.y)
                                    sens->zero_max.y = sens->data.y;
462                             if (sens->data.z > sens->zero_max.z)
                                    sens->zero_max.z = sens->data.z;
463
464                             if (string_output_pointer == NULL) {
465                                 XYZVector_to_string(&sens->data,
                                        string_output_buffer);
466                                 string_output_pointer =
                                        string_output_buffer;
467                             }
468                         }
```

```
469                     }
470
471                     if (ON_KEY_DOWN(BUTTON_CONFIRM)) {
472                         sensor_stop_continuous_reading();
473
474                         sens->e.zero.x = (sens->zero_min.x +
                               sens->zero_max.x) / 2;
475                         sens->e.zero.y = (sens->zero_min.y +
                               sens->zero_max.y) / 2;
476                         sens->e.zero.z = (sens->zero_min.z +
                               sens->zero_max.z) / 2;
477
478                         sens->e.zero_compensation = 1;
479
480                         // Saving to EEPROM
481                         // I could save the entire EEPROM block, but
                               instead
482                         // I'm saving only the boolean zero_compensation
                               and
483                         // the XYZVector zero.
484                         int_eeprom_write_block(
485                             &sens->e.zero_compensation,
486                             &eeprom_sensor.zero_compensation,
487                             (1 + sizeof(XYZVector))
488                         );
489
490                         ui_pop_state();
491                         ui_enter_widget(UI_ZERO_PRINT_WIDGET);
492                     }
493                 }
494                 break;  // }}}
495
496             ////////////////////
497             case UI_ZERO_TOGGLE_WIDGET:  // {{{
498                 // Toggling current state
499                 sens->e.zero_compensation = !sens->e.zero_compensation;
500
501                 // Saving to EEPROM
502                 int_eeprom_write_block(
503                     &sens->e.zero_compensation,
504                     &eeprom_sensor.zero_compensation,
505                     1
506                 );
507
508                 ui_pop_state();
509                 ui_enter_widget(UI_ZERO_PRINT_WIDGET);
510                 break;  // }}}
511
512
513             ////////////////////
514             case UI_CORNERS_PRINT_WIDGET:  // {{{
515                 if (string_output_pointer != NULL) {
516                     // Do nothing, let's wait the previous output...
517                 } else {
518                     if (ui.menu_item % 2 == 0) {
519                         // Print the corner name
520                         output_pgm_string(
521                             (PGM_VOID_P) pgm_read_word_near(
522                                 &corners_names[ui.menu_item / 2]
523                             )
524                         );
525                     } else {
526                         // Print the corner value
527                         XYZVector_to_string(&sens->e.corners[ui.menu_item
                               / 2], string_output_buffer);
528                         string_output_pointer = string_output_buffer;
529                     }
530
531                     ui.menu_item++;
532                     if (ui.menu_item >= 4 * 2) {
533                         ui_pop_state();
534                     }
```

```
535                 }
536                 break;  // }}}
537
538             ////////////////////
539             case UI_CORNERS_SET_TOPLEFT_WIDGET:  // {{{
540             case UI_CORNERS_SET_TOPRIGHT_WIDGET:
541             case UI_CORNERS_SET_BOTTOMLEFT_WIDGET:
542             case UI_CORNERS_SET_BOTTOMRIGHT_WIDGET:
543                 // Setting the ui.menu_item value to {0,1,2,3}, according
                        to
544                 // the selected corner
545                 ui.menu_item = ui.widget_id -
                        UI_CORNERS_SET_TOPLEFT_WIDGET;
546
547                 // Switching to the generic corner-saving code
548                 ui.widget_id = UI_CORNERS_SET_ANYTHING_WIDGET;
549                 sensor_start_continuous_reading();
550                 break;  // }}}
551
552             ////////////////////
553             case UI_CORNERS_SET_ANYTHING_WIDGET:  // {{{
554                 if (string_output_pointer == NULL
555                     && button.state & BUTTON_CONFIRM
556                     && sens->new_data_available
557                     && !sens->overflow
558                 ) {
559                     sensor_stop_continuous_reading();
560                     sens->new_data_available = 0;
561
562                     // Saving
563                     sens->e.corners[ui.menu_item] = sens->data;
564                     int_eeprom_write_block(
565                         &sens->e.corners[ui.menu_item],
566                         &eeprom_sensor.corners[ui.menu_item],
567                         sizeof(XYZVector)
568                     );
569
570                     // Printing
571                     XYZVector_to_string(&sens->data, string_output_buffer);
572                     string_output_pointer = string_output_buffer;
573
574                     ui_pop_state();
575                 }
576                 break;  // }}}
577
578
579 #if ENABLE_FULL_MENU
580             ////////////////////
581             case UI_SENSOR_ID_WIDGET:  // {{{
582                 if (string_output_pointer != NULL) {
583                     // Do nothing, let's wait the previous output...
584                     break;
585                 }
586                 if (ui.menu_item == 0) {
587                     sens->func_step = 0;
588                     ui.menu_item = 1;
589                 }
590
591                 return_code =
                        sensor_read_identification_string(string_output_buffer);
592
593                 if (return_code == SENSOR_FUNC_DONE) {
594                     string_output_buffer[3] = '\n';
595                     string_output_buffer[4] = '\0';
596                     // I could have used this function:
597                     //append_newline_to_str(string_output_buffer);
598                     // But it adds 18 bytes to the firmware
599
600                     string_output_pointer = string_output_buffer;
601                     ui_pop_state();
602                 } else if (return_code == SENSOR_FUNC_ERROR) {
```

```
603                     output_pgm_string(error_sensor_string);
604                     ui_pop_state();
605                 }
606                 break;  // }}}
607 #endif
608
609             ////////////////////
610             case UI_SENSOR_XYZ_ONCE_WIDGET:  // {{{
611             case UI_SENSOR_XYZ_CONT_WIDGET:
612                 if (ui.menu_item == 0) {
613                     if (string_output_pointer != NULL) {
614                         // Do nothing, let's wait the previous output...
615                         break;
616                     }
617                     sensor_start_continuous_reading();
618                     ui.menu_item = 1;  // Started reading, but nothing
                            printed yet.
619                 } else {
620                     if (string_output_pointer == NULL) {
621                         if (sens->new_data_available) {
622                             sens->new_data_available = 0;
623                             XYZVector_to_string(&sens->data,
                                    string_output_buffer);
624                             string_output_pointer = string_output_buffer;
625                             ui.menu_item = 2;  // At least one thing has
                                    been printed
626                         } else if (sens->error_while_reading) {
627                             output_pgm_string(error_sensor_string);
628                             sensor_stop_continuous_reading();
629                             ui_pop_state();
630                             break;
631                         }
632                     }
633                     if (ui.menu_item == 2 && (
634                         ui.widget_id == UI_SENSOR_XYZ_ONCE_WIDGET
635                         || ON_KEY_DOWN(BUTTON_CONFIRM)
636                     )) {
637                         sensor_stop_continuous_reading();
638                         ui_pop_state();
639                     }
640                 }
641                 break;  // }}}
642
643 #if ENABLE_FULL_MENU
644             ////////////////////
645             case UI_KEYBOARD_TEST_WIDGET:  // {{{
646                 if (string_output_pointer == NULL) {
647                     output_pgm_string(keyboard_test_string);
648                     ui_pop_state();
649                 }
650                 break;  // }}}
651 #endif
652
653             default:
654                 // Fallback in case of errors
655                 ui_pop_state();
656         }
657     }
658 }  // }}}
659
660 // }}}
661
662
663 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
        foldmarker={{{,}}}
```

APÊNDICE G – mouseemu.h e mouseemu.c

```
1  /* Name: mouseemu.h
2   *
3   * See the .c file for more information
4   */
5
6  #ifndef __mouseemu_h_included__
7  #define __mouseemu_h_included__
8
9  #include "common.h"
10 #include "sensor.h"
11
12
13 typedef struct MouseReport {
14     uchar report_id;
15     int x; // 0..32767
16     int y; // 0..32767
17     uchar buttons;
18 } MouseReport;
19
20 extern MouseReport mouse_report;
21
22
23 void init_mouse_emulation();
24 uchar mouse_prepare_next_report();
25
26
27 #endif  // __mouseemu_h_included____
28
29 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
       foldmarker={{{,}}}
```

```
/* Name: mouseemu.c
 * Project: atmega8-magnetometer-usb-mouse
 * Author: Denilson Figueiredo de Sa
 * Creation Date: 2011-10-18
 * Tabsize: 4
 * License: GNU GPL v2 or GNU GPL v3
 */


#include <math.h>

#include "buttons.h"
#include "common.h"
#include "mouseemu.h"


// HID report
MouseReport mouse_report;

float mouse_smooth[2];


int apply_smoothing(uchar index, float *value_ptr) {
    // http://en.wikipedia.org/wiki/Exponential_smoothing

#define AVG  (mouse_smooth[index])

    // This value was choosen empirically.
#define ALPHA  0.125

    AVG = AVG * (1 - ALPHA) + (*value_ptr) * ALPHA;

    if      (AVG < 0.0)  AVG = 0.0;
    else if (AVG > 1.0)  AVG = 1.0;

    return (int) round(AVG * 32767);

#undef ALPHA
#undef AVG
}


void init_mouse_emulation() {  // {{{
    // According to avr-libc FAQ, the compiler automatically initializes
    // all variables with zero.
    mouse_report.report_id = 2;
    mouse_report.x = -1;
    mouse_report.y = -1;
    //mouse_report.buttons = 0;
}  // }}}


static uchar mouse_update_buttons() {  // {{{
    // Return 1 if the button state has been updated (and thus the report
    // should be sent to the computer).

    // Since the 3 buttons are already at the 3 least significant bits, no
    // complicated conversion is need.
    // 0x07 = 0000 0111
    uchar new_state = button.state & 0x07;
    uchar modified = (new_state != mouse_report.buttons);

    mouse_report.buttons = new_state;
    return modified;
}  // }}}


static uchar mouse_axes_no_conversion() {  // {{{
    // Get X, Y, Z data from the sensor, discard the Z component and
    // directly use X, Y as the mouse position.
    // Only useful for debugging.

    SensorData *sens = &sensor;
```

```
74      FIX_POINTER(sens);
75
76      mouse_report.x = sens->data.x * 8 + 16384;
77      mouse_report.y = sens->data.y * 8 + 16384;
78
79      return 1;
80  }  // }}}
81
82
83  static void fill_matrix_from_sensor(float m[3][4]) {  // {{{
84      SensorData *sens = &sensor;
85      FIX_POINTER(sens);
86
87      // The linear system:
88      // -t*P + u*(B-A) + v*(C-A) = -A
89      // Where:
90      //   A = topleft
91      //   B = topright
92      //   C = bottomleft
93      //   P = current point
94      // The final, (x,y) screen coordinates are (u,v)
95      //
96      // That system is equivalent to this one:
97      // A + u*(B-A) + v*(C-A) = t*P
98      // Which means the current point is equal to the topleft corner, plus
99      // "u" times in the topleft/topright direction, plus "v" times in the
100     // topleft/bottomleft direction. "u" and "v" are between 0.0 and 1.0.
101     //
102     // It would have been better to normalize each vector before doing any
103     // math on them, in order to reduce deformations. However, I know
104     // (empirically) that all values from the sensor have about the same
105     // magnitude, and thus I don't need to normalize them.
106
107     // First column: - current_point
108     m[0][0] = -sens->data.x;
109     m[1][0] = -sens->data.y;
110     m[2][0] = -sens->data.z;
111
112     // Note: the values below are constant, and could have been
113     // pre-converted to float only once (either at boot, or after updating
114     // those values). It would save some cycles during this runtime.
115
116     // Second column: topright - topleft
117     m[0][1] = sens->e.corners[1].x - sens->e.corners[0].x;
118     m[1][1] = sens->e.corners[1].y - sens->e.corners[0].y;
119     m[2][1] = sens->e.corners[1].z - sens->e.corners[0].z;
120
121     // Third column: bottomleft - topleft
122     m[0][2] = sens->e.corners[2].x - sens->e.corners[0].x;
123     m[1][2] = sens->e.corners[2].y - sens->e.corners[0].y;
124     m[2][2] = sens->e.corners[2].z - sens->e.corners[0].z;
125
126     // Fourth column: - topleft
127     m[0][3] = -sens->e.corners[0].x;
128     m[1][3] = -sens->e.corners[0].y;
129     m[2][3] = -sens->e.corners[0].z;
130 }  // }}}
131
132 static void swap_rows(float a[4], float b[4]) {  // {{{
133     float tmp;
134     uchar i;
135
136     for (i=0; i<4; i++) {
137         tmp = a[i];
138         a[i] = b[i];
139         b[i] = tmp;
140     }
141 }  // }}}
142
143 static uchar mouse_axes_linear_equation_system() {  // {{{
144 #define W 4
```

```
#define H 3
    // Matrix of 3 lines and 4 columns
    float m[H][W];
    // The solution of this system
    float sol[H];

    uchar y;

    fill_matrix_from_sensor(m);

    // Gauss-Jordan elimination, based on:
    // http://elonen.iki.fi/code/misc-notes/python-gaussj/index.html

    for(y = 0; y < H; y++) {
        uchar maxrow;
        uchar y2;
        float pivot;

        // Finding the row with the maximum value at this column
        maxrow = y;
        pivot = fabs(m[maxrow][y]);
        for(y2 = y+1; y2 < H; y2++) {
            float newpivot;
            newpivot = fabs(m[y2][y]);
            if (newpivot > pivot) {
                pivot = newpivot;
                maxrow = y2;
            }
        }
        // If the maximum value in this column is zero
        if (pivot < 0.0009765625) {  // 2**-10 == 0.0009765625
            // Singular
            return 0;
        }
        if (maxrow != y) {
            swap_rows(m[y], m[maxrow]);
        }

        // Now we are ready to eliminate the column y, using m[y][y]
        for(y2 = y+1; y2 < H; y2++) {
            uchar x;
            float c;
            c = m[y2][y] / m[y][y];

            // Subtracting from every element in row y2
            for(x = y; x < W; x++) {
                m[y2][x] -= m[y][x] * c;
            }
        }
    }

    // The matrix is now in "row echelon form" (it is a triangular matrix).
    // Instead of using a loop for back-substituting all 3 elements from
    // sol[], I'm calculating only 2 of them, as only those are needed.

    sol[2] = m[2][3] / m[2][2];
    sol[1] = m[1][3] / m[1][1] - m[1][2] * sol[2] / m[1][1];
    // sol[0] is discarded

    if (   sol[1] < -0.25
        || sol[1] >  1.25
        || sol[2] < -0.25
        || sol[2] >  1.25
    ) {
        // Out-of-bounds
        return 0;
    }

    mouse_report.x = apply_smoothing(0, &sol[1]);
    mouse_report.y = apply_smoothing(1, &sol[2]);

```

```
216     return 1;
217 #undef W
218 #undef H
219 }  // }}}
220
221
222 static uchar mouse_update_axes() {  // {{{
223     // Update the report descriptor for the axes if new data is available
        from
224     // the sensor.
225
226     SensorData *sens = &sensor;
227     FIX_POINTER(sens);
228
229     if (sens->new_data_available && !sens->overflow) {
230         // Marking the data as "used"
231         sens->new_data_available = 0;
232
233         // Trying to convert the coordinates
234         if (
235             //mouse_axes_no_conversion()
236             mouse_axes_linear_equation_system()
237         ) {
238             return 1;
239         } else {
240             // But sometimes it will fail
241             return 0;
242         }
243     } else {
244         // Clearing the x, y to invalid values.
245         // Invalid values should be ignored by USB host.
246         //mouse_report.x = -1;
247         //mouse_report.y = -1;
248
249         // Well... Actually, Linux 2.6.38 does not behave that way.
250         // Instead, Linux moves the mouse pointer even if the supplied X,Y
251         // values are outside the LOGICAL_MINIMUM..LOGICAL_MAXIMUM range.
252         //
253         // As a workaround:
254         // If no data is available, I just leave the previous data in
            there.
255         //
256         // Note: Windows correctly ignores the invalid values.
257         return 0;
258     }
259
260     return 0;
261 }  // }}}
262
263
264 uchar mouse_prepare_next_report() {  // {{{
265     // Return 1 if a new report is available and should be sent to the
266     // computer.
267
268     if (button.recent_state_change) {
269         // Don't try to update the pointer coordinates after a click.
270         return mouse_update_buttons();
271     } else {
272         // I'm using a bitwise OR here because a boolean OR would
            short-circuit
273         // the expression and wouldn't run the second function. It's ugly,
            but
274         // it's simple and works.
275         return mouse_update_buttons() | mouse_update_axes();
276     }
277 }  // }}}
278
279
280 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
    foldmarker={{{,}}}
```

APÊNDICE H – sensor.h e sensor.c

```
1  /* Name: sensor.h
2   *
3   * See the .c file for more information
4   */
5
6  #ifndef __sensor_h_included__
7  #define __sensor_h_included__
8
9  #include <avr/eeprom.h>
10 #include "common.h"
11
12
13 // Return codes for the non-blocking functions
14 #define SENSOR_FUNC_STILL_WORKING 0
15 #define SENSOR_FUNC_DONE          1
16 #define SENSOR_FUNC_ERROR         2
17
18 // Value that means "overflow"
19 #define SENSOR_DATA_OVERFLOW -4096
20
21
22 // Definitions
23 typedef struct XYZVector {
24     int x, y, z;
25 } XYZVector;
26
27 typedef struct SensorEepromData {
28     // This struct is used for data at EEPROM and at SRAM
29
30     // Boolean to enable Zero compensation
31     uchar zero_compensation;
32
33     // Zero calibration value
34     XYZVector zero;
35
36     // Calibration corners (screen)
37     XYZVector corners[4];
38 } SensorEepromData;
39
40 typedef struct SensorData {
41     union {
42         uchar flags;
43         struct {
44             // Boolean that detects if the sensor have reported an overflow
45             uchar overflow:1;
46
47             // Set to 1 whenever new sensor data has been read and hasn't
48             // been used yet. This flag should be cleared elsewhere, after
49             // using the data.
50             uchar new_data_available:1;
51
52             // Almost the same as TWI_statusReg.lastTransOK.
53             // Gets set whenever a function returns with SENSOR_FUNC_ERROR.
54             // Gets reset whenever a function returns with
                   SENSOR_FUNC_DONE.
55             uchar error_while_reading:1;
```

```
56
57              // Enable continuous reading of sensor values
58              // This variable should be used in main() main loop (together
59              // with a timer) to detect when sensor_read_data_registers()
60              // should be called.
61              uchar continuous_reading:1;
62
63              uchar unused_bits:4;
64          };
65      };
66
67      // The X,Y,Z data from the sensor
68      XYZVector data;
69
70      SensorEepromData e;
71
72      // Zero calibration temporary values
73      XYZVector zero_min;
74      XYZVector zero_max;
75
76      // Used to determine the next step in non-blocking functions.
77      // Must be set to zero to ensure each function starts from the
            beginning.
78      uchar func_step;
79
80  } SensorData;
81
82
83  // Variable
84  extern SensorData sensor;
85
86
87  // EEPROM addresses
88  extern uchar EEMEM eeprom_sensor_unused;
89  extern SensorEepromData EEMEM eeprom_sensor;
90
91
92  // Functions
93  uchar sensor_read_data_registers();
94
95  void sensor_start_continuous_reading();
96  void sensor_stop_continuous_reading();
97
98  uchar sensor_read_identification_string(uchar *s);
99
100 void sensor_init_configuration();
101
102
103 #endif  // __sensor_h_included____
104
105 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
        foldmarker={{{,}}}
```

```
1  /* Name: sensor.c
2   * Project: atmega8-magnetometer-usb-mouse
3   * Author: Denilson Figueiredo de Sa
4   * Creation Date: 2011-09-28
5   * Tabsize: 4
6   * License: GNU GPL v2 or GNU GPL v3
7   *
8   * Communication with HMC5883L or HMC5883 from Honeywell.
9   * This sensor has "L883 2105" written on the chip.
10  * This sensor is a 3-axis magnetometer with I2C interface.
11  */
12
13
14 #include <avr/eeprom.h>
15
16 #include "avr315/TWI_Master.h"
17 #include "sensor.h"
18
19
20 SensorData sensor;
21
22
23 // Avoiding GCC optimizing-out these EEPROM vars
24 // Maybe I should put them inside a struct?
25 // http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=68621
26 #define X_EEMEM __attribute__((section(".eeprom"), used,
       externally_visible))
27
28 // "Default" EEPROM values:
29 uchar X_EEMEM eeprom_sensor_unused = 0;
30 SensorEepromData X_EEMEM eeprom_sensor = {
31     1,  // zero_compensation
32     {21, -108, 138},  // zero
33     {  // corners
34         {123, 219, 44}, // topleft
35         {-40, 245, 68}, // topright
36         {113, 166, 151}, // bottomleft
37         {-44, 190, 160} // bottomright
38     }
39 };
40
41
42 ////////////////////////////////////////////////////////
43 // Constant definitions                                {{{
44
45 #define SENSOR_I2C_READ_ADDRESS  0x3D
46 #define SENSOR_I2C_WRITE_ADDRESS 0x3C
47
48 // HMC5883L register definitions  {{{
49 // See page 11 of HMC5883L.pdf
50
51 // Read/write registers:
52 #define SENSOR_REG_CONF_A       0
53 #define SENSOR_REG_CONF_B       1
54 #define SENSOR_REG_MODE         2
55 // Read-only registers:
56 #define SENSOR_REG_DATA_START   3
57 #define SENSOR_REG_DATA_X_MSB   3
58 #define SENSOR_REG_DATA_X_LSB   4
59 #define SENSOR_REG_DATA_Z_MSB   5
60 #define SENSOR_REG_DATA_Z_LSB   6
61 #define SENSOR_REG_DATA_Y_MSB   7
62 #define SENSOR_REG_DATA_Y_LSB   8
63 #define SENSOR_REG_STATUS       9
64 #define SENSOR_REG_ID_A        10
65 #define SENSOR_REG_ID_B        11
66 #define SENSOR_REG_ID_C        12
67
68 // }}}
69
70 // HMC5883L status definitions  {{{
```

```
71  // See page 16 of HMC5883L.pdf
72
73  #define SENSOR_STATUS_LOCK  2
74  #define SENSOR_STATUS_RDY   1
75
76  // }}}
77
78  // HMC5883L configuration definitions  {{{
79  // See pages 12, 13, 14 of HMC5883L.pdf
80
81  // The Mode Register has 2 possible values for Idle mode.
82  #define SENSOR_MODE_CONTINUOUS  0
83  #define SENSOR_MODE_SINGLE      1
84  #define SENSOR_MODE_IDLE_A      2
85  #define SENSOR_MODE_IDLE_B      3
86  #define SENSOR_MODE_MASK        3
87
88  // How many samples averaged? Default=1
89  #define SENSOR_CONF_A_SAMPLES_1    0x00
90  #define SENSOR_CONF_A_SAMPLES_2    0x20
91  #define SENSOR_CONF_A_SAMPLES_4    0x40
92  #define SENSOR_CONF_A_SAMPLES_8    0x60
93  #define SENSOR_CONF_A_SAMPLES_MASK 0x60
94
95  // Data output rate for continuous mode. Default=15Hz
96  #define SENSOR_CONF_A_RATE_0_75     0x00
97  #define SENSOR_CONF_A_RATE_1_5      0x04
98  #define SENSOR_CONF_A_RATE_3        0x08
99  #define SENSOR_CONF_A_RATE_7_5      0x0C
100 #define SENSOR_CONF_A_RATE_15       0x10
101 #define SENSOR_CONF_A_RATE_30       0x14
102 #define SENSOR_CONF_A_RATE_75       0x18
103 #define SENSOR_CONF_A_RATE_RESERVED 0x1C
104 #define SENSOR_CONF_A_RATE_MASK     0x1C
105
106 // Measurement configuration, whether to apply bias. Default=Normal
107 #define SENSOR_CONF_A_BIAS_NORMAL   0x00
108 #define SENSOR_CONF_A_BIAS_POSITIVE 0x01
109 #define SENSOR_CONF_A_BIAS_NEGATIVE 0x02
110 #define SENSOR_CONF_A_BIAS_RESERVED 0x03
111 #define SENSOR_CONF_A_BIAS_MASK     0x03
112
113 // Gain configuration. Default=1.3Ga
114 #define SENSOR_CONF_B_GAIN_0_88 0x00
115 #define SENSOR_CONF_B_GAIN_1_3  0x20
116 #define SENSOR_CONF_B_GAIN_1_9  0x40
117 #define SENSOR_CONF_B_GAIN_2_5  0x60
118 #define SENSOR_CONF_B_GAIN_4_0  0x80
119 #define SENSOR_CONF_B_GAIN_4_7  0xA0
120 #define SENSOR_CONF_B_GAIN_5_6  0xC0
121 #define SENSOR_CONF_B_GAIN_8_1  0xE0
122 #define SENSOR_CONF_B_GAIN_MASK 0xE0
123
124 // Digital resolution (mG/LSb) for each gain
125 #define SENSOR_GAIN_SCALE_0_88  0.73
126 #define SENSOR_GAIN_SCALE_1_3   0.92
127 #define SENSOR_GAIN_SCALE_1_9   1.22
128 #define SENSOR_GAIN_SCALE_2_5   1.52
129 #define SENSOR_GAIN_SCALE_4_0   2.27
130 #define SENSOR_GAIN_SCALE_4_7   2.56
131 #define SENSOR_GAIN_SCALE_5_6   3.03
132 #define SENSOR_GAIN_SCALE_8_1   4.35
133
134 // }}}
135
136 // }}}
137
138
139 static void sensor_set_address_pointer(uchar reg) {  // {{{
140     // Sets the sensor internal register pointer.
```

```
141     // This is required before reading registers.
142     //
143     // This function is non-blocking (except if TWI is already busy).
144
145     uchar msg[2];
146     msg[0] = SENSOR_I2C_WRITE_ADDRESS;
147     msg[1] = reg;
148     TWI_Start_Transceiver_With_Data(msg, 2);
149 }  // }}}
150
151 static void sensor_set_register_value(uchar reg, uchar value) {  // {{{
152     // Sets one of those 3 writable registers to a value.
153     // Only useful for configuration.
154     //
155     // This function is non-blocking (except if TWI is already busy).
156
157     uchar msg[3];
158     msg[0] = SENSOR_I2C_WRITE_ADDRESS;
159     msg[1] = reg;
160     msg[2] = value;
161     TWI_Start_Transceiver_With_Data(msg, 3);
162 }  // }}}
163
164
165 uchar sensor_read_data_registers() {  // {{{
166     // Reads the X,Y,Z data registers and store them at global vars.
167     // In case of a transmission error, the previous values are not
        changed.
168     //
169     // This function is non-blocking.
170
171     // 1 address byte + 6 data bytes = 7 bytes
172     uchar msg[7];
173
174     uchar lastTransOK;
175
176     SensorData *sens = &sensor;
177     FIX_POINTER(sens);
178
179     switch(sens->func_step) {
180         case 0:  // Set address pointer
181             if (TWI_Transceiver_Busy()) return SENSOR_FUNC_STILL_WORKING;
182
183             sensor_set_address_pointer(SENSOR_REG_DATA_START);
184             sens->func_step = 1;
185         case 1:  // Start reading operation
186             if (TWI_Transceiver_Busy()) return SENSOR_FUNC_STILL_WORKING;
187
188             msg[0] = SENSOR_I2C_READ_ADDRESS;
189             TWI_Start_Transceiver_With_Data(msg, 7);
190
191             sens->func_step = 2;
192         case 2:  // Finished reading operation
193             if (TWI_Transceiver_Busy()) return SENSOR_FUNC_STILL_WORKING;
194
195             lastTransOK = TWI_Get_Data_From_Transceiver(msg, 7);
196             sens->func_step = 0;
197
198             if (lastTransOK) {
199                 // Copying data to sensor->data struct
200                 #define OFFSET(suffix) (1 + SENSOR_REG_DATA_##suffix -
                        SENSOR_REG_DATA_START)
201                 sens->data.x = (msg[OFFSET(X_MSB)] << 8) |
                        (msg[OFFSET(X_LSB)]);
202                 sens->data.y = (msg[OFFSET(Y_MSB)] << 8) |
                        (msg[OFFSET(Y_LSB)]);
203                 sens->data.z = (msg[OFFSET(Z_MSB)] << 8) |
                        (msg[OFFSET(Z_LSB)]);
204                 #undef OFFSET
205
206                 // Detecting overflow
207                 sens->overflow =
```

```
208                     (sens->data.x == SENSOR_DATA_OVERFLOW)
209                     || (sens->data.y == SENSOR_DATA_OVERFLOW)
210                     || (sens->data.z == SENSOR_DATA_OVERFLOW);
211
212                 // Applying zero compensation
213                 if (sens->e.zero_compensation && !sens->overflow) {
214                     sens->data.x -= sens->e.zero.x;
215                     sens->data.y -= sens->e.zero.y;
216                     sens->data.z -= sens->e.zero.z;
217                 }
218
219                 sens->new_data_available = 1;
220                 sens->error_while_reading = 0;
221                 return SENSOR_FUNC_DONE;
222             } else {
223                 sens->error_while_reading = 1;
224                 return SENSOR_FUNC_ERROR;
225             }
226         default:
227             sens->error_while_reading = 1;
228             return SENSOR_FUNC_ERROR;
229     }
230 }  // }}}
231
232 void sensor_start_continuous_reading() {  // {{{
233     SensorData *sens = &sensor;
234     FIX_POINTER(sens);
235
236     sens->func_step = 0;
237     sens->new_data_available = 0;
238     sens->error_while_reading = 0;
239     sens->continuous_reading = 1;
240 }  // }}}
241
242 void sensor_stop_continuous_reading() {  // {{{
243     SensorData *sens = &sensor;
244     FIX_POINTER(sens);
245
246     sens->func_step = 0;
247     //sens->new_data_available = 0;
248     //sens->error_while_reading = 0;
249     sens->continuous_reading = 0;
250 }  // }}}
251
252
253 uchar sensor_read_identification_string(uchar *s) {  // {{{
254     // Reads the 3 identification registers from the sensor.
255     // They should read as ASCII "H43".
256     //
257     // Receives a pointer to a string with at least 4 chars of size.
258     // After reading the registers, stores them at *s, followed by '\0'.
259     // In case of a transmission error, the *s is not touched.
260     //
261     // This function is non-blocking.
262
263     // 1 address byte + 3 chars
264     uchar msg[4];
265
266     uchar lastTransOK;
267
268     switch(sensor.func_step) {
269         case 0:  // Set address pointer
270             if (TWI_Transceiver_Busy()) return SENSOR_FUNC_STILL_WORKING;
271
272             sensor_set_address_pointer(SENSOR_REG_ID_A);
273             sensor.func_step = 1;
274         case 1:  // Start reading operation
275             if (TWI_Transceiver_Busy()) return SENSOR_FUNC_STILL_WORKING;
276
277             msg[0] = SENSOR_I2C_READ_ADDRESS;
278             TWI_Start_Transceiver_With_Data(msg, 4);
279
```

```
280             sensor.func_step = 2;
281         case 2:  // Finished reading operation
282             if (TWI_Transceiver_Busy()) return SENSOR_FUNC_STILL_WORKING;
283 
284             lastTransOK = TWI_Get_Data_From_Transceiver(msg, 4);
285             sensor.func_step = 0;
286 
287             if (lastTransOK) {
288                 s[0] = msg[1];
289                 s[1] = msg[2];
290                 s[2] = msg[3];
291                 s[3] = '\0';
292                 sensor.error_while_reading = 0;
293                 return SENSOR_FUNC_DONE;
294             } else {
295                 sensor.error_while_reading = 1;
296                 return SENSOR_FUNC_ERROR;
297             }
298         default:
299             sensor.error_while_reading = 1;
300             return SENSOR_FUNC_ERROR;
301     }
302 }  // }}}
303 
304 
305 void sensor_init_configuration() {  // {{{
306     // This must be called AFTER interrupts were enabled and AFTER
307     // TWI_Master has been initialized.
308 
309     // According to avr-libc FAQ, the compiler automatically initializes
        all
310     // variables with zero.
311     //sensor.func_step = 0;
312     //sensor.new_data_available = 0;
313     //sensor.error_while_reading = 0;
314 
315     // Reading from the EEPROM:
316     eeprom_read_block(&sensor.e, &eeprom_sensor, sizeof(SensorEepromData));
317 
318     sensor_set_register_value(
319         SENSOR_REG_CONF_A,
320         SENSOR_CONF_A_SAMPLES_8
321         | SENSOR_CONF_A_RATE_75
322         | SENSOR_CONF_A_BIAS_NORMAL
323     );
324     sensor_set_register_value(
325         SENSOR_REG_CONF_B,
326         SENSOR_CONF_B_GAIN_1_3
327     );
328     sensor_set_register_value(
329         SENSOR_REG_MODE,
330         SENSOR_MODE_CONTINUOUS
331     );
332 }  // }}}
333 
334 
335 // vim:noexpandtab tabstop=4 shiftwidth=4 foldmethod=marker
    foldmarker={{{,}}}
```

APÊNDICE I – hardwareconfig.h e usbconfig.h

```
1  /* Name: hardwareconfig.h
2   *
3   * This file is shared between the main project and the bootloader.
4   * It contains hardware definitions of which ports and pins are used.
5   * Having this separate file avoids copy-pasting these definitions.
6   */
7
8
9  #ifndef __hardwareconfig_h_included__
10 #define __hardwareconfig_h_included__
11
12 /* ---------------------------- Hardware Config
       ---------------------------- */
13
14 #define USB_CFG_IOPORTNAME      D
15 /* This is the port where the USB bus is connected. When you configure it
       to
16  * "B", the registers PORTB, PINB and DDRB will be used.
17  */
18 #define USB_CFG_DMINUS_BIT      0
19 /* This is the bit number in USB_CFG_IOPORT where the USB D- line is
       connected.
20  * This may be any bit in the port.
21  */
22 #define USB_CFG_DPLUS_BIT       2
23 /* This is the bit number in USB_CFG_IOPORT where the USB D+ line is
       connected.
24  * This may be any bit in the port. Please note that D+ must also be
       connected
25  * to interrupt pin INT0! [You can also use other interrupts, see section
26  * "Optional MCU Description" below, or you can connect D- to the
       interrupt, as
27  * it is required if you use the USB_COUNT_SOF feature. If you use D- for
       the
28  * interrupt, the USB interrupt will also be triggered at Start-Of-Frame
29  * markers every millisecond.]
30  */
31 #define USB_CFG_CLOCK_KHZ       (F_CPU/1000)
32 /* Clock rate of the AVR in kHz. Legal values are 12000, 12800, 15000,
       16000,
33  * 16500, 18000 and 20000. The 12.8 MHz and 16.5 MHz versions of the code
34  * require no crystal, they tolerate +/- 1% deviation from the nominal
35  * frequency. All other rates require a precision of 2000 ppm and thus a
36  * crystal!
37  * Since F_CPU should be defined to your actual clock rate anyway, you
       should
38  * not need to modify this setting.
39  */
40 #define USB_CFG_CHECK_CRC       0
41 /* Define this to 1 if you want that the driver checks integrity of
       incoming
42  * data packets (CRC checks). CRC checks cost quite a bit of code size and
       are
43  * currently only available for 18 MHz crystal clock. You must choose
44  * USB_CFG_CLOCK_KHZ = 18000 if you enable this option.
45  */
```

```
46
47 /* ------------------------ Optional Hardware Config
       ------------------------- */
48
49 /* #define USB_CFG_PULLUP_IOPORTNAME   D */
50 /* If you connect the 1.5k pullup resistor from D- to a port pin instead of
51  * V+, you can connect and disconnect the device from firmware by calling
52  * the macros usbDeviceConnect() and usbDeviceDisconnect() (see usbdrv.h).
53  * This constant defines the port on which the pullup resistor is
        connected.
54  */
55 /* #define USB_CFG_PULLUP_BIT          4 */
56 /* This constant defines the bit number in USB_CFG_PULLUP_IOPORT (defined
57  * above) where the 1.5k pullup resistor is connected. See description
58  * above for details.
59  */
60
61
62 #endif /* __hardwareconfig_h_included__ */
```

```
1  /* Name: usbconfig.h
2   * Project: V-USB, virtual USB port for Atmel's(r) AVR(r) microcontrollers
3   * Author: Christian Starkjohann
4   * Creation Date: 2005-04-01
5   * Tabsize: 4
6   * Copyright: (c) 2005 by OBJECTIVE DEVELOPMENT Software GmbH
7   * License: GNU GPL v2 (see License.txt), GNU GPL v3 or proprietary
      (CommercialLicense.txt)
8   * This Revision: $Id: usbconfig-prototype.h 785 2010-05-30 17:57:07Z cs $
9   */
10
11 #ifndef __usbconfig_h_included__
12 #define __usbconfig_h_included__
13
14 /*
15  This file was copied from vusb-20100715/usbdrv/usbconfig-prototype.h
16  and then modified to fit this project.
17 */
18
19 /* ---------------------------- Hardware Config
       ---------------------------- */
20
21 /* This section has been moved to "hardwareconfig.h", as it is now shared
22  * between the main project and the bootloader
23  */
24
25 #include "hardwareconfig.h"
26
27 /* ---------------------------- Functional Range
       ---------------------------- */
28
29 #define USB_CFG_HAVE_INTRIN_ENDPOINT    1
30 /* Define this to 1 if you want to compile a version with two endpoints:
       The
31  * default control endpoint 0 and an interrupt-in endpoint (any other
      endpoint
32  * number).
33  */
34 #define USB_CFG_HAVE_INTRIN_ENDPOINT3   0
35 /* Define this to 1 if you want to compile a version with three endpoints:
       The
36  * default control endpoint 0, an interrupt-in endpoint 3 (or the number
37  * configured below) and a catch-all default interrupt-in endpoint as
      above.
38  * You must also define USB_CFG_HAVE_INTRIN_ENDPOINT to 1 for this feature.
39  */
40 #define USB_CFG_EP3_NUMBER              3
41 /* If the so-called endpoint 3 is used, it can now be configured to any
       other
42  * endpoint number (except 0) with this macro. Default if undefined is 3.
43  */
44 /* #define USB_INITIAL_DATATOKEN           USBPID_DATA1 */
45 /* The above macro defines the startup condition for data toggling on the
46  * interrupt/bulk endpoints 1 and 3. Defaults to USBPID_DATA1.
47  * Since the token is toggled BEFORE sending any data, the first packet is
48  * sent with the oposite value of this configuration!
49  */
50 #define USB_CFG_IMPLEMENT_HALT          0
51 /* Define this to 1 if you also want to implement the ENDPOINT_HALT feature
52  * for endpoint 1 (interrupt endpoint). Although you may not need this
      feature,
53  * it is required by the standard. We have made it a config option because
      it
54  * bloats the code considerably.
55  */
56 #define USB_CFG_SUPPRESS_INTR_CODE      0
57 /* Define this to 1 if you want to declare interrupt-in endpoints, but
       don't
58  * want to send any data over them. If this macro is defined to 1,
      functions
59  * usbSetInterrupt() and usbSetInterrupt3() are omitted. This is useful if
```

```
60   * you need the interrupt-in endpoints in order to comply to an interface
61   * (e.g. HID), but never want to send any data. This option saves a couple
62   * of bytes in flash memory and the transmit buffers in RAM.
63   */
64  #define USB_CFG_INTR_POLL_INTERVAL      10
65  /* If you compile a version with endpoint 1 (interrupt-in), this is the
        poll
66   * interval. The value is in milliseconds and must not be less than 10 ms
         for
67   * low speed devices.
68   */
69  #define USB_CFG_IS_SELF_POWERED         0
70  /* Define this to 1 if the device has its own power supply. Set it to 0 if
        the
71   * device is powered from the USB bus.
72   */
73  #define USB_CFG_MAX_BUS_POWER           100
74  /* Set this variable to the maximum USB bus power consumption of your
        device.
75   * The value is in milliamperes. [It will be divided by two since USB
76   * communicates power requirements in units of 2 mA.]
77   */
78  #define USB_CFG_IMPLEMENT_FN_WRITE      0
79  /* Set this to 1 if you want usbFunctionWrite() to be called for
        control-out
80   * transfers. Set it to 0 if you don't need it and want to save a couple of
81   * bytes.
82   */
83  #define USB_CFG_IMPLEMENT_FN_READ       0
84  /* Set this to 1 if you need to send control replies which are generated
85   * "on the fly" when usbFunctionRead() is called. If you only want to send
86   * data from a static buffer, set it to 0 and return the data from
87   * usbFunctionSetup(). This saves a couple of bytes.
88   */
89  #define USB_CFG_IMPLEMENT_FN_WRITEOUT   0
90  /* Define this to 1 if you want to use interrupt-out (or bulk out)
        endpoints.
91   * You must implement the function usbFunctionWriteOut() which receives all
92   * interrupt/bulk data sent to any endpoint other than 0. The endpoint
         number
93   * can be found in 'usbRxToken'.
94   */
95  #define USB_CFG_HAVE_FLOWCONTROL        0
96  /* Define this to 1 if you want flowcontrol over USB data. See the
        definition
97   * of the macros usbDisableAllRequests() and usbEnableAllRequests() in
98   * usbdrv.h.
99   */
100 #define USB_CFG_DRIVER_FLASH_PAGE       0
101 /* If the device has more than 64 kBytes of flash, define this to the 64 k
        page
102  * where the driver's constants (descriptors) are located. Or in other
         words:
103  * Define this to 1 for boot loaders on the ATMega128.
104  */
105 #define USB_CFG_LONG_TRANSFERS          0
106 /* Define this to 1 if you want to send/receive blocks of more than 254
        bytes
107  * in a single control-in or control-out transfer. Note that the capability
108  * for long transfers increases the driver size.
109  */
110 /* #define USB_RX_USER_HOOK(data, len)     if(usbRxToken ==
        (uchar)USBPID_SETUP) blinkLED(); */
111 /* This macro is a hook if you want to do unconventional things. If it is
112  * defined, it's inserted at the beginning of received message processing.
113  * If you eat the received message and don't want default processing to
114  * proceed, do a return after doing your things. One possible application
115  * (besides debugging) is to flash a status LED on each packet.
116  */
117 /* #define USB_RESET_HOOK(resetStarts)
        if(!resetStarts){hadUsbReset();} */
```

```
118 /* This macro is a hook if you need to know when an USB RESET occurs. It
        has
119  * one parameter which distinguishes between the start of RESET state and
        its
120  * end.
121  */
122 /* #define USB_SET_ADDRESS_HOOK()              hadAddressAssigned(); */
123 /* This macro (if defined) is executed when a USB SET_ADDRESS request was
124  * received.
125  */
126 #define USB_COUNT_SOF                   0
127 /* define this macro to 1 if you need the global variable "usbSofCount"
        which
128  * counts SOF packets. This feature requires that the hardware interrupt is
129  * connected to D- instead of D+.
130  */
131 /* #ifdef __ASSEMBLER__
132  * macro myAssemblerMacro
133  *     in      YL, TCNT0
134  *     sts     timer0Snapshot, YL
135  *     endm
136  * #endif
137  * #define USB_SOF_HOOK                    myAssemblerMacro
138  * This macro (if defined) is executed in the assembler module when a
139  * Start Of Frame condition is detected. It is recommended to define it to
140  * the name of an assembler macro which is defined here as well so that
        more
141  * than one assembler instruction can be used. The macro may use the
        register
142  * YL and modify SREG. If it lasts longer than a couple of cycles, USB
        messages
143  * immediately after an SOF pulse may be lost and must be retried by the
        host.
144  * What can you do with this hook? Since the SOF signal occurs exactly
        every
145  * 1 ms (unless the host is in sleep mode), you can use it to tune OSCCAL
        in
146  * designs running on the internal RC oscillator.
147  * Please note that Start Of Frame detection works only if D- is wired to
        the
148  * interrupt, not D+. THIS IS DIFFERENT THAN MOST EXAMPLES!
149  */
150 #define USB_CFG_CHECK_DATA_TOGGLING     0
151 /* define this macro to 1 if you want to filter out duplicate data packets
152  * sent by the host. Duplicates occur only as a consequence of
        communication
153  * errors, when the host does not receive an ACK. Please note that you
        need to
154  * implement the filtering yourself in usbFunctionWriteOut() and
155  * usbFunctionWrite(). Use the global usbCurrentDataToken and a static
        variable
156  * for each control- and out-endpoint to check for duplicate packets.
157  */
158 #define USB_CFG_HAVE_MEASURE_FRAME_LENGTH   0
159 /* define this macro to 1 if you want the function usbMeasureFrameLength()
160  * compiled in. This function can be used to calibrate the AVR's RC
        oscillator.
161  */
162 #define USB_USE_FAST_CRC                0
163 /* The assembler module has two implementations for the CRC algorithm. One
        is
164  * faster, the other is smaller. This CRC routine is only used for
        transmitted
165  * messages where timing is not critical. The faster routine needs 31
        cycles
166  * per byte while the smaller one needs 61 to 69 cycles. The faster routine
167  * may be worth the 32 bytes bigger code size if you transmit lots of data
        and
168  * run the AVR close to its limit.
169  */
170
171 /* ---------------------------- Device Description
        ---------------------------- */
```

```
172
173 // Note: "obdev.at" does not have a VID/PID combination for devices that
        are
174 // both Mouse and Keyboard at the same time. The VID/PID listed below is
        not
175 // appropriate, but it's being used anyway for the lack of a better
        solution
176 // (and because this project is for academic purposes).
177
178 #define  USB_CFG_VENDOR_ID       0xc0, 0x16 /* = 0x16c0 = 5824 = voti.nl */
179 /* USB vendor ID for the device, low byte first. If you have registered
        your
180  * own Vendor ID, define it here. Otherwise you may use one of obdev's free
181  * shared VID/PID pairs. Be sure to read USB-IDs-for-free.txt for rules!
182  * *** IMPORTANT NOTE ***
183  * This template uses obdev's shared VID/PID pair for Vendor Class devices
184  * with libusb: 0x16c0/0x5dc.  Use this VID/PID pair ONLY if you understand
185  * the implications!
186  */
187
188 //#define  USB_CFG_DEVICE_ID       0xdb, 0x27 /* = 0x27db = 10203 = For
        USB Keyboards */
189 //#define  USB_CFG_DEVICE_ID       0xda, 0x27 /* = 0x27da = 10202 = For
        USB Mice */
190 #define  USB_CFG_DEVICE_ID       0xd9, 0x27 /* = 0x27d9 = 10201 = For
        generic HID class devices */
191 // XXX: obdev does not have a VID/PID for a device that works as both mouse
192 // and keyboard at the same time, and thus this project uses an inadequate
193 // VID/PID pair, for the lack of a more appropriate one. DO NOT let this
194 // device with this VID/PID pair go out of your lab.
195 /* This is the ID of the product, low byte first. It is interpreted in the
196  * scope of the vendor ID. If you have registered your own VID with usb.org
197  * or if you have licensed a PID from somebody else, define it here.
          Otherwise
198  * you may use one of obdev's free shared VID/PID pairs. See the file
199  * USB-IDs-for-free.txt for details!
200  * *** IMPORTANT NOTE ***
201  * This template uses obdev's shared VID/PID pair for Vendor Class devices
202  * with libusb: 0x16c0/0x5dc.  Use this VID/PID pair ONLY if you understand
203  * the implications!
204  */
205
206 #define USB_CFG_DEVICE_VERSION  0x00, 0x01
207 /* Version number of the device: Minor number first, then major number.
208  */
209
210 //#define USB_CFG_VENDOR_NAME     'o', 'b', 'd', 'e', 'v', '.', 'a', 't'
211 //#define USB_CFG_VENDOR_NAME_LEN 8
212 #define USB_CFG_VENDOR_NAME     'd', 'e', 'n', 'i', 'l', 's', 'o', 'n',
        's', 'a', '@', 'g', 'm', 'a', 'i', 'l', '.', 'c', 'o', 'm'
213 #define USB_CFG_VENDOR_NAME_LEN 20
214 /* These two values define the vendor name returned by the USB device. The
        name
215  * must be given as a list of characters under single quotes. The
          characters
216  * are interpreted as Unicode (UTF-16) entities.
217  * If you don't want a vendor name string, undefine these macros.
218  * ALWAYS define a vendor name containing your Internet domain name if you
          use
219  * obdev's free shared VID/PID pair. See the file USB-IDs-for-free.txt for
220  * details.
221  */
222
223 //#define USB_CFG_DEVICE_NAME     'T', 'e', 'm', 'p', 'l', 'a', 't', 'e'
224 //#define USB_CFG_DEVICE_NAME_LEN 8
225 #define USB_CFG_DEVICE_NAME     'A', 'T', 'm', 'e', 'g', 'a', '8', ' ',
        'M', 'a', 'g', 'n', 'e', 't', 'o', 'm', 'e', 't', 'e', 'r', ' ', 'U',
        'S', 'B', ' ', 'M', 'o', 'u', 's', 'e'
226 #define USB_CFG_DEVICE_NAME_LEN 30
227 /* Same as above for the device name. If you don't want a device name,
        undefine
228  * the macros. See the file USB-IDs-for-free.txt before you assign a name
```

```
      if
229  * you use a shared VID/PID.
230  */
231
232 /*#define USB_CFG_SERIAL_NUMBER   'N', 'o', 'n', 'e' */
233 /*#define USB_CFG_SERIAL_NUMBER_LEN   0 */
234 /* Same as above for the serial number. If you don't want a serial number,
235  * undefine the macros.
236  * It may be useful to provide the serial number through other means than
      at
237  * compile time. See the section about descriptor properties below for how
238  * to fine tune control over USB descriptors such as the string descriptor
239  * for the serial number.
240  */
241
242 #define USB_CFG_DEVICE_CLASS        0    /* set to 0 if deferred to
      interface */
243 #define USB_CFG_DEVICE_SUBCLASS     0
244 /* See USB specification if you want to conform to an existing device
      class.
245  * Class 0xff is "vendor specific".
246  */
247 #define USB_CFG_INTERFACE_CLASS     0x03    /* HID class */
248 #define USB_CFG_INTERFACE_SUBCLASS  0       /* no boot interface */
249 #define USB_CFG_INTERFACE_PROTOCOL  0       /* no protocol */
250 //#define USB_CFG_INTERFACE_SUBCLASS  1       /* boot interface */
251 //#define USB_CFG_INTERFACE_PROTOCOL  2       /* mouse protocol */
252 /* See USB specification if you want to conform to an existing device
      class or
253  * protocol. The following classes must be set at interface level:
254  * HID class is 3, no subclass and protocol required (but may be useful!)
255  * CDC class is 2, use subclass 2 and protocol 1 for ACM
256  */
257 #define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH    82
258 /* Define this to the length of the HID report descriptor, if you implement
259  * an HID device. Otherwise don't define it or define it to 0.
260  * If you use this define, you must add a PROGMEM character array named
261  * "usbHidReportDescriptor" to your code which contains the report
       descriptor.
262  * Don't forget to keep the array and this define in sync!
263  */
264
265 /* #define USB_PUBLIC static */
266 /* Use the define above if you #include usbdrv.c instead of linking
      against it.
267  * This technique saves a couple of bytes in flash memory.
268  */
269
270 /* ------------------- Fine Control over USB Descriptors
      ------------------- */
271 /* If you don't want to use the driver's default USB descriptors, you can
272  * provide our own. These can be provided as (1) fixed length static data
       in
273  * flash memory, (2) fixed length static data in RAM or (3) dynamically at
274  * runtime in the function usbFunctionDescriptor(). See usbdrv.h for more
275  * information about this function.
276  * Descriptor handling is configured through the descriptor's properties.
       If
277  * no properties are defined or if they are 0, the default descriptor is
       used.
278  * Possible properties are:
279  *   + USB_PROP_IS_DYNAMIC: The data for the descriptor should be fetched
280  *     at runtime via usbFunctionDescriptor(). If the usbMsgPtr mechanism
       is
281  *     used, the data is in FLASH by default. Add property USB_PROP_IS_RAM
       if
282  *     you want RAM pointers.
283  *   + USB_PROP_IS_RAM: The data returned by usbFunctionDescriptor() or
       found
284  *     in static memory is in RAM, not in flash memory.
285  *   + USB_PROP_LENGTH(len): If the data is in static memory (RAM or
       flash),
```

```
286  *     the driver must know the descriptor's length. The descriptor itself
       is
287  *     found at the address of a well known identifier (see below).
288  * List of static descriptor names (must be declared PROGMEM if in flash):
289  *   char usbDescriptorDevice[];
290  *   char usbDescriptorConfiguration[];
291  *   char usbDescriptorHidReport[];
292  *   char usbDescriptorString0[];
293  *   int usbDescriptorStringVendor[];
294  *   int usbDescriptorStringDevice[];
295  *   int usbDescriptorStringSerialNumber[];
296  * Other descriptors can't be provided statically, they must be provided
297  * dynamically at runtime.
298  *
299  * Descriptor properties are or-ed or added together, e.g.:
300  * #define USB_CFG_DESCR_PROPS_DEVICE   (USB_PROP_IS_RAM |
       USB_PROP_LENGTH(18))
301  *
302  * The following descriptors are defined:
303  *   USB_CFG_DESCR_PROPS_DEVICE
304  *   USB_CFG_DESCR_PROPS_CONFIGURATION
305  *   USB_CFG_DESCR_PROPS_STRINGS
306  *   USB_CFG_DESCR_PROPS_STRING_0
307  *   USB_CFG_DESCR_PROPS_STRING_VENDOR
308  *   USB_CFG_DESCR_PROPS_STRING_PRODUCT
309  *   USB_CFG_DESCR_PROPS_STRING_SERIAL_NUMBER
310  *   USB_CFG_DESCR_PROPS_HID
311  *   USB_CFG_DESCR_PROPS_HID_REPORT
312  *   USB_CFG_DESCR_PROPS_UNKNOWN (for all descriptors not handled by the
       driver)
313  *
314  * Note about string descriptors: String descriptors are not just strings,
       they
315  * are Unicode strings prefixed with a 2 byte header. Example:
316  * int  serialNumberDescriptor[] = {
317  *     USB_STRING_DESCRIPTOR_HEADER(6),
318  *     'S', 'e', 'r', 'i', 'a', 'l'
319  * };
320  */
321
322 #define USB_CFG_DESCR_PROPS_DEVICE                  0
323 #define USB_CFG_DESCR_PROPS_CONFIGURATION           0
324 #define USB_CFG_DESCR_PROPS_STRINGS                 0
325 #define USB_CFG_DESCR_PROPS_STRING_0                0
326 #define USB_CFG_DESCR_PROPS_STRING_VENDOR           0
327 #define USB_CFG_DESCR_PROPS_STRING_PRODUCT          0
328 #define USB_CFG_DESCR_PROPS_STRING_SERIAL_NUMBER    0
329 #define USB_CFG_DESCR_PROPS_HID                     0
330 #define USB_CFG_DESCR_PROPS_HID_REPORT              0
331 #define USB_CFG_DESCR_PROPS_UNKNOWN                 0
332
333 /* ------------------------ Optional MCU Description
       ------------------------ */
334
335 /* The following configurations have working defaults in usbdrv.h. You
336  * usually don't need to set them explicitly. Only if you want to run
337  * the driver on a device which is not yet supported or with a compiler
338  * which is not fully supported (such as IAR C) or if you use a differnt
339  * interrupt than INT0, you may have to define some of these.
340  */
341 /* #define USB_INTR_CFG            MCUCR */
342 /* #define USB_INTR_CFG_SET        ((1 << ISC00) | (1 << ISC01)) */
343 /* #define USB_INTR_CFG_CLR        0 */
344 /* #define USB_INTR_ENABLE         GIMSK */
345 /* #define USB_INTR_ENABLE_BIT     INT0 */
346 /* #define USB_INTR_PENDING        GIFR */
347 /* #define USB_INTR_PENDING_BIT    INTF0 */
348 /* #define USB_INTR_VECTOR         INT0_vect */
349
350 #endif /* __usbconfig_h_included__ */
```

APÊNDICE J – Makefile

```
### Microcontroller and programmer configuration ###

# $(GCC_MCU) is passed to GCC
GCC_MCU = atmega8

# $(AVRDUDE_MCU) is passed to avrdude
AVRDUDE_MCU = atmega8

# Microcontroller clock
F_CPU = 12000000

# For parallel port programmer:
#AVRDUDE_PARAMS = -c bsd -P /dev/parport0 -E noreset
# For USBasp
AVRDUDE_PARAMS = -c usbasp

# Should we reserve space for the bootloader?
# This option also sets the correct FUSE bytes.
BOOTLOADER_ENABLED = 0

# BOOTLOADER_ADDRESS is 1800 for 8k devices, 3800 for 16k and 7800 for 32k.
# The datasheet says the start address is at 0xC00, but that is indexed in
# words. Multiplying that number by 2 we get 0x1800, indexed in bytes.
BOOTLOADER_ADDRESS = 1800

# In order to fit into ATmega8 8K program space, some pieces of the firmware
# must be disabled.
ENABLE_MOUSE = 1
ENABLE_KEYBOARD = 1
ENABLE_FULL_MENU = 0

# ENABLE_MOUSE:
#   Enables the mouse-emulation code. Required if you want the firmware to work
#   as a mouse.
# ENABLE_KEYBOARD:
#   Enables the keyboard-emulation code and the built-in configuration menu.
#   Required for configuring the device. Also useful for testing and
#   development.
# ENABLE_FULL_MENU:
#   If disabled, removes a few less important items from the built-in menus.
#   Only makes sense when ENABLE_KEYBOARD is 1.
#
#
# Little table of firmware size, as of revision next to 309:a13540b0c33f
#
# MOUSE  KEYBOARD  FULL_MENU  "make combine"  "make all"
#   0       0         0/1       2792 bytes    2870 bytes  (useless)
#   0       1          0        5544 bytes    5492 bytes
#   0       1          1        5852 bytes    6016 bytes
#   1       0         0/1       5540 bytes    5644 bytes
#   1       1          0        8172 bytes    8180 bytes  (no space for bootloader)
```

```
52 #   1       1          1       !8462 bytes   !8700 bytes  (doesn't fit
       into 8K)
53 #
54 #
55 # Too many choices? I'll make this simple for you, just answer these
       questions:
56 #
57 # * How much memory does your device have?
58 # |-> More than 8K, it's better than ATmega8
59 # |   * Then enable everything and don't worry about size!
60 # '-> Exactly 8K, it's the ATmega8
61 #     * Go to the next question.
62 #
63 # * Do you want a bootloader?
64 # |-> YES, I want the bootloader!
65 # |   * Disable the mouse support and write the firmware with keyboard
       support
66 # |     and the full menu.
67 # |   * Use the built-in menu to configure the device.
68 # |   * After you're done with the configuration, enable the mouse support
       and
69 # |     disable the keyboard support, and rewrite the firmware.
70 # |   * If you need to reconfigure, you need to repeat all these steps
       again.
71 # '-> NO, I don't need a bootloader!
72 #     * Enable the mouse and the keyboard support, but disable the full
       menu.
73 #     * Enjoy! It fits into 8K.
74
75
76 ### Configurations that depend on the value of BOOTLOADER_ENABLED ###
77
78 # FUSE bytes
79 #
80 # Note: lfuse should work as either 0x9F or 0xEF
81 # Note: hfuse should be 0xC0 if bootloader is enabled
82 #                    or 0xC1 if bootloader is disabled
83 #
84 # Tip: This site is handy for calculating the fuse bytes:
85 #      http://www.engbedded.com/fusecalc/
86 #
87 # If you prefer, avrdude also supports binary (0b prefix), hexadecimal
88 # (0x prefix), octal (0 prefix) and decimal (no prefix)
89
90 ifeq ($(BOOTLOADER_ENABLED), 1)
91 AVRDUDE_PARAMS_FUSE = -U hfuse:w:0xC0:m -U lfuse:w:0x9F:m
92 else
93 AVRDUDE_PARAMS_FUSE = -U hfuse:w:0xC1:m -U lfuse:w:0x9F:m
94 endif
95
96 # Maximum firmware size
97 # ATmega8 has 8K of flash ROM, but 1024 words (2048 bytes) are reserved
       for the
98 # bootloader
99 ifdef BUILDING_BOOTLOADER
100  CHECKSIZE_CODELIMIT = 2048
101 else
102  ifeq ($(BOOTLOADER_ENABLED), 1)
103   CHECKSIZE_CODELIMIT = 6144
104  else
105   CHECKSIZE_CODELIMIT = 8192
106  endif
107 endif
108
109 # List of objects
110 ifdef BUILDING_BOOTLOADER
111 VUSBOBJS = $(VUSBDIR)/usbdrvasm.o $(VUSBDIR)/oddebug.o
112 MYOBJS =
113 else
114 VUSBOBJS = $(VUSBDIR)/usbdrvasm.o $(VUSBDIR)/oddebug.o $(VUSBDIR)/usbdrv.o
115 MYOBJS = buttons.o int_eeprom.o keyemu.o mouseemu.o menu.o sensor.o
       avr315/TWI_Master.o
116 endif
```

```
117
118 ALLOBJS = $(PROGNAME).o $(VUSBOBJS) $(MYOBJS)
119
120
121 ### Compiling tools configuration ###
122
123 AS  = avr-as
124 CC  = avr-gcc
125 CXX = avr-g++
126
127 NM      = avr-nm
128 OBJCOPY = avr-objcopy
129 OBJDUMP = avr-objdump
130 SIZE    = avr-size
131
132 AVRDUDE = avrdude
133
134 AVRDUDE_PARAMS += -p $(AVRDUDE_MCU)
135
136 ifeq ($(BOOTLOADER_ENABLED), 1)
137 # -D  = Disable auto erase for flash memory
138 #       This is required in order to write main firmware to flash without
139 #       erasing the bootloader.
140 AVRDUDE_PARAMS += -D
141 endif
142
143 # PROGNAME is the main project file (without extension)
144 # VUSBDIR is the V-USB driver source-code directory
145 # CHECKSIZE is the path to the checksize script
146 ifdef BUILDING_BOOTLOADER
147 PROGNAME  = bootloader
148 VUSBDIR   = ../vusb-20100715/usbdrv
149 CHECKSIZE = ../checksize
150 else
151 PROGNAME  = main
152 VUSBDIR   = ./vusb-20100715/usbdrv
153 CHECKSIZE = ./checksize
154 endif
155
156 # Starting with simple, straight-forward CFLAGS:
157 CFLAGS   = -mmcu=$(GCC_MCU) -DF_CPU=$(F_CPU)
158 CFLAGS  += -DBOOTLOADER_ENABLED=$(BOOTLOADER_ENABLED)
159 CFLAGS  += -DENABLE_MOUSE=$(ENABLE_MOUSE)
160 CFLAGS  += -DENABLE_KEYBOARD=$(ENABLE_KEYBOARD)
161 CFLAGS  += -DENABLE_FULL_MENU=$(ENABLE_FULL_MENU)
162 CFLAGS  += -std=c99 -pipe -Os -Wall
163 CFLAGS  += -I./ -I$(VUSBDIR)
164
165 # And other FLAGs as well
166 CXXFLAGS = $(CFLAGS)
167 ASFLAGS  = -Wa,-adhlns=$(subst $(suffix $<),.lst,$<)
168 LDFLAGS  = -Wl,-Map=$(PROGNAME).map
169 LIBS     = -lm
170
171 ifdef BUILDING_BOOTLOADER
172 LDFLAGS += -Wl,--section-start=.text=$(BOOTLOADER_ADDRESS)
173 endif
174
175
176 ### Compiler fine-tuning ###
177
178 # -Wno-pointer-sign because those "char*" and "uchar*" warnings are
      useless.
179 CFLAGS  += -Wno-pointer-sign
180
181 # With a non-returning void main, we can safely ignore this warning.
182 CFLAGS  += -Wno-main
183
184 # People say these are good flags:
185 CFLAGS  += -funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums
186
187 # Warn if the compiler adds padding inside any struct. This will break some
188 # code assumptions when loading/storing partial sensor data from EEPROM.
```

```
189 # -Wpadded
190 #   Warn if padding is included in a structure, either to align an element
        of
191 #   the structure or to align the whole structure.  Sometimes when this
192 #   happens it is possible to rearrange the fields of the structure to
        reduce
193 #   the padding and so make the structure smaller.
194 CFLAGS  += -Wpadded
195
196 # -Wmissing-field-initializers
197 #     Warn if a structure's initializer has some fields missing.
198 CFLAGS  += -Wmissing-field-initializers
199
200 # This saved 112 bytes
201 CFLAGS  += -fno-split-wide-types
202
203 # And I'm not very sure about enabling these:
204 #CFLAGS  += -fno-move-loop-invariants -fno-tree-scev-cprop
        -fno-inline-small-functions
205 # But these saved 118 bytes:
206 CFLAGS  += -fno-move-loop-invariants -fno-tree-scev-cprop
207
208 # From GCC manpage:
209 #   -mcall-prologues
210 #       Functions prologues/epilogues expanded as call to appropriate
        subroutines.
211 #       Code size will be smaller.
212 # However, in my experiment it actually increased the code size by 54~86
        bytes.
213 #CFLAGS  += -mcall-prologues
214
215 # From GCC manpage:
216 #   -mtiny-stack
217 #     Change only the low 8 bits of the stack pointer.
218 # This saves only 10 bytes.
219 CFLAGS  += -mtiny-stack
220
221 # -fms-extensions is required to accept anonymous structures and unions.
222 CFLAGS  += -fms-extensions
223
224 # Supposedly, -ffreestanding together with a non-returning main would save
        a
225 # few bytes. But in fact this flag increases the firmware by 8 or 14 bytes,
226 # even with a non-returning main.
227 # Also, it gives a linking error about 'fabs'.
228 #CFLAGS  += -ffreestanding
229
230 # Setting the cost of inlining small functions.
231 # My gcc 4.5.2 doesn't support inline-call-cost, maybe it was removed in
        newer
232 # GCC versions?
233 # Also, enabling those other options increased the firmware size.
234 #CFLAGS  += --param inline-call-cost=2 -finline-limit=3
        -fno-inline-small-functions
235 #CFLAGS  += -finline-limit=3 -fno-inline-small-functions
236
237 # Replace CALL statements with RCALL where possible to save a few bytes.
238 LDFLAGS += -Wl,--relax
239
240 # Don't include unused functions and data.
241 CFLAGS  += -ffunction-sections -fdata-sections
242 LDFLAGS += -Wl,--gc-sections -Wl,--print-gc-sections
243
244 # Compile all *.c files at once, allowing for better optimizations.
245 # Note: -combine has been removed in GCC 4.6, in favor of LTO
246 #       http://gcc.gnu.org/bugzilla/show_bug.cgi?id=29171#c7
247 #       http://gcc.gnu.org/wiki/LinkTimeOptimization
248 COMBINE_FLAGS = -combine -fwhole-program
249
250 # See this page for more compiler optimisation suggestions:
251 # http://www.tty1.net/blog/2008-04-29-avr-gcc-optimisations_en.html
252
253
```

```
254 #LDFLAGS += $(GENTOO_LD_PATH_WORKAROUND)
255 #GENTOO_LD_PATH_WORKAROUND = -L/usr/i686-pc-linux-gnu/avr/lib
256 #GENTOO_LD_PATH_WORKAROUND = -L/usr/x86_64-pc-linux-gnu/avr/lib
257 #
258 # See:
259 # http://bugs.gentoo.org/show_bug.cgi?id=147155
260 # http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=336170
261 #
262 # ln -snf ../../lib/binutils/avr/2.*/ldscripts/ /usr/avr/lib/ldscripts
263
264
265 ### Make targets ###
266
267 #Basic rules
268 .PHONY: all normal-build combine combine-build post-build help clean boot
        writeboot writeflash writeeeprom writefuse erase dump comments size
269
270 all: normal-build post-build
271
272 combine: combine-build post-build
273
274 normal-build: $(ALLOBJS)
275     $(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) -o $(PROGNAME).elf $^ $(LIBS)
276
277 combine-build: $(VUSBOBJS)
278 # Combined compiling does not work for VUSB source-code. Thus, we add
        VUSBOBJS
279 # as dependencies of this target.
280     $(CC) $(CPPFLAGS) $(CFLAGS) $(COMBINE_FLAGS) $(LDFLAGS) \
281         -o $(PROGNAME).elf \
282         $(PROGNAME).c $(MYOBJS:.o=.c) \
283         $(VUSBOBJS) $(LIBS)
284
285 post-build: $(PROGNAME).elf $(PROGNAME).hex $(PROGNAME).eep $(PROGNAME).lss
286     $(CHECKSIZE) $(PROGNAME).elf $(CHECKSIZE_CODELIMIT)
287
288 help:
289     @echo 'make all          - Builds the project'
290     @echo 'make combine      - Compiles all *.c at the same time, allowing
            some compiler optimizations'
291     @echo 'make clean        - Deletes all built files'
292     @echo
293     @echo 'make boot         - Builds the bootloader (please run "make
            clean" before)'
294     @echo 'make writeboot    - Writes the bootloader to flash memory
            (please run "make clean" after)'
295     @echo
296     @echo 'make writeflash   - Writes the flash memory of the
            microcontroller'
297     @echo 'make writeeeprom  - Writes the EEPROM of the microcontroller'
298     @echo 'make writefuse    - Writes the fuse bytes of the microcontroller'
299     @echo 'make erase        - Performs a chip erase'
300     @echo 'make dump         - Dumps all memory from the microcontroller'
301     @echo
302     @echo 'make comments     - Prints all TODO/FIXME/XXX comments'
303     @echo 'make size         - Prints the size of all functions/symbols'
304
305 clean:
306     rm -f $(PROGNAME).{o,s,elf,hex,eep,lss,sym,lst,map}
307 ifndef BUILDING_BOOTLOADER
308     rm -f $(ALLOBJS)
309     rm -f $(ALLOBJS:.o=.s)
310     rm -f $(ALLOBJS:.o=.lst)
311     rm -f $(ALLOBJS:.o=.map)
312     cd bootloader && $(MAKE) -f ../Makefile BUILDING_BOOTLOADER=1 clean
313 endif
314
315 boot:
316     cd bootloader && $(MAKE) -f ../Makefile BUILDING_BOOTLOADER=1 all
317
318 writeboot:
319     $(AVRDUDE) $(AVRDUDE_PARAMS) \
```

```
320         -U flash:w:bootloader/bootloader.hex:i
321
322 writeflash:
323     $(AVRDUDE) $(AVRDUDE_PARAMS) \
324         -U flash:w:$(PROGNAME).hex:i
325
326 writeeeprom:
327     $(AVRDUDE) $(AVRDUDE_PARAMS) \
328         -U eeprom:w:$(PROGNAME).eep:i
329
330 writefuse:
331     $(AVRDUDE) $(AVRDUDE_PARAMS) \
332         $(AVRDUDE_PARAMS_FUSE)
333
334 erase:
335     $(AVRDUDE) $(AVRDUDE_PARAMS) \
336         -e
337 dump:
338     $(AVRDUDE) $(AVRDUDE_PARAMS) \
339         -U flash:r:flash.dump:i \
340         -U eeprom:r:eeprom.dump:i \
341         -U hfuse:r:hfuse.dump:b \
342         -U lfuse:r:lfuse.dump:b \
343         -U lock:r:lock.dump:b \
344         -U signature:r:signature.dump:h
345 # efuse is available on newer AVR microcontrollers
346 #       -U efuse:r:efuse.dump:b
347
348 comments:
349     ack --nomake 'TODO|FIXME|XXX'
350 # I could have used grep, but ack is so much easier and prettier!
351 # http://betterthangrep.com/
352 # http://search.cpan.org/dist/ack/
353 #
354 # Note: Debian (and Ubuntu?) users have this tool installed as "ack-grep"
355
356 size:
357 # -t d  =>  use decimal radix
358 # -A    =>  also print the filename
359 # Sample output:
360 # main.o:00001546 T main
361 #
362 # It's possible to either pass *.o or *.elf
363     $(NM) -f bsd -t d -A --size-sort $(PROGNAME).elf | \
364         sed 's/^\([^:]\+\):\([0-9a-fA-F]\+\) \(.\) \(.\+\)$$/\2 \3 \4
               [\1]/' | \
365         sort -n
366
367
368 # Dependencies
369 # Note: Header dependencies for individual objects are not listed here.
370 #       Please run "make clean" after you edit any header.
371 $(PROGNAME).s: $(PROGNAME).c usbconfig.h $(MYOBJS:.o=.h)
372 $(PROGNAME).o: $(PROGNAME).c usbconfig.h $(MYOBJS:.o=.h)
373
374 # This has been commented-out in order to support combined compiling.
375 #$(PROGNAME).elf: $(ALLOBJS)
376
377
378 # The variables:
379 # $@ - The name of the target of the rule.
380 # $? - The names of all the prerequisites that are newer than the target.
381 # $< - The name of the first prerequisite.
382 # $^ - The names of all the prerequisites.
383
384 # Pattern-rules:
385 %.s: %.c
386     $(CC) -S $(CPPFLAGS) $(CFLAGS) $(ASFLAGS) -o $@ $<
387 %.o: %.c
388     $(CC) -c $(CPPFLAGS) $(CFLAGS) $(ASFLAGS) -o $@ $<
389 %.o: %.cpp
390     $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $(ASFLAGS) -o $@ $<
391 %.o: %.S
```

```
392     $(CC) -c $(CPPFLAGS) $(CFLAGS) $(ASFLAGS) -o $@ $<
393 # "-x assembler-with-cpp" should not be necessary since this is the default
394 # file type for the .S (with capital S) extension. However, upper case
395 # characters are not always preserved on Windows. Add this flag to ensure
396 # WinAVR compatibility.
397
398 %.o: %.h
399
400 #%.elf: %.o
401 #   $(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) -o $@ $^
402
403 # Create final output files (.hex, .eep) from ELF output file.
404 # Use -O ihex for Intel Hex format
405 # Use -O binary for binary format
406 %.hex: %.elf
407     $(OBJCOPY) -O ihex -R .eeprom $< $@
408 #   $(OBJCOPY) -j .text -j .data -O ihex $< $@
409 %.eep: %.elf
410     $(OBJCOPY) -j .eeprom --set-section-flags=.eeprom="alloc,load" \
411     --change-section-lma .eeprom=0 -O ihex $< $@
412
413 # Create extended listing file from ELF output file.
414 %.lss: %.elf
415     $(OBJDUMP) -h -S -C $< > $@
416
417 # Create a symbol table from ELF output file.
418 %.sym: %.elf
419     $(NM) -n $< > $@
```

APÊNDICE K – generate_sphere_vectors.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# vi:ts=4 sw=4 et

from __future__ import print_function

import sys
import time

from math import sin, cos, radians, degrees


# argparse is beautiful!
# This var will be written by parse_args()
options = None


def parse_args(args=None):
    global options

    import argparse

    class HackishFormatterClass(argparse.ArgumentDefaultsHelpFormatter,
        argparse.RawDescriptionHelpFormatter):
        pass

    parser = argparse.ArgumentParser(
        description='Generates 3D vectors (x,y,z) around a sphere',
        epilog=
        'When printing the calibration vectors, the "aperture angles"
            define how far\n'
        'apart are each corner of the pyramid, while the "offset angles"
            define the\n'
        'orientation of the pyramid.\n'
        '\n'
        'Remember: pitch=theta=vertical [-90, 90]; yaw=phi=horizontal
            [-180, 180]',
        # If the next line breaks, try one of the others:
        formatter_class=HackishFormatterClass
        #formatter_class=argparse.ArgumentDefaultsHelpFormatter
        #formatter_class=argparse.RawDescriptionHelpFormatter
    )

    parser.add_argument(
        '-r', '--radius',
        action='store',
        type=int,
        default=200,
        help='The radius of the sphere, i.e. the size (norm) of each
            vector'
    )
    parser.add_argument(
        '--phi-range', '--h-range',
        action='store',
        type=int,
        nargs=3,
```

```
52          default=(90, -91, -1),
53          metavar=('START', 'STOP', 'STEP'),
54          dest='phi_range',
55          help='Phi (horizontal) range when printing the vectors, passed
                directly to Python\'s range() function'
56      )
57      parser.add_argument(
58          '--theta-range', '--v-range',
59          action='store',
60          type=int,
61          nargs=3,
62          default=(45, -45, -2),
63          metavar=('START', 'STOP', 'STEP'),
64          dest='theta_range',
65          help='Theta (vertical) range when printing the vectors, passed
                directly to Python\'s range() function'
66      )
67
68      parser.add_argument(
69          '-P', '--PHI', '--h-aperture',
70          action='store',
71          type=int,
72          default=45,
73          metavar='ANGLE',
74          dest='phi_aperture',
75          help='Aperture angle (horizontal - phi) for the calibration
                pyramid'
76      )
77      parser.add_argument(
78          '-T', '--THETA', '--v-aperture',
79          action='store',
80          type=int,
81          default=45,
82          metavar='ANGLE',
83          dest='theta_aperture',
84          help='Aperture angle (vertical - theta) for the calibration
                pyramid'
85      )
86      parser.add_argument(
87          '-p', '--phi', '--h-offset',
88          action='store',
89          type=int,
90          default=0,
91          metavar='ANGLE',
92          dest='phi_offset',
93          help='Offset angle (horizontal - phi) for the calibration pyramid'
94      )
95      parser.add_argument(
96          '-t', '--theta', '--v-offset',
97          action='store',
98          type=int,
99          default=0,
100         metavar='ANGLE',
101         dest='theta_offset',
102         help='Offset angle (vertical - theta) for the calibration pyramid'
103     )
104     parser.add_argument(
105         '-C', '--no-calibration',
106         action='store_true',
107         dest='omit_calibration',
108         help='Don\'t print the calibration vectors at the start'
109     )
110     parser.add_argument(
111         '-s', '--sleep',
112         action='store',
113         type=float,
114         default=0.0,
115         metavar='MS',
116         dest='sleep_ms',
117         help='Sleep MS milliseconds after each theta value'
```

```
118     )
119
120     options = parser.parse_args(args)
121
122
123 def print_calibration():
124     global options
125
126     pitch = options.theta_aperture/2
127     yaw = options.phi_aperture/2
128
129     offsets = [
130         #(delta_theta, delta_phi, name)
131         (+pitch, +yaw, 'topleft'),
132         (+pitch, -yaw, 'topright'),
133         (-pitch, -yaw, 'bottomright'),
134         (-pitch, +yaw, 'bottomleft'),
135     ]
136     for delta_theta, delta_phi, name in offsets:
137         x, y, z = spherical_to_cartesian(
138             options.theta_offset + delta_theta,
139             options.phi_offset + delta_phi
140         )
141         print(name)
142         print('{0}\t{1}\t{2}'.format(x, y, z))
143
144 def spherical_to_cartesian(theta, phi):
145     global options
146
147     # These are not exactly the same equations as in:
148     # http://en.wikipedia.org/wiki/Spherical_coordinates
149
150     x = options.radius * cos(radians(theta)) * cos(radians(phi))
151     y = options.radius * cos(radians(theta)) * sin(radians(phi))
152     z = options.radius * sin(radians(theta))
153
154     x = int(round(x))
155     y = int(round(y))
156     z = int(round(z))
157
158     return x, y, z
159
160 def main():
161     global options
162
163     parse_args()
164
165     if not options.omit_calibration:
166         print_calibration()
167
168     for theta in xrange(*options.theta_range):
169     #for theta in xrange(45, -45, -2):
170     #for theta in xrange(75, -75, -5):
171         for phi in xrange(*options.phi_range):
172         #for phi in xrange(-90, 90, 1):
173         #for phi in xrange(0, 360, 5):
174             x, y, z = spherical_to_cartesian(theta, phi)
175             print('{0}\t{1}\t{2}'.format(x, y, z))
176
177         if options.sleep_ms > 0:
178             # Flushing the output
179             sys.stdout.flush()
180
181             # Printing the current theta value
182             #sys.stderr.write('theta={0}\n'.format(theta))
183             #sys.stderr.flush()
184
185             time.sleep(options.sleep_ms/1000.0)
186
187
188 if __name__ == "__main__":
189     main()
```

APÊNDICE L – convert_coordinates.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# vi:ts=4 sw=4 et

# This code is not very pretty, but works.

import numpy
import re
import sys
import time

from numpy import matrix, array, cross, dot, rad2deg, deg2rad
from numpy.linalg import norm


def vector(param):
    '''Constructor for a 3D vector using numpy.array'''
    return numpy.array(param, dtype='f')

def vector_from_string(s):
    try:
        x,y,z = [float(i) for i in s.split()]
        return vector([x,y,z])
    except:
        return None


cos_between_vectors = lambda A,B: dot(A,B)/norm(A)/norm(B)


class State(object):
    CALIBRATION_NAMES = ['topleft', 'topright', 'bottomright',
       'bottomleft']

    def __init__(self):
        self.DEBUG = False
        for name in self.CALIBRATION_NAMES:
            setattr(self, name, vector([0,0,0]))


    def single_edge_interpolation(self, A, B, C, using):
        # A and B are one of the corners.
        # C is the currently pointed value.
        #
        # Returns None in case of errors.
        #
        # Returns None if C is at the opposite side of N, when looking at
           the
        # plane AOB. In other words, if the angle between N and C is
           greater
        # than 90 degrees.
        #
        # Else, returns a value between 0.0 and 1.0, which means how close
           to A
        # is C, inside the segment AB.
        #
        #
```

```
54          # N ^
55          #   |   _
56          #   |  .'| B (pointing inside this drawing)
57          #   |.'
58          # O *--------> A
59          #
60          # N = A x B
61
62          # N is normal to the plane of A and B
63          N = cross(A, B)
64          # Converting to unit vector
65          N /= norm(N)
66
67          # Dot product between N and C
68          NdotC = dot(N, C)
69
70          # Checking the side of C, in relation to N and plane AB
71          if NdotC < 0:
72              return None
73
74
75          # Clinha is the projection of C onto AB plane.
76          Clinha = C - NdotC * N
77
78          # Clinha is the projection of C onto AB plane.
79          # I don't care about the size of this vector, only about the
               direction.
80          #
               http://www.euclideanspace.com/maths/geometry/elements/plane/lineOnPlane/in
81          #Clinha = cross(N, cross(C,N))
82
83          if state.DEBUG:
84              print "N", repr(N)
85              print "Clinha", repr(Clinha)
86
87          # Comparing the cossines...
88          # I could compare the angles, but that would need arccos()
               function,
89          # while I can calculate the cossines directly by dot-product and
90          # division.
91          # Bah... nevermind... I'm going to compare the angles until I get a
92          # better solution
93          cos_AB = cos_between_vectors(A, B)
94          cos_AC = cos_between_vectors(A, Clinha)
95          cos_BC = cos_between_vectors(B, Clinha)
96          if self.DEBUG:
97              print "cos_AB", cos_AB
98              print "cos_AC", cos_AC
99              print "cos_BC", cos_BC
100
101         # Check if inside the bounds of AB
102         #if cos_AC < cos_AB or cos_BC < cos_AB:
103         #    return None
104
105         if not (
106             -1 <= cos_AB <= 1 and
107             -1 <= cos_AC <= 1 and
108             -1 <= cos_BC <= 1
109         ):
110             return None
111
112         if using == 'angle':
113             # Calculate the proportion based on angle
114             ang_AB = numpy.arccos(cos_AB)
115             ang_AC = numpy.arccos(cos_AC)
116             ang_BC = numpy.arccos(cos_BC)
117             if self.DEBUG:
118                 print "ang_AB", ang_AB
119                 print "ang_AC", ang_AC
120                 print "ang_BC", ang_BC
121
122             # Return the proportion
```

```
123             return ang_AC / ang_AB
124         elif using == 'cos':
125             # Calculate the proportion based on cos
126             # Return the proportion
127             return (1 - cos_AC) / (1 - cos_AB)
128         elif using == 'sin':
129             # Calculate the proportion based on sin
130             sin_AB = numpy.sqrt(1 - cos_AB**2)
131             sin_AC = numpy.sqrt(1 - cos_AC**2)
132             sin_BC = numpy.sqrt(1 - cos_BC**2)
133             if self.DEBUG:
134                 print "sin_AB", sin_AB
135                 print "sin_AC", sin_AC
136                 print "sin_BC", sin_BC
137 
138             # Return the proportion
139             return sin_AC / sin_AB
140         elif using == 'tan':
141             # Calculate the proportion based on tangent
142             sin_AB = numpy.sqrt(1 - cos_AB**2)
143             sin_AC = numpy.sqrt(1 - cos_AC**2)
144             sin_BC = numpy.sqrt(1 - cos_BC**2)
145 
146             tan_AB = sin_AB / cos_AB
147             tan_AC = sin_AC / cos_AC
148             tan_BC = sin_BC / cos_BC
149 
150             if self.DEBUG:
151                 print "tan_AB", tan_AB
152                 print "tan_AC", tan_AC
153                 print "tan_BC", tan_BC
154 
155             # Return the proportion
156             return tan_AC / tan_AB
157         elif using == 'dist':
158             A /= norm(A)
159             B /= norm(B)
160             Clinha /= norm(Clinha)
161 
162             # Calculate the proportion based on distances
163             dist_AB = norm(A-B)
164             dist_AC = norm(A-Clinha)
165             dist_BC = norm(B-Clinha)
166 
167             if self.DEBUG:
168                 print "dist_AB", dist_AB
169                 print "dist_AC", dist_AC
170                 print "dist_BC", dist_BC
171 
172             # Return the proportion
173             return dist_AC / dist_AB
174         elif using == 'exact':
175             # We need to find the exact intersection between AB and Clinha
176             # We have two parametric lines:
177             #  A + alpha*(B-A)   =>  all points from the edge
178             #  0 + beta *Clinha  =>  the pointed direction (already
                    projected)
179             # We know these two lines are at the same plane.
180             # So, we use a 2D coordinate system like this:
181             X = A
182             Y = cross(X, N)
183             X /= norm(X)
184             Y /= norm(Y)
185             # The base vectors are unitary and orthogonal.
186 
187             # Projecting Clinha to this sytem:
188             Cx = dot(Clinha, X)
189             Cy = dot(Clinha, Y)
190 
191             Ax = dot(A, X)
192             Ay = dot(A, Y)
```

```
193
194             BAx = dot(B-A, X)
195             BAy = dot(B-A, Y)
196
197             #  A + alpha*(B-A)  =  0 + beta*Clinha
198             M = array([
199                 [BAx, -Cx],
200                 [BAy, -Cy],
201             ])
202             constant = array([-Ax, -Ay])
203
204             try:
205                 sol = numpy.linalg.solve(M, constant)
206                 # alpha = sol[0]
207                 # beta = sol[1]
208                 return sol[0]
209
210             except numpy.linalg.LinAlgError:
211                 return None
212
213     def interpolation_using_2_edges(self, pointer, using):
214         # This is a very bad approximation
215         x = self.single_edge_interpolation(self.topleft   , self.topright,
                pointer, using)
216         y = self.single_edge_interpolation(self.bottomleft, self.topleft ,
                pointer, using)
217         if None in [x, y]:
218             return (None, None)
219
220         y = 1 - y
221
222         return (x, y)
223
224     def interpolation_using_4_edges(self, pointer, using):
225         # Let:
226         #  A = topleft
227         #  B = topright
228         #  C = bottomright
229         #  D = bottomleft
230         # as 3D vector coordinates
231         #
232         # And let:
233         #  AB = pointer projection at AB edge
234         #  BC = pointer projection at BC edge
235         #  DC = pointer projection at DC edge
236         #  AD = pointer projection at AD edge
237         # as 1D coordinates, already normalized between 0 and 1
238         #
239         # Let's trace a line joining AB and DC, and another joining AD and
                BC.
240         # The intersection of these two lines should be at the 2D screen
241         # coordinate pointed by the user.
242         #
243         # y(x) = x * (BC - AD) + AD
244         #   y(0) = AD
245         #   y(1) = BC
246         #
247         # x(y) = y * (DC - AB) + AB
248         #   x(0) = AB
249         #   x(1) = DC
250         #
251         # Some math later:
252         # x = (AD * (DC - AB) + AB) / (1 - (BC - AD) * (DC - AB))
253
254         AB = self.single_edge_interpolation(self.topleft    ,
                self.topright   , pointer, using)
255         BC = self.single_edge_interpolation(self.topright   ,
                self.bottomright, pointer, using)
256         DC = self.single_edge_interpolation(self.bottomright,
                self.bottomleft , pointer, using)
257         AD = self.single_edge_interpolation(self.bottomleft , self.topleft
```

```
                , pointer, using)

        if self.DEBUG:
            print "AB, BC, DC, AD", AB, BC, DC, AD

        if None in [AB, BC, DC, AD]:
            return (None, None)

        DC = 1 - DC
        AD = 1 - AD

        x = (AD * (DC - AB) + AB) / (1 - (BC - AD) * (DC - AB))
        y = x * (BC - AD) + AD

        return (x, y)

    def interpolation_using_linear_equations(self, pointer):
        # Let ABD be the plane of the screen. (yes, I'm ignoring C)
        # Let A be the topleft, B the topright, and D the bottomleft.
        # Let P be the currently pointed direction.
        # Let X be the point that, at the same time, is contained into ABD
        # plane and P direction.
        # Let (u,v) be the 2D screen coordinates in range 0.0 to 1.0.
        #
        # Thus X = A + u*(B-A) + v*(D-A)
        # And  X = t*P
        # where t is a scalar that is not useful for this program.
        #
        # Thus we can build this linear system:
        # A + u*(B-A) + v*(D-A) = t*P
        # u*(B-A) + v*(D-A) - t*P = -A
        #
        # Or, in matrix notation:
        # [ (B-A) , (D-A) , -P] dot (u,v,t).T = -A

        A = self.topleft
        B = self.topright
        D = self.bottomleft
        P = pointer

        A /= norm(A)
        B /= norm(B)
        D /= norm(D)
        #P /= norm(P)

        col1 = (B-A)
        col2 = (D-A)
        col3 = -P

        #col1 /= norm(col1)
        #col2 /= norm(col2)
        #col3 /= norm(col3)

        M = array([col1, col2, col3])
        M = M.T

        try:
            X = numpy.linalg.solve(M, -A)
            return (X[0], X[1])

        except numpy.linalg.LinAlgError:
            return (None, None)


def reset():
    global state
    state = State()


def parse_args():
    import argparse

    parser = argparse.ArgumentParser(
```

```
330         description='Converts 3D vector coordinates to 2D screen
                coordinates',
331         epilog='3D coordinates are arbitrary, and only the direction is
                taken into account. Before starting the conversion, this
                program needs calibration by setting the 4 corners of the
                screen to 3D coordinates. This program prints 2D coordinates
                between 0.0 and 1.0.',
332         formatter_class=argparse.ArgumentDefaultsHelpFormatter
333     )
334
335     parser.add_argument(
336         '-a', '--algorithm',
337         action='store',
338         type=int,
339         default=7,
340         choices=tuple(range(1,1+13)),
341         help='Use a different algorithm for 3D->2D conversion, read the
                source code to learn the available algorithms'
342     )
343     parser.add_argument(
344         '-s', '--sleep',
345         action='store',
346         type=float,
347         default=0.0,
348         metavar='MS',
349         dest='sleep_ms',
350         help='Sleep MS milliseconds after each printed coordinate'
351     )
352     parser.add_argument(
353         '-f', '--flush',
354         action='store_true',
355         dest='force_flush',
356         help='Force stdout flush after each printed coordinate
                (automatically enabled if --sleep is set)'
357     )
358
359     args = parser.parse_args()
360     return args
361
362
363 def main():
364     re_vector_line = re.compile(r'^\s*([-\d]+)\s*([-\d]+)\s*([-\d]+)')
365
366     global options
367     options = parse_args()
368
369     global state
370     reset()
371     line_number = 0
372
373     while True:
374         line_number += 1
375
376         try:
377             line = raw_input().strip()
378         except EOFError:
379             break
380
381         match_vector_line = re_vector_line.match(line)
382
383         if line == '':
384             # Ignoring empty line
385             pass
386         elif line[0] == '#':
387             # Ignoring comment line
388             pass
389
390         # Commands...
391         elif line.lower() == 'reset':
392             reset()
393         elif line.lower() == 'quit':
394             break
```

```
395         elif line.lower() == 'debug':
396             state.DEBUG = not state.DEBUG
397             print "Debug is now {0}".format("ON" if state.DEBUG else "OFF")
398             sys.stdout.flush()
399         elif line.lower() == 'calibration':
400             # Prints the calibration vectors
401             for name in State.CALIBRATION_NAMES:
402                 print "{0}:\t{1}".format(
403                     name,
404                     repr(getattr(state, name))
405                 )
406             sys.stdout.flush()
407 
408         # Calibration coordinates
409         elif line.lower() in State.CALIBRATION_NAMES:
410             value = raw_input().strip()
411             pointer = vector_from_string(value)
412             if pointer is not None:
413                 setattr(state, line.lower(), pointer)
414 
415         # Reading a coordinate
416         elif match_vector_line:
417             pointer = vector_from_string(line)
418             if pointer is None:
419                 continue
420 
421             if state.DEBUG:
422                 print repr(pointer)
423 
424             # Doing the 3D->2D conversion
425             #
426             # Okay, this sequence of if statements is ugly. A list or dict
427             # would probably be cleaner.
428             if options.algorithm == 1:
429                 x, y = state.interpolation_using_2_edges(pointer,
                        using='angle')
430             elif options.algorithm == 2:
431                 x, y = state.interpolation_using_2_edges(pointer,
                        using='cos')
432             elif options.algorithm == 3:
433                 x, y = state.interpolation_using_2_edges(pointer,
                        using='sin')
434             elif options.algorithm == 4:
435                 x, y = state.interpolation_using_2_edges(pointer,
                        using='tan')
436             elif options.algorithm == 5:
437                 x, y = state.interpolation_using_2_edges(pointer,
                        using='dist')
438             elif options.algorithm == 6:
439                 x, y = state.interpolation_using_2_edges(pointer,
                        using='exact')
440 
441             elif options.algorithm == 7:
442                 x, y = state.interpolation_using_4_edges(pointer,
                        using='angle')
443             elif options.algorithm == 8:
444                 x, y = state.interpolation_using_4_edges(pointer,
                        using='cos')
445             elif options.algorithm == 9:
446                 x, y = state.interpolation_using_4_edges(pointer,
                        using='sin')
447             elif options.algorithm == 10:
448                 x, y = state.interpolation_using_4_edges(pointer,
                        using='tan')
449             elif options.algorithm == 11:
450                 x, y = state.interpolation_using_4_edges(pointer,
                        using='dist')
451             elif options.algorithm == 12:
452                 x, y = state.interpolation_using_4_edges(pointer,
                        using='exact')
```

```
453
454             elif options.algorithm == 13:
455                 x, y = state.interpolation_using_linear_equations(pointer)
456
457             if state.DEBUG:
458                 print "x,y", x, y
459
460             # Printing the final 2D coordinates
461             if x is not None and y is not None:
462                 print "{0} {1}".format(x, y)
463
464                 if options.sleep_ms > 0 or options.force_flush:
465                     sys.stdout.flush()
466
467                 if options.sleep_ms > 0:
468                     #time.sleep(0.015625)  # 2**-6
469                     #time.sleep(2**-8)
470                     time.sleep(options.sleep_ms/1000.0)
471             else:
472                 print "discarded"
473
474         # Fallback for unknown lines
475         else:
476             print "Unrecognized line {0}: {1}".format(line_number, line)
477             sys.stdout.flush()
478
479
480 if __name__ == "__main__":
481     main()
```

APÊNDICE M – draw_points.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# vi:ts=4 sw=4 et

# How to use:
# ./something_that_generates_points | ./draw_points.py
# ./draw_points.py < some_data.txt
#
# Once the window is opened, it can be resized, or it can be closed by
# pressing Esc or Q, or by just closing it.
#
# Upon reading EOF, it will stop reading from stdin, but the window will
# remain open until you close it.

from __future__ import division
from __future__ import print_function

import select
import sys
from itertools import izip
from math import isnan

# Easy way of importing pygame:
# import pygame

# Hard way of importing pygame:
import pygame.display
import pygame.event
import pygame.image
import pygame.time

from pygame.locals import *


class DrawPoints(object):

    def init(self, args):
        # pygame.init() will try to initialize all pygame modules,
        # including Cdrom and Audio. I'm not using such things, so let's
        # initialize the only really required module:
        pygame.display.init()

        if args.persist:
            self.FADE_POINTS = False
            self.MAX_POINTS = 0
            self.colors = []
            self.points = []
        else:
            self.FADE_POINTS = True
            self.MAX_POINTS = 256
            self.colors = [
                Color(i,i,i) for i in range(256)
            ]
            self.points = [ (None, None) ] * self.MAX_POINTS

        self.BLACK = Color(0, 0, 0)
```

```
57          self.WHITE = Color(255, 255, 255)
58
59          self.resolution = args.window_size
60          self.thickness = args.size
61
62          self.poll = select.poll()
63          self.poll.register(sys.stdin, select.POLLIN)
64
65          self.save_as = args.output
66          self.quit_after_eof = args.quit_after_eof
67
68
69      def run(self):
70          self.screen = pygame.display.set_mode(self.resolution, RESIZABLE)
71          pygame.display.set_caption("Drawing points from stdin")
72
73          # Inject a USEREVENT every 10ms
74          pygame.time.set_timer(USEREVENT, 10)
75
76          while True:
77              for event in [pygame.event.wait(),]+pygame.event.get():
78                  if event.type == QUIT:
79                      self.quit()
80                  elif event.type == KEYDOWN and event.key in [K_ESCAPE,
                        K_q]:
81                      self.quit()
82
83                  elif event.type == VIDEORESIZE:
84                      self.resolution = event.size
85                      self.screen = pygame.display.set_mode(self.resolution,
                            RESIZABLE)
86                      self.redraw()
87
88                  elif event.type == USEREVENT:
89                      # poll() returns a list of file-descriptors that are
                            "ready"
90                      while self.poll.poll(0):
91                          line = sys.stdin.readline()
92                          if line == "":
93                              # EOF
94                              self.poll.unregister(sys.stdin)
95                              pygame.time.set_timer(USEREVENT, 0)
96                              if self.quit_after_eof:
97                                  self.quit()
98                          try:
99                              x,y = [float(i) for i in line.strip().split()]
100                         except:
101                             continue
102                         if not isnan(x) and not isnan(y):
103                             self.add_point(x,y)
104                     self.redraw()
105
106
107     def add_point(self, x, y):
108         if self.FADE_POINTS:
109             self.points.append( (x,y) )
110             if len(self.points) > self.MAX_POINTS:
111                 self.points.pop(0)
112         else:
113             self.draw_point(x, y, self.WHITE)
114
115     def draw_point(self, x, y, color):
116         x *= self.resolution[0]
117         y *= self.resolution[1]
118         rect = Rect(
119             x - self.thickness, y - self.thickness,
120             1 + 2 * self.thickness, 1 + 2 * self.thickness
121         )
122         self.screen.fill(color, rect=rect)
123
124     def redraw(self):
```

```
125         if self.FADE_POINTS:
126             self.screen.fill(self.BLACK)
127             for point, color in izip(self.points, self.colors):
128                 if point[0] is None or point[1] is None:
129                     continue
130                 self.draw_point(point[0], point[1], color)
131         else:
132             pass
133 
134         pygame.display.flip()
135 
136 
137     def quit(self, status=0):
138         if self.save_as:
139             print("Saving image to '{0}'".format(self.save_as))
140             pygame.image.save(self.screen, self.save_as)
141 
142         pygame.quit()
143         sys.exit(status)
144 
145 
146 def parse_args():
147     import argparse
148 
149     parser = argparse.ArgumentParser(
150         description="Draws 2D points based on coordinates between 0.0 and
                1.0",
151         formatter_class=argparse.ArgumentDefaultsHelpFormatter
152     )
153     parser.add_argument(
154         '-p', '--persist',
155         action='store_true',
156         help='Persist the drawing, instead of fading the older points.'
157     )
158     parser.add_argument(
159         '-s', '--size',
160         action='store',
161         type=int,
162         default=2,
163         help='The thickness of each "dot". The actual size is "1 +
                2*SIZE".'
164     )
165     parser.add_argument(
166         '-w', '--window',
167         action='store',
168         type=int,
169         nargs=2,
170         default=(640, 480),
171         dest='window_size',
172         help='The size of the window'
173     )
174     parser.add_argument(
175         '-o', '--output',
176         metavar='FILE',
177         action='store',
178         type=str,
179         help='Save the final drawing to a file.'
180     )
181     parser.add_argument(
182         '-q', '--quit-after-eof',
183         action='store_true',
184         dest='quit_after_eof',
185         help='Close the program after finding EOF in stdin'
186     )
187     args = parser.parse_args()
188     return args
189 
190 if __name__ == "__main__":
191     program = DrawPoints()
192     program.init(parse_args())
193     program.run()  # This function should never return
```

APÊNDICE N – render_images.sh

```
#!/bin/bash

# Wanna render the images without having a window popping up for every image?
# Run:
#   xvfb-run ./render_images.sh

# phi and theta offsets
p=0
t=0

# PHI and THETA apertures
#P=80
#T=80

# White dot size
DOT_SIZE=4

# Destination directory
IMAGE_DIR="resultados"


mkdir -p "${IMAGE_DIR}"

#for abertura in {10..90..5} ; do
for abertura in 30 45 60 75 85 ; do
#for abertura in {6..150..2}; do
    P=${abertura}
    T=${abertura}

    # C program
    #./generate_sphere_vectors.py -P ${P} -T ${T} -p ${p} -t ${t} \
    #| ./linear_eq_conversion \
    #| ./draw_points.py -p -s ${DOT_SIZE} -q -o "${IMAGE_DIR}/P${P}T${T}p${p}t${t}_cleq.png"

    # Python program
    #for a in {1..13} ; do
    for a in {7..13} ; do
        ./generate_sphere_vectors.py -P ${P} -T ${T} -p ${p} -t ${t} \
        | ./convert_coordinates.py -a ${a} \
        | ./draw_points.py -p -s ${DOT_SIZE} -w 640 640 -q -o "${IMAGE_DIR}/P${P}T${T}p${p}t${t}_a${a}.png"
    done
done

echo "If you want to save space, also run this command:"
echo "optipng -o7 images/*"
```

ANEXO A – ATmega8 datasheet

Páginas 1-6, 9-10, 17-19, 157-160, 168-170, 213, 216-217 [10].

Features

- High-performance, Low-power Atmel®AVR® 8-bit Microcontroller
- Advanced RISC Architecture
  - 130 Powerful Instructions – Most Single-clock Cycle Execution
  - 32 × 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 16MIPS Throughput at 16MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory segments
  - 8Kbytes of In-System Self-programmable Flash program memory
  - 512Bytes EEPROM
  - 1Kbyte Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C[1]
  - Optional Boot Code Section with Independent Lock Bits
    In-System Programming by On-chip Boot Program
    True Read-While-Write Operation
  - Programming Lock for Software Security
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescaler, one Compare Mode
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Three PWM Channels
  - 8-channel ADC in TQFP and QFN/MLF package
    Eight Channels 10-bit Accuracy
  - 6-channel ADC in PDIP package
    Six Channels 10-bit Accuracy
  - Byte-oriented Two-wire Serial Interface
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated RC Oscillator
  - External and Internal Interrupt Sources
  - Five Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, and Standby
- I/O and Packages
  - 23 Programmable I/O Lines
  - 28-lead PDIP, 32-lead TQFP, and 32-pad QFN/MLF
- Operating Voltages
  - 2.7V - 5.5V (ATmega8L)
  - 4.5V - 5.5V (ATmega8)
- Speed Grades
  - 0 - 8MHz (ATmega8L)
  - 0 - 16MHz (ATmega8)
- Power Consumption at 4Mhz, 3V, 25°C
  - Active: 3.6mA
  - Idle Mode: 1.0mA
  - Power-down Mode: 0.5µA

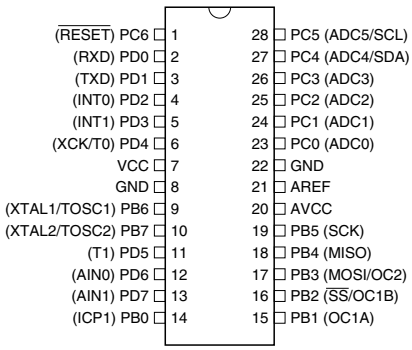
8-bit AVR® with 8KBytes In-System Programmable Flash

ATmega8
ATmega8L

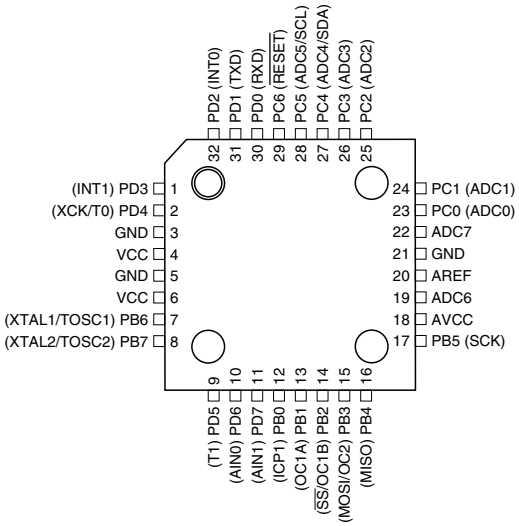
Rev.2486Z–AVR–02/11

Pin Configurations

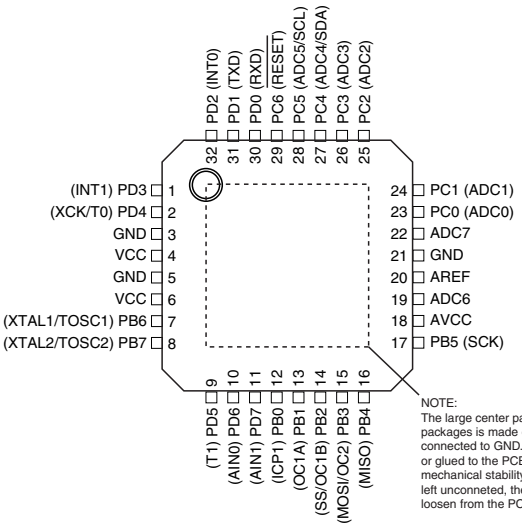
PDIP



TQFP Top View



MLF Top View



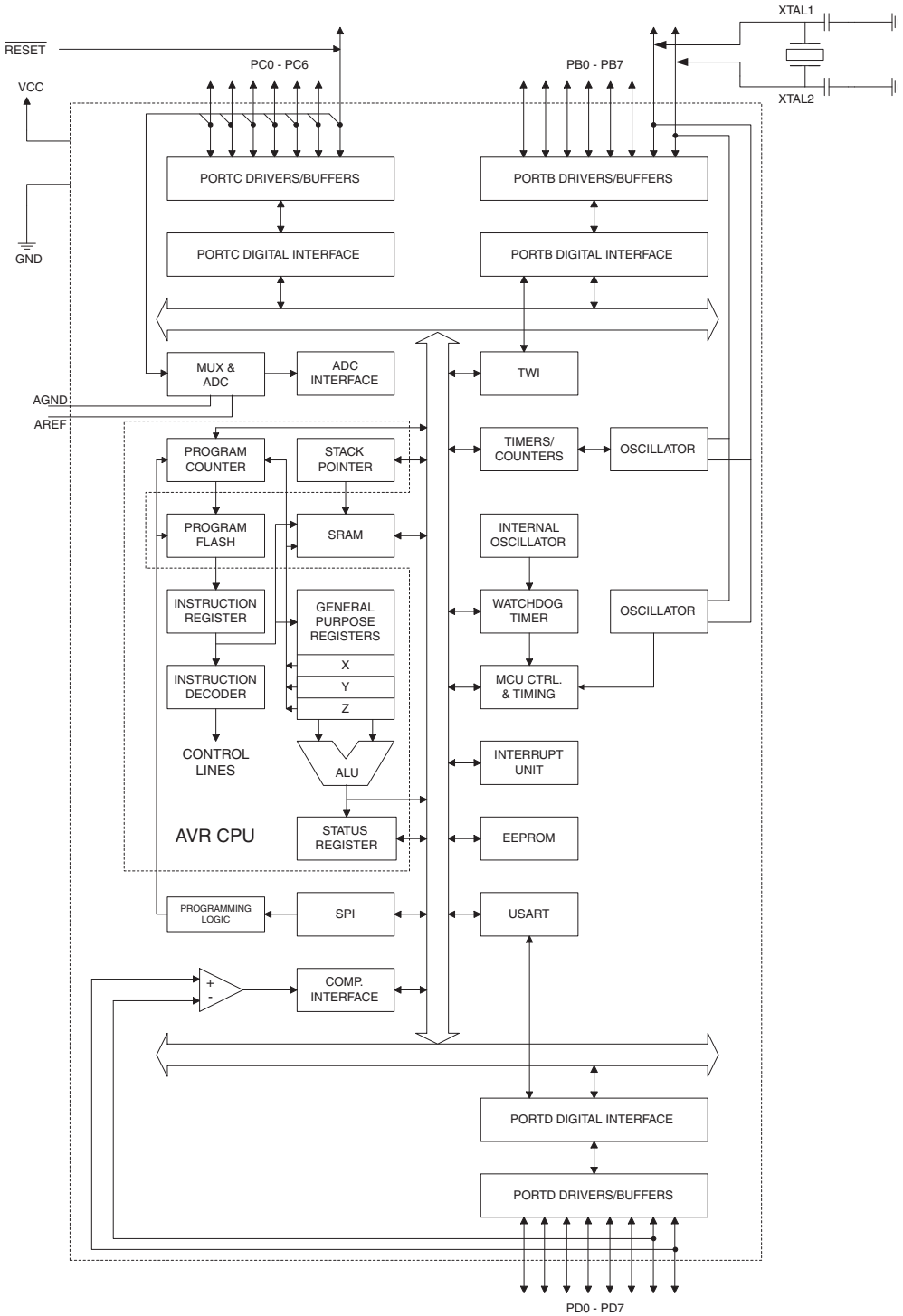
NOTE:
The large center pad underneath the MLF packages is made of metal and internally connected to GND. It should be soldered or glued to the PCB to ensure good mechanical stability. If the center pad is left unconneted, the package might loosen from the PCB.

Overview

The Atmel®AVR® ATmega8 is a low-power CMOS 8-bit microcontroller based on the AVR RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega8 achieves throughputs approaching 1MIPS per MHz, allowing the system designer to optimize power consumption versus processing speed.

Block Diagram

Figure 1. Block Diagram



The Atmel®AVR® core combines a rich instruction set with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

The ATmega8 provides the following features: 8 Kbytes of In-System Programmable Flash with Read-While-Write capabilities, 512 bytes of EEPROM, 1 Kbyte of SRAM, 23 general purpose I/O lines, 32 general purpose working registers, three flexible Timer/Counters with compare modes, internal and external interrupts, a serial programmable USART, a byte oriented Two-wire Serial Interface, a 6-channel ADC (eight channels in TQFP and QFN/MLF packages) with 10-bit accuracy, a programmable Watchdog Timer with Internal Oscillator, an SPI serial port, and five software selectable power saving modes. The Idle mode stops the CPU while allowing the SRAM, Timer/Counters, SPI port, and interrupt system to continue functioning. The Power-down mode saves the register contents but freezes the Oscillator, disabling all other chip functions until the next Interrupt or Hardware Reset. In Power-save mode, the asynchronous timer continues to run, allowing the user to maintain a timer base while the rest of the device is sleeping. The ADC Noise Reduction mode stops the CPU and all I/O modules except asynchronous timer and ADC, to minimize switching noise during ADC conversions. In Standby mode, the crystal/resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low-power consumption.

The device is manufactured using Atmel's high density non-volatile memory technology. The Flash Program memory can be reprogrammed In-System through an SPI serial interface, by a conventional non-volatile memory programmer, or by an On-chip boot program running on the AVR core. The boot program can use any interface to download the application program in the Application Flash memory. Software in the Boot Flash Section will continue to run while the Application Flash Section is updated, providing true Read-While-Write operation. By combining an 8-bit RISC CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATmega8 is a powerful microcontroller that provides a highly-flexible and cost-effective solution to many embedded control applications.

The ATmega8 is supported with a full suite of program and system development tools, including C compilers, macro assemblers, program debugger/simulators, In-Circuit Emulators, and evaluation kits.

Disclaimer

Typical values contained in this datasheet are based on simulations and characterization of other AVR microcontrollers manufactured on the same process technology. Minimum and Maximum values will be available after the device is characterized.

Pin Descriptions

VCC — Digital supply voltage.

GND — Ground.

Port B (PB7..PB0) XTAL1/XTAL2/TOSC1/TOSC2 — Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Depending on the clock selection fuse settings, PB6 can be used as input to the inverting Oscillator amplifier and input to the internal clock operating circuit.

Depending on the clock selection fuse settings, PB7 can be used as output from the inverting Oscillator amplifier.

If the Internal Calibrated RC Oscillator is used as chip clock source, PB7..6 is used as TOSC2..1 input for the Asynchronous Timer/Counter2 if the AS2 bit in ASSR is set.

The various special features of Port B are elaborated in “Alternate Functions of Port B” on page 58 and “System Clock and Clock Options” on page 25.

Port C (PC5..PC0) — Port C is an 7-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running.

PC6/$\overline{\text{RESET}}$ — If the RSTDISBL Fuse is programmed, PC6 is used as an I/O pin. Note that the electrical characteristics of PC6 differ from those of the other pins of Port C.

If the RSTDISBL Fuse is unprogrammed, PC6 is used as a Reset input. A low level on this pin for longer than the minimum pulse length will generate a Reset, even if the clock is not running. The minimum pulse length is given in Table 15 on page 38. Shorter pulses are not guaranteed to generate a Reset.

The various special features of Port C are elaborated on page 61.

Port D (PD7..PD0) — Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port D also serves the functions of various special features of the ATmega8 as listed on page 63.

$\overline{\text{RESET}}$ — Reset input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running. The minimum pulse length is given in Table 15 on page 38. Shorter pulses are not guaranteed to generate a reset.

AV_{CC}	AV_{CC} is the supply voltage pin for the A/D Converter, Port C (3..0), and ADC (7..6). It should be externally connected to V_{CC}, even if the ADC is not used. If the ADC is used, it should be connected to V_{CC} through a low-pass filter. Note that Port C (5..4) use digital supply voltage, V_{CC}.
AREF	AREF is the analog reference pin for the A/D Converter.
ADC7..6 (TQFP and QFN/MLF Package Only)	In the TQFP and QFN/MLF package, ADC7..6 serve as analog inputs to the A/D converter. These pins are powered from the analog supply and serve as 10-bit ADC channels.

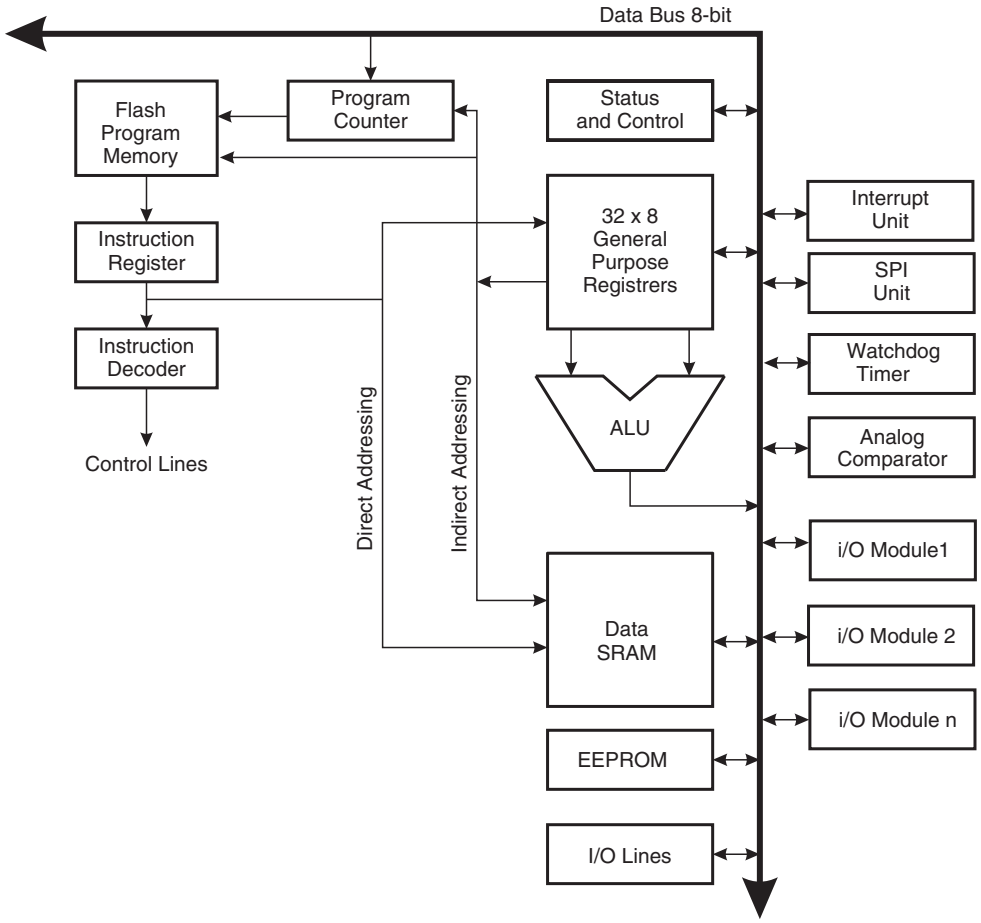
Atmel AVR CPU Core

Introduction

This section discusses the Atmel®AVR® core architecture in general. The main function of the CPU core is to ensure correct program execution. The CPU must therefore be able to access memories, perform calculations, control peripherals, and handle interrupts.

Architectural Overview

Figure 2. Block Diagram of the AVR MCU Architecture



In order to maximize performance and parallelism, the AVR uses a Harvard architecture – with separate memories and buses for program and data. Instructions in the Program memory are executed with a single level pipelining. While one instruction is being executed, the next instruction is pre-fetched from the Program memory. This concept enables instructions to be executed in every clock cycle. The Program memory is In-System Reprogrammable Flash memory.

The fast-access Register File contains 32 × 8-bit general purpose working registers with a single clock cycle access time. This allows single-cycle Arithmetic Logic Unit (ALU) operation. In a typical ALU operation, two operands are output from the Register File, the operation is executed, and the result is stored back in the Register File – in one clock cycle.

Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing – enabling efficient address calculations. One of the these address pointers

can also be used as an address pointer for look up tables in Flash Program memory. These added function registers are the 16-bit X-register, Y-register, and Z-register, described later in this section.

The ALU supports arithmetic and logic operations between registers or between a constant and a register. Single register operations can also be executed in the ALU. After an arithmetic operation, the Status Register is updated to reflect information about the result of the operation.

The Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most AVR instructions have a single 16-bit word format. Every Program memory address contains a 16-bit or 32-bit instruction.

Program Flash memory space is divided in two sections, the Boot program section and the Application program section. Both sections have dedicated Lock Bits for write and read/write protection. The SPM instruction that writes into the Application Flash memory section must reside in the Boot program section.

During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the Stack. The Stack is effectively allocated in the general data SRAM, and consequently the Stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the reset routine (before subroutines or interrupts are executed). The Stack Pointer SP is read/write accessible in the I/O space. The data SRAM can easily be accessed through the five different addressing modes supported in the AVR architecture.

The memory spaces in the AVR architecture are all linear and regular memory maps.

A flexible interrupt module has its control registers in the I/O space with an additional global interrupt enable bit in the Status Register. All interrupts have a separate Interrupt Vector in the Interrupt Vector table. The interrupts have priority in accordance with their Interrupt Vector position. The lower the Interrupt Vector address, the higher the priority.

The I/O memory space contains 64 addresses for CPU peripheral functions as Control Registers, SPI, and other I/O functions. The I/O Memory can be accessed directly, or as the Data Space locations following those of the Register File, 0x20 - 0x5F.

AVR ATmega8 Memories

This section describes the different memories in the Atmel®AVR® ATmega8. The AVR architecture has two main memory spaces, the Data memory and the Program Memory space. In addition, the ATmega8 features an EEPROM Memory for data storage. All three memory spaces are linear and regular.

In-System Reprogrammable Flash Program Memory

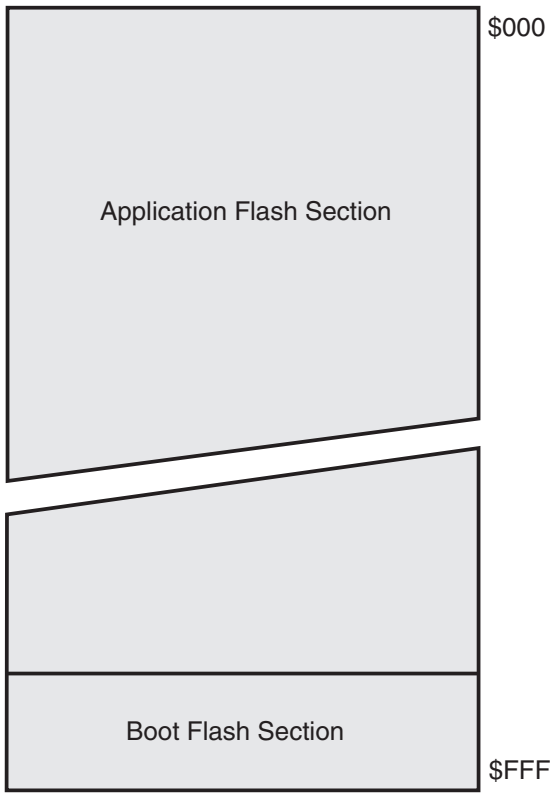
The ATmega8 contains 8Kbytes On-chip In-System Reprogrammable Flash memory for program storage. Since all AVR instructions are 16-bits or 32-bits wide, the Flash is organized as 4K × 16 bits. For software security, the Flash Program memory space is divided into two sections, Boot Program section and Application Program section.

The Flash memory has an endurance of at least 10,000 write/erase cycles. The ATmega8 Program Counter (PC) is 12 bits wide, thus addressing the 4K Program memory locations. The operation of Boot Program section and associated Boot Lock Bits for software protection are described in detail in “Boot Loader Support – Read-While-Write Self-Programming” on page 202. “Memory Programming” on page 215 contains a detailed description on Flash Programming in SPI- or Parallel Programming mode.

Constant tables can be allocated within the entire Program memory address space (see the LPM – Load Program memory instruction description).

Timing diagrams for instruction fetch and execution are presented in “Instruction Execution Timing” on page 13.

Figure 7. Program Memory Map



SRAM Data Memory

Figure 8 shows how the Atmel®AVR® SRAM Memory is organized.

The lower 1120 Data memory locations address the Register File, the I/O Memory, and the internal data SRAM. The first 96 locations address the Register File and I/O Memory, and the next 1024 locations address the internal data SRAM.

The five different addressing modes for the Data memory cover: Direct, Indirect with Displacement, Indirect, Indirect with Pre-decrement, and Indirect with Post-increment. In the Register File, registers R26 to R31 feature the indirect addressing pointer registers.

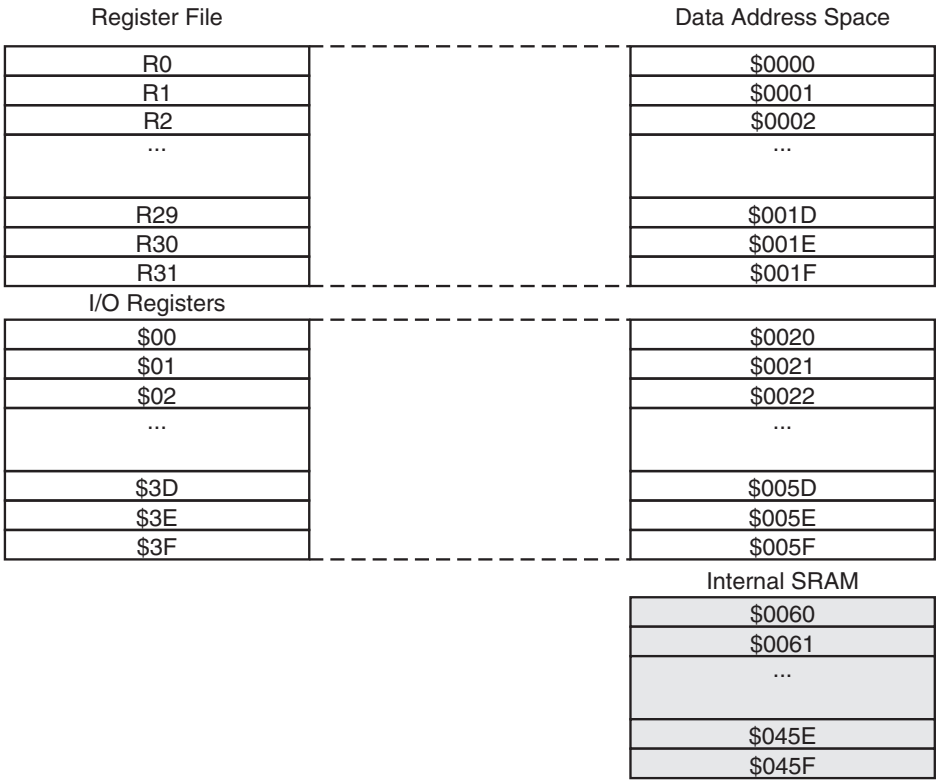
The direct addressing reaches the entire data space.

The Indirect with Displacement mode reaches 63 address locations from the base address given by the Y-register or Z-register.

When using register indirect addressing modes with automatic pre-decrement and post-increment, the address registers X, Y and Z are decremented or incremented.

The 32 general purpose working registers, 64 I/O Registers, and the 1024 bytes of internal data SRAM in the ATmega8 are all accessible through all these addressing modes. The Register File is described in “General Purpose Register File” on page 12.

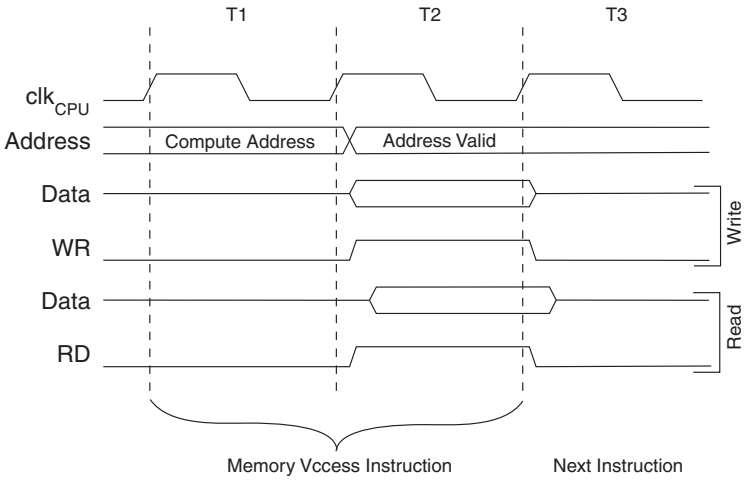
Figure 8. Data Memory Map



Data Memory Access Times

This section describes the general access timing concepts for internal memory access. The internal data SRAM access is performed in two clk_{CPU} cycles as described in Figure 9.

Figure 9. On-chip Data SRAM Access Cycles



EEPROM Data Memory

The ATmega8 contains 512bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles. The access between the EEPROM and the CPU is described below, specifying the EEPROM Address Registers, the EEPROM Data Register, and the EEPROM Control Register.

"Memory Programming" on page 215 contains a detailed description on EEPROM Programming in SPI- or Parallel Programming mode.

EEPROM Read/Write Access

The EEPROM Access Registers are accessible in the I/O space.

The write access time for the EEPROM is given in Table 1 on page 21. A self-timing function, however, lets the user software detect when the next byte can be written. If the user code contains instructions that write the EEPROM, some precautions must be taken. In heavily filtered power supplies, V_{CC} is likely to rise or fall slowly on Power-up/down. This causes the device for some period of time to run at a voltage lower than specified as minimum for the clock frequency used. See "Preventing EEPROM Corruption" on page 23. for details on how to avoid problems in these situations.

In order to prevent unintentional EEPROM writes, a specific write procedure must be followed. Refer to "The EEPROM Control Register – EECR" on page 20 for details on this.

When the EEPROM is read, the CPU is halted for four clock cycles before the next instruction is executed. When the EEPROM is written, the CPU is halted for two clock cycles before the next instruction is executed.

Two-wire Serial Interface

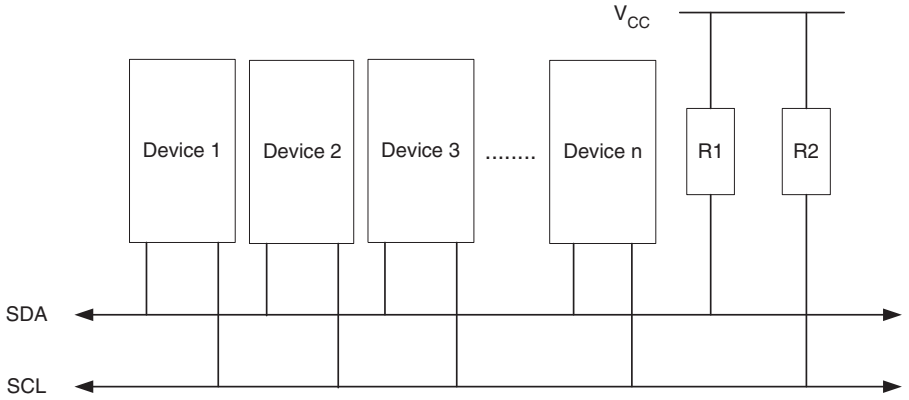
Features

- **Simple Yet Powerful and Flexible Communication Interface, only two Bus Lines Needed**
- **Both Master and Slave Operation Supported**
- **Device can Operate as Transmitter or Receiver**
- **7-bit Address Space Allows up to 128 Different Slave Addresses**
- **Multi-master Arbitration Support**
- **Up to 400kHz Data Transfer Speed**
- **Slew-rate Limited Output Drivers**
- **Noise Suppression Circuitry Rejects Spikes on Bus Lines**
- **Fully Programmable Slave Address with General Call Support**
- **Address Recognition Causes Wake-up When AVR is in Sleep Mode**

Two-wire Serial Interface Bus Definition

The Two-wire Serial Interface (TWI) is ideally suited for typical microcontroller applications. The TWI protocol allows the systems designer to interconnect up to 128 different devices using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA). The only external hardware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines. All devices connected to the bus have individual addresses, and mechanisms for resolving bus contention are inherent in the TWI protocol.

Figure 68. TWI Bus Interconnection



TWI Terminology

The following definitions are frequently encountered in this section.

Table 64. TWI Terminology

Term	Description
Master	The device that initiates and terminates a transmission. The Master also generates the SCL clock
Slave	The device addressed by a Master
Transmitter	The device placing data on the bus
Receiver	The device reading data from the bus

Electrical Interconnection

As depicted in Figure 68 on page 157, both bus lines are connected to the positive supply voltage through pull-up resistors. The bus drivers of all TWI-compliant devices are open-drain or open-collector. This implements a wired-AND function which is essential to the operation of the interface. A low level on a TWI bus line is generated when one or more TWI devices output a zero. A high level is output when all TWI devices tri-state their outputs, allowing the pull-up resistors to pull the line high. Note that all AVR devices connected to the TWI bus must be powered in order to allow any bus operation.

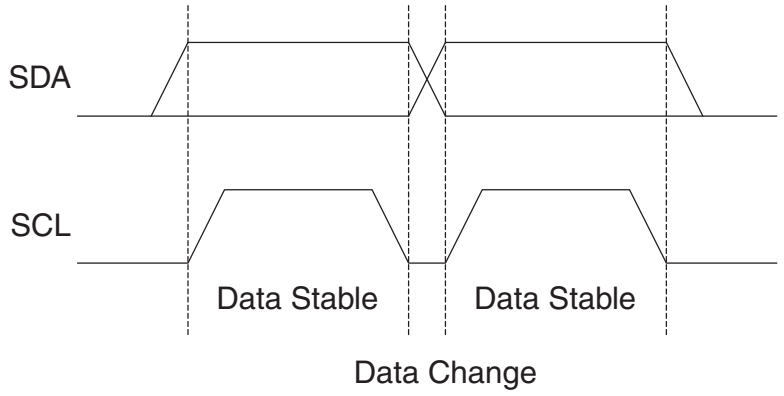
The number of devices that can be connected to the bus is only limited by the bus capacitance limit of 400pF and the 7-bit slave address space. A detailed specification of the electrical characteristics of the TWI is given in "Two-wire Serial Interface Characteristics" on page 238. Two different sets of specifications are presented there, one relevant for bus speeds below 100kHz, and one valid for bus speeds up to 400kHz.

Data Transfer and Frame Format

Transferring Bits

Each data bit transferred on the TWI bus is accompanied by a pulse on the clock line. The level of the data line must be stable when the clock line is high. The only exception to this rule is for generating start and stop conditions.

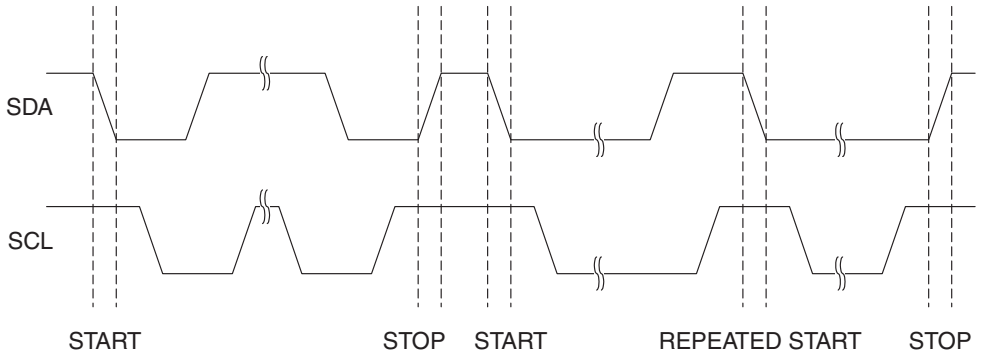
Figure 69. Data Validity



START and STOP Conditions

The Master initiates and terminates a data transmission. The transmission is initiated when the Master issues a START condition on the bus, and it is terminated when the Master issues a STOP condition. Between a START and a STOP condition, the bus is considered busy, and no other master should try to seize control of the bus. A special case occurs when a new START condition is issued between a START and STOP condition. This is referred to as a REPEATED START condition, and is used when the Master wishes to initiate a new transfer without relinquishing control of the bus. After a REPEATED START, the bus is considered busy until the next STOP. This is identical to the START behavior, and therefore START is used to describe both START and REPEATED START for the remainder of this datasheet, unless otherwise noted. As depicted below, START and STOP conditions are signalled by changing the level of the SDA line when the SCL line is high.

Figure 70. START, REPEATED START and STOP conditions



Address Packet Format

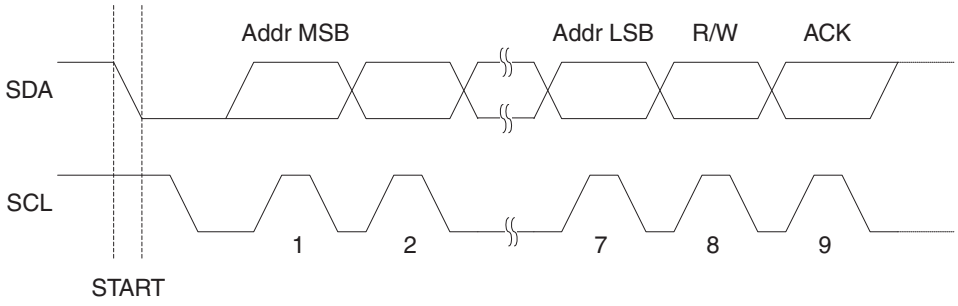
All address packets transmitted on the TWI bus are 9 bits long, consisting of 7 address bits, one READ/WRITE control bit and an acknowledge bit. If the READ/WRITE bit is set, a read operation is to be performed, otherwise a write operation should be performed. When a Slave recognizes that it is being addressed, it should acknowledge by pulling SDA low in the ninth SCL (ACK) cycle. If the addressed Slave is busy, or for some other reason can not service the Master's request, the SDA line should be left high in the ACK clock cycle. The Master can then transmit a STOP condition, or a REPEATED START condition to initiate a new transmission. An address packet consisting of a slave address and a READ or a WRITE bit is called SLA+R or SLA+W, respectively.

The MSB of the address byte is transmitted first. Slave addresses can freely be allocated by the designer, but the address 0000 000 is reserved for a general call.

When a general call is issued, all slaves should respond by pulling the SDA line low in the ACK cycle. A general call is used when a Master wishes to transmit the same message to several slaves in the system. When the general call address followed by a Write bit is transmitted on the bus, all slaves set up to acknowledge the general call will pull the SDA line low in the ack cycle. The following data packets will then be received by all the slaves that acknowledged the general call. Note that transmitting the general call address followed by a Read bit is meaningless, as this would cause contention if several slaves started transmitting different data.

All addresses of the format 1111 xxx should be reserved for future purposes.

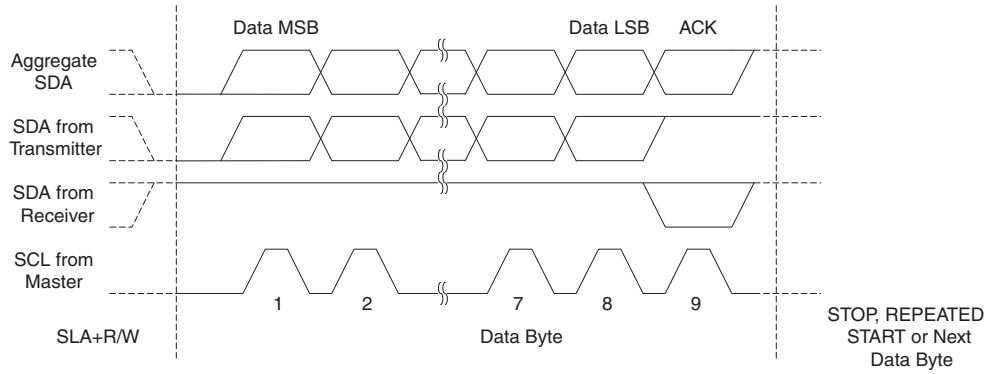
Figure 71. Address Packet Format



Data Packet Format

All data packets transmitted on the TWI bus are nine bits long, consisting of one data byte and an acknowledge bit. During a data transfer, the Master generates the clock and the START and STOP conditions, while the Receiver is responsible for acknowledging the reception. An Acknowledge (ACK) is signalled by the Receiver pulling the SDA line low during the ninth SCL cycle. If the Receiver leaves the SDA line high, a NACK is signalled. When the Receiver has received the last byte, or for some reason cannot receive any more bytes, it should inform the Transmitter by sending a NACK after the final byte. The MSB of the data byte is transmitted first.

Figure 72. Data Packet Format

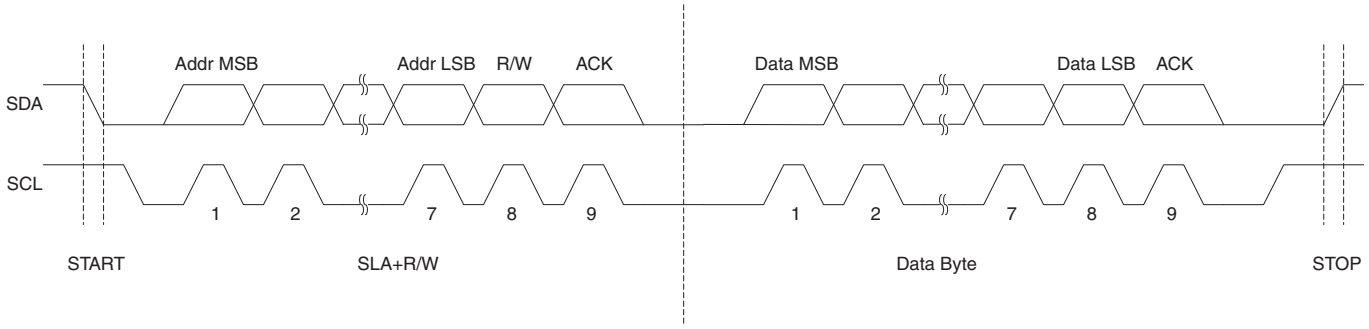


Combining Address and Data Packets into a Transmission

A transmission basically consists of a START condition, a SLA+R/W, one or more data packets and a STOP condition. An empty message, consisting of a START followed by a STOP condition, is illegal. Note that the Wired-ANDing of the SCL line can be used to implement handshaking between the Master and the Slave. The Slave can extend the SCL low period by pulling the SCL line low. This is useful if the clock speed set up by the Master is too fast for the Slave, or the Slave needs extra time for processing between the data transmissions. The Slave extending the SCL low period will not affect the SCL high period, which is determined by the Master. As a consequence, the Slave can reduce the TWI data transfer speed by prolonging the SCL duty cycle.

Figure 73 shows a typical data transmission. Note that several data bytes can be transmitted between the SLA+R/W and the STOP condition, depending on the software protocol implemented by the application software.

Figure 73. Typical Data Transmission



- **Bits 7..1 – TWA: TWI (Slave) Address Register**

These seven bits constitute the slave address of the TWI unit.

- **Bit 0 – TWGCE: TWI General Call Recognition Enable Bit**

If set, this bit enables the recognition of a General Call given over the Two-wire Serial Bus.

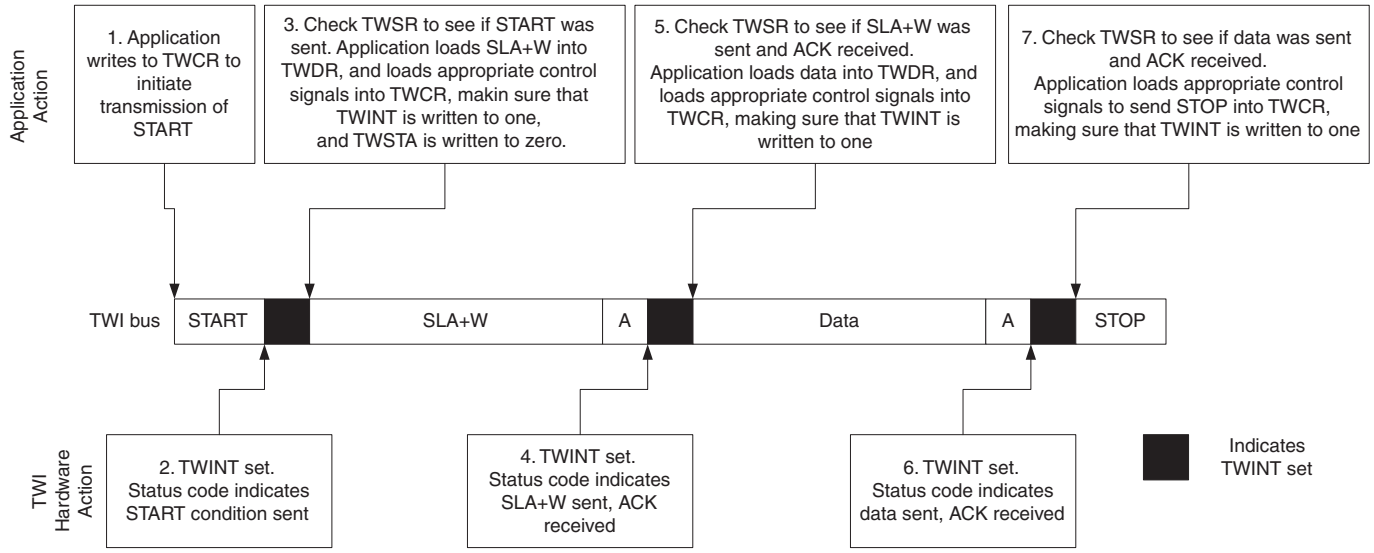
Using the TWI

The AVR TWI is byte-oriented and interrupt based. Interrupts are issued after all bus events, like reception of a byte or transmission of a START condition. Because the TWI is interrupt-based, the application software is free to carry on other operations during a TWI byte transfer. Note that the TWI Interrupt Enable (TWIE) bit in TWCR together with the Global Interrupt Enable bit in SREG allow the application to decide whether or not assertion of the TWINT Flag should generate an interrupt request. If the TWIE bit is cleared, the application must poll the TWINT Flag in order to detect actions on the TWI bus.

When the TWINT Flag is asserted, the TWI has finished an operation and awaits application response. In this case, the TWI Status Register (TWSR) contains a value indicating the current state of the TWI bus. The application software can then decide how the TWI should behave in the next TWI bus cycle by manipulating the TWCR and TWDR Registers.

Figure 77 is a simple example of how the application can interface to the TWI hardware. In this example, a Master wishes to transmit a single data byte to a Slave. This description is quite abstract, a more detailed explanation follows later in this section. A simple code example implementing the desired behavior is also presented.

Figure 77. Interfacing the Application to the TWI in a Typical Transmission



1. The first step in a TWI transmission is to transmit a START condition. This is done by writing a specific value into TWCR, instructing the TWI hardware to transmit a START condition. Which value to write is described later on. However, it is important that the TWINT bit is set in the value written. Writing a one to TWINT clears the flag. The TWI will not start any operation as long as the TWINT bit in TWCR is set. Immediately after the application has cleared TWINT, the TWI will initiate transmission of the START condition
2. When the START condition has been transmitted, the TWINT Flag in TWCR is set, and TWSR is updated with a status code indicating that the START condition has successfully been sent

3. The application software should now examine the value of TWSR, to make sure that the START condition was successfully transmitted. If TWSR indicates otherwise, the application software might take some special action, like calling an error routine. Assuming that the status code is as expected, the application must load SLA+W into TWDR. Remember that TWDR is used both for address and data. After TWDR has been loaded with the desired SLA+W, a specific value must be written to TWCR, instructing the TWI hardware to transmit the SLA+W present in TWDR. Which value to write is described later on. However, it is important that the TWINT bit is set in the value written. Writing a one to TWINT clears the flag. The TWI will not start any operation as long as the TWINT bit in TWCR is set. Immediately after the application has cleared TWINT, the TWI will initiate transmission of the address packet
4. When the address packet has been transmitted, the TWINT Flag in TWCR is set, and TWSR is updated with a status code indicating that the address packet has successfully been sent. The status code will also reflect whether a Slave acknowledged the packet or not
5. The application software should now examine the value of TWSR, to make sure that the address packet was successfully transmitted, and that the value of the ACK bit was as expected. If TWSR indicates otherwise, the application software might take some special action, like calling an error routine. Assuming that the status code is as expected, the application must load a data packet into TWDR. Subsequently, a specific value must be written to TWCR, instructing the TWI hardware to transmit the data packet present in TWDR. Which value to write is described later on. However, it is important that the TWINT bit is set in the value written. Writing a one to TWINT clears the flag. The TWI will not start any operation as long as the TWINT bit in TWCR is set. Immediately after the application has cleared TWINT, the TWI will initiate transmission of the data packet
6. When the data packet has been transmitted, the TWINT Flag in TWCR is set, and TWSR is updated with a status code indicating that the data packet has successfully been sent. The status code will also reflect whether a Slave acknowledged the packet or not
7. The application software should now examine the value of TWSR, to make sure that the data packet was successfully transmitted, and that the value of the ACK bit was as expected. If TWSR indicates otherwise, the application software might take some special action, like calling an error routine. Assuming that the status code is as expected, the application must write a specific value to TWCR, instructing the TWI hardware to transmit a STOP condition. Which value to write is described later on. However, it is important that the TWINT bit is set in the value written. Writing a one to TWINT clears the flag. The TWI will not start any operation as long as the TWINT bit in TWCR is set. Immediately after the application has cleared TWINT, the TWI will initiate transmission of the STOP condition. Note that TWINT is NOT set after a STOP condition has been sent

Even though this example is simple, it shows the principles involved in all TWI transmissions. These can be summarized as follows:

- When the TWI has finished an operation and expects application response, the TWINT Flag is set. The SCL line is pulled low until TWINT is cleared
- When the TWINT Flag is set, the user must update all TWI Registers with the value relevant for the next TWI bus cycle. As an example, TWDR must be loaded with the value to be transmitted in the next bus cycle
- After all TWI Register updates and other pending application software tasks have been completed, TWCR is written. When writing TWCR, the TWINT bit should be set. Writing a one to TWINT clears the flag. The TWI will then commence executing whatever operation was specified by the TWCR setting

In the following an assembly and C implementation of the example is given. Note that the code below assumes that several definitions have been made, for example by using include-files.

	Assembly Code Example	C Example	Comments
1	`ldi r16, (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)` `out TWCR, r16`	`TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)`	Send START condition

2	`wait1:`
	`in r16,TWCR`
	`sbrs r16,TWINT`
	`rjmp wait1`
		`while (!(TWCR & (1<<TWINT)))`
		`;`
			Wait for TWINT Flag set. This indicates that the START condition has been transmitted

3	`in r16,TWSR`
	`andi r16, 0xF8`
	`cpi r16, START`
	`brne ERROR`
		`if ((TWSR & 0xF8) != START)`
		`ERROR();`
			Check value of TWI Status Register. Mask prescaler bits. If status different from START go to ERROR

	`ldi r16, SLA_W`
	`out TWDR, r16`
	`ldi r16, (1<<TWINT) | (1<<TWEN)`
	`out TWCR, r16`
		`TWDR = SLA_W;`
		`TWCR = (1<<TWINT) | (1<<TWEN);`
			Load SLA_W into TWDR Register. Clear TWINT bit in TWCR to start transmission of address

4	`wait2:`
	`in r16,TWCR`
	`sbrs r16,TWINT`
	`rjmp wait2`
		`while (!(TWCR & (1<<TWINT)))`
		`;`
			Wait for TWINT Flag set. This indicates that the SLA+W has been transmitted, and ACK/NACK has been received.

5	`in r16,TWSR`
	`andi r16, 0xF8`
	`cpi r16, MT_SLA_ACK`
	`brne ERROR`
		`if ((TWSR & 0xF8) != MT_SLA_ACK)`
		`ERROR();`
			Check value of TWI Status Register. Mask prescaler bits. If status different from MT_SLA_ACK go to ERROR

	`ldi r16, DATA`
	`out TWDR, r16`
	`ldi r16, (1<<TWINT) | (1<<TWEN)`
	`out TWCR, r16`
		`TWDR = DATA;`
		`TWCR = (1<<TWINT) | (1<<TWEN);`
			Load DATA into TWDR Register. Clear TWINT bit in TWCR to start transmission of data

6	`wait3:`
	`in r16,TWCR`
	`sbrs r16,TWINT`
	`rjmp wait3`
		`while (!(TWCR & (1<<TWINT)))`
		`;`
			Wait for TWINT Flag set. This indicates that the DATA has been transmitted, and ACK/NACK has been received.

7	`in r16,TWSR`
	`andi r16, 0xF8`
	`cpi r16, MT_DATA_ACK`
	`brne ERROR`
		`if ((TWSR & 0xF8) != MT_DATA_ACK)`
		`ERROR();`
			Check value of TWI Status Register. Mask prescaler bits. If status different from MT_DATA_ACK go to ERROR

	`ldi r16, (1<<TWINT)|(1<<TWEN)|(1<<TWSTO)` `out TWCR, r16`	`TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTO);`	Transmit STOP condition

```
  ; return to RWW section
  ; verify that RWW section is safe to read
Return:
  in   temp1, SPMCR
  sbrs temp1, RWWSB           ; If RWWSB is set, the RWW section is not
ready yet
  ret
  ; re-enable the RWW section
  ldi  spmcrval, (1<<RWWSRE) | (1<<SPMEN)
  rcallDo_spm
  rjmp Return

Do_spm:
  ; check for previous SPM complete
Wait_spm:
  in   temp1, SPMCR
  sbrc temp1, SPMEN
  rjmp Wait_spm
  ; input: spmcrval determines SPM action
  ; disable interrupts if enabled, store status
  in   temp2, SREG
  cli
  ; check that no EEPROM write access is present
Wait_ee:
  sbic EECR, EEWE
  rjmp Wait_ee
  ; SPM timed sequence
  out  SPMCR, spmcrval
  spm
  ; restore SREG (to enable interrupts if originally enabled)
  out  SREG, temp2
  ret
```

ATmega8 Boot Loader Parameters

In Table 82 through Table 84 on page 214, the parameters used in the description of the self programming are given.

Table 82. Boot Size Configuration

BOOTSZ1	BOOTSZ0	Boot Size	Pages	Application Flash Section	Boot Loader Flash Section	End Application Section	Boot Reset Address (Start Boot Loader Section)
1	1	128 words	4	0x000 - 0xF7F	0xF80 - 0xFFF	0xF7F	0xF80
1	0	256 words	8	0x000 - 0xEFF	0xF00 - 0xFFF	0xEFF	0xF00
0	1	512 words	16	0x000 - 0xDFF	0xE00 - 0xFFF	0xDFF	0xE00
0	0	1024 words	32	0x000 - 0xBFF	0xC00 - 0xFFF	0xBFF	0xC00

Note: The different BOOTSZ Fuse configurations are shown in Figure 102 on page 204

Table 86. Lock Bit Protection Modes[2] (Continued)

Memory Lock Bits			Protection Type
BLB1 Mode	BLB12	BLB11	
1	1	1	No restrictions for SPM or LPM accessing the Boot Loader section
2	1	0	SPM is not allowed to write to the Boot Loader section
3	0	0	SPM is not allowed to write to the Boot Loader section, and LPM executing from the Application section is not allowed to read from the Boot Loader section. If Interrupt Vectors are placed in the Application section, interrupts are disabled while executing from the Boot Loader section
4	0	1	LPM executing from the Application section is not allowed to read from the Boot Loader section. If Interrupt Vectors are placed in the Application section, interrupts are disabled while executing from the Boot Loader section

Notes: 1. Program the Fuse Bits before programming the Lock Bits
2. "1" means unprogrammed, "0" means programmed

Fuse Bits

The ATmega8 has two fuse bytes. Table 87 and Table 88 on page 217 describe briefly the functionality of all the fuses and how they are mapped into the fuse bytes. Note that the fuses are read as logical zero, "0", if they are programmed.

Table 87. Fuse High Byte

Fuse High Byte	Bit No.	Description	Default Value
RSTDISBL[4]	7	Select if PC6 is I/O pin or RESET pin	1 (unprogrammed, PC6 is RESET-pin)
WDTON	6	WDT always on	1 (unprogrammed, WDT enabled by WDTCR)
SPIEN[1]	5	Enable Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)
CKOPT[2]	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed, EEPROM not preserved)
BOOTSZ1	2	Select Boot Size (see Table 82 on page 213 for details)	0 (programmed)[3]
BOOTSZ0	1	Select Boot Size (see Table 82 on page 213 for details)	0 (programmed)[3]
BOOTRST	0	Select Reset Vector	1 (unprogrammed)

Notes: 1. The SPIEN Fuse is not accessible in Serial Programming mode
2. The CKOPT Fuse functionality depends on the setting of the CKSEL bits, see "Clock Sources" on page 26 for details
3. The default value of BOOTSZ1..0 results in maximum Boot Size. See Table 82 on page 213
4. When programming the RSTDISBL Fuse Parallel Programming has to be used to change fuses or perform further programming

Table 88. Fuse Low Byte

Fuse Low Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown out detector trigger level	1 (unprogrammed)
BODEN	6	Brown out detector enable	1 (unprogrammed, BOD disabled)
SUT1	5	Select start-up time	1 (unprogrammed)(1)
SUT0	4	Select start-up time	0 (programmed)(1)
CKSEL3	3	Select Clock source	0 (programmed)(2)
CKSEL2	2	Select Clock source	0 (programmed)(2)
CKSEL1	1	Select Clock source	0 (programmed)(2)
CKSEL0	0	Select Clock source	1 (unprogrammed)(2)

Notes: 1. The default value of SUT1..0 results in maximum start-up time. See Table 10 on page 30 for details
2. The default setting of CKSEL3..0 results in internal RC Oscillator @ 1MHz. See Table 2 on page 26 for details

The status of the Fuse Bits is not affected by Chip Erase. Note that the Fuse Bits are locked if lock bit1 (LB1) is programmed. Program the Fuse Bits before programming the Lock Bits.

Latching of Fuses

The fuse values are latched when the device enters Programming mode and changes of the fuse values will have no effect until the part leaves Programming mode. This does not apply to the EESAVE Fuse which will take effect once it is programmed. The fuses are also latched on Power-up in Normal mode.

ANEXO B – HMC5883L datasheet

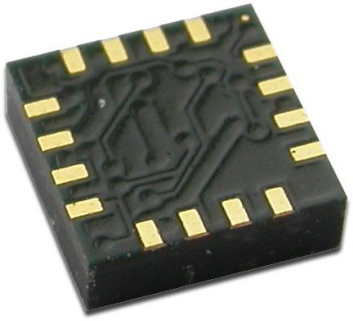
Todas as páginas [14].

3-Axis Digital Compass IC HMC5883L

Honeywell

Advanced Information

The Honeywell HMC5883L is a surface-mount, multi-chip module designed for low-field magnetic sensing with a digital interface for applications such as low-cost compassing and magnetometry. The HMC5883L includes our state-of-the-art, high-resolution HMC118X series magneto-resistive sensors plus an ASIC containing amplification, automatic degaussing strap drivers, offset cancellation, and a 12-bit ADC that enables 1° to 2° compass heading accuracy. The I^2C serial bus allows for easy interface. The HMC5883L is a 3.0x3.0x0.9mm surface mount 16-pin leadless chip carrier (LCC). Applications for the HMC5883L include Mobile Phones, Netbooks, Consumer Electronics, Auto Navigation Systems, and Personal Navigation Devices.



The HMC5883L utilizes Honeywell's Anisotropic Magnetoresistive (AMR) technology that provides advantages over other magnetic sensor technologies. These anisotropic, directional sensors feature precision in-axis sensitivity and linearity. These sensors' solid-state construction with very low cross-axis sensitivity is designed to measure both the direction and the magnitude of Earth's magnetic fields, from milli-gauss to 8 gauss. Honeywell's Magnetic Sensors are among the most sensitive and reliable low-field sensors in the industry.

FEATURES	BENEFITS
3-Axis Magnetoresistive Sensors and ASIC in a 3.0x3.0x0.9mm LCC Surface Mount Package	Small Size for Highly Integrated Products. Just Add a Micro-Controller Interface, Plus Two External SMT Capacitors Designed for High Volume, Cost Sensitive OEM Designs Easy to Assemble & Compatible with High Speed SMT Assembly
12-Bit ADC Coupled with Low Noise AMR Sensors Achieves 2 milli-gauss Field Resolution in ±8 Gauss Fields	Enables 1° to 2° Degree Compass Heading Accuracy
Built-In Self Test	Enables Low-Cost Functionality Test after Assembly in Production
Low Voltage Operations (2.16 to 3.6V) and Low Power Consumption (100 µA)	Compatible for Battery Powered Applications
Built-In Strap Drive Circuits	Set/Reset and Offset Strap Drivers for Degaussing, Self Test, and Offset Compensation
I^2C Digital Interface	Popular Two-Wire Serial Data Interface for Consumer Electronics
Lead Free Package Construction	RoHS Compliance
Wide Magnetic Field Range (+/-8 Oe)	Sensors Can Be Used in Strong Magnetic Field Environments with a 1° to 2° Degree Compass Heading Accuracy
Software and Algorithm Support Available	Compassing Heading, Hard Iron, Soft Iron, and Auto Calibration Libraries Available
Fast 160 Hz Maximum Output Rate	Enables Pedestrian Navigation and LBS Applications

SPECIFICATIONS (* Tested at 25°C except stated otherwise.)

Characteristics	Conditions*	Min	Typ	Max	Units
Power Supply					
Supply Voltage	VDD Referenced to AGND VDDIO Referenced to DGND	2.16 1.71	2.5 1.8	3.6 VDD+0.1	Volts Volts
Average Current Draw	Idle Mode Measurement Mode (7.5 Hz ODR; No measurement average, MA1:MA0 = 00) VDD = 2.5V, VDDIO = 1.8V (Dual Supply) VDD = VDDIO = 2.5V (Single Supply)	- -	2 100	- -	µA µA
Performance					
Field Range	Full scale (FS)	-8		+8	gauss
Mag Dynamic Range	3-bit gain control	±1		±8	gauss
Sensitivity (Gain)	VDD=3.0V, GN=0 to 7, 12-bit ADC	230		1370	LSb/gauss
Digital Resolution	VDD=3.0V, GN=0 to 7, 1-LSb, 12-bit ADC	0.73		4.35	milli-gauss
Noise Floor (Field Resolution)	VDD=3.0V, GN=0, No measurement average, Standard Deviation 100 samples (See typical performance graphs below)		2		milli-gauss
Linearity	±2.0 gauss input range			0.1	±% FS
Hysteresis	±2.0 gauss input range		±25		ppm
Cross-Axis Sensitivity	Test Conditions: Cross field = 0.5 gauss, Happlied = ±3 gauss		±0.2%		%FS/gauss
Output Rate (ODR)	Continuous Measurment Mode Single Measurement Mode	0.75		75 160	Hz Hz
Measurement Period	From receiving command to data ready		6		ms
Turn-on Time	Ready for I2C commands Analog Circuit Ready for Measurements		200 50		µs ms
Gain Tolerance	All gain/dynamic range settings		±5		%
I^2C Address	8-bit read address 8-bit write address		0x3D 0x3C		hex hex
I^2C Rate	Controlled by I^2C Master			400	kHz
I^2C Hysteresis	Hysteresis of Schmitt trigger inputs on SCL and SDA - Fall (VDDIO=1.8V) Rise (VDDIO=1.8V)		0.2*VDDIO 0.8*VDDIO		Volts Volts
Self Test	X & Y Axes Z Axis		±1.16 ±1.08		gauss
	X & Y & Z Axes (GN=5) Positive Bias X & Y & Z Axes (GN=5) Negative Bias	243 -575		575 -243	LSb
Sensitivity Tempco	T_A = -40 to 125°C, Uncompensated Output		-0.3		%/°C
General					
ESD Voltage	Human Body Model (all pins) Charged Device Model (all pins)			2000 750	Volts
Operating Temperature	Ambient	-30		85	°C
Storage Temperature	Ambient, unbiased	-40		125	°C

Characteristics	Conditions*	Min	Typ	Max	Units
Reflow Classification	MSL 3, 260 °C Peak Temperature				
Package Size	Length and Width	2.85	3.00	3.15	mm
Package Height		0.8	0.9	1.0	mm
Package Weight			18		mg

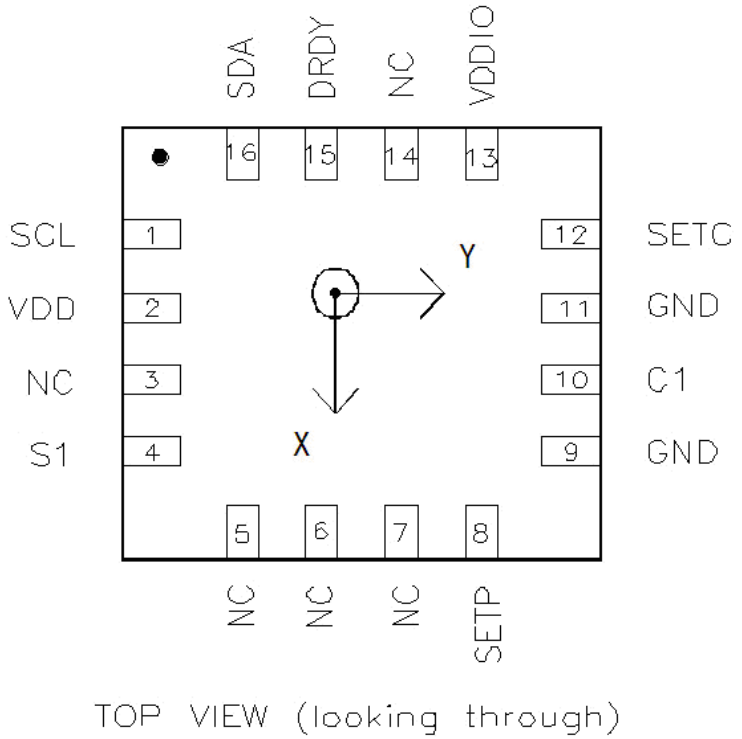
Absolute Maximum Ratings (* Tested at 25°C except stated otherwise.)

Characteristics	Min	Max	Units
Supply Voltage VDD	-0.3	4.8	Volts
Supply Voltage VDDIO	-0.3	4.8	Volts

PIN CONFIGURATIONS

Pin	Name	Description
1	SCL	Serial Clock – I^2C Master/Slave Clock
2	VDD	Power Supply (2.16V to 3.6V)
3	NC	Not to be Connected
4	S1	Tie to VDDIO
5	NC	Not to be Connected
6	NC	Not to be Connected
7	NC	Not to be Connected
8	SETP	Set/Reset Strap Positive – S/R Capacitor (C2) Connection
9	GND	Supply Ground
10	C1	Reservoir Capacitor (C1) Connection
11	GND	Supply Ground
12	SETC	S/R Capacitor (C2) Connection – Driver Side
13	VDDIO	IO Power Supply (1.71V to VDD)
14	NC	Not to be Connected
15	DRDY	Data Ready, Interrupt Pin. Internally pulled high. Optional connection. Low for 250 µsec when data is placed in the data output registers.
16	SDA	Serial Data – I^2C Master/Slave Data

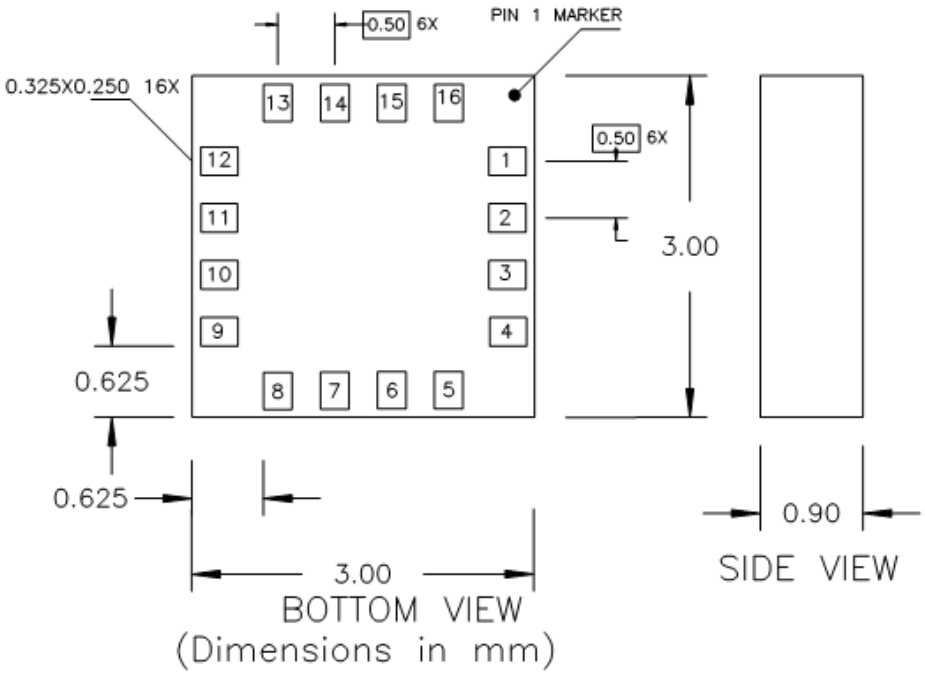
Table 1: Pin Configurations



Arrow indicates direction of magnetic field that generates a positive output reading in Normal Measurement configuration.

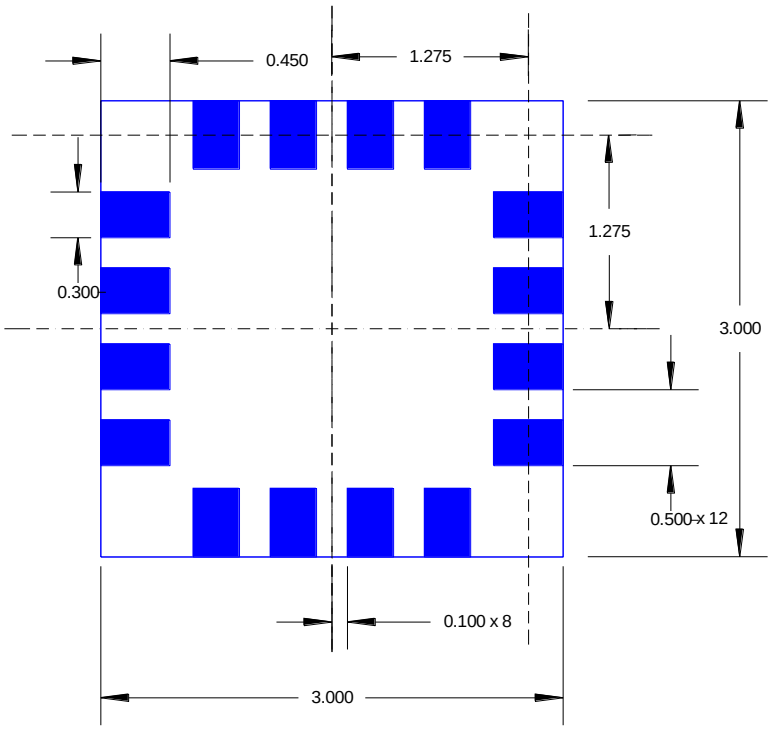
PACKAGE OUTLINES

PACKAGE DRAWING HMC5883L (16-PIN LPCC, dimensions in millimeters)



MOUNTING CONSIDERATIONS

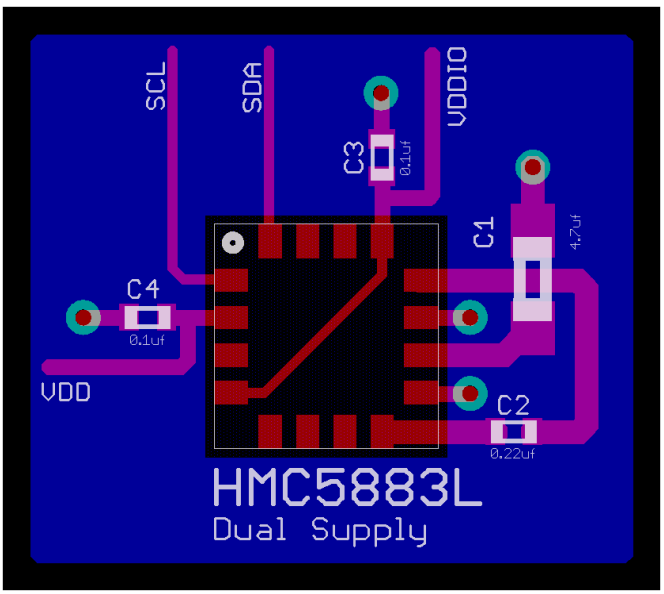
The following is the recommend printed circuit board (PCB) footprint for the HMC5883L.

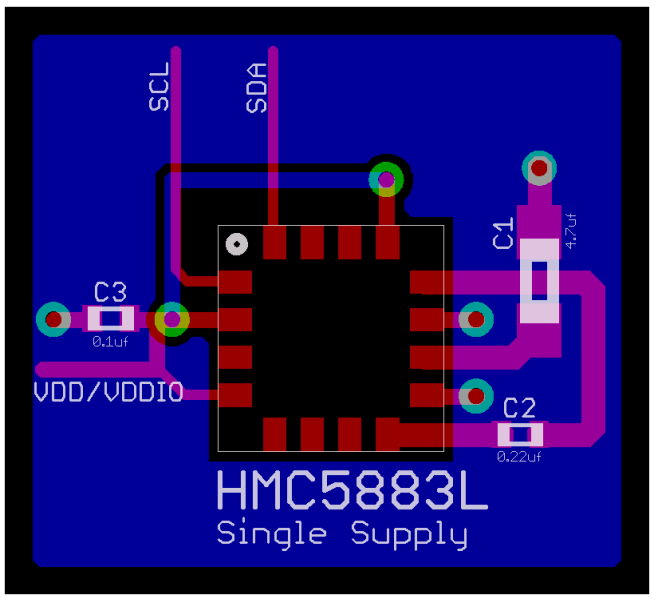


HMC5883 Land Pad Pattern
(All dimensions are in mm)

LAYOUT CONSIDERATIONS

Besides keeping all components that may contain ferrous materials (nickel, etc.) away from the sensor on both sides of the PCB, it is also recommended that there is no conducting copper under/near the sensor in any of the PCB layers. See recommended layout below. Notice that the one trace under the sensor in the dual supply mode is not expected to carry active current since it is for pin 4 pull-up to VDDIO. Power and ground planes are removed under the sensor to minimize possible source of magnetic noise. For best results, use non-ferrous materials for all exposed copper coding.





PCB Pad Definition and Traces

The HMC5883L is a fine pitch LCC package. Refer to previous figure for recommended PCB footprint for proper package centering. Size the traces between the HMC5883L and the external capacitors (C1 and C2) to handle the 1 ampere peak current pulses with low voltage drop on the traces.

Stencil Design and Solder Paste

A 4 mil stencil and 100% paste coverage is recommended for the electrical contact pads.

Reflow Assembly

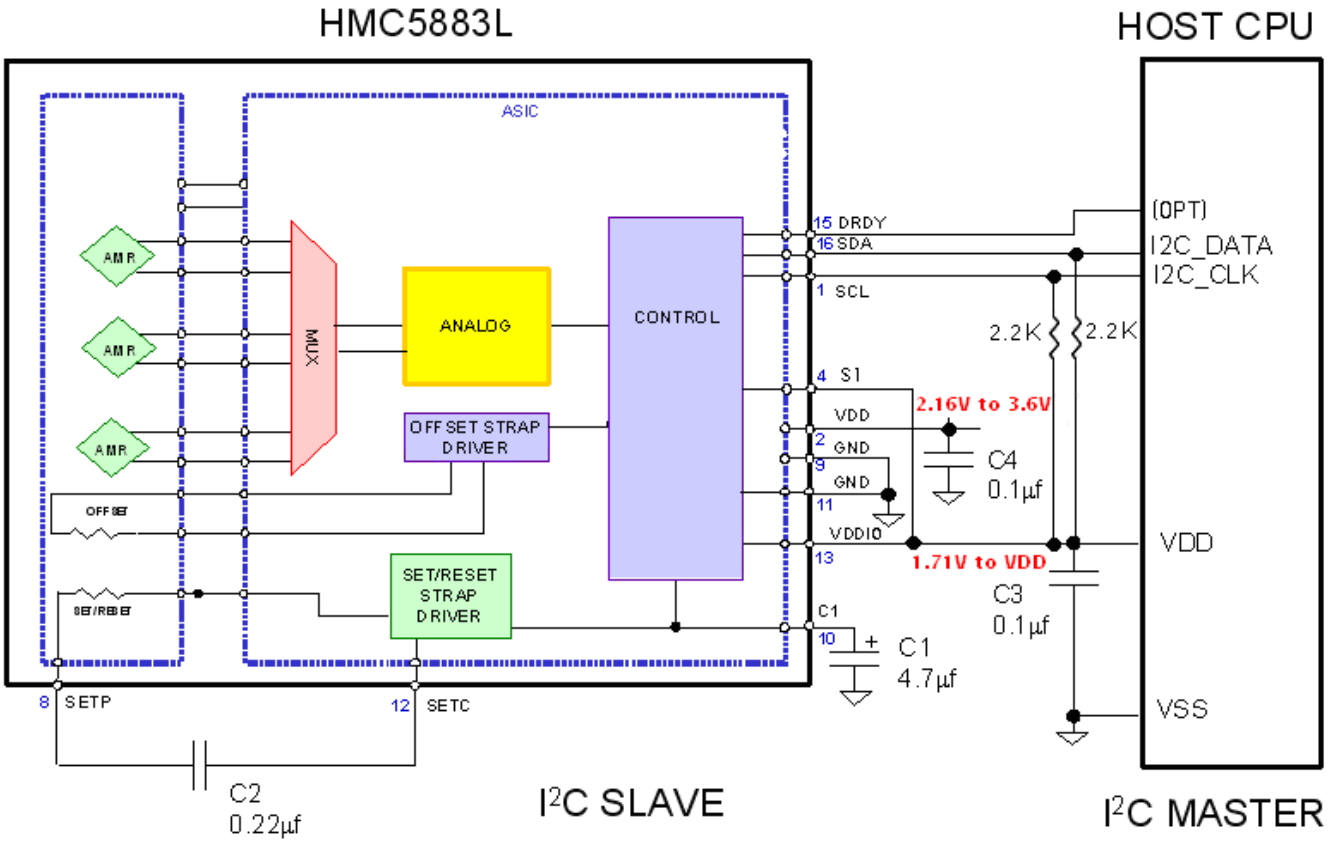
This device is classified as MSL 3 with 260°C peak reflow temperature. A baking process (125°C, 24 hrs) is required if device is not kept continuously in a dry (< 10% RH) environment before assembly. No special reflow profile is required for HMC5883L, which is compatible with lead eutectic and lead-free solder paste reflow profiles. Honeywell recommends adherence to solder paste manufacturer's guidelines. Hand soldering is not recommended. Built-in self test can be used to verify device functionalities after assembly.

External Capacitors

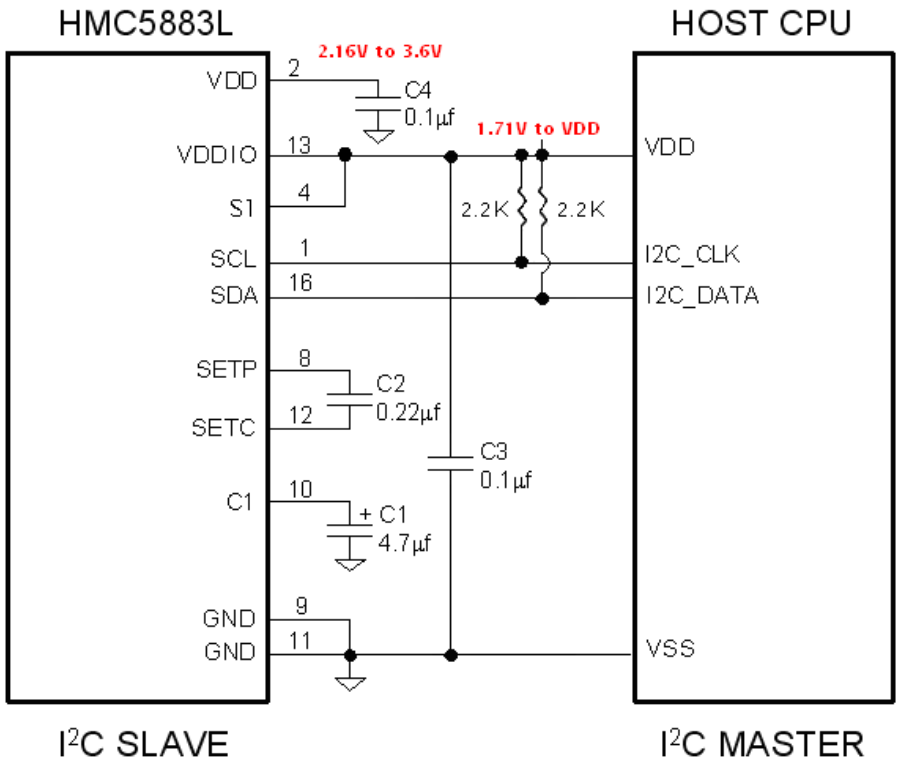
The two external capacitors should be ceramic type construction with low ESR characteristics. The exact ESR values are not critical but values less than 200 milli-ohms are recommended. Reservoir capacitor C1 is nominally 4.7 µF in capacitance, with the set/reset capacitor C2 nominally 0.22 µF in capacitance. Low ESR characteristics may not be in many small SMT ceramic capacitors (0402), so be prepared to up-size the capacitors to gain Low ESR characteristics.

INTERNAL SCHEMATIC DIAGRAM

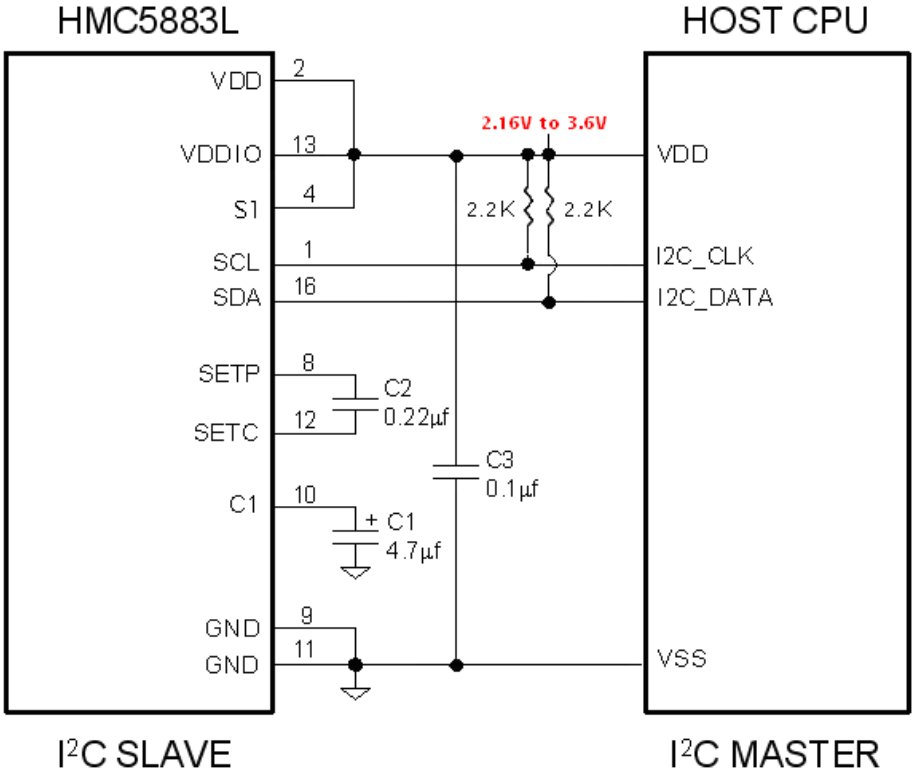
HMC5883L



DUAL SUPPLY REFERENCE DESIGN



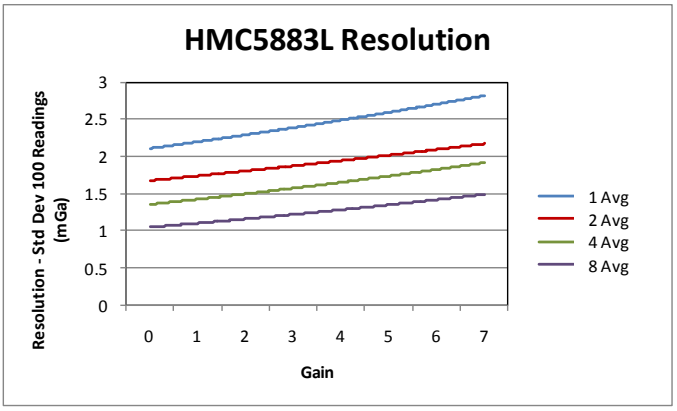
SINGLE SUPPLY REFERENCE DESIGN

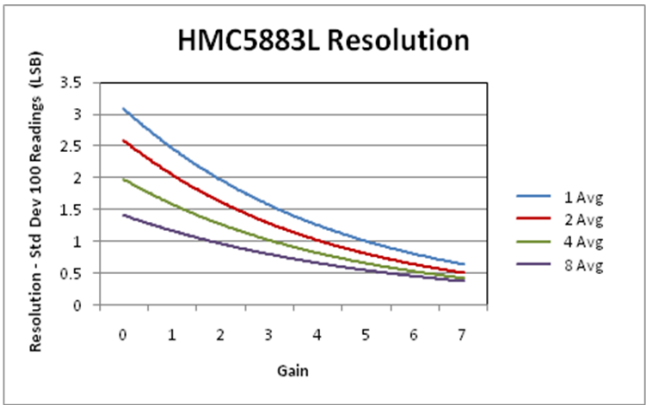


PERFORMANCE

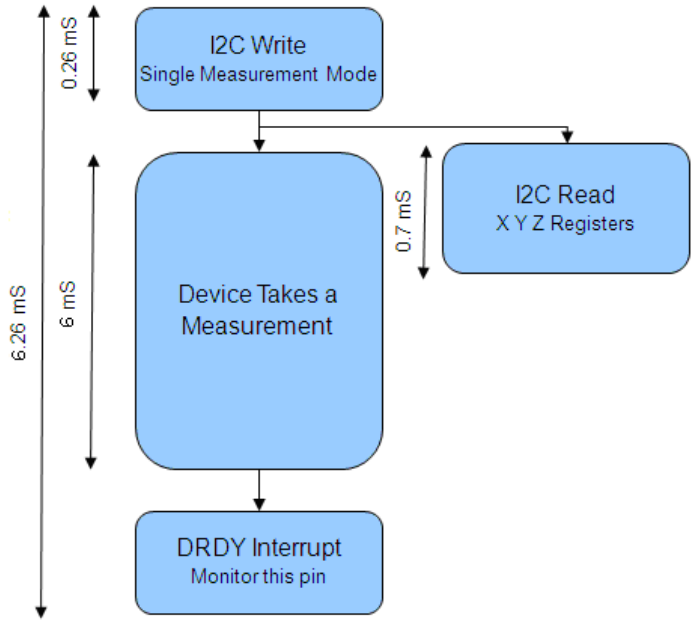
The following graph(s) highlight HMC5883L's performance.

Typical Noise Floor (Field Resolution)





Typical Measurement Period in Single-Measurement Mode



** Monitoring of the DRDY Interrupt pin is only required if maximum output rate is desired.*

BASIC DEVICE OPERATION

Anisotropic Magneto-Resistive Sensors

The Honeywell HMC5883L magnetoresistive sensor circuit is a trio of sensors and application specific support circuits to measure magnetic fields. With power supply applied, the sensor converts any incident magnetic field in the sensitive axis directions to a differential voltage output. The magnetoresistive sensors are made of a nickel-iron (Permalloy) thin-film and patterned as a resistive strip element. In the presence of a magnetic field, a change in the bridge resistive elements causes a corresponding change in voltage across the bridge outputs.

These resistive elements are aligned together to have a common sensitive axis (indicated by arrows in the pinout diagram) that will provide positive voltage change with magnetic fields increasing in the sensitive direction. Because the output is only proportional to the magnetic field component along its axis, additional sensor bridges are placed at orthogonal directions to permit accurate measurement of magnetic field in any orientation.

Self Test

To check the HMC5883L for proper operation, a self test feature in incorporated in which the sensor is internally excited with a nominal magnetic field (in either positive or negative bias configuration). This field is then measured and reported. This function is enabled and the polarity is set by bits MS[n] in the configuration register A. An internal current source generates DC current (about 10 mA) from the VDD supply. This DC current is applied to the offset straps of the magneto-resistive sensor, which creates an artificial magnetic field bias on the sensor. The difference of this measurement and the measurement of the ambient field will be put in the data output register for each of the three axes. By using this built-in function, the manufacturer can quickly verify the sensor's full functionality after the assembly without additional test setup. The self test results can also be used to estimate/compensate the sensor's sensitivity drift due to temperature.

For each "self test measurement", the ASIC:

1. Sends a "Set" pulse
2. Takes one measurement (M1)
3. Sends the (~10 mA) offset current to generate the (~1.1 Gauss) offset field and takes another measurement (M2)
4. Puts the difference of the two measurements in sensor's data output register:

   Output = [M2 – M1] (i.e. output = offset field only)

See SELF TEST OPERATION section later in this datasheet for additional details.

Power Management

This device has two different domains of power supply. The first one is VDD that is the power supply for internal operations and the second one is VDDIO that is dedicated to IO interface. It is possible to work with VDDIO equal to VDD; Single Supply mode, or with VDDIO lower than VDD allowing HMC5883L to be compatible with other devices on board.

I^2C Interface

Control of this device is carried out via the I^2C bus. This device will be connected to this bus as a slave device under the control of a master device, such as the processor.

This device is compliant with *I^2C-Bus Specification*, document number: 9398 393 40011. As an I^2C compatible device, this device has a 7-bit serial address and supports I^2C protocols. This device supports standard and fast modes, 100kHz and 400kHz, respectively, but does not support the high speed mode (Hs). External pull-up resistors are required to support these standard and fast speed modes.

Activities required by the master (register read and write) have priority over internal activities, such as the measurement. The purpose of this priority is to not keep the master waiting and the I^2C bus engaged for longer than necessary.

Internal Clock

The device has an internal clock for internal digital logic functions and timing management. This clock is not available to external usage.

H-Bridge for Set/Reset Strap Drive

The ASIC contains large switching FETs capable of delivering a large but brief pulse to the Set/Reset strap of the sensor. This strap is largely a resistive load. There is no need for an external Set/Reset circuit. The controlling of the Set/Reset function is done automatically by the ASIC for each measurement. One half of the difference from the measurements taken after a set pulse and after a reset pulse will be put in the data output register for each of the three axes. By doing so, the sensor's internal offset and its temperature dependence is removed/cancelled for all measurements. The set/reset pulses also effectively remove the past magnetic history (magnetism) in the sensor, if any.

For each "measurement", the ASIC:

1. Sends a "Set" pulse
2. Takes one measurement (Mset)
3. Sends a "Reset" pulse
4. Takes another measurement (Mreset)
5. Puts the following result in sensor's data output register:

   Output = [Mset – Mreset] / 2

Charge Current Limit

The current that reservoir capacitor (C1) can draw when charging is limited for both single supply and dual supply configurations. This prevents drawing down the supply voltage (VDD).

MODES OF OPERATION

This device has several operating modes whose primary purpose is power management and is controlled by the Mode Register. This section describes these modes.

Continuous-Measurement Mode

During continuous-measurement mode, the device continuously makes measurements, at user selectable rate, and places measured data in data output registers. Data can be re-read from the data output registers if necessary; however, if the master does not ensure that the data register is accessed before the completion of the next measurement, the data output registers are updated with the new measurement. To conserve current between measurements, the device is placed in a state similar to idle mode, but the Mode Register is not changed to Idle Mode. That is, MD[n] bits are unchanged. Settings in the Configuration Register A affect the data output rate (bits DO[n]), the measurement configuration (bits MS[n]), when in continuous-measurement mode. All registers maintain values while in continuous-measurement mode. The I^2C bus is enabled for use by other devices on the network in while continuous-measurement mode.

Single-Measurement Mode

This is the default power-up mode. During single-measurement mode, the device makes a single measurement and places the measured data in data output registers. After the measurement is complete and output data registers are updated, the device is placed in idle mode, and the Mode Register *is* changed to idle mode by setting MD[n] bits. Settings in the configuration register affect the measurement configuration (bits MS[n])when in single-measurement mode. All registers maintain values while in single-measurement mode. The I^2C bus is enabled for use by other devices on the network while in single-measurement mode.

Idle Mode

During this mode the device is accessible through the I^2C bus, but major sources of power consumption are disabled, such as, but not limited to, the ADC, the amplifier, and the sensor bias current. All registers maintain values while in idle mode. The I^2C bus is enabled for use by other devices on the network while in idle mode.

REGISTERS

This device is controlled and configured via a number of on-chip registers, which are described in this section. In the following descriptions, *set* implies a logic 1, and *reset* or *clear* implies a logic 0, unless stated otherwise.

Register List
The table below lists the registers and their access. All address locations are 8 bits.

Address Location	Name	Access
00	Configuration Register A	Read/Write
01	Configuration Register B	Read/Write
02	Mode Register	Read/Write
03	Data Output X MSB Register	Read
04	Data Output X LSB Register	Read
05	Data Output Z MSB Register	Read
06	Data Output Z LSB Register	Read
07	Data Output Y MSB Register	Read
08	Data Output Y LSB Register	Read
09	Status Register	Read
10	Identification Register A	Read
11	Identification Register B	Read
12	Identification Register C	Read

Table2: Register List

Register Access

This section describes the process of reading from and writing to this device. The devices uses an address pointer to indicate which register location is to be read from or written to. These pointer locations are sent from the master to this slave device and succeed the 7-bit address (0x1E) plus 1 bit read/write identifier, i.e. 0x3D for read and 0x3C for write.

To minimize the communication between the master and this device, the address pointer updated automatically without master intervention. The register pointer will be incremented by 1 automatically after the current register has been read successfully.

The address pointer value itself cannot be read via the I^2C bus.
Any attempt to read an invalid address location returns 0's, and any write to an invalid address location or an undefined bit within a valid address location is ignored by this device.

To move the address pointer to a random register location, first issue a "write" to that register location with no data byte following the commend. For example, to move the address pointer to register 10, send 0x3C 0x0A.

Configuration Register A

The configuration register is used to configure the device for setting the data output rate and measurement configuration. CRA0 through CRA7 indicate bit locations, with *CRA* denoting the bits that are in the configuration register. CRA7 denotes the first bit of the data stream. The number in parenthesis indicates the default value of that bit.CRA default is 0x10.

CRA7	CRA6	CRA5	CRA4	CRA3	CRA2	CRA1	CRA0
(0)	MA1(0)	MA0(0)	DO2 (1)	DO1 (0)	DO0 (0)	MS1 (0)	MS0 (0)

Table 3: Configuration Register A

Location	Name	Description
CRA7	CRA7	Bit CRA7 is reserved for future function. Set to 0 when configuring CRA.
CRA6 to CRA5	MA1 to MA0	Select number of samples averaged (1 to 8) per measurement output. 00 = 1(Default); 01 = 2; 10 = 4; 11 = 8
CRA4 to CRA2	DO2 to DO0	Data Output Rate Bits. These bits set the rate at which data is written to all three data output registers.
CRA1 to CRA0	MS1 to MS0	Measurement Configuration Bits. These bits define the measurement flow of the device, specifically whether or not to incorporate an applied bias into the measurement.

Table 4: Configuration Register A Bit Designations

The Table below shows all selectable output rates in continuous measurement mode. All three channels shall be measured within a given output rate. Other output rates with maximum rate of 160 Hz can be achieved by monitoring DRDY interrupt pin in single measurement mode.

DO2	DO1	DO0	Typical Data Output Rate (Hz)
0	0	0	0.75
0	0	1	1.5
0	1	0	3
0	1	1	7.5
1	0	0	15 (Default)
1	0	1	30
1	1	0	75
1	1	1	Reserved

Table 5: Data Output Rates

MS1	MS0	Measurement Mode
0	0	Normal measurement configuration (Default). In normal measurement configuration the device follows normal measurement flow. The positive and negative pins of the resistive load are left floating and high impedance.
0	1	Positive bias configuration for X, Y, and Z axes. In this configuration, a positive current is forced across the resistive load for all three axes.
1	0	Negative bias configuration for X, Y and Z axes. In this configuration, a negative current is forced across the resistive load for all three axes..
1	1	This configuration is reserved.

Table 6: Measurement Modes

Configuration Register B

The configuration register B for setting the device gain. CRB0 through CRB7 indicate bit locations, with *CRB* denoting the bits that are in the configuration register. CRB7 denotes the first bit of the data stream. The number in parenthesis indicates the default value of that bit. CRB default is 0x20.

CRB7	CRB6	CRB5	CRB4	CRB3	CRB2	CRB1	CRB0
GN2 (0)	GN1 (0)	GN0 (1)	(0)	(0)	(0)	(0)	(0)

Table 7: Configuration B Register

Location	Name	Description
CRB7 to CRB5	GN2 to GN0	Gain Configuration Bits. These bits configure the gain for the device. The gain configuration is common for all channels.
CRB4 to CRB0	0	These bits must be cleared for correct operation.

Table 8: Configuration Register B Bit Designations

The table below shows nominal gain settings. Use the "Gain" column to convert counts to Gauss. The "Digital Resolution" column is the theoretical value in term of milli-Gauss per count (LSb) which is the inverse of the values in the "Gain" column. The effective resolution of the usable signal also depends on the noise floor of the system, i.e.

Effective Resolution = Max (Digital Resolution, Noise Floor)

Choose a lower gain value (higher GN#) when total field strength causes overflow in one of the data output registers (saturation). Note that the very first measurement after a gain change maintains the same gain as the previous setting. **The new gain setting is effective from the second measurement and on.**

GN2	GN1	GN0	Recommended Sensor Field Range	Gain (LSb/ Gauss)	Digital Resolution (mG/LSb)	Output Range
0	0	0	± 0.88 Ga	1370	0.73	0xF800–0x07FF (-2048–2047)
0	0	1	± 1.3 Ga	1090 (default)	0.92	0xF800–0x07FF (-2048–2047)
0	1	0	± 1.9 Ga	820	1.22	0xF800–0x07FF (-2048–2047)
0	1	1	± 2.5 Ga	660	1.52	0xF800–0x07FF (-2048–2047)
1	0	0	± 4.0 Ga	440	2.27	0xF800–0x07FF (-2048–2047)
1	0	1	± 4.7 Ga	390	2.56	0xF800–0x07FF (-2048–2047)
1	1	0	± 5.6 Ga	330	3.03	0xF800–0x07FF (-2048–2047)
1	1	1	± 8.1 Ga	230	4.35	0xF800–0x07FF (-2048–2047)

Table 9: Gain Settings

Mode Register

The mode register is an 8-bit register from which data can be read or to which data can be written. This register is used to select the operating mode of the device. MR0 through MR7 indicate bit locations, with *MR* denoting the bits that are in the mode register. MR7 denotes the first bit of the data stream. The number in parenthesis indicates the default value of that bit. Mode register default is 0x01.

MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
(0)	(0)	(0)	(0)	(0)	(0)	MD1 (0)	MD0 (1)

Table 10: Mode Register

Location	Name	Description
MR7 to MR2	0	Bit MR7 is set to 1 internally after each single-measurement operation. Set to 0 when configuring mode register.
MR1 to MR0	MD1 to MD0	Mode Select Bits. These bits select the operation mode of this device.

Table 11: Mode Register Bit Designations

MD1	MD0	Operating Mode
0	0	Continuous-Measurement Mode. In continuous-measurement mode, the device continuously performs measurements and places the result in the data register. RDY goes high when new data is placed in all three registers. After a power-on or a write to the mode or configuration register, the first measurement set is available from all three data output registers after a period of $2/f_{DO}$ and subsequent measurements are available at a frequency of f_{DO}, where f_{DO} is the frequency of data output.
0	1	Single-Measurement Mode (Default). When single-measurement mode is selected, device performs a single measurement, sets RDY high and returned to idle mode. Mode register returns to idle mode bit values. The measurement remains in the data output register and RDY remains high until the data output register is read or another measurement is performed.
1	0	Idle Mode. Device is placed in idle mode.
1	1	Idle Mode. Device is placed in idle mode.

Table 12: Operating Modes

Data Output X Registers A and B

The data output X registers are two 8-bit registers, data output register A and data output register B. These registers store the measurement result from channel X. Data output X register A contains the MSB from the measurement result, and data output X register B contains the LSB from the measurement result. The value stored in these two registers is a 16-bit value in 2's complement form, whose range is 0xF800 to 0x07FF. DXRA0 through DXRA7 and DXRB0 through DXRB7 indicate bit locations, with *DXRA* and *DXRB* denoting the bits that are in the data output X registers. DXRA7 and DXRB7 denote the first bit of the data stream. The number in parenthesis indicates the default value of that bit.

In the event the ADC reading overflows or underflows for the given channel, or if there is a math overflow during the bias measurement, this data register will contain the value -4096. This register value will clear when after the next valid measurement is made.

DXRA7	**DXRA6**	**DXRA5**	**DXRA4**	**DXRA3**	**DXRA2**	**DXRA1**	**DXRA0**
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
DXRB7	**DXRB6**	**DXRB5**	**DXRB4**	**DXRB3**	**DXRB2**	**DXRB1**	**DXRB0**
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Table 13: Data Output X Registers A and B

Data Output Y Registers A and B

The data output Y registers are two 8-bit registers, data output register A and data output register B. These registers store the measurement result from channel Y. Data output Y register A contains the MSB from the measurement result, and data output Y register B contains the LSB from the measurement result. The value stored in these two registers is a 16-bit value in 2's complement form, whose range is 0xF800 to 0x07FF. DYRA0 through DYRA7 and DYRB0 through DYRB7 indicate bit locations, with *DYRA* and *DYRB* denoting the bits that are in the data output Y registers. DYRA7 and DYRB7 denote the first bit of the data stream. The number in parenthesis indicates the default value of that bit.

In the event the ADC reading overflows or underflows for the given channel, or if there is a math overflow during the bias measurement, this data register will contain the value -4096. This register value will clear when after the next valid measurement is made.

DYRA7	**DYRA6**	**DYRA5**	**DYRA4**	**DYRA3**	**DYRA2**	**DYRA1**	**DYRA0**
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
DYRB7	**DYRB6**	**DYRB5**	**DYRB4**	**DYRB3**	**DYRB2**	**DYRB1**	**DYRB0**
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Table 14: Data Output Y Registers A and B

Data Output Z Registers A and B

The data output Z registers are two 8-bit registers, data output register A and data output register B. These registers store the measurement result from channel Z. Data output Z register A contains the MSB from the measurement result, and data output Z register B contains the LSB from the measurement result. The value stored in these two registers is a 16-bit value in 2's complement form, whose range is 0xF800 to 0x07FF. DZRA0 through DZRA7 and DZRB0 through DZRB7 indicate bit locations, with *DZRA* and *DZRB* denoting the bits that are in the data output Z registers. DZRA7 and DZRB7 denote the first bit of the data stream. The number in parenthesis indicates the default value of that bit.

In the event the ADC reading overflows or underflows for the given channel, or if there is a math overflow during the bias measurement, this data register will contain the value -4096. This register value will clear when after the next valid measurement is made.

DZRA7	DZRA6	DZRA5	DZRA4	DZRA3	DZRA2	DZRA1	DZRA0
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
DZRB7	**DZRB6**	**DZRB5**	**DZRB4**	**DZRB3**	**DZRB2**	**DZRB1**	**DZRB0**
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Table 15: Data Output Z Registers A and B

Data Output Register Operation

When one or more of the output registers are read, new data cannot be placed in any of the output data registers until all six data output registers are read. This requirement also impacts DRDY and RDY, which cannot be cleared until new data is placed in all the output registers.

Status Register

The status register is an 8-bit read-only register. This register is used to indicate device status. SR0 through SR7 indicate bit locations, with *SR* denoting the bits that are in the status register. SR7 denotes the first bit of the data stream.

SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0
(0)	(0)	(0)	(0)	(0)	(0)	LOCK (0)	RDY(0)

Table 16: Status Register

Location	Name	Description
SR7 to SR2	0	These bits are reserved.
SR1	LOCK	Data output register lock. This bit is set when: 1.some but not all for of the six data output registers have been read, 2. Mode register has been read. When this bit is set, the six data output registers are locked and any new data will not be placed in these register until one of these conditions are met: 1.all six bytes have been read, 2. the mode register is changed, 3. the measurement configuration (CRA) is changed, 4. power is reset.
SR0	RDY	Ready Bit. Set when data is written to all six data registers. Cleared when device initiates a write to the data output registers and after one or more of the data output registers are written to. When RDY bit is clear it shall remain cleared for a 250 µs. DRDY pin can be used as an alternative to the status register for monitoring the device for measurement data.

Table 17: Status Register Bit Designations

Identification Register A

The identification register A is used to identify the device. IRA0 through IRA7 indicate bit locations, with *IRA* denoting the bits that are in the identification register A. IRA7 denotes the first bit of the data stream. The number in parenthesis indicates the default value of that bit.

The identification value for this device is stored in this register. This is a read-only register.
Register values. ASCII value *H*

IRA7	IRA6	IRA5	IRA4	IRA3	IRA2	IRA1	IRA0
0	1	0	0	1	0	0	0

Table 18: Identification Register A Default Values

Identification Register B

The identification register B is used to identify the device. IRB0 through IRB7 indicate bit locations, with *IRB* denoting the bits that are in the identification register A. IRB7 denotes the first bit of the data stream.

Register values. ASCII value *4*

IRB7	IRB6	IRB5	IRB4	IRB3	IRB2	IRB1	IRB0
0	0	1	1	0	1	0	0

Table 19: Identification Register B Default Values

Identification Register C

The identification register C is used to identify the device. IRC0 through IRC7 indicate bit locations, with *IRC* denoting the bits that are in the identification register A. IRC7 denotes the first bit of the data stream.

Register values. ASCII value *3*

IRC7	IRC6	IRC5	IRC4	IRC3	IRC2	IRC1	IRC0
0	0	1	1	0	0	1	1

Table 20: Identification Register C Default Values

I^2C COMMUNICATION PROTOCOL

The HMC5883L communicates via a two-wire I^2C bus system as a slave device. The HMC5883L uses a simple protocol with the interface protocol defined by the I^2C bus specification, and by this document. The data rate is at the standard-mode 100kbps or 400kbps rates as defined in the I^2C Bus Specifications. The bus bit format is an 8-bit Data/Address send and a 1-bit acknowledge bit. The format of the data bytes (payload) shall be case sensitive ASCII characters or binary data to the HMC5883L slave, and binary data returned. Negative binary values will be in two's complement form. The default (factory) HMC5883L 8-bit slave address is 0x3C for write operations, or 0x3D for read operations.

The HMC5883L Serial Clock (SCL) and Serial Data (SDA) lines require resistive pull-ups (Rp) between the master device (usually a host microprocessor) and the HMC5883L. Pull-up resistance values of about 2.2K to 10K ohms are recommended with a nominal VDDIO voltage. Other resistor values may be used as defined in the I^2C Bus Specifications that can be tied to VDDIO.

The SCL and SDA lines in this bus specification may be connected to multiple devices. The bus can be a single master to multiple slaves, or it can be a multiple master configuration. All data transfers are initiated by the master device, which is responsible for generating the clock signal, and the data transfers are 8 bit long. All devices are addressed by I^2C's unique 7-bit address. After each 8-bit transfer, the master device generates a 9^{th} clock pulse, and releases the SDA line. The receiving device (addressed slave) will pull the SDA line low to acknowledge (ACK) the successful transfer or leave the SDA high to negative acknowledge (NACK).

Per the I^2C spec, all transitions in the SDA line must occur when SCL is low. This requirement leads to two unique conditions on the bus associated with the SDA transitions when SCL is high. Master device pulling the SDA line low while the SCL line is high indicates the Start (S) condition, and the Stop (P) condition is when the SDA line is pulled high while the SCL line is high. The I^2C protocol also allows for the Restart condition in which the master device issues a second start condition without issuing a stop.

All bus transactions begin with the master device issuing the start sequence followed by the slave address byte. The address byte contains the slave address; the upper 7 bits (bits7-1), and the Least Significant bit (LSb). The LSb of the address byte designates if the operation is a read (LSb=1) or a write (LSb=0). At the 9^{th} clock pulse, the receiving slave device will issue the ACK (or NACK). Following these bus events, the master will send data bytes for a write operation, or the slave will clock out data with a read operation. All bus transactions are terminated with the master issuing a stop sequence.

I^2C bus control can be implemented with either hardware logic or in software. Typical hardware designs will release the SDA and SCL lines as appropriate to allow the slave device to manipulate these lines. In a software implementation, care must be taken to perform these tasks in code.

OPERATIONAL EXAMPLES

The HMC5883L has a fairly quick stabilization time from no voltage to stable and ready for data retrieval. The nominal 56 milli-seconds with the factory default single measurement mode means that the six bytes of magnetic data registers (DXRA, DXRB, DZRA, DZRB, DYRA, and DYRB) are filled with a valid first measurement.

To change the measurement mode to continuous measurement mode, after the power-up time send the three bytes:

0x3C 0x02 0x00

This writes the 00 into the second register or mode register to switch from single to continuous measurement mode setting. With the data rate at the factory default of 15Hz updates, a 67 milli-second typical delay should be allowed by the I^2C master before querying the HMC5883L data registers for new measurements. To clock out the new data, send:

0x3D, and clock out DXRA, DXRB, DZRA, DZRB, DYRA, and DYRB located in registers 3 through 8. The HMC5883L will automatically re-point back to register 3 for the next 0x3D query. All six data registers must be read properly before new data can be placed in any of these data registers.

Below is an example of a (power-on) initialization process for “continuous-measurement mode”:

1. Write CRA (00) – send **0x3C 0x00 0x70** (8-average, 15 Hz default, normal measurement)
2. Write CRB (01) – send **0x3C 0x01 0xA0** (Gain=5, or any other desired gain)
3. Write Mode (02) – send **0x3C 0x02 0x00** (Continuous-measurement mode)
4. Wait 6 ms or monitor status register or DRDY hardware interrupt pin
5. Loop
   Send **0x3D 0x06** (Read all 6 bytes. If gain is changed then this data set is using previous gain)
   Convert three 16-bit 2’s compliment hex values to decimal values and assign to X, Z, Y, respectively.
   Send **0x3C 0x03** (point to first data register 03)
   Wait about 67 ms (if 15 Hz rate) or monitor status register or DRDY hardware interrupt pin

   End_loop

Below is an example of a (power-on) initialization process for “single-measurement mode”:

1. Write CRA (00) – send **0x3C 0x00 0x70** (8-average, 15 Hz default or any other rate, normal measurement)
2. Write CRB (01) – send **0x3C 0x01 0xA0** (Gain=5, or any other desired gain)
3. For each measurement query:
   Write Mode (02) – send **0x3C 0x02 0x01** (Single-measurement mode)
   Wait 6 ms or monitor status register or DRDY hardware interrupt pin
   Send **0x3D 0x06** (Read all 6 bytes. If gain is changed then this data set is using previous gain)
   Convert three 16-bit 2’s compliment hex values to decimal values and assign to X, Z, Y, respectively.

SELF TEST OPERATION

To check the HMC5883L for proper operation, a self test feature in incorporated in which the sensor offset straps are excited to create a nominal field strength (bias field) to be measured. To implement self test, the least significant bits (MS1 and MS0) of configuration register A are changed from 00 to 01 (positive bias) or 10 (negetive bias).

Then, by placing the mode register into single or continuous-measurement mode, two data acquisition cycles will be made on each magnetic vector. The first acquisition will be a set pulse followed shortly by measurement data of the external field. The second acquisition will have the offset strap excited (about 10 mA) in the positive bias mode for X, Y, and Z axes to create about a 1.1 gauss self test field plus the external field. The first acquisition values will be subtracted from the second acquisition, and the net measurement will be placed into the data output registers.

Since self test adds ~1.1 Gauss additional field to the existing field strength, using a reduced gain setting prevents sensor from being saturated and data registers overflowed. For example, if the configuration register B is set to 0xA0 (Gain=5), values around +452 LSb (1.16 Ga * 390 LSb/Ga) will be placed in the X and Y data output registers and around +421 (1.08 Ga * 390 LSb/Ga) will be placed in Z data output register. To leave the self test mode, change MS1 and MS0 bit of the configuration register A back to 00 (Normal Measurement Mode). Acceptable limits of the self test values depend on the gain setting. Limits for Gain=5 is provided in the specification table.

Below is an example of a “positive self test” process using continuous-measurement mode:

1. Write CRA (00) – send **0x3C 0x00 0x71** (8-average, 15 Hz default, positive self test measurement)
2. Write CRB (01) – send **0x3C 0x01 0xA0** (Gain=5)
3. Write Mode (02) – send **0x3C 0x02 0x00** (Continuous-measurement mode)
4. Wait 6 ms or monitor status register or DRDY hardware interrupt pin
5. Loop
   Send **0x3D 0x06** (Read all 6 bytes. If gain is changed then this data set is using previous gain)
   Convert three 16-bit 2’s compliment hex values to decimal values and assign to X, Z, Y, respectively.
   Send **0x3C 0x03** (point to first data register 03)
   Wait about 67 ms (if 15 Hz rate) or monitor status register or DRDY hardware interrupt pin
   End_loop
6. Check limits –
   If all 3 axes (X, Y, and Z) are within reasonable limits (243 to 575 for Gain=5, adjust these limits basing on the gain setting used. See an example below.) Then
   All 3 axes pass positive self test
   Write CRA (00) – send **0x3C 0x00 0x70** (Exit self test mode and this procedure)
   Else
   If Gain<7
   Write CRB (01) – send **0x3C 0x01 0x_0** (Increase gain setting and retry, skip the next data set)
   Else
   At least one axis did not pass positive self test
   Write CRA (00) – send **0x3C 0x00 0x70** (Exit self test mode and this procedure)
   End If

Below is an example of how to adjust the “positive self” test limits basing on the gain setting:

1. If Gain = 6, self test limits are:
   Low Limit = 243 * 330/390 = 206
   High Limit = 575 * 330/390 = 487

2. If Gain = 7, self test limits are:
   Low Limit = 243 * 230/390 = 143
   High Limit = 575 * 230/390 = 339

SCALE FACTOR TEMPERATURE COMPENSATION

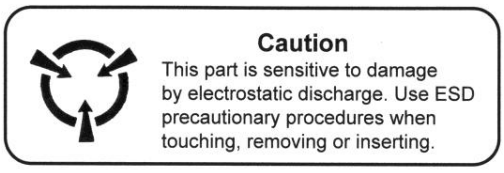
The built-in self test can also be used to periodically compensate the scaling errors due to temperature variations. A compensation factor can be found by comparing the self test outputs with the ones obtained at a known temperature. For example, if the self test output is 400 at room temperature and 300 at the current temperature then a compensation factor of (400/300) should be applied to all current magnetic readings. A temperature sensor is not required using this method.

Below is an example of a temperature compensation process using positive self test method:

1. If self test measurement at a temperature “when the last magnetic calibration was done”:
   X_STP = 400
   Y_STP = 410
   Z_STP = 420
2. If self test measurement at a different tmperature:
   X_STP = 300 (Lower than before)
   Y_STP = 310 (Lower than before)
   Z_STP = 320 (Lower than before)
   Then
   X_TempComp = 400/300
   Y_TempComp = 410/310
   Z_TempComp = 420/320
3. Applying to all new measurements:
   X = X * X_TempComp
   Y = Y * Y_TempComp
   Z = Z * Z_TempComp
   Now all 3 axes are temperature compensated, i.e. sensitivity is same as “when the last magnetic calibration was done”; therefore, the calibration coefficients can be applied without modification.
4. Repeat this process periodically or,for every Δt degrees of temperature change measured, if available.

ORDERING INFORMATION

Ordering Number	Product
HMC5883L-TR	Tape and Reel 4k pieces/reel



CAUTION: ESDS CAT. 1B

FIND OUT MORE

For more information on Honeywell’s Magnetic Sensors visit us online at www.honeywell.com/magneticsensors or contact us at 800-323-8295 (763-954-2474 internationally).

Honeywell
12001 Highway 55
Plymouth, MN 55441
Tel: 800-323-8295
www.honeywell.com/magneticsensors

Form # 900405 Rev D
March 2011

Honeywell

ANEXO C – 3V Tips ’n Tricks

Páginas 1, 4 [24].

CHAPTER 8

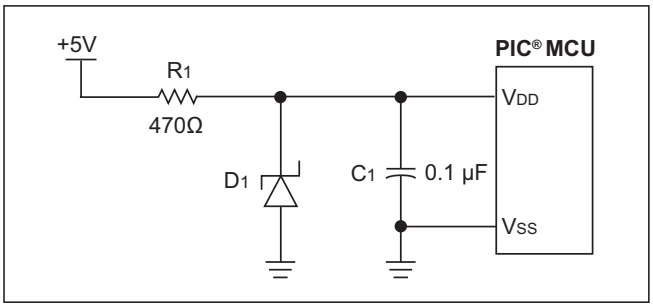
3V Tips ‘n Tricks

Table Of Contents

TIP #2 Low-Cost Alternative Power System Using a Zener Diode

Details a low-cost regulator alternative using a Zener diode.

Figure 2-1: Zener Supply



A simple, low-cost 3.3V regulator can be made out of a Zener diode and a resistor as shown in Figure 2-1. In many applications, this circuit can be a cost-effective alternative to using a LDO regulator. However, this regulator is more load sensitive than a LDO regulator. Additionally, it is less energy efficient, as power is always being dissipated in R1 and D1.

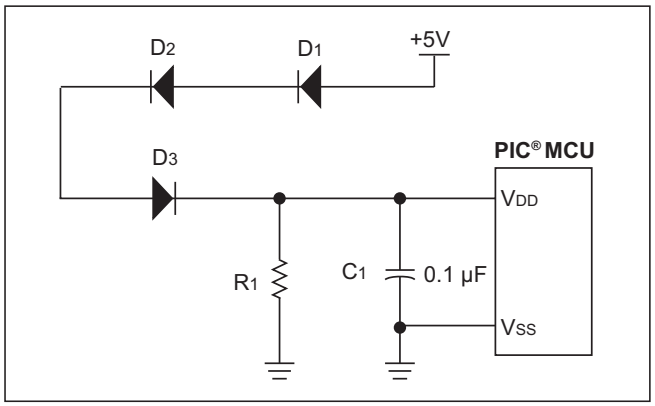
R1 limits the current to D1 and the PIC MCU so that V_{DD} stays within the allowable range. Because the reverse voltage across a Zener diode varies as the current through it changes, the value of R1 needs to be considered carefully.

R1 must be sized so that at maximum load, typically when the PIC MCU is running and is driving its outputs high, the voltage drop across R1 is low enough so that the PIC MCU has enough voltage to operate. Also, R1 must be sized so that at minimum load, typically when the PIC MCU is in Reset, that V_{DD} does not exceed either the Zener diode's power rating or the maximum V_{DD} for the PIC MCU.

TIP #3 Lower Cost Alternative Power System Using 3 Rectifier Diodes

Figure 3-1 details a lower cost regulator alternative using 3 rectifier diodes.

Figure 3-1: Diode Supply



We can also use the forward drop of a series of normal switching diodes to drop the voltage going into the PIC MCU. This can be even more cost-effective than the Zener diode regulator. The current draw from this design is typically less than a circuit using a Zener.

The number of diodes needed varies based on the forward voltage of the diode selected. The voltage drop across diodes D1-D3 is a function of the current through the diodes. R1 is present to keep the voltage at the PIC MCUs V_{DD} pin from exceeding the PIC MCUs maximum V_{DD} at minimum loads (typically when the PIC MCU is in Reset or sleeping). Depending on the other circuitry connected to V_{DD}, this resistor may have its value increased or possibly even eliminated entirely. Diodes D1-D3 must be selected so that at maximum load, typically when the PIC is running and is driving its outputs high, the voltage drop across D1-D3 is low enough to meet the PIC MCUs minimum V_{DD} requirements.

ANEXO D – Bi-directional level shifter for I^2C-bus and other systems

Páginas 8-12 [25].

2. INTERCONNECTION OF DEVICES WITH DIFFERENT LOGIC LEVELS.

2.1 Logic levels of the I²C-bus.

An overview of the different logic levels, used in I^2C-bus systems, is given below.

The I^2C-bus specifies two types of logic levels:

a) fixed levels,

b) supply voltage related levels.

a) The fixed levels are intended for non-CMOS devices and/or devices with higher supply voltages than 5 Volt, e.g. 12 Volt. The I/O levels for fixed level devices are:

LOW level input voltage V_{IL}	min. -0.5V	max. 1.5V
HIGH level input voltage V_{IH}	min. 3.0V	max. V_{DDmax}+0.5V
LOW level output voltage V_{OL1}	min. 0V	max. 0.4V
HIGH level output voltage V_{OH}	open drain output, determined by V_{DD} via an external pull-up resistor.	

b) The supply voltage related levels are intended for CMOS devices and/or devices with supply voltages of 5V or lower. Their I/O levels are:

LOW level input voltage V_{IL}	min. -0.5V	max. $0.3V_{DD}$
HIGH level input voltage V_{IH}	min. $0.7V_{DD}$	max. V_{DDmax}+0.5V
LOW level output voltage V_{OL1}	min. 0V	max. 0.4V
HIGH level output voltage V_{OH}	open drain output, determined by V_{DD} via an external pull-up resistor.	

The logic levels of the bus lines depends on the pull-up resistors to V_{DD}, leakage current and, if present, series resistors to the I/O pins of the devices. Their values must be chosen in such a way that during the LOW level a minimum noise margin of 0.1 V_{DD} is present and 0.2 V_{DD} during the HIGH level.

2.2 I²C-bus devices with different supply voltages and 5V tolerant I/O's.

From the listed values in 2.1 can be concluded that fixed level devices and 5V supply voltage related devices can be connected directly to the same bus lines, without additional components. Also 3.3V supply voltage related devices can be connected directly to these bus lines as long as they have 5V tolerant I/O pins, because the pull-up resistors have to be connected to the 5V supply voltage (see Figure 1).

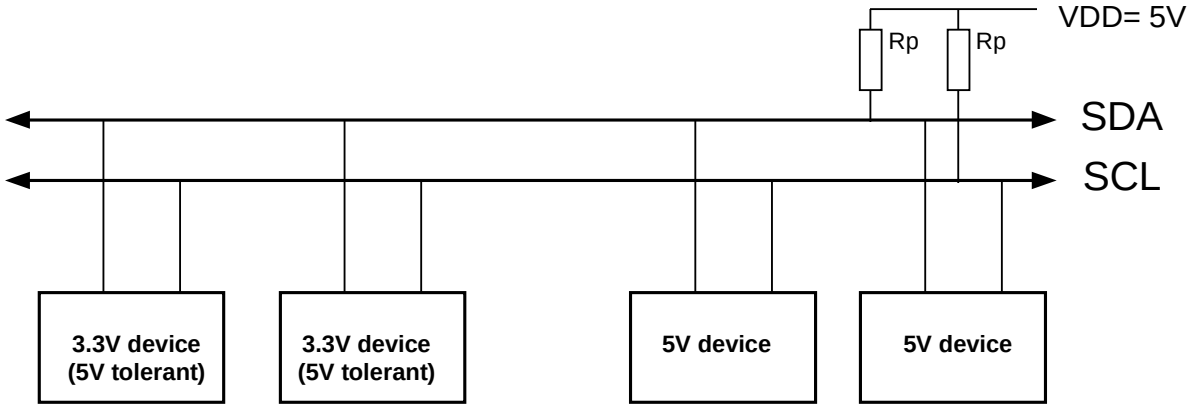


Figure 1. I²C-bus system with 3.3V devices (5V tolerant) and 5V devices, all connected to the same bus lines.

This is the most simple solution, but the lower voltage devices must be 5 Volt tolerant, which may make them more expensive to manufacture. Lower voltage devices with a supply voltage as low as 2 Volt still meet the I²C-bus specification and can be connected to the bus system of Figure 1.

If devices with a supply voltage lower than 2.7 Volt are connected via series resistors to the bus lines, then attention must be paid to meet the 0.1 V_{DD} noise margin during the LOW level, required by the I²C-bus specification. The required 0.2 V_{DD} noise margin during the HIGH level does not depend on the supply voltage.

Devices with a supply voltage lower than 2 Volt do not meet the noise margin requirement of 0.1 V_{DD} because the LOW level on the bus lines is 0.4V and their input level of 0.3 V_{DD} is less than 0.6V. This will be solved in the next update of the I²C-bus specification.

2.3 Devices with different logic levels connected via the bi-directional level shifter.

The bi-directional level shifter is used to interconnect two sections of an I^2C-bus system, each section with a different supply voltage and different logic levels. In the bus system of Figure 2 the left section has pull-up resistors and devices connected to a 3.3 Volt supply voltage, the right section has pull-up resistors and devices connected to a 5 Volt supply voltage. The devices of each section have I/O's with supply voltage related logic input levels and an open drain output configuration.

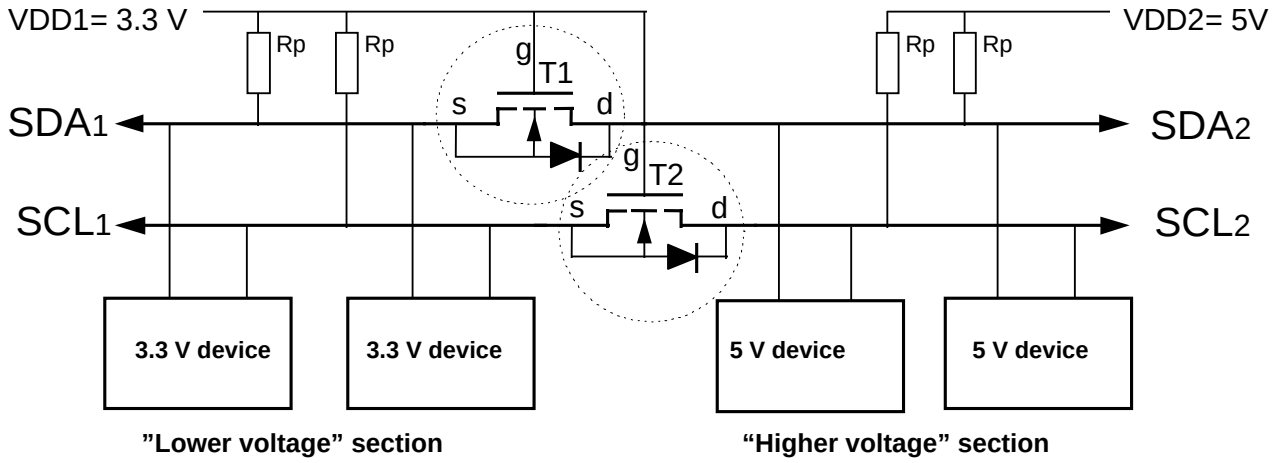


Figure 2. Bi-directional level shifter circuit connects two different voltage sections of an I^2C-bus system.

The level shifter for each bus line is identical and consists of one discrete N-channel enhancement MOS-FET, T1 for the serial data line SDA and T2 for the serial clock line SCL. The gates (g) has to be connected to the lowest supply voltage VDD1 , the sources (s) to the bus lines of the "Lower voltage" section, and the drains (d) to the bus lines of the "Higher voltage" section. Many MOS-FET's have the substrate internally already connected with its source, otherwise it should be done externally. The diode between the drain (d) and substrate is inside the MOS-FET present as n-p junction of drain and substrate.

2.3.1 Description of the level shift operation.

For the level shift operation three states has to be considered:

- State 1. No device is pulling down the bus line and the bus line of the "Lower voltage" section is pulled up by its pull-up resistors Rp to 3.3 V. The gate and the source of the MOS-FET are both at 3.3 V, so its V_{GS} is below the threshold voltage and the MOS-FET is not conducting. This allows that the bus line at the "Higher voltage" section is pulled up by its pull-up resistor Rp to 5V. So the bus lines of both sections are HIGH, but at a different voltage level.
- State 2. A 3.3 V device pulls down the bus line to a LOW level. The source of the MOS-FET becomes also LOW, while the gate stay at 3.3 V. The V_{GS} rises above the threshold and the MOS-FET becomes conducting. Now the bus line of the "Higher voltage" section is also pulled down to a LOW level by the 3.3 V device via the conducting MOS-FET. So the bus lines of both sections become LOW at the same voltage level.

- State 3. A 5 V device pulls down the bus line to a LOW level. Via the drain-substrate diode of the MOS-FET the “Lower voltage” section is in first instance pulled down until V_{GS} passes the threshold and the MOS-FET becomes conducting. Now the bus line of the “Lower voltage” section is further pulled down to a LOW level by the 5 V device via the conducting MOS-FET. So the bus lines of both sections become LOW at the same voltage level.

The three states show that the logic levels are transferred in both directions of the bus system, independent of the driving section. State 2 and state 3 perform the “wired AND” function between the bus lines of both sections as required by the I^2C-bus specification.

Other supply voltages than 3.3V for VDD1 and 5V for VDD2 can be applied, e.g. 2V for VDD1 and 10V for VDD2 is feasible. In normal operation VDD2 must be equal to or higher than VDD1.

The MOS-FET’s allow that VDD2 is lower than VDD1 during switching power on/off, of course the bus system is not operational during that time.

The maximum VDD2 is not critical as long as the drain of the MOS-FET can withstand this voltage. At a higher VDD2 a slower falling edge for both bus sections has to be taken in account, both in state 2 and state 3, because it takes more discharge time of the bus line.

The lowest possible supply voltage VDD1 depends on the threshold voltage $V_{GS(th)}$ of the MOS-FET’s. With a threshold voltage of about 1 Volt below the lowest VDD1, the level shifter circuit will operate properly. If for example the lowest VDD1 is 3 Volt, a threshold voltage $V_{GS(th)}$ of maximum 2 Volt is allowed.

2.3.2 Protection of the “Lower voltage” section against high voltage spikes.

If an I^2C-bus system has to be connected with e.g. external bus lines on which high voltage spikes can be expected, the level shifter circuit may be used as protection circuit as long as the drain of the MOS-FET can withstand these high voltage spikes. The ”Lower voltage” section is the protected side, the “Higher voltage” section the is the side of the external bus lines (see Figure 2). If in this application no level shifting is required, VDD1 and VDD2 can be interconnected.

2.3.3 The level shifter used in point to point connections.

The circuit of figure 2 can also be used as a one-directional level shifter between an output signal and one or more inputs, which have higher or lower logic levels than that output signal.

If the output signal is generated by a push-pull stage, then the Rp at the output circuit side, (“Lower voltage” or “Higher voltage” section) can be omitted. The Rp at the input circuit(s) side remains needed.

The protection and isolation features, described in 2.3.2 and 2.3.4 also apply here.

2.3.4 Isolation of the powered-down “Lower voltage” section.

An additional feature of the level shifter circuit in figure 2 is the isolation of the “Lower voltage” section when VDD1 is switched off. In that case VDD1 is about 0 Volt and the MOS-FET’s are switched off because V_{GS} is below the threshold voltage. The “Higher voltage” section is not hindered and stays operational. To assure a noise margin, the MOS-FET’s should have a minimum threshold voltage $V_{GS(th)}$ of e.g. 0.4V and VDD1 must stay below this value. The isolation feature can also be applied if no level shifting is required, VDD1 and VDD2 may have the same value, e.g. both 3.3V or both 5V.

2.3.5 Extended circuit for isolation of the powered-down “higher voltage” section.

If it is necessary to isolate also the “Higher voltage” section when it is powered off, then the level shifter circuit can be extended as shown in figure 3.

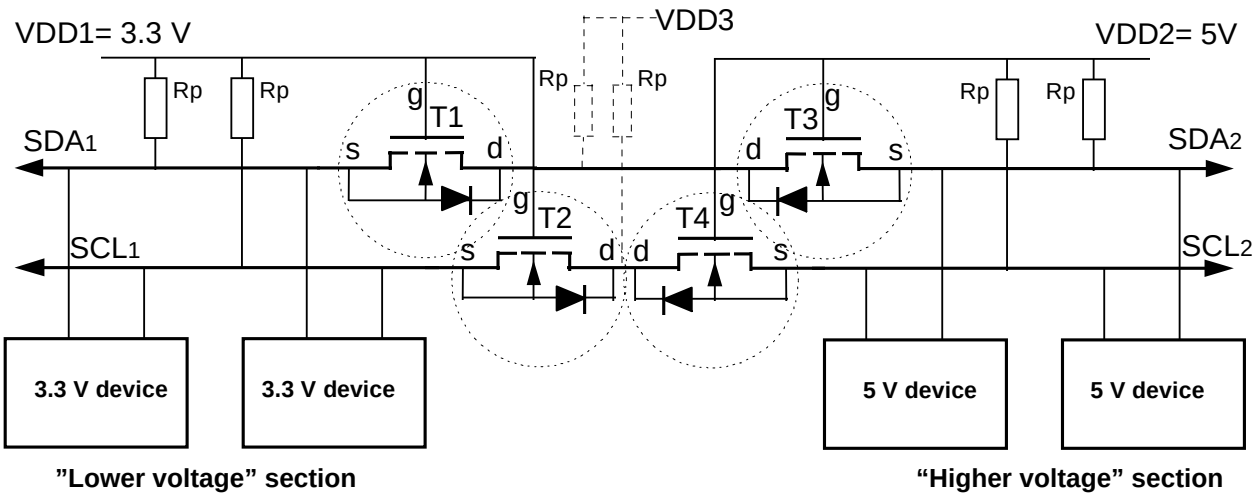


Figure 3. I^2C-bus system in which the “Higher voltage” section is isolated at power-off.

If VDD2 is switched-off then T3 and T4 becomes not conducting and the “Higher voltage” section is isolated from the other part of the bus system. The pull-up resistors Rp to VDD3 are not necessary for the proper operation and may have a high resistance value, they can be added to prevent the MOS-FET drains become floating at a HIGH level. VDD3 is preferably connected to the highest supply voltage. If VDD3 has a lower value, then care must be taken that the logic HIGH levels of the bus lines are not decreased too much.

The “Lower voltage” section is isolated if VDD1 is switched off, in the same way as in Figure 2 and described in 2.3.4, but now independent of the value of VDD2.

Because this level shifter circuit is symmetrical, the “Lower voltage” section and “Higher voltage” section can be chosen arbitrary at the left or right side in figure 3. Even more sections with a higher, a lower or a same supply voltage value can be added by connecting these sections via additional MOS-FET’s to the common drain terminals (d) in the same way as the other sections in Figure 3. Every section is isolated from the rest of the bus system when its supply voltage is switched off, while level shifting between all other sections remain operational.

ANEXO E – Level shifting techniques in I^2C-bus design

Páginas 3-4 [26].

1. Introduction

Present technology processes for integrated circuits with clearances of 0.5 μm and less limit the maximum supply voltage and consequently the logic levels for the digital I/O signals. To interface these lower voltage circuits with existing 5 V devices, a level shifter is needed. For bidirectional bus systems like the I²C-bus, such a level shifter must also be bidirectional, without the need of a direction control signal. The simplest way to solve this problem is by connecting a discrete MOS-FET to each bus line.

2. Bidirectional level shifter for Fast-mode and Standard-mode I²C-bus systems

In spite of its surprising simplicity, such a solution not only fulfils the requirement of bidirectional level shifting without a direction control signal, it also:

- isolates a powered-down bus section from the rest of the bus system
- protects the ‘lower voltage’ side against high voltage spikes from the ‘higher-voltage’ side.

The bidirectional level shifter can be used for both Standard-mode (up to100 kbit/s) or in Fast-mode (up to 400 kbit/s) I²C-bus systems. It is not intended for Hs-mode systems, which may have a bridge with a level shifting possibility.

2.1 Connecting devices with different logic levels

Different voltage devices could be connected to the same bus by using pull-up resistors to the supply voltage line. Although this is the simplest solution, the lower voltage devices must be 5 V tolerant, which can make them more expensive to manufacture. By using a bidirectional level shifter, however, it is possible to interconnect two sections of an I²C-bus system, with each section having a different supply voltage and different logic levels. Such a configuration is shown in Figure 1. The left ‘low-voltage’ section has pull-up resistors and devices connected to a 3.3 V supply voltage; the right ‘high-voltage’ section has pull-up resistors and devices connected to a 5 V supply voltage. The devices of each section have I/Os with supply voltage related logic input levels and an open-drain output configuration.

The level shifter for each bus line is identical and consists of one discrete N-channel enhancement MOS-FET; TR1 for the serial data line SDA and TR2 for the serial clock line SCL. The gates (g) have to be connected to the lowest supply voltage V_{DD1}, the sources (s) to the bus lines of the ‘lower-voltage’ section, and the drains (d) to the bus lines of the ‘higher-voltage’ section. Many MOS-FETs have the substrate internally connected with its source, if this is not the case, an external connection should be made. Each MOS-FET has an integral diode (n-p junction) between the drain and substrate.

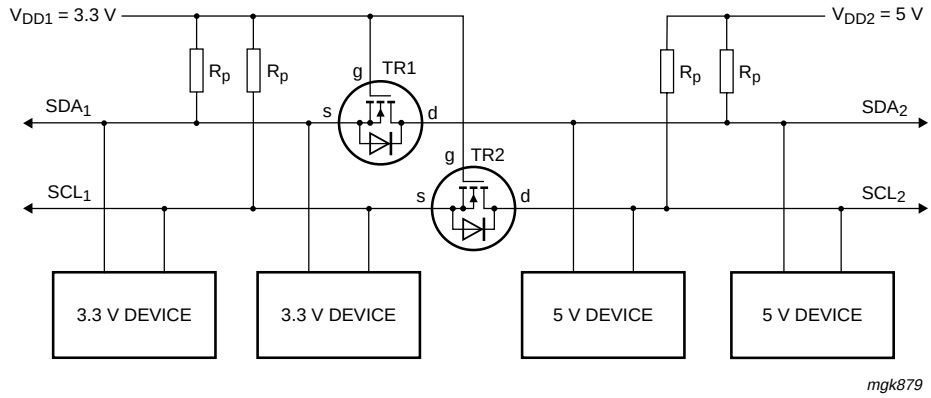


Fig 1. Bidirectional level shifter circuit connecting two different voltage sections in an I²C-bus system

2.1.1 Operation of the level shifter

The following three states should be considered during the operation of the level shifter:

1. No device is pulling down the bus line. The bus line of the 'lower-voltage' section is pulled up by its pull-up resistors R_p to 3.3 V. The gate and the source of the MOS-FET are both at 3.3 V, so its V_{GS} is below the threshold voltage and the MOS-FET is not conducting. This allows the bus line at the 'higher-voltage' section to be pulled up by its pull-up resistor R_p to 5 V. So the bus lines of both sections are HIGH, but at a different voltage level.
2. A 3.3 V device pulls down the bus line to a LOW level. The source of the MOS-FET also becomes LOW, while the gate stays at 3.3 V. V_{GS} rises above the threshold and the MOS-FET starts to conduct. The bus line of the 'higher-voltage' section is then also pulled down to a LOW level by the 3.3 V device via the conducting MOS-FET. So the bus lines of both sections go LOW to the same voltage level.
3. A 5 V device pulls down the bus line to a LOW level. The drain-substrate diode of the MOS-FET the 'lower-voltage' section is pulled down until V_{GS} passes the threshold and the MOS-FET starts to conduct. The bus line of the 'lower-voltage' section is then further pulled down to a LOW level by the 5 V device via the conducting MOS-FET. So the bus lines of both sections go LOW to the same voltage level.

The three states show that the logic levels are transferred in both directions of the bus system, independent of the driving section. State 1 performs the level shift function. States 2 and 3 perform a 'wired-AND' function between the bus lines of both sections as required by the I²C-bus specification.

Supply voltages other than 3.3 V for V_{DD1} and 5 V for V_{DD2} can also be applied, e.g., 2 V for V_{DD1} and 10 V for V_{DD2} is feasible. In normal operation V_{DD2} must be equal to or higher than V_{DD1} (V_{DD2} is allowed to fall below V_{DD1} during switching power on/off).