

Descobrindo o Microcontrolador STM32

Geoffrey Brown
©2012

6 de junho de 2018

This work is covered by the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) license.
<http://creativecommons.org/licenses/by-nc-sa/3.0/>



Este livro foi traduzido pela turma de Engenharia de Automação e Controle do Instituto Federal do Espírito Santo - Campus Serra, como um trabalho para a disciplina Arquitetura de Computadores em 2014/2. (Amanda Fernandes Vilaça Martins, Felippe Gaede, Pedro Freitas, Gabriel Magnago Alves, Henrique Silva Costa, José Smith, Vivian Azeredo Gusella, Guilherme Ferrari)
Correções, sugestões e comentários: envie para serafini@ifes.edu.br ou jiserafini@gmail.com

Sumário

Lista de Exercícios	7
Prefácio	11
1 Iniciando	13
1.1 Hardware Necessário	16
STM32 VL Discovery	18
Asynchronous Serial	19
SPI	20
I2C	21
Time Based	23
Interface Analógica	23
Power Supply	24
Materiais para Prototipagem	25
Equipamento de Testes	25
1.2 Instalação do Software	27
A Tool-chain da GNU	27
A Biblioteca Firmware STM32	28
Template de Código	28
Servidor GDB	29
1.3 Referências Principais	30
2 Introdução ao STM32 F1	33
2.1 O Cortex-M3	36
2.2 STM32 F1	40
3 Programa Esqueleto	49
Programa Demo	50
Scripts Make	52
O Modelo de Memória do STM32 e a sequencia de Boot	54

SUMÁRIO

4 Configuração do STM32	59
4.1 Distribuição de Clock	63
4.2 Pinos de I/O	65
4.3 Funções Alternativas	67
4.4 Remapeamento	68
4.5 Atribuições de Pinos para os Exemplos e Exercícios	68
4.6 Configuração de Periféricos	70
5 Asynchronous Serial Communication	73
5.1 Implementação de Polling no STM32	79
5.2 Inicialização	80
6 SPI	89
6.1 Protocolo	89
6.2 Periféricos SPI do STM32	91
6.3 Testando a Interface SPI	94
6.4 Interface EEPROM	96
7 SPI : Display LCD	103
7.1 Módulo LCD a Cores	103
7.2 Informações sobre Copyright	115
7.3 Comandos de Inicialização (Restantes)	115
8 Cartões de Memória SD	117
8.1 Organização do FatFs	120
8.2 SD Driver	121
8.3 Copyright do FatFs	127
9 I²C – Wii Nunchuk	129
9.1 Protocolo I ² C	130
9.2 Wii Nunchuk	132
9.3 Interface STM32 I ² C	137
10 Temporizadores ou Timers	145
10.1 Saída PWM	148
7735 Backlight	149
10.2 Captura de Entrada (Input)	152
11 Interrupções	157
11.1 Modelo de Exceção do Cortex-M3	161
11.2 Habilitando Interrupções e Definindo suas Prioridade	165

SUMÁRIO

11.3 Configuração do NVIC	166
11.4 Exemplo: Interrupções do Timer	167
11.5 Exemplo: Comunicação Serial com Interrupção	168
Filas de Interrupções Seguras	173
Controle de Fluxo por Hardware	175
11.6 Interrupções Externas	180
12 DMA: Direct Memory Access	187
12.1 Arquitetura DMA do STM32	189
12.2 Suporte DMA para o SPI	191
13 DAC : Conversão Digital Analógica	197
Advertência:	198
13.1 Exemplo: DMA Driven DAC	202
14 ADC : Conversão Analógica Digital	209
14.1 Sobre ADCs de Aproximação Sucessivas	210
15 NewLib	219
15.1 Hello World	220
15.2 Construindo a newlib	226
16 Sistemas Operacionais de Tempo-Real	227
16.1 Threads	229
16.2 Configuração do FreeRTOS	234
16.3 Sincronização	235
16.4 Manipuladores de Interrupção	237
16.5 SPI	240
16.6 FatFS	243
16.7 FreeRTOS API	244
16.8 Discussão	245
17 Próximos Passos	247
17.1 Processadores	248
17.2 Sensores	250
Medições de Posição/Inercial	250
Sensores Ambientais	251
Sensores de Força e Movimento	251
Identificação – Código de Barras/RFID	251
Proximidade	251
17.3 Comunicação	252

SUMÁRIO

17.4 Discussão	252
Atribuições	254
Referências Bibliográficas	255

SUMÁRIO

List of exercises

Exercise 3.1 <i>GDB no STM32</i>	52
Exercise 4.1 <i>Blinking Lights</i>	62
Exercise 4.2 <i>Blinking Lights com Pushbutton</i>	67
Exercise 4.3 <i>Configuração sem a Standard Peripheral Library</i>	70
Exercise 5.1 <i>Testando a Interface USB/UART</i>	75
Exercise 5.2 <i>Hello World!</i>	83
Exercise 5.3 <i>Echo</i>	87
Exercise 6.1 <i>SPI Loopback</i>	95
Exercise 6.2 <i>Write e Teste de um Módulo EEPROM</i>	101
Exercise 7.1 <i>Código Completo da Interface</i>	109
Exercise 7.2 <i>Exibindo Textos</i>	109
Exercise 7.3 <i>Graficos</i>	110
Exercise 8.1 <i>FAT File System</i>	124
Exercise 9.1 <i>Lendo o Wii Nunchuk</i>	136
Exercise 10.1 <i>Ramping LED</i>	149
Exercise 10.2 <i>Controle de um Servo Hobby</i>	149
Exercise 10.3 <i>Sensor Ultrassônico</i>	156
Exercise 11.1 <i>Timer Interrupt – Blinking LED</i>	167
Exercise 11.2 <i>Comunicação Serial dirigida a Interrupções</i>	179
Exercise 11.3 <i>Interrupção Externa</i>	182
Exercise 12.1 <i>Módulo SPI DMA</i>	193
Exercise 12.2 <i>Mostrando imagens BMP a partir do Fat File System</i>	193
Exercise 13.1 <i>Gerador de Formas de Onda</i>	199
Exercise 13.2 <i>Application Software Driven Conversion</i>	199
Exercise 13.3 <i>Conversão Interrupt Driven</i>	200
Exercise 13.4 <i>Audio Player</i>	204
Exercise 14.1 <i>Amostragem Continua</i>	213
Exercise 14.2 <i>Conversão Dirigida pelo Timer</i>	216

SUMÁRIO

Exercise 14.3 <i>Gravador de Voz</i>	217
Exercise 15.1 <i>Hello World</i>	223
Exercise 16.1 <i>RTOS – Blinking Lights</i>	235
Exercise 16.2 <i>Multiplos Threads</i>	237
Exercise 16.3 <i>Multithreaded Queues</i>	239
Exercise 16.4 <i>Multithreaded SPI</i>	243
Exercise 16.5 <i>Multithreaded FatFS</i>	244

Agradecimentos

Eu tive muita ajuda de várias pessoas da Escola de Informática da Universidade de Indiana no desenvolvimento deste material. Principalmente Caled Hess que desenvolveu a protoboard que usamos em nosso laboratório, e ele, junto com Bryce Himebaugh fizeram contribuições significativas no desenvolvimento de várias experiências. Tracey Theriault forneceu muitas das fotografias.

Estou agradecido a ST Microelectronics por várias doações que possibilitaram o desenvolvimento deste laboratório. Eu quero agradecer particularmente a Andrew Dostie que sempre respondeu rapidamente a qualquer pergunta que lhe fiz.

STM32 F1, STM32 F2, STM32F3, STM32 F4, STM32 L1, Discovery Kit, Cortex, ARM e outras são marcas registradas e são propriedades de seus donos.

Prefácio

Este livro pretende ser um manual hands-on para aprender como projetar sistemas utilizando a família de microcontroladores STM32 F1. Ele foi escrito para suportar um curso de ciência da computação para iniciantes na Universidade de Indiana. O foco deste livro está no desenvolvimento de código para utilizar os vários periféricos disponíveis no microcontrolador STM32 F1 e em particular na placa STM32VL Discovery. Como existem outras boas fontes de informação sobre o Cortex-M3, que é o core processor para os microcontroladores STM32 F1, não iremos examinar este core em detalhes; uma excelente referência é “The Definitive Guide to the ARM CORTEX-M3.” [5]

Este livro não é exaustivo, mas fornece uma “trilha” simples para o aprendizado da programação do microcontrolador STM32 por meio de uma série de exercícios de laboratórios. Uma decisão chave foi utilizar um modelo de hardware off-the-shelf para todos os experimentos apresentados.

Eu ficaria feliz em tornar disponível para qualquer instrutor os outros materiais desenvolvidos para o curso C335 (Computer Structures) da Universidade de Indiana, mas as restrições de copyright limitam minha capacidade de torná-los amplamente disponível.

Geoffrey Brown
Indiana University

Capítulo 1

Iniciando

Nos últimos anos temos visto um renascimento de entusiastas e inventores construindo dispositivos eletrônicos personalizados. Estes sistemas utilizam componentes e módulos comerciais (off-the-shelf) cujo desenvolvimento foi abastecido por uma explosão tecnológica de sensores e atuadores integrados que incorporam muita eletrônica analógica que anteriormente aparentava ser uma barreira para desenvolvimentos de sistemas por não engenheiros. Micro-controladores com firmwares personalizados oferecem a cola para vincular sofisticados módulos comerciais em sistemas complexos customizados. Este livro apresenta uma série de tutoriais objetivando o ensino de programação de sistemas embarcados e interfaces de hardware necessárias para a utilização da família de micro-controladores STM32 no desenvolvimento de dispositivos eletrônicos. O livro é direcionado para leitores com experiência em programação na linguagem “C”, mas sem experiência com sistemas embarcados.

A família de micro-controladores STM32, baseado no processador ARM Cortex-M3, oferece uma estrutura para criação de uma vasta quantidade de sistemas embarcados, desde simples dongles alimentados a bateria até sistemas complexos de tempo real como helicópteros auto-pilotados. Esta família de componentes inclui dezenas configurações diferentes, provendo amplas opções de tamanho de memória, periféricos, performance e consumo de energia. Os componentes são razoavelmente baratos em pequenas quantidades -- poucos dólares pelos dispositivos mais complexos -- e justificam seu uso em aplicações de baixo volume de produção. Na verdade, os componentes iniciais da “Value Line” são comparáveis em valores aos componentes da ATmega, usados nas placas de desenvolvimentos Arduíno, e que ainda apresentam melhorias significativas de desempenho e periféricos mais potentes. Além disso, os periféricos utilizados são compartilhados por outros membros da família (por exemplo,

CAPÍTULO 1. INICIANDO

os módulos USART são compatíveis com todos componentes da STM32 F1) e são suportados por um único firmware. Portanto, aprender como programar um único dispositivo da família STM32 F1 o capacita a programar todos eles.
¹

Infelizmente, poder e flexibilidade são obtidos com um custo — o desenvolvimento de softwares para a família STM32 pode ser extremamente desafiador para um iniciante com uma vasta documentação e bibliotecas para vasculhar. Por exemplo, o RM0041, manual de referência para uma grande quantidade de dispositivos STM32 F1, possui 675 páginas e sequer cobre o núcleo de processamento do Cortex-M3! Felizmente, não é necessário ler todo este livro para começar a desenvolver aplicações para o STM32, embora seja uma referência importante. Além disto, um iniciante se depara com um conjunto de opções de ferramentas de programação (*tool-chain*)² Em contraste, a plataforma Arduino oferece bibliotecas para aplicações simples e ambientes de programação acessíveis a programadores inexperientes. Para muitos sistemas simples, oferece um rápido atalho para construir protótipos. No entanto, simplicidade traz custos — as plataformas de software do Arduino não se adaptam bem ao gerenciamento de atividades concorrentes em sistemas complexos de tempo real e, para interação com softwares externos, depende de bibliotecas projetadas fora do ambiente de programação do Arduíno, utilizando técnicas e ferramentas semelhantes às utilizadas para o STM32. Além do mais, a plataforma Arduíno não prove capacidade de depuração que limita severamente o desenvolvimento de sistemas mais complexos. Novamente, a depuração requer a saída do ambiente confinado da plataforma Arduíno. Finalmente, o ambiente Arduíno não suporta sistemas operacionais de tempo-real (RTOS), essenciais para a construção de sistemas embarcados mais complexos.

Para leitores com experiência em programação com linguagem “C”, a família STM32 é uma plataforma muito superior que a Arduíno em sistemas micro-controlados se as dificuldades iniciais puderem ser reduzidas. O objetivo deste livro é ajudar aprendizes em sistemas embarcados a dar um salto inicial com a programação para a família STM32. Eu assumo que tenha competências básicas na linguagem C em ambiente LINUX — leitores sem experiência são melhores servidos com a plataforma Arduíno. Parto do princípio também que

¹Existem atualmente cinco famílias de MCUs STM32 – STM32 F0, STM32 F1, STM32 L1, STM32 F2, e STM32 F4 suportadas por diferentes, mas estruturalmente semelhantes, biblioteca de firmware. Embora estas famílias compartilhem vários periféricos, algum cuidado deve ser tomado quando for movimentar os projetos entre estas famílias. [18, 17, 16]

²Uma tool-chain inclui um compilador, assembler, linker, debugger e varias ferramentas para processamento de arquivos binários.

esteja familiarizado com editores de texto, e experiência em escrever, compilar e depurar códigos em C. Não assumo que tenha familiaridade significativa com o hardware — as pequenas quantidades de “ligações” requeridas neste livro podem ser facilmente realizadas por um iniciante.

Os projetos que descrevo neste livro utilizam uma pequena quantidade de módulos disponíveis comercialmente e de baixo custo. Entre eles, incluem o poderoso STM32 VL Discovery Board (uma placa de US\$10 que inclui um processador STM F100 e um hardware para depuração), um pequeno display LCD, uma porta USB/UART, um Wii Nunchuk, um alto-falante e um microfone. Com estes poucos componentes podemos explorar as três mais importantes interfaces — serial, SPI e I2C — interfaces de entrada e saída analógica, e desenvolvimento de firmware utilizando interrupções e DMA. Todos os componentes requeridos para construção estão disponíveis por fornecedores locais ou vendedores do Ebay. Escolhi não utilizar um único “evaluation board” abrangente, normalmente utilizados em tutoriais, porque desejo que os leitores desde livro vejam que poucos componentes aliados a técnicas de softwares provêm os conceitos necessários para adaptar vários outros componentes comerciais. Ao longo do livro irei sugerir outros módulos e descreverei como adaptar as técnicas introduzidas neste livro para sua utilização.

O software de desenvolvimento usado neste livro é todo open-source. Nosso recurso principal é a tool-chain de desenvolvimento de software GNU que inclui o gcc, gas, objcopy, objdump e o depurador gdb. Não utilizo uma IDE como o eclipse. Acredito que a maioria das IDEs possuem um custo inicial alto embora possam agilizar o processo de desenvolvimento para sistemas de grande porte. IDEs também ocultam o processo de compilação de uma maneira a dificultar a determinação do que realmente está acontecendo, quando meu objetivo aqui é detalhar o processo de desenvolvimento. Enquanto o leitor é livre para usar uma IDE, eu não ofereço orientações para a utilização de uma. Não se pode assumir que open-source significa baixa qualidade — muitas aplicações comerciais para sistemas embarcados utilizam o software GNU e uma significante parcela de desenvolvimento de software comercial é feita com software da GNU. Finalmente, virtualmente todos os processadores embarcados são suportados pela tool-chain de software GNU. Aprender como usar esta ferramenta em um processador literalmente abre portas para o desenvolvimento de sistemas embutidos.

O desenvolvimento de Firmwares diferem significativamente de desenvolvimento de aplicações porque geralmente é excessivamente difícil de determinar o que realmente está acontecendo no código que interage com hardwares periféricos apenas examinando o estado do programa. Além disto, em muitas

CAPÍTULO 1. INICIANDO

situações é impraticável a parada do programa (exemplo: com um depurador) porque poderia invalidar o comportamento em tempo-real. Por exemplo, no desenvolvimento de código para interface com um Wii Nunchuk (um dos projetos descritos neste livro), eu tive dificuldades em rastrear um bug (erro) de temporização relacionado ao quanto rápido os dados estavam sendo “clockeados” pela interface de hardware. Nenhum depurador de software poderia isolar este problema — tinha que encontrar uma maneira de verificar o comportamento do hardware. Similarmente, ao desenvolver código para prover controle de fluxo para uma interface serial, fiz suposições erradas sobre a ponte USB/U-AUT especifica que estava comunicando. Somente ao observar a interface de hardware é que encontrei este problema.

Neste livro, introduzo um processo de desenvolvimento de firmware que combina métodos tradicionais de depuração (com o GDB), com o uso de um “analisador lógico” de baixo custo que permite capturar o comportamento em tempo real das interfaces de hardware.

1.1 Hardware Necessário

Uma lista do hardware necessário para os tutoriais deste livro é apresentada na Figura 1.1. A lista de componentes está organizada por categorias correspondendo às várias interfaces cobertas por este livro seguido pelos materiais de teste e prototipagem necessários. No restante desta seção, descrevemos cada componente e, quando existente, as características chave que precisam serem satisfeitas. Alguns destes componentes necessitam de soldagem de pinos. Esta é uma tarefa simples que pode ser completada com um ferro de solda simples e barato. A quantidade de solda necessária é mínima e recomendo pegar emprestado o equipamento necessário se possível. Existem muitos tutoriais de solda na web.

O componente de maior custo é um analisador lógico. Utilizo o “Saleae Logic”, que para alguns iniciantes pode ser caro (US\$150).³ Uma alternativa, o OpenBench Logic Sniffer, é consideravelmente mais barato (US\$50) e provavelmente adequado. Minha escolha foi ditada pelas necessidades de um laboratório de ensino, cujos equipamentos sofrem diferentes danos — as partes eletrônicas expostas e pinos do Logic Sniffer são muito vulneráveis neste ambiente. Um osciloscópio pode ajudar nas interfaces de áudio, mas está longe de ser essencial.

³ Atualmente a Saleae oferece um desconto para alunos e professores.

1.1. HARDWARE NECESSÁRIO

Componente	Fornecedor	Custo
Processador		
STM32 VL discovery	Mouser, Digikey, Future Electronics	\$10
Asynchronous Serial		
USB/UART breakout	Sparkfun, Pololu, ebay	\$7-\$15
SPI		
EEPROM (25LC160)	Digikey, Mouser, others	\$0.75
LCD (ST7735)	ebay and adafruit	\$16-\$25
Micro SD card (1-2G)	Various	\$5
I2C		
Wii Nunchuk	ebay (clones), Amazon	\$6-\$12
Nunchuk Adaptor	Sparkfun, Adafruit	\$3
Time Based		
Hobby Servo (HS-55 micro)	ebay	\$5
Ultrasonic range finder (HC-SR04)	ebay	\$4
Analog		
Potenciometro	Digikey, Mouser, ebay	\$1
Audio amplifier	Sparkfun (TPA2005D1)	\$8
Speaker	Sparkfun COM-10722	\$1
Microphone Module	Sparkfun (BOB-09868 or BOB-09964)	\$8-\$10
Power Supply (optional)		
Step Down Regulator (2110)	Pololu	\$15
9V Battery Holder		
9V Battery		
Prototyping Materials		
Solderless 700 point breadboard (2)	ebay	\$6
Jumper wires	ebay	\$5-\$10
Equipamento de Tests		
Saleae Logic ou	Saleae	\$150
OpenBench Logic Sniffer	Seeed studio	\$50
Osciloscópio	opcional para testes de saída analógica	

Figura 1.1: Required Prototype Hardware and Suppliers

CAPÍTULO 1. INICIANDO

STM32 VL Discovery

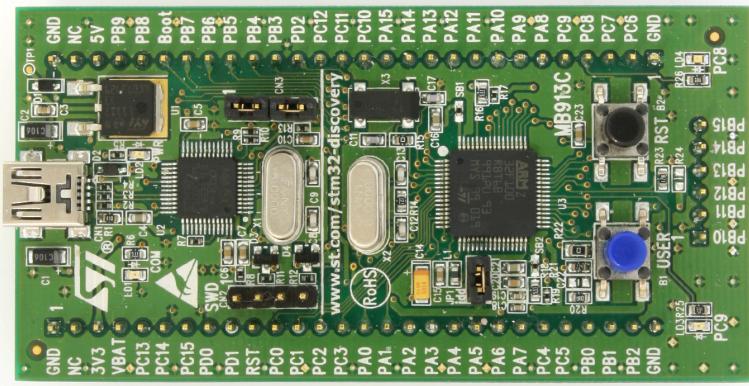


Figura 1.2: STM32 VL Discovery Board

O componente essencial usado nos tutoriais é o STM32 VL Discovery produzido pela STMicroelectronics (ST) e disponível em muitos distribuidores de eletrônicos por aproximadamente US\$10.⁴ Esta placa, mostrada na Figura 1.2 inclui um microcontrolador configurável STM32 F100 com 128KB de flash e 8KB de RAM, assim como um hardware de interface de depuração integrado, baseada em um STM32 F103 dedicado conectado via USB. Com software adequado rodando no PC (host) é possível conectar o processador STM32 F100 para baixar, executar e depurar o código do usuário. Além disto, a interface depuradora integrada é acessível através de pinos e pode ser usada para depurar qualquer membro da família STM32 — efetivamente, ST está dando de graça uma interface de hardware de depuração (debugger) com a placa de prototipação. O STM32 VL Discovery é distribuído com uma documentação completa incluindo esquemáticos. [14].

Na fotografia, há uma linha branca vertical um pouco para a esquerda da metade do CI. Na parte à direita da linha está o STM32 F100, cristais osciladores, dois LED's acessíveis ao usuário, um botão também acessível e outro botão de reset. À esquerda está presente a interface depuradora de hardware, incluindo um STM32 F103, regulador de tensão e outros componentes. O regulador converte 5V fornecido pela porta USB para 3,3V para o processador e outras conexões da placa. Este regulador é capaz de fornecer corrente suficiente para os hardwares adicionais usados nos tutoriais.

⁴<http://www.st.com/internet/evalboard/product/250863.jsp>

1.1. HARDWARE NECESSÁRIO

Todos os pinos do STM32 F100 trazem consigo rótulos — como iremos ver, os rótulos dos pinos correspondem diretamente aos nomes lógicos utilizados na documentação do STM32 em vez de associar os pinos físicos com códigos particulares utilizados. Este uso de nomes lógicos é consistente através da família e simplifica a tarefa de definir softwares portáteis.

O STM32 F100 é um membro da value line de processadores STM32 e a velocidade de execução é relativamente lenta (para processadores Cortex-M3) em 24Mhz, mesmo assim provê mais computação e portas I/O do que requerido pelos tutoriais utilizados neste livro. Além disto, todos os periféricos providos pelo STM32 F100 são compatíveis a outros membros da família STM32, e o código desenvolvido desde componente é completamente portável entre os micro controladores da família.

Asynchronous Serial

Uma das técnicas mais utilizadas para depuração de software é exibir mensagens no terminal. O micro-controlador STM32 provê capacidade necessária para comunicação serial através de dispositivos USART (Universal Synchronous Asynchronous Receiver Transmitter), mas não a conexão física necessária para comunicar com um PC (host). Nos tutoriais utilizamos uma bridge USB/UART comum. A mais comum dessas é conhecida como substitutos para a porta serial dos PCs e são impróprios para nosso propósito porque incluem conversores de nível de tensão para satisfazer a especificação RS-232. Em vez disto, necessitamos um dispositivo que provê acesso direto aos pinos dos dispositivos da bridge USB/UART.



Figura 1.3: Pololu CP2102 Breakout Board

Um exemplo de dispositivo, mostrado na Figura 1.3, é a placa breakout Pololu CP2102. Uma alternativa é a placa breakout Sparkfun FT232RL (BOB-00718) que utiliza o chip bridge FTDI FT232RL. Comprei uma placa CP2102 no Ebay a preço baixo e que funciona bem. Enquanto uma placa com portas para dispositivos parece ser viável, é importante notar que nem todas



Figura 1.4: EEPROM em package PDIP

as placas são utilizáveis. As placas CP2102 mais comuns, que possuem seis pinos, não provem acesso aos pinos de controle de fluxo do hardware, o que é essencial para confiabilidade em conexões de alta velocidade. Um tutorial importante neste livro cobre a implementação de uma interface serial de alta velocidade confiável. Você deve procurar entre os pinos de qualquer placa deste tipo para garantir que pelo menos os seguintes sinais estão disponíveis – `rx`, `tx`, `rts` e `cts`.

Interfaces seriais assíncronas são utilizadas normalmente em muitos módulos receptores GPS (Global Positioning System), modems de GSM de celular e interfaces wireless bluetooth.

SPI

A mais simples das interfaces seriais síncronas que examinaremos neste livro é a SPI. Os módulos principais que consideramos são um display LCD colorido e um cartão de memória SD. Como estes representam uma relativa complexidade da utilização da interface SPI, então inicialmente discutiremos um dispositivo mais simples — uma EEPROM serial (Electrically Erasable Programmable Memory). Muitos sistemas embarcados a utilizam para armazenamento permanente e é relativamente simples de desenvolver o código necessário para acessá-la.

Existem muitas EEPROMs disponíveis, embora com diferentes interfaces. Recomendo começar com o Microchip 25LC160 com um package PDIP (Figura 1.4). Outros packages podem ser desafiadores de serem usados em um ambiente básico de protótipo. EEPROMs com diferentes densidades de armazenamento frequentemente requerem diferentes protocolos de comunicação.

O segundo dispositivo SPI que consideramos é um display — utilizamos um módulo TFT colorido barato que inclui uma entrada para cartão micro SD. Enquanto utilizei aquele mostrado na Figura 1.1, um módulo equivalente

1.1. HARDWARE NECESSÁRIO

está disponível pela Adafruit. A restrição mais importante é que os exemplos no livro assumem que o display é um ST7735 com uma interface SPI. Nós utilizamos o adaptador SD card, embora seja possível encontrar adaptadores alternativos da Sparkfun ou outros fornecedores.

O display tem 128x160 pixels colorido similar aos de dispositivos como ipods ou câmeras digitais. As cores são brilhantes e podem facilmente exibir imagens com boa fidelidade. Uma limitação significativa dos displays baseados em SPI é a largura da banda de passagem de comunicação — Para gráficos de alta velocidade é recomendado utilizar um display com interface paralela. Embora os componentes da linha value line das Discovery Board não possuem um periférico interno para suportar interfaces paralelas, muitos outros componentes STM32 possuem.

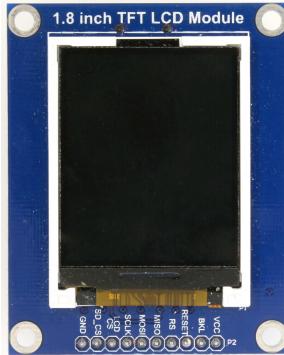


Figura 1.5: Color Display Module

Finalmente você necessitará um cartão de memória SD de 1G-2G juntamente com um adaptador para programar o cartão em um computador desktop. A velocidade e marca não são parâmetros críticos. O módulo TFT recomendado inclui um slot de cartão SD.

I2C

A segunda interface serial síncrona que estudaremos é o I2C. Para ilustrar a utilização do I2C utilizaremos um Wii Nunchuk (Figura 1.6). Ele foi desenvolvido para consoles de vídeo Wii, mas foi reutilizado por hobbystas. Ele contém um acelerômetro ST LIS2L02AL de 3-eixos, um joy-stick analógico de 2-eixos, e dois botões que podem ser utilizados (polled) via I2C. Eles estão disponíveis tanto no modelo genuíno quanto nos clones. Devo dizer que existem algumas diferenças sutis entre vários clones que podem impactar no

CAPÍTULO 1. INICIANDO



Figura 1.6: Wii Nunchuk

desenvolvimento de software. O problema especificamente é a diferença nas sequências de inicialização e na codificação de dados.



Figura 1.7: Wii Nunchuk Adaptor

O conector do Nunchuk é proprietário do Wii e não encontrei um fornecedor para um conector semelhante. Existem placas adaptadoras simples disponíveis que funcionam bem para os objetivos deste tutorial. Eles estão dis-

1.1. HARDWARE NECESSÁRIO

poníveis em vários fornecedores; a versão da Sparkfun é mostrada na Figura 1.7.

Time Based

Temporizadores são componentes chaves da maioria dos micro-controladores. Além de serem utilizados para medir a passagem do tempo — por exemplo, criar um alerta em intervalos regulares — temporizadores são usados para gerar e decodificar trens de pulsos complexos. Um uso comum é na geração do sinal PWM para controle de velocidade de motor. Os temporizadores STM32 são bastante sofisticados e fornecem medições e geração de tempo complexas. Demonstraremos como os temporizadores podem ser usados para definir a posição de um servo motor comum (Figura 1.8) e como medir o “tempo de voo” de um sensor ultrassônico (Figura 1.9). O sensor ultrassônico usado é conhecido genericamente como HC-SR04 e está disponível por muitos fornecedores — comprei um de um vendedor do Ebay. Virtualmente qualquer servo motor funcionará, entretanto, devido a limitações do USB, é desejável que utilize um “micro” servo para os experimentos descritos neste livro.

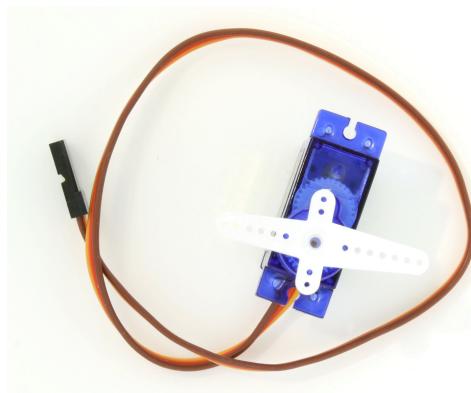


Figura 1.8: Servo

Interface Analógica

A última interface a ser considerada é a analógica — tanto de entrada (analógica para digital) como de saída (digital para analógica). Um conversor Digital-Analógico (DAC) converte um valor digital em tensão. Para ilustrar esta capacidade, usamos um DAC para controlar um pequeno alto-falante através de um amplificador (Figura 1.11). Este experimento, lendo arquivos

CAPÍTULO 1. INICIANDO

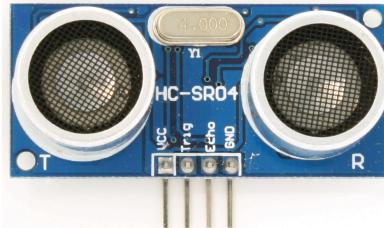


Figura 1.9: Ultrasonic Sensor

de áudio de um cartão de memória SD e tocando-os em um alto-falante, requer o uso de múltiplas interfaces bem como timers e DMA.

Para ilustrar o uso de conversão Analógica-Digital, utilizaremos um pequeno potenciômetro (Figura 1.10) para fornecer uma tensão de entrada variável e um microfone (Figura 1.12) para fornecer um sinal analógico.



Figura 1.10: Potenciômetro comum



Figura 1.11: Auto-falante e Amplificador

Power Supply

Em nosso laboratório utilizamos uma fonte USB para a maioria dos experimentos. Entretanto, se necessário construir uma fonte por bateria, então

1.1. HARDWARE NECESSÁRIO

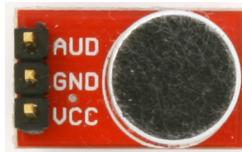


Figura 1.12: Microfone

será necessário o uso de um regulador de tensão (conversor) entre a bateria desejada e 5V. O STM32 VL Discovery possui um regulador linear para converter 5V para 3.3V. Utilizei um conversor Buck simples -- A Figura 1.13 mostra um disponível da Pololu -- para converter uma bateria de 9V para 5V. De posse do conversor e da bateria, todos os experimentos descritos neste livro podem ser feitos de modo portátil.

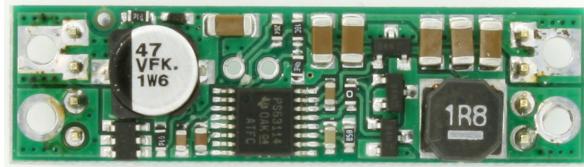


Figura 1.13: Fonte de Alimentação

Materiais para Prototipagem

Need pictures

De modo a prover uma plataforma para conexão de vários componentes, recomendo comprar duas placas breadboards de 700-pontos juntamente com um número de jumpers tanto na configuração macho-macho quanto fêmea-fêmea. Todos estes estão disponíveis no Ebay com preços altamente competitivos.

Equipamento de Testes

O Analisador Lógico Saleae é mostrado na Figura 1.14. Este dispositivo fornece um analisador lógico de 8 canais capaz de capturar dados digitais a 10-20MHz, o que é suficientemente rápido para depurar os protocolos seriais básicos utilizados pelos tutoriais. Enquanto o hardware é muito simples -- até um pouco primitivo -- o software provido é muito sofisticado. Mais importante, ele tem a capacidade de analisar vários protocolos de comunicação e

CAPÍTULO 1. INICIANDO

mostrar os dados resultantes de maneira significativa. A Figura 1.15 demonstra a saída de dados seriais -- neste caso “hello world” (Você talvez precise dar zoom no PDF para ver os detalhes).



Figura 1.14: Saleae Logic

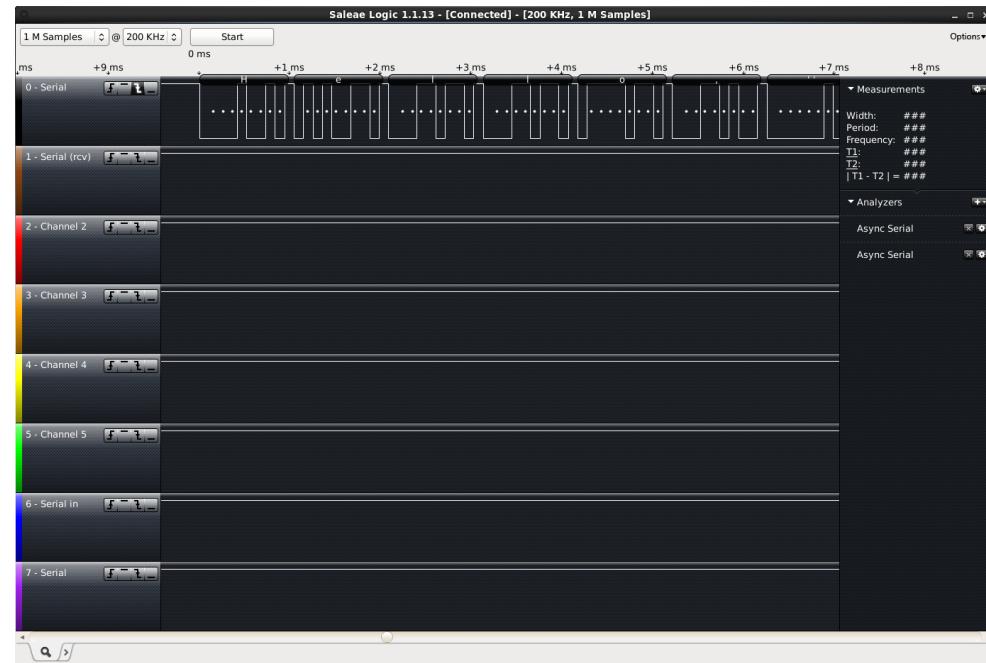


Figura 1.15: Saleae Logic Software

1.2. INSTALAÇÃO DO SOFTWARE

Quando desenvolvemos softwares em ambientes embarcados, o cenário mais comum quando testamos uma nova interface de hardware é ... não acontecer nada. A não ser que as coisas funcionem perfeitamente, é difícil saber onde começar a procurar por problemas. Com um analisador lógico, podemos capturar e visualizar qualquer dado que está sendo transmitido. Por exemplo, trabalhando com um software para acionar uma porta serial , é possível determinar se alguma coisa está sendo transmitida, e se for o caso, o quê. Isto é especialmente importante quando o processador embarcado está comunicando com um dispositivo externo (p. ex.: um Wii Nunchuk) -- onde cada comando requer a transmissão e recepção de uma sequência binária específica. Um analisador lógico fornece os meios para a observação dos eventos comunicados em tempo real (se algum !).

1.2 Instalação do Software

O processo de desenvolvimento de software descrito neste livro utiliza as bibliotecas de firmware distribuídas pela STMicroelectronics, que fornecem acesso de baixo nível a todos os periféricos da família STM32. Enquanto estas bibliotecas são relativamente complicadas, este livro fornece um “caminho das pedras” para seu uso, bem como alguns atalhos iniciais. As vantagens de utilizar estas bibliotecas de firmware são que elas abstraem muito dos detalhes em nível de bit necessários para programar o STM32, são relativamente maduras e testadas exaustivamente, e possibilitam o desenvolvimento de aplicações portáveis entre os membros da família STM32. Em contraste, examinamos um código base distribuído com o processador NXP LPC13XX Cortex-M3 e descobrimos que está incompleto e relativamente em estado imaturo.

A Tool-chain da GNU

O desenvolvimento de software para este livro foi executado usando ferramentas de desenvolvimento para sistemas embarcados GNU, incluindo gcc, gas, gdb e gld. Nós utilizamos com sucesso duas distribuições diferentes dessas ferramentas. No ambiente Linux usamos a Sourcery (uma subsidiária da Mentor Graphics) CodeBench Lite Edition for ARM (EABI). Ela pode ser obtida através do endereço: <https://sourcery.mentor.com/sgpp/lite/arm/portal/subscription?@template=lite>. Recomendo o uso do instalador GNU/Linux. O site inclui documentação em PDF para a tool-chain GNU juntamente com um documento “getting started” que mostra instruções detalhadas de instalação.

CAPÍTULO 1. INICIANDO

Adicionar a linha de comandos Linux no arquivo de inicialização do bash irá tornar a utilização mais fácil

```
export PATH=path-to/codesourcery/bin:$PATH
```

Nos sistemas OS X (Macs) utilizamos a distribuição yagarto (www.yagarto.de) da toolchain da GNU. Existe um instalador simples disponível para download.

A Biblioteca Firmware STM32

Os componentes STM32 são suportados pela ST Standard Peripheral Library⁵ que fornece o firmware para suportar todos os periféricos dos vários STM32. Esta biblioteca, embora de fácil instalação, tem o seu uso desafiador. Existem muitos módulos separados (um para cada periférico), bem como um grande número de funções e definições para cada módulo. Além disto, compilar usando estes módulos requer sinalizações apropriadas aos compiladores, assim como alguns arquivos externos (um arquivo de configuração, e um pequena quantidade de código). A abordagem utilizada nesta documentação é fornecer um ambiente de construção básico (makefiles, arquivos de configuração, etc.) que podem facilmente serem estendidos ao explorar os vários periféricos. Ao invés de tentar descrever completamente a biblioteca de periféricos, apresento os módulos a medida em que são necessários e outras funções/definições necessárias.

Template de Código

Enquanto o firmware fornecido pela STMicroelectronics fornece uma fundação sólida de desenvolvimento de software da família STM32, ele pode ser difícil de iniciar o desenvolvimento. Infelizmente, os exemplos distribuídos na placa STM32 VL Discovery estão extremamente entrelaçados com as IDEs comerciais baseada em Windows para o desenvolvimento de código para o STM32, e é desafiador extrair e usar em um ambiente Linux. Criei um pequeno template de exemplo que utiliza makefiles padrão para Linux e em que todos os aspectos de criação do processo são expostos ao usuário.

Este template pode ser baixado em:

```
git clone git://github.com/geoffreymbrown/STM32-Template.git
```

⁵<http://www.st.com/web/en/catalog/tools/PF257890>

1.2. INSTALAÇÃO DO SOFTWARE

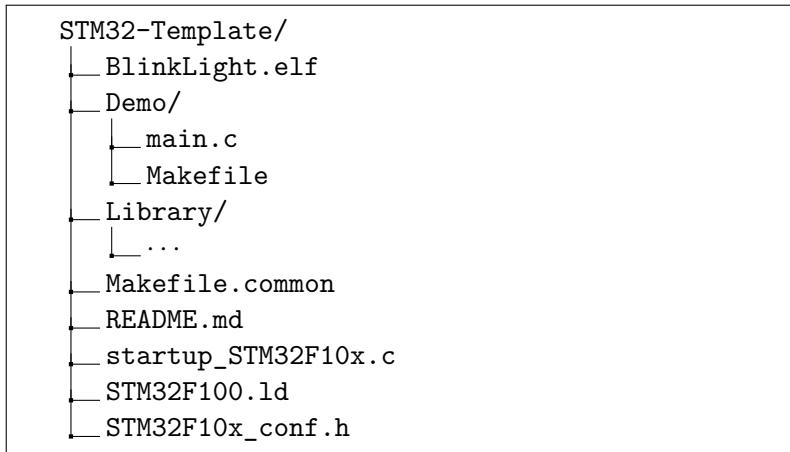


Figura 1.16: Template STM32VL

O diretório do template (mostrado na Figura 1.16) consiste na parte específica de código inicial, de uma parte específica de linker script, um makefile, e um arquivo header requerido pela biblioteca padrão do periférico. Um subdiretório contém o código e exemplos específicos do makefile. O diretório inclui um arquivo binário (executável) para o STM32 VL Discovery. O programa demo é discutido mais adiante no Capítulo 3.

Servidor GDB

Para baixar e debugar o código na placa STM32 VL Discovery, podemos explorar a interface USB de depuração stlink, que comunica com o módulo depurador do STM32. A interface stlink pode ser usada tanto pelo processador da placa Discovery, e selecionando os pinos corretamente para processadores off-board. A ST também vende uma versão stand-alone desta interface depuradora. Infelizmente, a interface stlink é suportada apenas em Windows e a ST não publicou ainda uma especificação da interface. É amplamente conhecido que a interface stlink é implementada usando um dispositivo da classe USB Mass Storage e sabe-se ainda que esta aplicação é incompatível com o OS X e com os drivers do kernel do Linux. Ainda assim, a interface sofreu uma suficiente engenharia reversa que um servidor gdb executando em um Linux ou OS X está disponível para download:

```
git clone git://github.com/texane/stlink.git
```

CAPÍTULO 1. INICIANDO

O arquivo README descreve o processo de instalação. A placa STM32 VL Discovery utiliza o protocolo STLINKv1 que é um pouco problemático devido à maneira com que interage com o Kernel OS. Devido aos problemas do Kernel, é importante seguir as instruções fornecidas. No caso do OS X, há também um “mac os x driver” que deve ser construída e instalada.

Para executar o servidor gdb, conecte uma placa STM32 VL Discovery. Verifique se o “/dev/stlink” existe e então execute:

```
st-util -1
```

Observação: Versões anteriores do st-util necessitam uma sequência inicial diferente

```
st-util 4242 /dev/stlink
```

Para fazer o download do exemplo pisca-led, inicie uma instancia de arm-none-eabi-gdb em uma janela separada e execute os seguinte comandos

```
arm-none-eabi-gdb BlinkingLights.elf
(gdb) target extended-remote :4242
(gdb) load
(gdb) continue
```

Isto fará o download do programa para a memoria flash e iniciar a execução.

GDB pode também ser usado para configurar breakpoints e watchpoints.

1.3 Referências Principais

Existe um extenso numero de documentos pertencentes à família STM32 da Cortex-M3 MCUs. A lista à seguir inclui os documentos chaves referenciados neste livro. A maioria deles está disponível online em www.st.com. Os documentos técnicos de referencia do Cortex-M3 estão disponíveis em www.arm.com.

1.3. REFERÊNCIAS PRINCIPAIS

RM0041 Reference manual for STM32F100x Advanced ARM-based 32-bit MCUs [20]. Este documento fornece informações de referência de todos os periféricos usados nos processadores STM32 incluindo o processador usado na placa STM32 VL Discovery.

PM0056 STM32F10xx/20xx/21xx/L1xxx [19]. Referencia da ST para programar o processador Cortex-M3. Inclui modelo de execução e instruções, e códigos para periféricos (p. ex. O controlador de interrupções).

Cortex-M3 ARM Cortex-M3 (revision r1p1) Technical Reference Manual. Fonte definitiva para informações pertinentes para o Cortex-M3. [1].

Data Sheet Low & Medium-density Value Line STM32 data sheet [15]. Fornece informações de pinos – especialmente o mapeamento entre GPIO e funções alternativas. Existem data sheets para um grande número de MCUs da família STM32 – este se aplica somente ao MCU da STM32 VL Discovery.

UM0919 User Manual STM32 Value Line Discovery [14]. Fornece informações detalhadas, incluindo diagramas de circuito, para a STM32 VL Discovery.

Capítulo 2

Introdução ao STM32 F1

Os micro-controladores STM32 F1xx são baseados no núcleo ARM Cortex-M3. O Cortex-M3 é também a base para micro-controladores de vários outros fabricantes, incluindo TI, NXP, Toshiba e Atmel. Compartilhar um núcleo comum significa que as ferramentas de desenvolvimento de software, incluindo o compilador e depurador são comuns através de uma vasta gama de micro-controladores. O Cortex-M3 difere de gerações anteriores de processadores ARM pela definição de um número de periféricos-chave como parte da arquitetura de núcleo incluindo controlador de interrupção, temporizador do sistema, e hardware de depuração e trace (incluindo as interfaces externas). Este nível de integração adicional significa que os softwares do sistema, tais como sistemas operacionais em tempo real e ferramentas de desenvolvimento de hardware tal como as interfaces de depuração podem ser compartilhadas por toda a família de processadores. As várias famílias de micro-controladores baseado no Cortex-M3 diferem significativamente em termos de periféricos de hardware e memória – os periféricos da família STM32 são completamente diferentes arquitetonicamente da família de periféricos da NXP, mesmo quando eles têm uma funcionalidade semelhante. Neste capítulo apresentamos os principais aspectos do núcleo Cortex-M3 e dos micro-controladores STM32 F1xx.

Um diagrama de blocos do processador de STM32F100 utilizado na placa value line discovery board é mostrada na Figura 2.1. A CPU Cortex-M3 é mostrada no canto superior esquerdo. Os componentes da value line têm uma frequência máxima de 24 MHz – outros processadores STM32 pode suportar um clock de até 72 MHz. A maior parte da figura mostra os periféricos e sua interligação. O processador Discovery tem 8K bytes de SRAM e 128K bytes de flash. Há dois barramentos de comunicação com periféricos – APB2 e APB1 que suportam uma ampla variedade de periféricos.

CAPÍTULO 2. INTRODUÇÃO AO STM32 F1

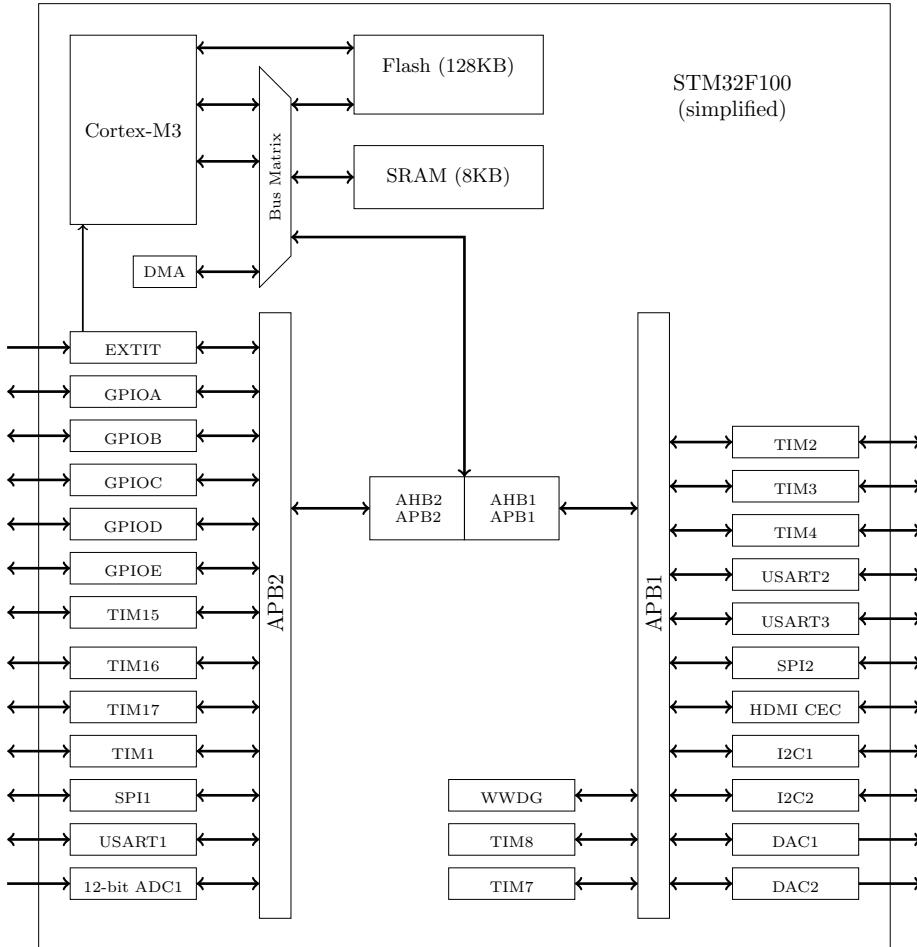


Figura 2.1: Arquitetura do STM32 F100

A arquitetura do núcleo do Cortex-M3 consiste de um processador de 32 bits (CM3) com um pequeno conjunto de periféricos principais – uma versão simplificada deste núcleo é mostrada na Figura 2.2. O núcleo CM3 tem uma arquitetura Harvard significando que usa interfaces separadas para buscar instruções (Inst) e (Data). Isso ajuda a garantir que o processador não tenha um gargalo de acesso a memória, uma vez que permite o acesso a dados e memórias de instruções simultaneamente. Do ponto de vista do CM3, tudo parece memória — ele somente diferencia entre a busca de instruções e acesso

a dados. A interface entre o Cortex-M3 e um hardware específico de um fabricante é feita através de três barramentos de memória – ICode, Dcode, e de Sistema - que são definidos para acessar diferentes regiões da memória.

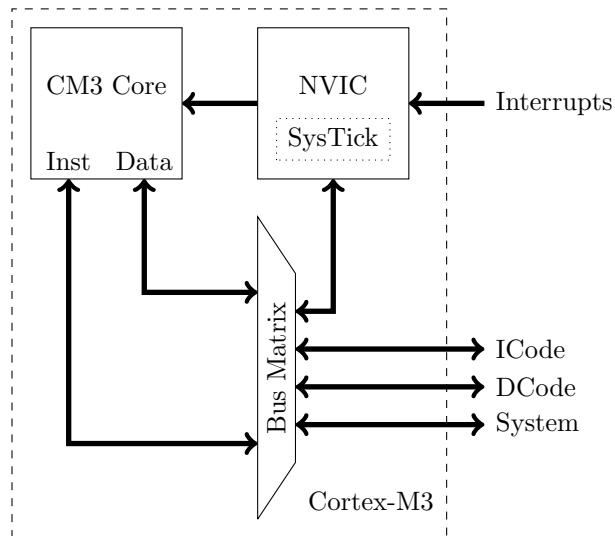


Figura 2.2: Arquitetura Simplificada no núcleo Cortex-M3

O STM32, mostrado na Figura 2.3 liga os três barramentos definidos pela Cortex-M3 através de uma matriz de barramento a nível do micro-controlador. No STM32, o barramento ICode conecta a interface de instrução do CM3 a Memória Flash, o barramento DCode conecta a Memória Flash para buscar dados e o barramento System fornece acesso de leitura/gravação para SRAM e para os periféricos do STM32. O sub-sistema periférico é suportado pelo barramento AHB, que é dividido em dois sub-barramentos AHB1 e AHB2. O STM32 fornece um controlador sofisticado de acesso direto à memória (DMA) que suporta a transferência direta de dados entre periféricos e memória.

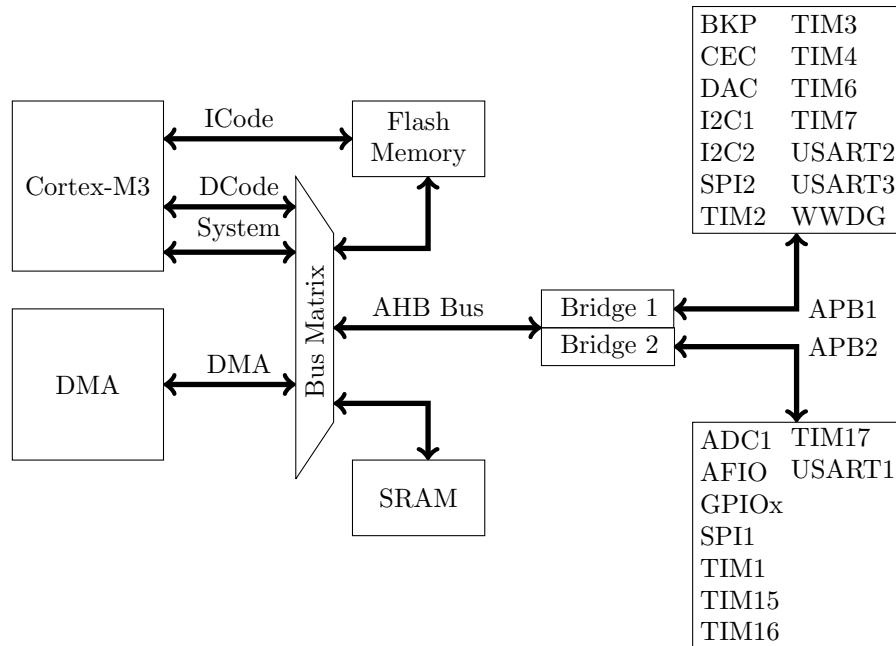


Figura 2.3: STM32 Arquitetura de Barramento Medium Density Value-Line

2.1 O Cortex-M3

O processador CM3 implementa o conjunto de instruções Thumb-2 que fornece um grande conjunto de instruções de 16 bits, permitindo acesso a 2 instruções por acesso a memória, junto com um pequeno conjunto de instruções de 32 bits para apoiar as operações mais complexas. Os detalhes específicos deste conjunto de instruções são em grande parte irrelevantes para este livro porque nós estaremos realizando toda a nossa programação em C. No entanto, existem algumas ideias-chave que discutiremos a seguir.

Tal como acontece com todos os processadores RISC, o Cortex-M3 é uma arquitetura de load/store (carrega/armazena) com três tipos básicos de instruções — operações registrador-para-registrador para processamento de dados, operações de memória que move dados entre a memória e os registradores, e as operações de controle de fluxo permitindo o controle do fluxo das linguagens de programação, tais como os comandos if, while e a chamadas de procedimentos. Por exemplo, suponha que nós definimos o seguinte procedimento em C bastante trivial:

2.1. O CORTEX-M3

```
int counter;

int counterInc(void){
    return counter++;
}
```

O código em linguagem Assembly resultante (anotado) e o código de máquina correspondente é mostrado a seguir:

```
counterInc:
  0: f240 0300  movw    r3, #:lower16:counter // r3 = &counter
  4: f2c0 0300  movt    r3, #:upper16:counter
  8: 6818        ldr     r0, [r3, #0]           // r0 = *r3
  a: 1c42        adds   r2, r0, #1             // r2 = r0 + 1
  c: 601a        str    r2, [r3, #0]           // *r3 = r2
  e: 4740        bx      lr                  // return r0
```

Duas instruções de 32 bits (movw, movt) são usadas para carregar as metades superiores/inferiores do endereço do **counter** (**contador**) (conhecido em tempo de linkedição, e, portanto, 0 na listagem de código). Em seguida, três instruções de 16 bits de carga (ldr) do valor do contador, incrementa (adiciona) o valor, e escrever de volta (str) o valor atualizado. Finalmente, o procedimento retorna o contador inicial.

Não se espera que o leitor deste livro entenda o conjunto de instruções Cortex-M3, ou mesmo este exemplo em grande detalhe. Os pontos principais são que o Cortex-M3 utiliza uma mistura de instruções de 32 bits e de 16 bits (principalmente o último) e que o núcleo interage com memória exclusivamente através de instruções de load/store (carga e armazenamento). Embora existam instruções que fazem o load/store de grupos de registradores (em vários ciclos) não existem instruções que operam diretamente sobre posições de memória.

O núcleo Cortex-M3 tem 16 registradores visíveis ao usuário (mostrados na Figura 2.4) – todo o processamento ocorre nesses registradores. Três destes registradores têm funções específicas, incluindo o contador de programa (program counter)(PC), que guarda o endereço da próxima instrução a ser executada, o registrador de link (link register)(LR), que guarda o endereço a partir do qual o procedimento atual foi chamado, e “o” ponteiro da pilha (stack pointer) (SP), que guarda o endereço do topo da pilha atual (como veremos no Capítulo 11, o CM3 suporta múltiplos modos de execução, cada um com seu próprio stack pointer privado). Mostrado separadamente temos um registrador de status do processador (processor status register) (PSR), que é implicitamente acessado por muitas instruções.

CAPÍTULO 2. INTRODUÇÃO AO STM32 F1

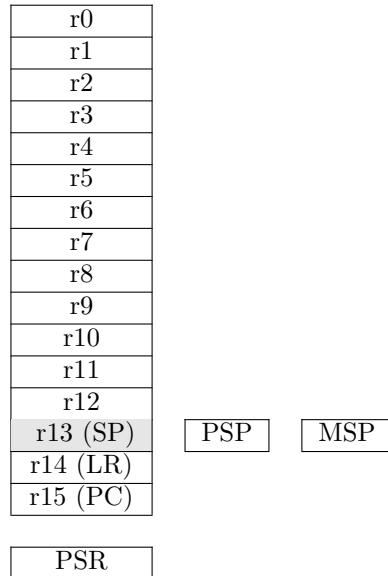


Figura 2.4: Conjunto de Registradores do Processador

O Cortex-M3, como outros processadores ARM foi projetado para ser programado (quase) totalmente em uma linguagem de alto nível, como C. Uma consequência disso é um bem desenvolvido “padrão chamada de procedimentos” (“procedure call standard”) (geralmente chamada de ABI ou Application Binary Interface) que dita como os registradores são usados. [2] Este modelo assume explicitamente que a memória RAM para executar um programa é dividida em três regiões, tal como mostrado na Figura 2.5. Os dados na RAM são alocados durante o processo de linkedição e inicializados pelo código de inicialização no reset (veja o Capítulo 3). O Heap (opcional) é gerenciado em tempo de execução pela biblioteca de código que implementa as funções como `malloc` e `free` que fazem parte da biblioteca C padrão. A stack (pilha) é gerenciada em tempo de execução pelo código gerado pelo compilador que gera stack frames por chamada a procedimentos contendo variáveis locais e registradores salvos.

O Cortex-M3 possui um espaço de endereçamento “físico” de 2^{32} bytes. O ARM Cortex-M3 Technical Reference Manual define como este espaço de endereçamento deve ser utilizado. [1] Isso é mostrado (parcialmente) na Figura 2.6. Como mencionado, a região de “Código” é acessada através dos barramentos ICode (instruções) e DCode (dados constantes). As áreas de SRAM e de Periféricos são acessadas através do barramento System. A po-

2.1. O CORTEX-M3

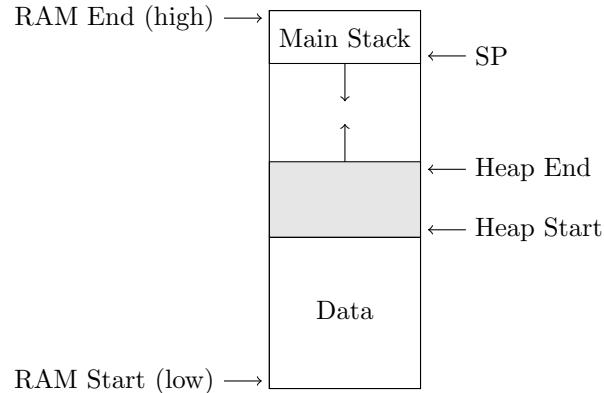


Figura 2.5: Modelo de Memória de Programa

pulação física destas regiões é dependente da implementação. Por exemplo, os processadores STM32 têm memória flash 8K–1M com endereço base em (0x08000000).¹ O processador STM32F100 na placa Discovery tem 8K de SRAM com endereço base em 0x20000000. Não são mostrados neste mapa de endereçamento os periféricos Cortex-M3 internos, como o NVIC, que está localizado a partir do endereço 0xE000E000; estes são definidos no manual de referência Cortex-M3. [1] Iremos discutir o NVIC mais adiante no Capítulo 11.

Como mencionado, o núcleo Cortex-M3 inclui um controlador de interrupção vetorizado (NVIC) (veja o Capítulo 11 para maiores detalhes). O NVIC é um dispositivo programável que fica entre o núcleo CM3 e o micro-controlador. O Cortex-M3 usa um modelo de interrupção vetorizado priorizado – a tabela do vetor é definida para residir a partir de local de memória 0. As primeiras 16 entradas nesta tabela são definidas para todas as implementações Cortex-M3, enquanto o restante, até 240, são de específicas da implementação; por exemplo, os dispositivos STM32F100 definem 60 vetores adicionais. O NVIC suporta a redefinição dinâmica de prioridades com até 256 níveis de prioridade – o STM32 suporta apenas 16 níveis de prioridade. Duas entradas na tabela de vetor são especialmente importantes: o endereço 0 contém o endereço do ponteiro de pilha inicial e o endereço 4 contém o endereço do “reset handler” que será executado em tempo de inicialização.

O NVIC também fornece registradores chave para o controle do sistema, incluindo o temporizador do sistema (System Timer) (SysTick) que

¹Esta memória é “ajustada” (“aliased”) para 0x00000000 durante o boot.

CAPÍTULO 2. INTRODUÇÃO AO STM32 F1

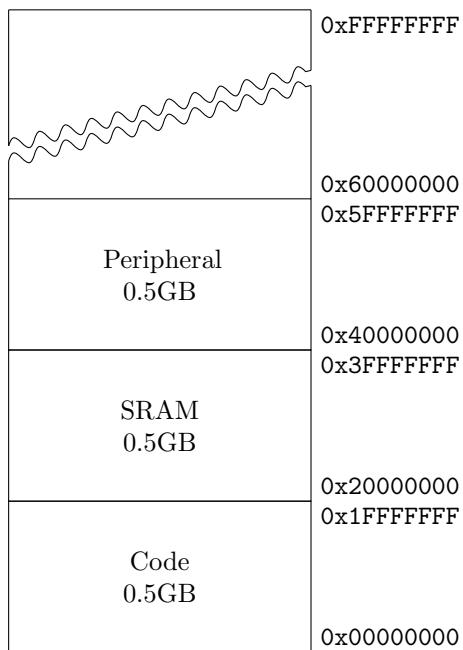


Figura 2.6: Espaço de Enderçamento da Memória do Cortex-M3

fornecer uma interrupção normal de temporização. Fornecer um temporizador integrado em toda a família Cortex-M3 tem a vantagem significativa de tornar o código do sistema operacional altamente portátil – todos os sistemas operacionais precisam de pelo menos um temporizador central para time-slicing. Os registros utilizados para controlar o NVIC são definidos para residir no endereço 0xE000E000 e são definidos pela especificação do Cortex-M3. Esses registros são acessados com o barramento do sistema.

2.2 STM32 F1

O STM32 é uma família de microcontroladores. Os microcontroladores STM32 F1xx são baseadas no Cortex-M3 e incluem o micro-controlador STM32F100 value-line usado na placa Discovery considerada neste livro. A série STM32 L1 é derivada da série STM32 F1, mas com um consumo de energia reduzido. A série STM32 F2 também é baseado no Cortex-M3, mas tem um conjunto avançado de periféricos e um núcleo de processador mais rápido. Muitos dos periféricos da série STM32 F1 são compatíveis para a

2.2. STM32 F1

frente, mas não todos. A série STM32 F4 de processadores usam o núcleo Cortex-M4, que é uma melhoria significativa do Cortex-M3. Finalmente, há uma nova família STM32 – o STM32 F0 baseado no Cortex-M0. Cada uma dessas famílias – STM32 F0, STM32 F1, STM32 L1, STM32 F2, e STM32 F4 são suportados por bibliotecas de firmware diferentes. Enquanto existe uma sobreposição significativa entre as famílias e seus periféricos, também existem diferenças importantes a serem consideradas. Neste livro, vamos nos concentrar na família STM32 F1.

Como mostrado na Figura 2.3, os micro-controladores STM32 F1 são baseados no núcleo Cortex-M3 com um conjunto de periféricos distribuídos em três barramentos – AHB e seu dois sub-barramentos APB1 e APB2. Estes periféricos são controlados pelo núcleo com instruções de carga e armazenamento que acessam registradores mapeados a memória (memory mapped). Os periféricos podem “interromper” o núcleo para solicitar atenção através de pedidos de interrupção específicos de periféricos encaminhados através do NVIC. Finalmente, as transferências de dados entre os periféricos e memória podem ser automatizadas usando DMA. No Capítulo 4 será discutida a configuração básica dos periféricos, e no Capítulo 11, mostramos como as interrupções podem ser usadas para construir software eficaz, e no Capítulo 12, mostramos como usar DMA para melhorar o desempenho e permitir o processamento em paralelo com a transferência de dados.

Ao longo deste livro, utilizamos a ST Standard Peripheral Library para os processadores STM32 F10xx. É útil compreender o layout desta biblioteca de software. A Figura 2.7 fornece uma visão simplificada da estrutura de diretórios. A biblioteca consiste em dois sub-diretórios principais – STM32F10x_StdPeriph_Driver e CMSIS. CMSIS significa “Cortex Micro-controller Software Interface Standard” e fornece o software de baixo nível comum necessário para todas as partes ARM Cortex. Por exemplo, o arquivos core_cm3.* fornecem acesso ao controlador de interrupção, ao timer tick do sistema, e aos módulos de depuração e trace. O diretório STM32F10x_StdPeriph_Driver fornece cerca de um módulo (23 no total) para cada um dos periféricos disponíveis na família STM32 F10x. Na figura, eu inclui módulos para uso geral I/O (GPIO), I2C, SPI, e IO (USART) serial. Ao longo deste livro vou apresentar os módulos conforme necessário.

Há diretórios adicionais distribuídos com as bibliotecas de firmware que fornecem código de exemplo que não são mostrados. A figura fornecida mostra os caminhos para todos os principais componentes necessários para construir os tutoriais neste livro.

CAPÍTULO 2. INTRODUÇÃO AO STM32 F1

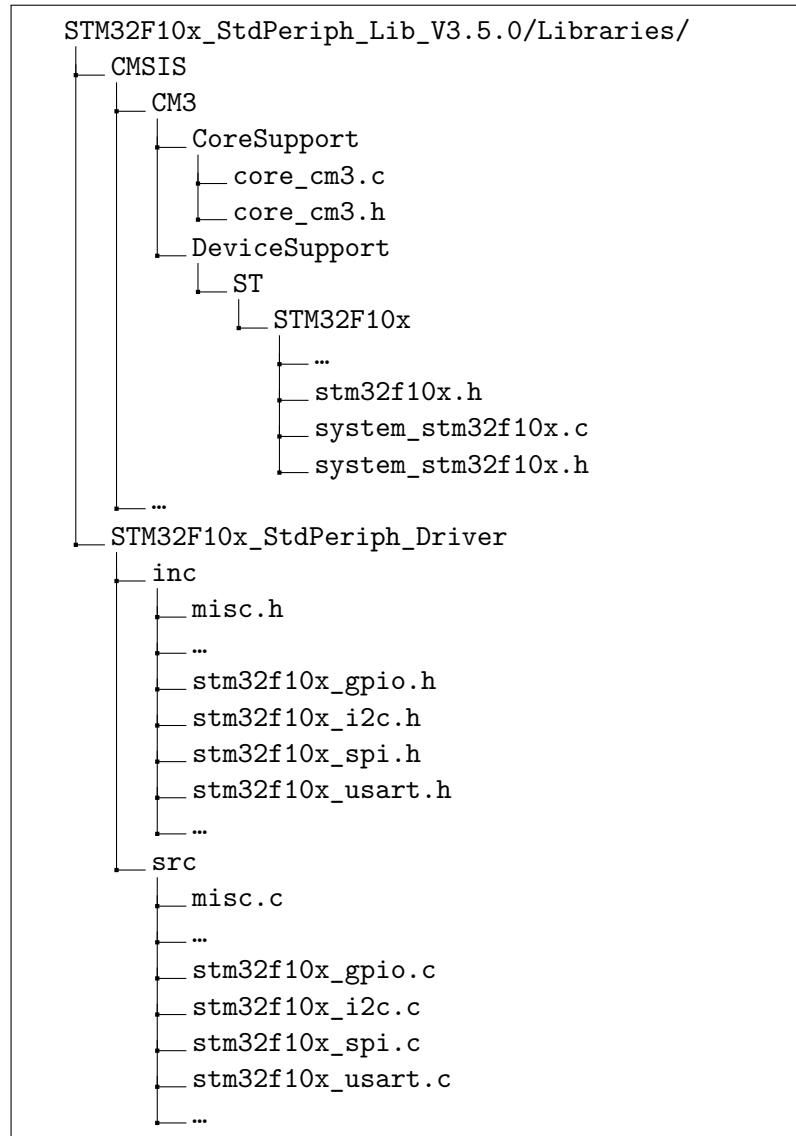


Figura 2.7: ST Standard Peripheral Library

O STM32 F1 tem um sistema de clock (relógio) sofisticado. Há duas fontes primárias de temporização – HSE e LSE. O sinal de HSE é derivado a partir de um cristal de 8MHz ou outro ressonador, e o sinal de LSE é derivado a partir de um cristal de 32,768 kHz. Internamente, o HSE é multiplicada em frequência por meio do uso de um PLL; a saída deste, SYSCLK

2.2. STM32 F1

é utilizada para derivar (por divisão) várias fontes de temporização on-chip e incluem clocks para os periféricos ABP1 e APB2, bem como para os diversos temporizadores programáveis. O LSE é usado para gerenciar um relógio de tempo real de baixo consumo de energia. Os micro-controladores STM32F100 podem suportar uma frequência máxima SYSCLK de 24MHz, enquanto os outros micro-controladores STM32 F1xx suportam uma frequência de 72MHz SYSCLK. Felizmente, a maior parte do código necessário para gerenciar esses temporizadores é fornecida no módulo de biblioteca para periféricos (`system_stm32f10x.[ch]`), que oferece uma função de inicialização – `SystemInit(void)` a ser chamada na inicialização. Este módulo também exporta uma variável `SystemCoreClock` que contém a frequência SYSCLK; isso simplifica a tarefa de desenvolver um código que seja portátil através da família STM32F1.

Os micro-controladores STM32 F1 tem uma variedade de periféricos – nem todos são suportados pelas partes do STM32F100. Os seguintes periféricos são considerados amplamente neste livro.

ADC Conversor Analógico para Digital – Chapter 14.

DAC Conversor Digital para Analógico – Capítulo 13.

GPIO General Purpose I/O – Capítulo 4.

I2C barramento I2C – Capítulo 9.

SPI barramento SPI – Capítulo 6.

TIM Timers (vários) – Capítulo 10.

USART Universal synchronous asynchronous receiver transmitter – Capítulo 5.

Os seguintes periféricos não são considerados neste livro.

CAN Controller area network. Não suportada pelo STM32F100

CEC Consumer electronics control.

CRC Unidade de cálculo Cyclic Redundancy Check.

ETH Ethernet interface. Não suportada pelo STM32F100

CAPÍTULO 2. INTRODUÇÃO AO STM32 F1

FSMC Flexible static memory controller. Não suportada pelos medium density STMF100.

PWR Power control (sleep and low power mode).

RTC Real time clock.

IWDG Independent watchdog.

USB Universal serial bus. Não suportada pelo STM32F100

WWDG Windowing watchdog

Como mencionado anteriormente todos os periféricos são “mapeados na memória”, que significa que o núcleo interage com o hardware periférico lendo e escrevendo nos “registradores” dos periféricos usando instruções de carga e armazenamento.² Todos os registradores de periféricos são documentados nos vários manuais de referência do STM32 ([20, 21]). A documentação inclui definições a nível de bits dos vários registradores e um texto para ajudar a interpretar esses bits. Os endereços físicos reais também são encontradas nos manuais de referência.

A tabela a seguir fornece o endereço para um subconjunto dos periféricos que consideramos neste livro. Observe que todos estes se situam na área do espaço de endereçamento do Cortex-M3 definido para os periféricos.

0x40013800 – 0x40013BFF	USART1
0x40013000 – 0x400133FF	SPI1
0x40012C00 – 0x40012FFF	TIM1 timer
0x40012400 – 0x400127FF	ADC1
...	...

Felizmente, não é necessário para um programador procurar todos estes valores já que são definidos no arquivo `stm32f10x.h` como `USART1_BASE`, `SPI1_BASE`, `TIM1_BASE` `ADC1_BASE`, etc.

Normalmente, cada periférico terá registradores de controle para configurar o periférico, registradores de status para determinar o status atual do periférico, e registradores de dados para ler e gravar dados no periférico. Cada porta GPIO (GPIOA, GPIOB, etc.) tem sete registradores. Dois são usados para configurar os 16 bits da porta individualmente, dois são usados

²A terminologia pode ser confusa – da perspectiva do núcleo do CM3, os registradores dos periféricos são somente locais específicos de memória.

2.2. STM32 F1

para ler/escrever os 16 bits da porta em paralelo, dois são usados para setar/-resetar os 16 bits da porta individualmente, e um é usado para implementar uma “sequência de bloqueio” (“locking sequence”) que destina-se a evitar que o código acidentalmente modifique a configuração da porta. Esta última característica pode ajudar a minimizar a possibilidade de que erros de software levem a falhas de hardware; p. ex.: causando um curto-circuito acidental.

Além de fornecer os endereços dos periféricos, `stm32f10x.h` também fornece estruturas em linguagem C que podem ser usados para acessar cada periférico. Por exemplo, as portas GPIO são definidas pela seguinte estrutura do tipo registro.

```
typedef struct
{
    volatile uint32_t CRL;
    volatile uint32_t CRH;
    volatile uint32_t IDR;
    volatile uint32_t ODR;
    volatile uint32_t BSRR;
    volatile uint32_t BRR;
    volatile uint32_t LCKR;
} GPIO_TypeDef;
```

Os endereços dos registradores das várias portas são definidas na biblioteca como (as seguintes definições são de `stm32f10x.h`)

```
#define PERIPH_BASE      ((uint32_t)0x40000000)
#define APB2PERIPH_BASE   (PERIPH_BASE + 0x10000)
#define GPIOA_BASE         (APB2PERIPH_BASE + 0x0800)
#define GPIOA              ((GPIO_TypeDef *) GPIOA_BASE)
```

Para ler os 16 bits de GPIOA em paralelo podemos usar o seguinte código:

```
uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx) {
    return ((uint16_t)GPIOx->IDR);
}
```

O exemplo anterior é um pouco enganador em sua simplicidade. Considere que, para configurar um pino GPIO requer escrever dois campos de 2 bits no local correto no registro da configuração correta. Em geral, os detalhes necessários podem ser severos.

Felizmente, a biblioteca padrão para periféricos fornece módulos para cada periférico que pode simplificar esta tarefa. Por exemplo, o seguinte é um subconjunto dos procedimentos disponíveis para o gerenciamento de portas GPIO:

CAPÍTULO 2. INTRODUÇÃO AO STM32 F1

```
void GPIO_Init(GPIO_TypeDef* GPIOx,
                GPIO_InitTypeDef* GPIO_InitStruct);
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx,
                               uint16_t GPIO_Pin);
uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx,
                               uint16_t GPIO_Pin);
uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
                   BitAction BitVal);
void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
```

A função de inicialização (`GPIO_Init`) fornece uma interface para configurar bits individuais das portas. As restantes funções fornecem interfaces para ler e escrever (também setar e resetar) ambos os bits individuais e os bits da porta 16 em paralelo.

Nós usamos as funções da biblioteca padrão para periféricos ao longo deste livro.

Existe uma desvantagem significativa para usar esta biblioteca – os módulos são enormes. O módulo do GPIO `stm32f10x_gpio.o` quando compilado com o parâmetro de depuração tem 4K onde uma aplicação simples pode usar em torno de 100 bytes de código personalizado. Além disso, o código pode ser lento – muitas vezes chamadas múltiplas de procedimento são utilizadas pela biblioteca e que não seriam exigidas pelo código personalizado. No entanto, a biblioteca oferece um caminho muito mais rápido para um código do protótipo correto. Para o trabalho de protótipos, provavelmente é melhor utilizar hardware extra (memória, velocidade do clock) em um problema que transparece os detalhes. Para o desenvolvimento sério do produto, pode ser sábio refinar o projeto para reduzir a dependência dessas bibliotecas.

Para ter uma noção do custo de utilização da biblioteca considere o código da Figura 2.1 que configura PC8 e PC9 como saídas (para acionar os LEDs) e PA0 como uma entrada (para ler opush button). ³. Código semelhante baseado em biblioteca é apresentado como um exercício no Capítulo 4. Na Tabela 2.1 comparo os requisitos de espaço das duas versões deste programa: com e sem o uso da biblioteca para periféricos. A primeira coluna (text) fornece o tamanho do “segmento de texto” (“text segment”) (código e de dados de inicialização), os dados alocados na memória RAM na inicialização são a soma de dados (dados inicializados) e bss (dados zerados). Os

³Este é um trecho do `blinky.c` de Paul Robson

2.2. STM32 F1

requisitos totais de memória são fornecidos na coluna text. Os arquivos .elf são os binários completos. Excluindo os 256 bytes pre-alocados da stack de runtime (bss), a versão da biblioteca é quase 3 vezes maior. Ao contrário do original, que fez a inicialização do sistema mínima, eu inclui dois arquivos de inicialização comum para ambas as versões. Além disso, a biblioteca padrão de periféricos tem ampla verificação de parâmetros que eu desativei para esta comparação.

Code With Libraries				
text	data	bss	dec	filename
200	0	0	200	main.o
576	0	0	576	startup_stm32f10x.o
832	0	0	832	stm32f10x_gpio.o
1112	20	0	1132	stm32f10x_rcc.o
484	20	0	504	system_stm32f10x.o
3204	40	256	3500	blinky2-lib.elf
Code Without Libraries				
text	data	bss	dec	filename
136	0	0	136	main.o
576	0	0	576	startup_stm32f10x.o
484	20	0	504	system_stm32f10x.o
1196	20	256	1472	blinky2.elf

Tabela 2.1: Tamanho do Código: Com e Sem as Standard Libraries

CAPÍTULO 2. INTRODUÇÃO AO STM32 F1

```
#include <stm32f10x.h>
int main(void){
    int n = 0;
    int button;

    /* Enable the GPIOA (bit 2) and GPIOC (bit 4) */
    /* See 6.3.7 in stm32f100x reference manual */

    RCC->APB2ENR |= 0x10 | 0x04;

    /* Set GPIOC Pin 8 and Pin 9 to outputs          */
    /* 7.2.2 in stm32f100x reference manual         */

    GPIOC->CRH = 0x11;

    /* Set GPIOA Pin 0 to input floating             */
    /* 7.2.1 in stm32f100x reference manual         */

    GPIOA->CRL = 0x04;

    while(1){
        delay();
        // Read the button - the button pulls down PA0 to logic 0
        button = ((GPIOA->IDR & 0x1) == 0);
        n++;

        /* see 7.2.5 in stm32f100x reference manual */

        if (n & 1) {
            GPIOC->BSRR = 1<<8 ;
        } else {
            GPIOC->BSRR = 1<<24;
        }
        if ((n & 4) && button) {
            GPIOC->BSRR = 1<<9 ;
        } else {
            GPIOC->BSRR = 1<<25;
        }
    }
}

void delay(void){
    int i = 100000; /* About 1/4 second delay */
    while (i-- > 0)
        asm("nop");
}
```

Listing 2.1: Programação Sem utilizar a Standard Peripheral Library

Capítulo 3

Programa Esqueleto

Neste capítulo discutimos o processo de criação, compilação, carregamento, execução, e depuração de um programa com a placa STM32 VL Discovery e a ferramenta Sourcery. Para PCs, o primeiro exemplo é o programa padrão “hello world”:

```
#include <stdio.h>
main() {
    printf("hello world\n");
}
```

que pode ser compilado e executado numa única etapa

```
$ gcc -o hello hello.c; ./hello
hello world
$
```

Este programa simples esconde uma quantidade enorme de complexidades que vão desde a inclusão automática das bibliotecas padrão, à vinculação de código de inicialização, para interagir com o mundo através do shell. No mundo embarcado, muitas dessas complexidades são visíveis para o programador e por isso é necessário entender um pouco mais sobre o ambiente de execução, mesmo para o programa mais simples (e “hello world” não é um programa simples).

No mundo embarcado, o programa mais simples em C, é aquele que não necessita de quaisquer bibliotecas padrão e não interage com o mundo:

```
main {
}
```

CAPÍTULO 3. PROGRAMA ESQUELETO

No entanto, isso é um pouco limitado demais para os nossos propósitos. Em vez disso, nós estruturamos este capítulo em torno de um programa que tem alguns dados e que funcione para sempre:

```
int i;
int off = 5;

void inc(void){
    i += off;
}

int main(void){
    while (1) {
        inc();
    }
}
```

Embora não podermos observar diretamente este programa enquanto ele é executado, podemos anexar um depurador e controlar sua execução através de breakpoints e watchpoints. Observe que esse programa tem duas variáveis globais (`i` e `off`) e uma é iniciada em zero e a outra tem uma inicialização diferente de zero. Além disso, o programa tem uma única função além da principal que é chamada repetidamente.

Antes de executarmos o programa, há uma série de obstáculos que devemos superar. Primeiro, precisamos compilar o programa em um formato binário adequado para carrega-lo na placa Discovery. Em segundo lugar, temos de carregar este binário na memória flash. Finalmente, a fim de observar o programa, é preciso interagir com a placa Discovery por meio de um depurador (GDB). Apesar de usarmos o GDB como um carregador e como um depurador, em geral, os dois últimos passo podem ser feitos por ferramentas distintas.

Programa Demo

O processo de compilação de um programa para processadores embarcados como o STM32 pode envolver alguns detalhes tais como os flags de compilação específica para o processador, os caminhos (paths) para as ferramentas de compilação, etc. Em geral, a melhor abordagem é construir um script “make” para guiar o processo. Antes de avançar neste capítulo, você deve baixar o modelo de código, conforme descrito na Seção 1.2, que contém os scripts necessários. O layout deste diretório é mostrado na Figura 1.16.

Para criar este exemplo no seu sistema, você precisará modificar duas constantes no arquivo `Makefile.common` – `TOOLROOT` que deve indicar o dire-

tório bin da instalação do Sourcery e LIBROOT que deve indicar a instalação da biblioteca padrão para periféricos STM32.

Para compilar este programa, altere os diretórios para o diretório Demo e execute “make”. Isso deve criar um arquivo chamado `Demo.ELF` que contém o binário compilado.

Para baixar e executar esse binário vamos precisar de dois programas – `gdb` (`arm-none-eabi-gdb`), que é parte da distribuição Sourcery, e `st-util`, que fornece um servidor `gdb` que comunica com o stub de depuração `stlink` na placa Discovery através de uma conexão USB. Descrevemos como instalar `st-util` na Seção 1.2. Vamos assumir que você instalou e testou a conexão. Você deve abrir duas janelas. Em um delas, execute:

```
st-util -1
```

Observação: versões anteriores de `st-util` precisam de uma sequencia de startup diferente

```
st-util 4242 /dev/stlink
```

que inicia um servidor `gdb` na porta 4242. Você deverá ver uma saída tal como a seguinte:

```
Chip ID is 00000420, Core ID is 1ba01477.  
KARL - should read back as 0x03, not 60 02 00 00  
Listening at *:4242...
```

Na outra janela, execute (as linhas que iniciam com “`(gdb)`” estão dentro do depurador):

```
arm-none-eabi-gdb Demo.elf  
(gdb) target extended-remote :4242  
(gdb) load  
(gdb) break main  
(gdb) break inc  
(gdb) continue
```

O comando “`target`” deve se conectar ao servidor `gdb` na porta 4242; o “`load`” baixa o executável para a memória flash do STM32. Os próximos dois comandos definem os breakpoints nos procedimentos `main` e `inc`, e o comando “`continue`” executa até o próximo breakpoint. Você pode então executar repetidamente e examinar o valor de `i`:

CAPÍTULO 3. PROGRAMA ESQUELETO

```
(gdb) print i  
(gdb) continue  
...
```

Exercise 3.1 GDB no STM32

Experimente com o GDB para testar os vários comandos disponíveis, como por exemplo:

1. Exibir os valores atuais dos registradores (p. ex.: `print /x $sp` exibe o stack pointer em hexadecimal).
2. Tente configurar um watchpoint em `i`.
3. Tente usar um breakpoint que exibe `i` e continua pouco antes das chamadas de `main` a `inc`

Scripts Make

Enquanto que para baixar e executar um binário é extremamente fácil, o processo de criação não é. As ferramentas de compilação requerem opções não-óbvias, as bibliotecas de firmware STM32 exigem várias definições, e a criação de um executável a partir de binários requer um “linker script” dedicado. Além disso, “`main.c`” não é, em si, um programa completo – há sempre os passos necessários para inicializar variáveis e configurar o ambiente de execução. No mundo Unix, cada programa em C é linkeditado com “`crt0.o`” para realizar essa inicialização. No mundo embarcado, é necessário uma inicialização adicional para configurar o ambiente de hardware. Nesta seção, vamos discutir o processo de criação e na próxima, a função desempenhada pelo código de inicialização STM32 (que está incluído com as bibliotecas de firmware).

Os arquivos make incluídos com o programa demo são divididos em duas partes – `Makefile.common` faz o trabalho pesado e é reutilizável para outros projetos, enquanto `Demo/Makefile` é o específico do projeto. Na verdade, a única função do makefile específico do projeto é definir os arquivos objetos necessários e suas dependências. O Makefile para o projeto demo é mostrado na Listagem 3.1. Ele pode ser modificado para outros projetos, adicionando objetos e modificando os flags de compilação. A variável `TEMPLATEROOT` deve ser modificada para apontar o diretório template.

```

TEMPLATEROOT = ..

# compilation flags for gdb

CFLAGS = -O1 -g
ASFLAGS = -g

# object files

OBJS= $(STARTUP) main.o

# include common make file

include $(TEMPLATEROOT)/Makefile.common

```

Listing 3.1: Makefile Demo

A maior parte do `Makefile.common` define os caminhos para as ferramentas e bibliotecas. As únicas partes notáveis deste arquivo são as definições específicas do processador e os flags de compilação. As definições específicas do processador incluem o linker script `LDSCRIPT` que informa ao linkeditor o arquivo correto do linker script a ser usado – vamos discutir esse arquivo brevemente na próxima seção. O tipo de processador é definido pelo `PTYPE` que controla a compilação condicional das bibliotecas de firmware, e dois arquivos de inicialização – um genérico (`system_stm32f10x.o`) e um específico para os processadores STM32 Value Line (`startup_stm32f10x.o`). As bibliotecas STM32 são projetados para oferecer suporte a vários membros da família STM32, exigindo várias chaves de compilação. O processador da placa STM32 VL Discovery é do tipo “medium density value line” – isso se reflete na definição em tempo de compilação `STM32F10X_MD_VL`.

```

PTYPE = STM32F10X_MD_VL
LDSCRIPT = $(TEMPLATEROOT)/stm32f100.ld
STARTUP= startup_stm32f10x.o system_stm32f10x.o

# Compilation Flags

FULLASSERT = -DUSE_FULL_ASSERT

LDFLAGS+= -T$(LDSCRIPT) -mthumb -mcpu=cortex-m3
CFLAGS+= -mcpu=cortex-m3 -mthumb
CFLAGS+= -I$(TEMPLATEROOT) -I$(DEVICE) -I$(CORE) -I$(PERIPH)/inc -I.
CFLAGS+= -D$(PTYPE) -DUSE_STDPERIPH_DRIVER $(FULLASSERT)

```

CAPÍTULO 3. PROGRAMA ESQUELETO

Os flags `-mcpu=cortex-m3 -mthumb` informam ao gcc sobre o núcleo do processador. `-DUSE_STDPERIPH_DRIVER -DUSE_FULL_ASSERT` afeta a compilação do código da biblioteca de firmware.

O Modelo de Memória do STM32 e a sequencia de Boot

A memória dos processadores STM32 é composta por duas grandes áreas – memória flash (efetivamente somente leitura) começa no endereço 0x08000000 enquanto que a memória RAM estática (leitura / gravação) começa no endereço 0x20000000. O tamanho destas áreas é específico do processador. Quando um programa está sendo executado, o código de máquina (em geral) fica na memória flash e o estado variável (variáveis e a run-time stack) ficam na memória RAM estática (SRAM). Além disso, a primeira parte da memória flash, a partir de 0x08000000, contém uma tabela de vetor consistindo de ponteiros para os vários manipuladores (handlers) de exceção. O mais importante deles é o endereço do manipulador de reset (armazenado em 0x80000004), que é executado sempre que o processador é reiniciado, e o valor inicial da stack pointer (armazenada em 0x80000000).

Esta estrutura de memória é refletida no fragmento do linker script mostrado na Figura 3. O script começa por definir o ponto de entrada de código (`Reset_Handler`) e as duas regiões de memória – flash e ram. Em seguida, ele coloca as seções chamadas a partir dos objetos de arquivos que estão sendo linkados em locais apropriados nestas duas regiões de memória. Do ponto de vista de um executável, há três seções relevantes – “.text” que é sempre alocado em flash, “.data” e “.bss” que sempre são alocados na região de memória RAM. As constantes necessárias para inicializar .data em tempo de execução (runtime) são colocados em flash, bem como o código de inicialização para copiar. Observe ainda que o script linker define os rótulos chaves `_etext`, `_sidata`, ... que são referenciados pelo código de inicialização para inicializar a RAM.

O linker GNU é instruído a colocar a seção de dados em FLASH – especificamente “na” localização de `_sidata`, mas linka as referências de dados para locais em RAM pelo seguinte fragmento de código:

```
.data : AT ( _sidata )
{
    ...
} >RAM
```

A ideia chave é que o linker GNU faça distinção entre o endereço virtual (VMA) e de carga (LMA). O VMA é o endereço que uma seção tem quando o

```

ENTRY(Reset_Handler)
MEMORY
{
    RAM  (rwx)  : ORIGIN = 0x20000000, LENGTH = 8K
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 128K
}

SECTIONS
{
    .text :
    {
        KEEP(*(.isr_vector)) /* vector table */
        *(.text)              /* code */
        *(.text.*)
        *(.rodata)            /* read-only data (constants) */
        ...
    } >FLASH
    ...
    _etext = .;
    _sidata = _etext;
    /* Init data goes in RAM, but is copied after code as well */
    .data : AT ( _sidata )
    {
        ...
        _sdata = .;
        *(.data)
        ...
        _edata = . ; /* used to init data section */
    } >RAM

    .bss :
    {
        ...
        _sbss = .;      /* used to init bss */
        __bss_start__ = _sbss;
        *(.bss)
        ...
        . = ALIGN(4);
        _ebss = . ;    /* used to init bss */
        __bss_end__ = _ebss;
    } >RAM
}

```

Figura 3.1: Fragmento do Linker Script

programa é executado, o LMA é o endereço em que a seção é carregada. Para dados, nosso linker script coloca o LMA da seção de dados dentro de flash e

CAPÍTULO 3. PROGRAMA ESQUELETO

o VMA dentro de RAM – observe que `_sidata = _etext`.

A primeira parte da seção `.text` é carregada com os vetores de exceção (`.isr_vector`) que são posteriormente definidos no código de inicialização. Esses vetores de exceção começam em `0x08000000`, como é exigido quando a inicialização STM32 é a partir da memória flash.

Há um número de detalhes omitidos que asseguram a criação de rótulos assumidos pelas bibliotecas de compilação e padrão bem como manipulação de seções de depuração do binário.

O linker script e o código de inicialização colaboram para criar um ambiente executável significativo. O linker script é responsável por garantir que as várias partes do arquivo executável (p. ex.: os vetores de exceção) estejam em seus devidos lugares e para associar rótulos significativos com regiões específicas de memória utilizada pelo código de inicialização. No reset, o manipulador de reset é chamado. O manipulador (handler) de reset (definido em `startup_stm32f10x.c`) copia os valores iniciais das variáveis da flash (onde o linker coloca-os) para SRAM e zera a chamada da parte não inicializada de SRAM. (veja a Listagem 3.2). Estas etapas são necessárias sempre que o processador reseta de modo a inicializar o ambiente “C”. Em seguida o reset handler chama `SystemInit` (definido em `system_stm32f10x.c` da biblioteca firmware) que inicializa o sistema de clock, desativa e limpa as interrupções. O flag de compilação `STM32F10X_MD_VL` definido em nosso makefile é fundamental porque o código de inicialização do clock é específico do processador. Finalmente, o reset handler chama a função `main` definida no código do usuário. As variáveis externas exigidas pelo reset handler para inicializar a memória (p. ex.: `_sidata`, `_sdata...`) são definidas pelo linker script.

Outra função importante do código de inicialização é para definir a tabela de vetores de interrupção padrão (Listagem 3.3). A fim de permitir que o código da aplicação redefina convenientemente os vários handlers de interrupção, cada vetor de interrupção necessário é atribuído a um apelido substituível (fraco) para um handler padrão (que é um loop eterno). Para criar um handler de interrupção personalizado no código do aplicativo, é suficiente definir um procedimento com o nome do handler apropriado. Uma precaução – você deve ter o cuidado de usar exatamente os nomes definidos na tabela de vetor para o handler ou então não será vinculado na tabela do vetor carregado!

```

// Linker supplied pointers

extern unsigned long _sidata;
extern unsigned long _sdata;
extern unsigned long _edata;
extern unsigned long _sbss;
extern unsigned long _ebss;

extern int main(void);

void Reset_Handler(void) {

    unsigned long *src, *dst;

    src = &_sidata;
    dst = &_sdata;

    // Copy data initializers

    while (dst < &_edata)
        *(dst++) = *(src++);

    // Zero bss

    dst = &_sbss;
    while (dst < &_ebss)
        *(dst++) = 0;

    SystemInit();
    __libc_init_array();
    main();
    while(1) {}
}

```

Listing 3.2: Reset Handler no `startup_stm32f10x.c`

CAPÍTULO 3. PROGRAMA ESQUELETO

```
static void default_handler (void) { while(1); }

void NMI_Handler (void) __attribute__((weak, alias
    →(``default_handler'')));
void HardFault_Handler (void) __attribute__((weak, alias
    →(``default_handler'')));
void MemMange_Handler (void) __attribute__((weak, alias
    →(``default_handler'')));
void BusFault_Handler (void) __attribute__((weak, alias
    →(``default_handler'')));

...
__attribute__((section(``.isr_vector'')))

void (* const g_pfnVectors[])(void) = {
    _estack,
    Reset_Handler,
    NMI_Handler,
    HardFault_Handler,
};

...
```

Listing 3.3: Vetores de Interrupção

Capítulo 4

Configuração do STM32

Os processadores STM32 são sistemas complexos com diversos periféricos. Antes que qualquer um desses periféricos possam ser utilizados eles precisam ser configurados. Algumas dessas configurações são genéricas – por exemplo a distribuição de clock, e a configuração dos pinos – enquanto outras são específicas de cada periférico. Ao longo desse capítulo, nós utilizaremos o programa simples “blinking lights” como um exemplo-guia.

Os passos fundamentais de inicialização que são necessários para utilizar qualquer periférico do STM32 são:

1. Habilitar clock para o periférico.
2. Configurar os pinos necessários ao periféricos.
3. Configurar o hardware do periférico.

Os processadores STM32, como membros da família Cortex-M3, tem todos eles um timer do sistema no núcleo que pode ser usados para fornecer intervalo de tempo regular, “tick”. Nós utilizamos esse timer para fornecer uma taxa constante de pulso (piscadas) para o led no nosso exemplo. A estrutura geral desse programa está ilustrada na Figura 4. O programa inicia incluindo cabeçalhos importantes da biblioteca de firmware – nesse caso para a configuração do clock e dos pinos. A rotina principal segue os passos de inicialização descritos anteriormente e então entra em um loop que troca o estado anterior do LED e espera 250ms. A função main é seguida por um código que implementa a função delay que utiliza o timer do sistema. Finalmente, uma função de ajuda é utilizada para lidar com uma possível violação no firmware da biblioteca (necessária se USE_FULL_ASSERT é definida quando for compilado o

CAPÍTULO 4. CONFIGURAÇÃO DO STM32

firmware da biblioteca dos módulos). Se por um lado o handler assert_failed não faz nada, por outro ela é muito útil quando estamos depurando novos projetos já que a biblioteca de firmware vai fazer uma checagem de parâmetros extensiva. No evento de uma violação da declaração, o GDB pode ser usada para examinar os parâmetros dessa rotina para determinar o ponto de falha.

```
#include <stm32f10x.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_gpio.h>

void Delay(uint32_t nTime);

int main(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // Enable Peripheral Clocks
    ... (1) ...
    // Configure Pins
    ... (2) ...
    // Configure SysTick Timer
    ... (3) ...
    while (1) {
        static int ledval = 0;

        // toggle led
        ... (4) ...

        Delay(250); // wait 250ms
    }
}

// Timer code
... (5) ...

#ifndef USE_FULL_ASSERT
void assert_failed(uint8_t* file, uint32_t line)
{
    /* Infinite loop */
    /* Use GDB to find out why we're here */
    while (1);
}
#endif
```

Figura 4.1: Blinking Lights

A STM32 VL Discovery Board tem um LED ligado ao pino I/O PC9. [14] Ao configurar esse pino, o clock deve primeiro ser ativado para a GPIO Port C com o seguinte comando de biblioteca (descrito em maiores detalhes na Seção 4.1).

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); // (1)
```

Após habilitar os clocks, é necessário configurar qualquer pino que será utilizado. Nesse caso, um único pino (PC9) deve ser configurado como saída (descrito com maiores detalhes na Seção 4.2).

```
/* (2) */  
  
GPIO_StructInit(&GPIO_InitStructure);  
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;  
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

Quando essa configuração está completa, o pino pode ser setado ou resetado com o seguinte código:

```
/* (4) */  
GPIO_WriteBit(GPIOC, GPIO_Pin_9, (ledval) ? Bit_SET : Bit_RESET);  
ledval = 1 - ledval;
```

O Pisca-Led utiliza um “timer tick” para medir a passagem do tempo. Embora esse timer tick utilize uma interrupção, nós não vamos discutir antes do Capítulo 11, o uso nesse caso pode ser tratado simplesmente como um padrão. O núcleo do Cortex-M3 usado no processador STM32 possui um timer dedicado para essa função. O timer funciona como um múltiplo do clock do sistema (que é definido em ticks/second) – aqui nos configuraremos para interrupções de 1ms (a constante `SystemCoreClock` é definida na biblioteca de firmware para ser o numero de ciclos de clock do sistema por segundo):

```
/* (3) */  
if (SysTick_Config(SystemCoreClock / 1000))  
    while (1);
```

A cada 1ms, o timer aciona o gatilho para chamar o `SysTick_Handler`. Para a demonstração do “blinking light”, nós simplesmente decrementaremos um contador comum – declarado como `__IO` para garantir que o compilador não faça uma otimização indesejada.

CAPÍTULO 4. CONFIGURAÇÃO DO STM32

```
/* (5) */
static __IO uint32_t TimingDelay;

void Delay(uint32_t nTime){
    TimingDelay = nTime;
    while(TimingDelay != 0);
}

void SysTick_Handler(void){
    if (TimingDelay != 0x00)
        TimingDelay--;
}
```

Esse programa simples de “blinking light” requer suporte de dois módulos de biblioteca (`stm32_gpio.c` e `stm32_rcc.c`). Para incluir isso no projeto, nós temos que modificar levemente o MakeFile fornecido com o projeto demo.

```
TEMPLATEROOT = ../../stm32vl_template

# additional compilation flags

CFLAGS += -O0 -g
ASFLAGS += -g

# project files

OBJS=      $(STARTUP) main.o
OBJS+=     stm32f10x_gpio.o stm32f10x_rcc.o

# include common make file

include $(TEMPLATEROOT)/Makefile.common
```

No restante desse capítulo nós examinaremos as configurações de clock e de pinos com maiores detalhes.

Exercise 4.1 Blinking Lights

Complete o programa `main.c` do “Blinking Light” e crie um projeto usando o programa de demonstração descrito no Capítulo 3 como um exemplo. Você deverá compilar e rodar o seu programa.

Modifique seu programa para causar uma violação de declaração (assertion violation) – por exemplo troque GPIOC com 66 quando inicializar o pino -- e use o GDB para achar o local na biblioteca de código fonte onde houve a falha de assertion.

4.1. DISTRIBUIÇÃO DE CLOCK

4.1 Distribuição de Clock

No mundo dos processadores embarcados, o consumo de energia é muito importante; assim, a maioria dos processadores embarcados sofisticados possuem mecanismos para desligar quaisquer recursos que não sejam necessários para uma aplicação particular. O STM32 tem uma rede de distribuição de clock complexa que garante que somente os periféricos que são utilizados na aplicação serão ligados. Esse sistema, chamado Reset and Clock Control (RCC) é controlado pelo módulo de firmware `stm32f10x_rcc.[ch]`. Enquanto esse módulo pode ser usado para controlar o clock principal do sistema e PLLs, qualquer configuração necessária por eles é tratada pelo código de inicialização fornecido com os exemplos nesse livro. Nossa preocupação aqui é simplesmente como habilitar o clock dos periféricos.

Os periféricos do STM32 são organizados em três grupos diferentes chamados APB1, APB2, e AHB. Periféricos APB1 incluem os dispositivos I2C, USARTs 2-5, e dispositivos SPI; dispositivos APB2 incluem as portas GPIO, controladores ADC e a USART1. Dispositivos AHB são primariamente orientados a memória incluindo os controladores DMA e interfaces de memória externa (para alguns dispositivos).

Para diversos periféricos o clock pode ser controlado com três rotinas de firmware:

```
RCC_APB1PeriphClockCmd(uint32_t RCC_APB1PERIPH,  
                      FunctionalState NewState)  
RCC_APB2PeriphClockCmd(uint32_t RCC_APB2PERIPH,  
                      FunctionalState NewState)  
RCC_AHBPeriphClockCmd(uint32_t RCC_AHBPERIPH,  
                      FunctionalState NewState)
```

Cada rotina recebe dois parâmetros – um vetor de bits do periférico a qual o estado será modificado, e uma ação – ENABLE ou DISABLE. Por exemplo, as portas do GPIO A e B podem ser habilitadas com o seguinte chamado:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |  
                      RCC_APB2Periph_GPIOB, ENABLE);
```

As constantes apropriadas são definidas em `stm32f10x_rcc.h`; o nome das constantes, mostradas na Tabela 4.1, são relativamente auto-explicativas e correspondem aos nomes dos dispositivos utilizados em vários manuais de referência do STM32 ([20, 21]), limitado a aqueles que estão presentes na MCU

CAPÍTULO 4. CONFIGURAÇÃO DO STM32

STM32 Discovery Board. Isto é um testemunho de que o projeto da família STM32 tem as mesmas constantes e bibliotecas principais que se aplicam através de uma ampla família de dispositivos. (Observe que os componentes `stm32f2xx` e `stm32f4xx` tem semelhanças e diferenças na biblioteca padrão de periféricos).

APB1 Devices	APB2 Devices
<code>RCC_APB1Periph_BKP</code>	<code>RCC_APB2Periph_ADC1</code>
<code>RCC_APB1Periph_CEC</code>	<code>RCC_APB2Periph_AFIO</code>
<code>RCC_APB1Periph_DAC</code>	<code>RCC_APB2Periph_GPIOA</code>
<code>RCC_APB1Periph_I2C1</code>	<code>RCC_APB2Periph_GPIOB</code>
<code>RCC_APB1Periph_I2C2</code>	<code>RCC_APB2Periph_GPIOC</code>
<code>RCC_APB1Periph_PWR</code>	<code>RCC_APB2Periph_GPIOD</code>
<code>RCC_APB1Periph_SPI2</code>	<code>RCC_APB2Periph_GPIOE</code>
<code>RCC_APB1Periph_TIM2</code>	<code>RCC_APB2Periph_SPI1</code>
<code>RCC_APB1Periph_TIM3</code>	<code>RCC_APB2Periph_TIM1</code>
<code>RCC_APB1Periph_TIM4</code>	<code>RCC_APB2Periph_TIM15</code>
<code>RCC_APB1Periph_TIM5</code>	<code>RCC_APB2Periph_TIM16</code>
<code>RCC_APB1Periph_TIM6</code>	<code>RCC_APB2Periph_TIM17</code>
<code>RCC_APB1Periph_TIM7</code>	<code>RCC_APB2Periph_USART1</code>
<code>RCC_APB1Periph_USART2</code>	
<code>RCC_APB1Periph_USART3</code>	
<code>RCC_APB1Periph_WWDG</code>	
AHB Devices	
<code>RCC_AHBPeriph_CRC</code>	<code>RCC_AHBPeriph_DMA</code>

Tabela 4.1: Clock Distribution Constant Names (`stm32f10x_rcc.h`)

Os códigos da biblioteca padrão de periféricos para habilitar clocks não faz nenhuma “magica”, mas sim libera o programador da necessidade de ser íntimo com os registradores do microcontrolador. Por exemplo, os periféricos APB2 são habilitados através de um único registrador (`RCC_APB2ENR`) (mostrador na Figura 4.2). ¹; cada periférico é habilitado, ou desabilitado, pelo estado de um único bit. Por exemplo, Bit 2 determina quando o GPIOA está habilitado (1) ou desabilitado (0). Aplicando a estrutura e definições de registrador em `<stm32f10x.h>` nós podemos habilitar GPIOA e GPIOB como segue:

¹Os vários registradores de controle estão completamente documentados em [20].

4.2. PINOS DE I/O

```
ABP2ENR |= 0x0C;
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	URT1	Res.	SPI1	TIM1	Res.	ADC1	IOPG	IOPF	IOPE	IOPD	IOPC	IOPB	IOPA	Res.	AFIO

Figura 4.2: APB2 Peripheral Clock Enable Register

4.2 Pinos de I/O

A maioria dos pinos do STM32 pode ser configurado como entrada ou saída e poderão ser conectado tanto para as portas GPIO quanto para “funções alternativas” (outros periféricos). Como uma convenção de nomenclatura padrão, os pinos são nomeados por sua função GPIO – por exemplo PA0 (bit 0 da porta A) ou PB9 (bit 9 da porta B). De fato, a nomenclatura da Discovery Board segue essa convenção. Dentro das restrições específicas do hardware, cada pino pode ser configurado nos modos mostrados na Tabela 4.2.

Function	Library Constant
Alternate function open-drain	GPIO_Mode_AF_OD
Alternate function push-pull	GPIO_Mode_AF_PP
Analog	GPIO_Mode_AIN
Input floating	GPIO_Mode_IN_FLOATING
Input pull-down	GPIO_Mode_IPD
Input pull-up	GPIO_Mode_IPU
Output open-drain	GPIO_Mode_Out_OD
Output push-pull	GPIO_Mode_Out_PP

Tabela 4.2: Pin Modes (`stm32f10x_gpio.h`)

Por default a maioria dos pinos está resetada para “Entrada Flutuante” (“Input Floating”) — isso garante que não ocorrerá nenhum conflito de hardware quando o sistema for ligado. A biblioteca de Firmware fornece uma rotina de inicialização em `stm32f10x_gpio.[ch]` que pode ser usada para reconfigurar os pinos. Por exemplo, para o Blinking Light nós configuramos PC9 como uma saída (lenta) como mostrado na Listagem 4.1.

CAPÍTULO 4. CONFIGURAÇÃO DO STM32

```
// see stm32f10x_gpio.h
GPIO_InitTypeDef GPIO_InitStructure;

GPIO_StructInit(&GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

Listing 4.1: Configuração de PC9

Quando configuramos uma saída como mostrado anteriormente, nós temos três escolhas de “velocidade” de saída – 50MHz, 10MHz, e 2MHz. Em geral, por razões de consumo de energia e ruído, é desejável usar a menor velocidade em relação aos requisitos de I/O. O campo `GPIO_Pin` é um vetor de bits e é possível configurar múltiplos pinos associados com uma única porta em um único passo.

Observação: A pinagem (pin-outs) para partes específicas (incluindo as atribuições dos periféricos aos pinos) são definidos nos seus respectivos data sheet (p. ex. [15]) e NÃO no manual de programação! As atribuições de pinos para a Discovery Board são documentados no manual do usuário [14].

Como nós vimos, nós podemos atribuir um valor ao pino que controla o LED com o seguinte procedimento:

```
GPIO_WriteBit(GPIOC, GPIO_Pin_9, x); // x is Bit_SET or Bit_RESET
```

A biblioteca dos módulos gpio fornece funções para ler e escrever tanto pinos individualmente quanto portas inteiras – o ultimo é especificamente útil para capturar dados paralelos. É instrutivo ler o arquivo `stm32f10x_gpio.h`.

Para (re-)configurar o pino associado com o botão da Discovery Board, nós podemos usar o seguinte código (após configurar o clock para porta A!):

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

Para ler o valor atual do botão nós podemos executar:

```
GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
```

4.3. FUNÇÕES ALTERNATIVAS

Relembre do Capítulo 2 que cada porta GPIO é controlada por 7 registradores:

```
typedef struct
{
    volatile uint32_t CRL;
    volatile uint32_t CRH;
    volatile uint32_t IDR;
    volatile uint32_t ODR;
    volatile uint32_t BSRR;
    volatile uint32_t BRR;
    volatile uint32_t LCKR;
} GPIO_TypeDef;
```

Os 16 bits de cada porta são configurados com CRL (bits 0-7) e CRH (pins 8-15). Para suportar os vários modos de I/O, 4 bits de configuração são necessários para cada bit do GPIO. Os 16 bits GPIO podem ser lidos em paralelo (IDR) e escritos em paralelo (ODR). Por conveniência, registradores BSRR e BRR fornecem um mecanismo para ligar ou desligar bits individualmente. O registrador de trava LCKR fornece um mecanismo para “travar” (“lock”) a configuração de bits individuais contra a reconfiguração de software e consequentemente proteger o hardware de bugs de software.

Exercise 4.2 Blinking Lights com Pushbutton

Modifique o seu programa Pisca-Led adicionando a leitura do estado do botão do usuário (PA0) no Led azul (PC8). Veja se você consegue descobrir como configurar ambos os Leds com uma única chamada de `GPIO_Init`.

4.3 Funções Alternativas

Periféricos como as USARTs compartilham pinos com os dispositivos GPIO. Antes destes periféricos poderem ser utilizados, qualquer saída requerida pelo periférico necessita ser configurada para um “modo alternativo” (“alternative mode”). Por exemplo, o pino Tx (saída de dados) da USART1 é configurado da seguinte maneira:

```
GPIO_InitStruct.GPIO_PIN = GPIO_Pin_9;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

CAPÍTULO 4. CONFIGURAÇÃO DO STM32

A configuração específica requerida por cada periférico é descrita na seção 7.1.11 do manual de referência RM0041 [20] do stm32f10xx (seção 9.1.11 para manual de referência RM0008 [21] do stm32f103xx).

4.4 Remapeamento

Também é possível “remapear” os pinos de modo que pinos não-padrão sejam usados para vários periféricos de modo a minimizar conflitos. Esses remapeamentos, que estão além do escopo desse livro, são descritos em um manual apropriado do STM32 ([20, 21]).

4.5 Atribuições de Pinos para os Exemplos e Exercícios

Nesse livro nós desenvolvemos uma série de exemplos e exercícios baseados na STM32VL Discovery Board. Para garantir que esses exemplos possam funcionar juntos são necessários alguns cuidados na seleção de dispositivos da STM32 e nos pinos GPIO -- por exemplo nós utilizamos o dispositivo SPI2 em vez do SPI1 devido um conflito de recursos. Na Tabela 4.5 enumeramos todos as atribuições de pinos usados ao longo desse livro com as configurações requeridas para esses usos.

4.5. ATRIBUIÇÕES DE PINOS PARA OS EXEMPLOS E EXERCÍCIOS

Device	Pino	Função	Configuração
User Button	PA0		Input floating
LCD Backlight	PA1	Backlight	Output/Alternative function push-pull
DAC1	PA4	DAC Output	Analog
ADC	PA6 PA7	IN6 IN7	Input floating Input floating
Timer 1	PA8	Channel 1	Input floating
USART 1	PA9	TX	Alternative function push-pull
	PA10	RX	Input Pull-up
	PA11	nCTS	Input Pull-up
	PA12	nRTS	Alternative function push-pull
Timer 3	PB0	Channel 3	Alternative function push-pull
	PB1	Channel 4	Alternative function push-pull
I2C1	PB6 PB7	SCK SDA	Alternative function open-drain Alternative function open-drain
Timer 4	PB9	Channel 4	Alternative function push-pull
I2C2	PB10 PB11	SCK SDA	Alternative function open-drain Alternative function open-drain
SPI2	PB13	CLK	Alternative function push-pull
	PB14	MISO	Input pull-up
	PB15	MOSI	Alternative function push-pull
LCD control	PC0	LCD Select	Output push-pull
	PC1	Reset	Output push-pull
	PC2	Data/Control	Output push-pull
SD Card	PC6	Select	Output push-pull
Blue LED	PC8		Output push-pull
Green LED	PC9		Output push-pull
SPI EEPROM CS	PC10		Output push-pull

Tabela 4.3: Atribuições de Pinos para os Exercícios

CAPÍTULO 4. CONFIGURAÇÃO DO STM32

4.6 Configuração de Periféricos

Como mencionado, o terceiro estágio de configuração, após a distribuição de clock e configuração de pinos, é a configuração de periféricos. Enquanto nós adiamos a discussão da configuração de periféricos específicos, a biblioteca padrão de firmware oferece um modo padrão para o processo de configuração. Nós já vimos um pouco disso antes, com a configuração GPIO onde estruturas de dispositivos específicos eram cheios de diversos parâmetros e um ou mais pinos para uma dada porta eram inicializados:

```
GPIO_StructInit(&GPIO_InitStructure);  
... fill in structure ...  
GPIO_Init(GPIOx, &GPIO_InitStructure);
```

Também é possível “des-inicializar” uma porta, retornando todas as configurações de pinos para o estado de reset de hardware com uma chamada para:

```
GPIO_DeInit(GPIOx)
```

A função DeInit reseta os registradores de periféricos, mas isso não desabilita o clock de periférico — isso requer uma chamada em separado para o comando de clock (com DISABLE no lugar de ENABLE).

Esse padrão — uma estrutura de inicialização, uma função de inicialização, e a função de des-inicialização é repetida do começo ao fim da biblioteca padrão do periférico. A convenção de nomenclatura básica para periférico “ppp” é:

Files	stm32f10x_ppp.[c h]
Init Structure	ppp_InitTypeDef
Zero Structure	ppp_StructInit(ppp_InitTypeDef*)
Initialize Peripheral	ppp_Init([sub-device], ppp_InitTypeDef*)
De-initialize Peripheral	ppp_DeInit([sub-device])

Exemplos de dispositivos com “sub dispositivos” (“sub device”) opcionais são USART, SPI, I2C. Timer são um caso um pouco complicado porque cada timer é tipicamente diversos dispositivos — um “time base”, zero ou mais comparadores de saída, zero ou mais captações de entrada. Existem outras exceções, mas a maioria para periféricos que não são suportados para os componentes da medium density value-line.

Exercise 4.3 Configuração sem a Standard Peripheral Library

4.6. CONFIGURAÇÃO DE PERIFÉRICOS

Escreva um programa usando somente as constantes definidas no manual de referência de programação ([20]) que: configure os pinos para o botão do usuário e Led azul, e, em um loop infinito, mostre o estado do botão no LED.

Seu código deverá ser semelhante com o seguinte:

```
main()
{
    // configure button
    // configure led

    while (1)
    {
        if (read(button))
            led = on;
        else
            led = off;
    }
}
```


Capítulo 5

Asynchronous Serial Communication

Depois dos LEDs e chaves, o método mais básico para comunicação com um processador embarcado é a serial assíncrona. A comunicação serial assíncrona em sua forma mais primitiva é implementada por um par simétrico de fios que conecta dois dispositivos -- aqui vou me referir a eles como o host (hospedeiro) e o target (alvo), embora esses termos sejam arbitrários. Sempre que o host tem dados para enviar para o target, o faz através do envio de um fluxo de bits codificados sobre seu fio de transmissão (TX); estes dados são recebidos pelo destino através de seu fio de recepção (RX). Da mesma forma, quando o target tem dados a enviar para o host, ele transmite o fluxo de bits codificados sobre seu fio TX e estes dados são recebidos pelo host sobre seu fio RX. Tal arranjo é ilustrado na Figura 5. Este modo de comunicação se chama “assíncrono” porque o host e o target não compartilham nenhuma referência de tempo. Em vez disso, as propriedades temporais são codificadas no fluxo de bits pelo transmissor e devem ser decodificadas pelo receptor.

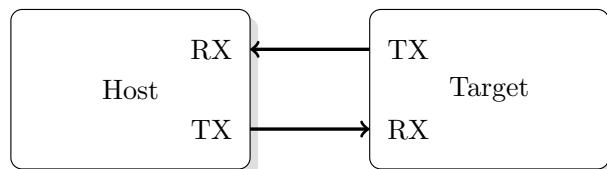


Figura 5.1: Topologia Básica de Comunicação Serial

CAPÍTULO 5. ASYNCHRONOUS SERIAL COMMUNICATION

Um dispositivo comumente utilizado para codificar e decodificar esses fluxos assíncronos de bits é a Universal Asynchronous Receiver/Transmitter (UART) que converte bytes de dados fornecidos pelo software em uma sequência de bits individuais, e, inversamente, converte essa sequência de bits de dados em bytes a serem passado para o software. Os processadores STM32 incluem (até) cinco desses dispositivos chamados USARTs (para Universal Synchronous/Asynchronous Receiver/Transmitter) porque suportam modos de comunicação adicionais além da comunicação assíncrona básica. Neste capítulo vamos explorar a comunicação serial entre STM32 USART (target) e uma ponte USB/UART conectada a um PC host.

UART também pode ser utilizada para fazer a interface de uma ampla variedade de outros periféricos. Por exemplo, modems amplamente disponíveis de telefones celulares GSM/GPRS e modems Bluetooth podem ser conectados a UART de um micro-controlador. Da mesma forma, receptores GPS frequentemente suportam interfaces UART. O protocolo serial fornece acesso a uma ampla variedade de dispositivos, tal como acontece com outras interfaces que consideraremos nos próximos capítulos.

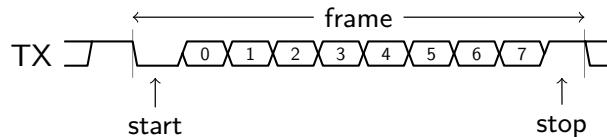


Figura 5.2: Protocolo de Comunicação Serial

Uma das codificações básicas utilizadas para comunicação serial assíncrona é mostrada na Figura 5. Cada caractere é transmitido em um “frame” (quadro), que começa com um bit “low” de início (start), seguido por oito bits de dados e termina com um bit “high” de parada (stop). Os bits de dados são codificados como sinais “high” ou “low” para (1) e (0), respectivamente. Entre os “frames”, uma condição de “idle” (“repouso”) é sinalizada pela transmissão de um sinal alto contínuo. Assim, é garantido que cada “frames” comece com uma transição de alto para baixo e contenha, pelo menos, uma transição baixo-alto. Alternativas para esta estrutura básica de “frames” incluem diferentes números de bits de dados (p. ex: 9), um bit de paridade após o último bit de dados, que permite a detecção de erros e condições de longas de parada.

Não há um clock diretamente codificado no sinal (em contraste com os

protocolos de sinalização tais como a codificação Manchester) — a transição de início fornece a única informação temporal no fluxo de dados. O transmissor e o receptor mantêm, cada um, independentemente, clocks que funcionam em (um múltiplos de) uma frequência concordada — de forma imprecisa, comumente chamada de taxa de transmissão (baud rate). Estes dois relógios (clocks) não estão sincronizados e não é garantido que estarão exatamente na mesma frequência, mas devem ser próximas o suficiente, numa frequência (menos de 2%) capaz de recuperar os dados.

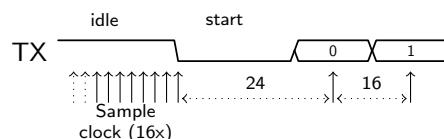


Figura 5.3: Decodificação do sinal UART

Para entender como o receptor extraí os dados codificados, assumimos que há um clock funcionando em um múltiplo da taxa de transmissão (p. ex.: 16x) a partir de um estado de repouso (idle) como mostrado na Figura 5. O receptor RX amostra (“samples”) seu sinal até detectar uma transição alta-baixa. Em seguida, ele espera 1,5 períodos de bit (24 períodos do clock) para fazer um sample do sinal RX até que ele estime ser o centro de dados do bit 0. O receptor, então, sample o RX em intervalos de bit (16 períodos do clock) até que tenha lido os 7 bits de dados restantes e o bit de parada. A partir desse ponto, o processo se repete. A extração bem sucedida dos dados por um frame requer que, ao longo de 10,5 períodos de bits, o desvio do relógio do receptor em relação ao relógio do transmissor seja menor que 0,5 períodos, a fim de detectar corretamente o bit de parada.

Exercise 5.1 Testando a Interface USB/UART

Antes de discutir a implementação da comunicação UART com a STM32, pode ser útil usar o Saleae Logic para “ver” os sinais assíncronos. Vamos usar a ponte USB-UART para gerar sinais que iremos, então, capturar com o analisador lógico. Mais tarde, esta será uma ferramenta essencial para depuração do código em execução no STM32.

É extremamente difícil depurar os drivers do dispositivo de hardware usando apenas um software depurador, tais como GDB, porque esses depu-

CAPÍTULO 5. ASYNCHRONOUS SERIAL COMMUNICATION

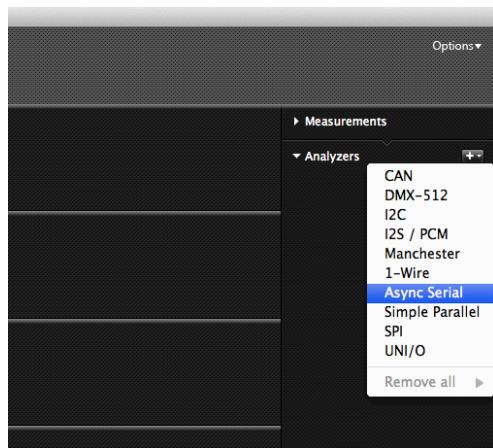


Figura 5.4: Adicionando um Analisador de Protocolos

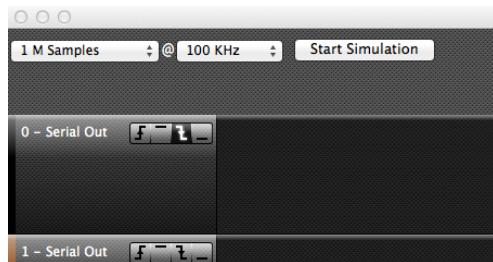


Figura 5.5: Configurando a taxa de Captura e o Trigger

radores não fornecem nenhuma visibilidade sobre o que o hardware está realmente fazendo. Por exemplo, se alguém deixa de configurar os clocks para um hardware periférico específico, tal periférico não fará nada em resposta a comandos do software, mesmo que todo o software específico do periférico esteja completamente correto. Talvez ainda mais irritante seja quando a interface do hardware quase funciona. Por exemplo, no desenvolvimento dos exemplos I2C usando o Wii Nunchuk, descobri que enquanto o STM32 e o Nunchuk estavam se comunicando, os dados trocados estavam errados — em última análise, eu acabei descobrindo uma velocidade incorreta do clock. Neste caso do GDB foi um pouco útil no isolamento do problema, mas não para seu diagnóstico.

Para o desenvolvimento de sistemas embarcados com hardware de comunicação, um analisador lógico fornece a capacidade de observar os acontecimentos reais de comunicação entre dois componentes de hardware. Esse-

cialmente, um analisador lógico “escuta” um conjunto de sinais digitais até um evento disparador ocorrer, para então captar uma janela de dados em torno do evento de gatilho (trigger). O software associado a um analisador lógico nos permite interpretar os dados capturados. No caso mais simples, um evento de gatilho pode ser uma ocorrência de uma transição em um sinal — no caso de comunicação USART, cada evento de transmissão começa com uma queda de transição. Ao longo deste livro, usamos o analisador lógico Saleae Logic. Analisadores lógicos profissionais podem capturar sequências complexas de eventos para formar uma condição de gatilho, mas o analisador Saleae é limitado a eventos simples. Além disso, o Saleae só pode capturar sinais a uma taxa de amostragem modesta. No entanto, é mais do que suficiente para os exemplos que apresentamos.

Os primeiros analisadores lógicos eram relativamente limitados em mostrar as informações capturadas como uma sequência simples de tempo dos sinais. Sistemas mais modernos, incluindo o Saleae Logic, fornecem capacidade de analisar protocolos que sobrepõem as sequências de tempo com interpretações. Por exemplo, com analisadores de protocolo serial, os caracteres reais transmitidos são exibidos sobrepostos a uma sequência de tempo.

O software Saleae Logic fornece um modo de simulação, que é útil para aprender como funciona o sistema. Vamos começar usando o modo de simulação para avaliar o protocolo serial. Você precisará baixar e instalar o software Saleae Logic apropriado para seu sistema em <http://www.saleae.com/downloads/>.

Inicie o software Logic. No lado direito da tela, adicione (“add”) um analisador “Async Serial” — as opções default são boas — e renomeie o canal 0 para “Saída Serial” (ou “Serial Out”). Isso é mostrado na Figura 5.4.

Defina a taxa de captura a 100 KHz, o número de amostras a 1 M e a condição de gatilho na Serial Out para a seta para baixo (uma transição de ‘1’ para ‘0’) como mostrado na Figura 5.5. Ao configurar o Saleae para outros protocolos, você precisará selecionar uma taxa de captura que seja um múltiplo (preferencialmente de 10x ou mais) da taxa de dados. Você perceberá que o Saleae Logic é um pouco limitante a este respeito. Por exemplo, achei necessário depurar o protocolo SPI com uma taxa um pouco reduzida, para garantir que o Saleae Logic fosse capaz de realizar a buferização dos dados. Em última análise, pode ser necessário depurar a uma taxa modesta e, quando confiante de que tudo está funcionando, aumentar a velocidade do protocolo subjacente.

Finalmente, inicie a simulação. Após a simulação, faça um zoom nos

CAPÍTULO 5. ASYNCHRONOUS SERIAL COMMUNICATION

dados (clicando com o ponteiro do mouse), até que os frames seriais individuais estejam visíveis. Você pode querer ler o guia do usuário Saleae Logic para aprender a navegar pelos dados exibidos.

Depois de entender a interface básica, aterre e conecte os pinos TX de um adaptador USB/UART nos fios cinza e preto (canal 0) da lógica. Conecte o adaptador USB/UART e o módulo do Saleae Logic no seu computador. Nós usaremos programa “screen” do Unix como uma interface para a ponte USB-UART.

Primeiro, você precisa determinar o nome do dispositivo para a ponte USB-UART. Liste o conteúdo de `/dev/tty*` para encontrar todos os dispositivos tty. Você está procurando um dispositivo com USB em seu nome (p. ex.: no OS X, `tty.SLAB_USBtoUART`, ou `ttyUSB0` no Linux).

Uma vez que você encontrou o dispositivo (p.ex.: `/dev/ttYXX`), é importante ter a certeza de que ele está configurado corretamente. Para determinar a configuração atual execute

```
stty -f /dev/ttYXX
```

No Linux `-f` deve ser substituído por `-F`. Isto irá listar a configuração atual. Se a taxa de transmissão for diferente de 9600, você tem duas opções -- alterar a taxa de transmissão em seu programa ou modificar a taxa de transmissão do dispositivo (consulte a página man para `stty` aprender a fazer isso.). Uma vez que a configuração do seu programa e do dispositivo corresponderem, execute o seguinte na tela do terminal.

```
screen /dev/ttYXX
```

Agora, qualquer coisa que você digitar no programa screen será transmitida pelo adaptador USB/UART. No programa Logic, “start” (Iniciar) uma captura — você deverá ver uma janela que diz “waiting for a trigger” (Esperando por um gatilho). Em seguida, na janela do screen digite o alfabeto tão rápido quanto puder. Uma vez concluída a captura, amplie e examine os dados capturados — você deve ver os caracteres digitados. Se os dados são marcados com erros de framing, então é provável que o USB/UART esteja transmitindo a uma taxa diferente do que espera o Saleae Logic. Corrija isso (reconfigurando o Logic ou o USB/UART) antes de prosseguir.

Para sair da tela, digite `C-a k` (“control a k”).

5.1. IMPLEMENTAÇÃO DE POLLING NO STM32

5.1 Implementação de Polling no STM32

A forma mais simples de comunicação USART baseia-se no polling de estado do dispositivo USART. Cada USART do STM32 tem 6 registradores – dos quais 4 são usados para configuração através das rotinas de inicialização fornecidas pela biblioteca do driver, conforme mostrado na Seção 5.2. Os 2 registradores restantes são os registradores de “dados” e de “status”. Enquanto o registrador de dados ocupar uma única palavra na memória, são, na verdade, dois locais distintos; quando o registrador de dados é gravado, o caractere escrito é transmitido pela USART. Quando o registrador de dados é lido, o caractere mais recentemente recebido pela USART é retornado. O registrador de status contém um número de flags para determinar o estado atual da USART. Os mais importantes são:

```
USART_FLAG_TXE -- Registrador de Transmissão de dados vazio  
USART_FLAG_RXNE -- Registrador Recepção de dados não vazio
```

Outros flags fornecem informações de erro que incluem paridade, framing e erros de overrun que devem ser verificados em uma implementação robusta e eficiente.

A fim de transmitir um caractere, o software de aplicação deve esperar até que o registrador de dados de transmissão estar vazio e em seguida, escrever o caractere a ser transmitido nesse registrador. O estado do registrador de dados a transmitir é determinado verificando o flag USART_FLAG_TXE do registrador de status da USART. O código a seguir implementa um procedimento básico putchar utilizando USART1.

```
int putchar(int c){  
    while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);  
    USART1->DR = (c & 0xff);  
    return 0;  
}
```

Com um único registro de transmissão de dados, esta implementação é tão rápida quanto a velocidade de transmissão subjacente e deve esperar (pelo “polling” do flag de status) entre os caracteres. Para programas simples isto pode ser aceitável, mas em geral, uma implementação não-bloqueante como a descrita na Seção 11.5 é preferida. O par de putchar é getchar.

```
int getchar(void){  
    while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);  
    return USART1->DR & 0xff;
```

{}

Observe o uso do sinalizador de status `USART_IT_RXNE` para determinar quando um caractere é recebido (“receive data not empty”). Além disso, observe o uso de uma máscara para selecionar os 8 bits inferiores retornados pelo `USART_ReceiveData` – este procedimento da biblioteca retorna 9 bits por padrão porque a configuração da USART permite 9 bits de dados. Por exemplo, o nono bit pode conter informações de paridade que poderiam ser usadas para verificar a validade do caractere recebido.

Enquanto a execução de sondagem de `putchar` tem a deficiência de ser lenta, a execução de pooling de `getchar` tem uma falha fatal — se o código do aplicativo não recebe caracteres assim que eles chegam, mas o host continua a enviar caracteres, um “*overrun*” (“transbordamento”) irá ocorrer, resultando na perda de caracteres. O STM32 recebe um único buffer de dados, enquanto muitos micro-controladores possuem buffers de 8 ou 16 caracteres. Isto fornece muito pouco espaço para variação de temporização no código do aplicativo, que é responsável pelo monitoramento do receptor USART. No Capítulo 11 discutimos o uso do código baseado em interrupção, para atenuar essa deficiência.

5.2 Inicialização

Tal como acontece com todos os periféricos do STM32, as USARTs devem ser inicializadas antes que elas possam ser utilizadas. Esta inicialização inclui a configuração de pinos, distribuição de clocks, e inicialização do dispositivo. A inicialização é tratada mais convenientemente com a biblioteca Standard Peripheral Driver — assumimos a versão 3.5.0. O componente `stm32f100` na Discovery Board tem 3 USARTs chamadas USART1 — USART3 por toda a documentação e biblioteca do driver. A seguir, utilizaremos a USART1, mas os princípios são os mesmos para as outras USARTs.

Existem três módulos (além do cabeçalho geral) que fazem parte da biblioteca de drivers e são necessários para aplicações USART (você terá de incluir os arquivos de objetos associados em seu arquivo de make).

```
#include <stm32f10x.h>
#include <stm32f10x_gpio.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_usart.h>
```

5.2. INICIALIZAÇÃO

A primeira etapa de inicialização é habilitar os sinais RCC (Reset e Clock Control) para os vários blocos funcionais necessários à utilização da USART – que incluem portas GPIO (Port A para USART1), o componente USART e o módulo AF (Alternative Function – função alternativa). Para USART1, a etapa de configuração necessária da RCC é:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 |  
                        RCC_APB2Periph_AFIO |  
                        RCC_APB2Periph_GPIOA, ENABLE);
```

USART2 e USART3 são periféricos da “APB1”, por isso a sua inicialização do RCC difere ligeiramente. Observe que os vários flags de APB2 são configurados ao mesmo tempo com operações “OR”; também é aceitável habilitar os clocks em etapas separadas.

Função	Pino		
	x=1	x=2	x=3
USARTx_TX	PA9	PA2	PB10
USARTx_RX	PA10	PA3	PB11
USARTx_CK	PA8	PA4	PB12
USARTx_RTS	PA12	PA1	PB14
USARTx_CTS	PA11	PA0	PB13

Tabela 5.1: Pinos USART

Uma vez que o clock é ativado, é necessário configurar os pinos USART — os pinos default para todas as três USARTs são mostrados na Tabela 5.1.

Os manuais de referência STM32 fornecem informações fundamentais para a configuração dos pinos GPIO para os vários dispositivos. Para as USARTs, esta informação é reproduzida aqui na Tabela 5.2. Informações mais completas de configuração dos pinos estão disponíveis no datasheet do dispositivo. [15].

Como mencionado anteriormente, as USARTs no STM32 são capazes de suportar um modo operacional adicional — serial síncrono — que requer um sinal de clock separado (USARTx_CK) que não iremos utilizar. Além disso, as USARTs tem a capacidade de suportar “hardware flow control” (sinais USARTx_RTS e USARTx_CTS) (“controle de fluxo por hardware”) que vamos discutir na Seção 11.5. Para comunicações seriais básicas devemos configurar dois pinos – USART1_Tx e USART1_Rx. O primeiro é uma saída dirigida

CAPÍTULO 5. ASYNCHRONOUS SERIAL COMMUNICATION

Pinos USART		Configuração	Configuração GPIO
USARTx_TX	Full Duplex	Alternate function push-pull	
	Half duplex Synchronous mode	Alternate function push-pull	
USARTx_RX	Full Duplex	Input floating/Input Pull-up	
	Half duplex Synchronous mode	Not used. Can be used as General IO	
USARTx_CK	Synchronous mode	Alternate function push-pull	
USARTx_RTS	Hardware flow control	Alternate function push-pull	
USARTx_CTS	Hardware flow control	Input floating/Input pull-up	

Tabela 5.2: USART Pin Configuration

(“driven”) pelo componente USART (uma “alternative function” na documentação do STM32) enquanto a posterior é uma entrada que pode ser configurada como flutuante ou “pulled up”. A configuração dos pinos é realizada com funções e constantes definidas na `stm32f10x_gpio.h`.

5.2. INICIALIZAÇÃO

```
GPIO_InitTypeDef GPIO_InitStruct;  
  
GPIO_StructInit(&GPIO_InitStruct);  
  
// Initialize USART1_Tx  
  
GPIO_InitStruct.GPIO_PIN = GPIO_Pin_9;  
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;  
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;  
GPIO_Init(GPIOA, &GPIO_InitStruct);  
  
// Initialize USART1_RX  
  
GPIO_InitStruct.GPIO_PIN = GPIO_Pin_10;  
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING;  
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

A etapa final de inicialização é configurar o USART. Nós usamos a inicialização USART padrão com 9600 “baud”, 8 bits de dados, 1 bit de parada, sem paridade, e sem controle de fluxo fornecido pelo procedimento da biblioteca `USART_StructInit`. As alterações no default de inicialização são feitas modificando campos específicos da `USART_InitStructure`.¹

```
// see stm32f10x_usart.h  
  
USART_InitTypeDef USART_InitStructure;  
  
// Initialize USART structure  
  
USART_StructInit(&USART_InitStructure);  
  
// Modify USART_InitStructure for non-default values, e.g.  
// USART_InitStructure.USART_BaudRate = 38400;  
  
USART_InitStructure.USART_BaudRate = 9600;  
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;  
USART_Init(USART1, &USART_InitStructure);  
USART_Cmd(USART1, ENABLE);
```

Exercise 5.2 Hello World!

O código nas seções anteriores fornecem toda a funcionalidade básica necessária para usar uma USART da STM32. Nesta seção, vou orientá-lo

¹Como em vários procedimentos da biblioteca, pode ser esclarecedor ler o código fonte - neste caso o módulo `stm32f10x_usart.c`.

CAPÍTULO 5. ASYNCHRONOUS SERIAL COMMUNICATION

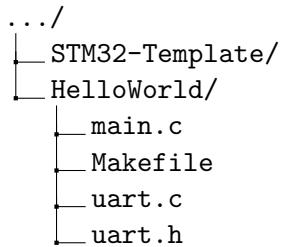


Figura 5.6: Projeto Hello World

```
int uart_open(USART_TypeDef* USARTx, uint32_t baud, uint32_t flags);
int uart_close(USART_TypeDef* USARTx);
int uart_putc(int c, USART_TypeDef* USARTx);
int uart_getc(USART_TypeDef* USARTx);
```

Listing 5.1: Uart Interface

pelo processo de desenvolvimento de uma aplicação simples que envia repetidamente a linha “hello world” a partir do STM32 para um computador host. Este é o primeiro aplicativo que requer realmente fiação de componentes de hardware juntos, por isso vou guia-lo através desse processo. A seguir eu suponho que você tenha acesso a uma ponte USB/UART conforme descrito na Seção 1.1.

Você estará desenvolvendo um programa usando o ambiente de construção fornecido com o exemplo Blinking Light. A estrutura de diretório para o projeto (completo) é ilustrada na Figura 5.6.

Os arquivos `uart.[ch]` fornecem a interface do software básico para a USART1 do STM32. Sua primeira tarefa é implementar as funções especificadas pelo arquivo `uart.h` (Listagem 5.1).

A função `uart_open` deve:

1. Inicializar os clocks da usart/gpio.
2. Configurar os pinos da usart.
3. Configurar e habilitar a USART1

5.2. INICIALIZAÇÃO

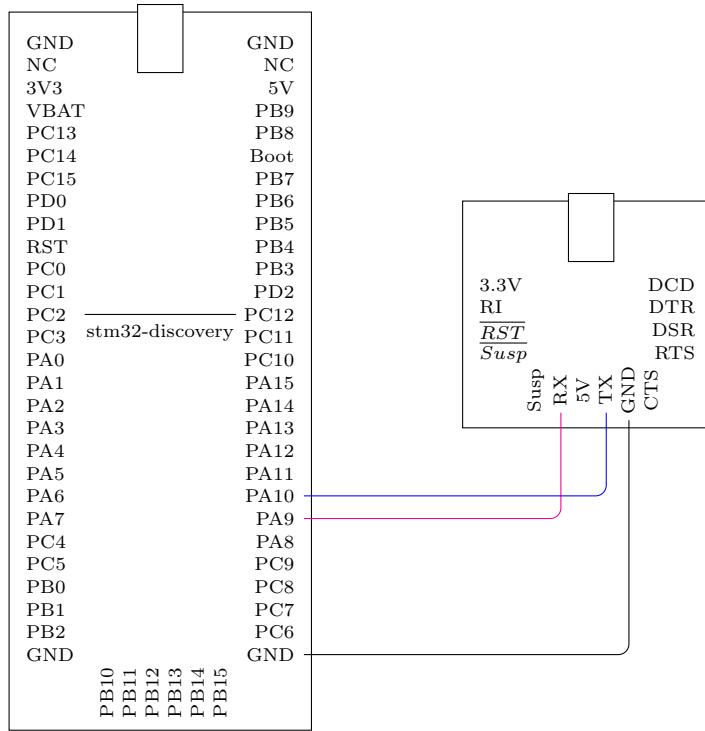


Figura 5.7: Wiring Uart Polling

Você vai precisar do seguinte “include” em seu arquivo `uart.c`:

```
#include <stm32f10x.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_gpio.h>
#include <stm32f10x_usart.h>
#include "uart.h"
```

As funções `uart_putc` e `uart_getc` devem ler ou escrever caracteres individuais, respectivamente. Elas devem se comportar como o `getc` e `putc` padrão do Linux. Seu arquivo `main.c` deve:

1. Inicializar o timer tick.
2. Inicializar a uart.
3. Escrever `Hello World!\n\r''` a cada 250 ms.

CAPÍTULO 5. ASYNCHRONOUS SERIAL COMMUNICATION

Você precisará incluir `stm32f10x.h`, `stm32f10x_usart.h`, e `uart.h` em seu arquivo `main.c`.

Você deve copiar e modificar o `Makefile` do diretório `BlinkingLight`. Você deve adicionar `uart.o` e `stm32f10x_usart.o` à lista OBJS dos arquivos do projeto. Você pode precisar modificar a variável `TEMPLATEROOT` para apontar para o diretório que contém o programa Demo.

Neste ponto, você deve se certificar que seu projeto compila corretamente (digite `make !`). Se a compilação for bem-sucedida, você deve ter um executável chamado “`HelloWorld.elf`”.

Existem apenas três fios que têm de ser ligados como esta mostrado na Figura 5.7. Os sinais aterrados da Discovery Board e da ponte uart precisam estar conectados (estes são rotulados com `gnd`) – há vários pinos de terra na Discovery Board, todos os quais são conectados eletricamente pelo PCB. Assim, apenas um único fio (preto é tradicional) é necessário para ligar os sinais de aterramento. Você precisará conectar o pino de RX (TX) da USART1 ao pino TX (RX) do USB/uart.

Para testar o programa, abra três janelas de terminal. Uma será usada para se comunicar com o adaptador USB/UART, uma segunda será usada para executar o servidor `gdb`, e a terceira para executar o `gbd`.

Inicie o programa `screen` para se comunicar com o adaptador USB/UART. Na janela `gdbserver`, execute

```
st-util -1
```

Finalmente, no terminal `gdb`, navegue até o diretório que contém o seu programa e execute:

```
arm-none-eabi-gdb HelloWorld.elf
(gdb) target extended-remote :4242
(gdb) load
(gdb) continue
```

Com alguma sorte, você será recompensado com “hello world” na janela do programa `screen`. Se não, é hora de alguma depuração séria.

Mesmo que você não encontre problemas, será instrutivo usar o Saleae Logic para observar o comportamento de baixo nível da UART do STM32. Para capturar dados da placa de trabalho, é necessário conectar dois fios do Saleae Logic em seu circuito — terra (cinza) e canal 0 (preto). O fio terra deve ser ligado a um dos pinos GND e o canal 0 ao pino USART1 TX (PA9).

5.2. INICIALIZAÇÃO

Conecte o analisador Saleae Logic em seu computador e configure o software para Async serial no canal 0.

Compile o projeto e faça o download do arquivo executável (siga as instruções da Seção 1.2). Inicie o processo de captura na Saleae Logic, e execute o seu programa. Com sorte, você será recompensado com uma captura de `Hello World\n\r` sendo transmitida. Se nenhum dado for capturado, verifique cuidadosamente se o seu software configurou corretamente os periféricos necessários e, em seguida, usando GDB, tente determinar se o seu código está realmente tentando transmitir a cadeia de caracteres.

Exercise 5.3 Echo

Modifique o seu programa para receber e ecoar a entrada do host. Há duas maneiras de fazer isso -- uma linha de cada vez e um caractere de cada vez (Experimente ambas !). Tente testar seu código usando cat para enviar um arquivo inteiro ao seu programa de uma só vez enquanto captura a saída em outro arquivo. Você pode verificar sua saída usando “diff”. É provável que a entrada e a saída não sejam iguais (especialmente se você executar o echo de uma linha de cada vez). Voltaremos a este problema no Capítulo 11.

Capítulo 6

SPI

O SPI é uma interface serial amplamente utilizada para se comunicar com dispositivos de hardwares comuns, incluindo displays, cartões de memória e sensores. Os processadores STM32 tem várias interfaces SPI e cada uma dessas interfaces pode se comunicar com vários dispositivos. Neste capítulo vamos mostrar como usar o barramento SPI para fazer interface com vários chips de memória EEPROM amplamente disponíveis. Estes chips fornecem memória não-volátil para sistemas embarcados e são usados frequentemente para guardar parâmetros chave e dados de estado. No Capítulo 7 mostraremos como usar o barramento SPI para fazer a interface com um módulo de display LCD colorido e no Capítulo 8 vamos usar a mesma interface para se comunicar com uma memória flash.

6.1 Protocolo

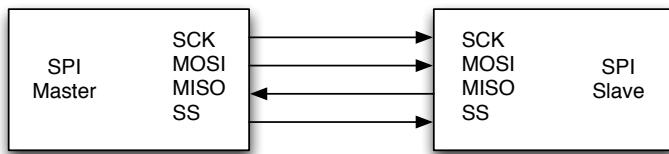


Figura 6.1: Diagrama de Blocos do Protocolo SPI

A interface básica do SPI é mostrada na Figura 6.1. Cada instância de um barramento SPI tem um mestre e um ou mais escravos. Enquanto o STM32 pode ser configurado para qualquer papel, aqui nós consideramos

CAPÍTULO 6. SPI

apenas o caso quando este é configurado como um mestre. Na figura existem quatro sinais de um fio – três do mestre e um do escravo. Um desses sinais, SS (de “seleção de escravo” - “slave select”) deve ser replicado para todos os escravos conectados ao barramento. Toda comunicação é controlada pelo mestre, que seleciona o escravo com que ele deseja se comunicar baixando a linha SS apropriada, e em seguida faz com que uma única palavra (geralmente um byte) seja transferido para o escravo em série sobre o sinal MOSI (“master out, slave in”) e simultaneamente aceita um byte simples do escravo sob o sinal MISO (“master in, slave out”). Esta transferência é realizada por meio da geração de 8 pulsos clock no sinal SCK (“serial clock”).

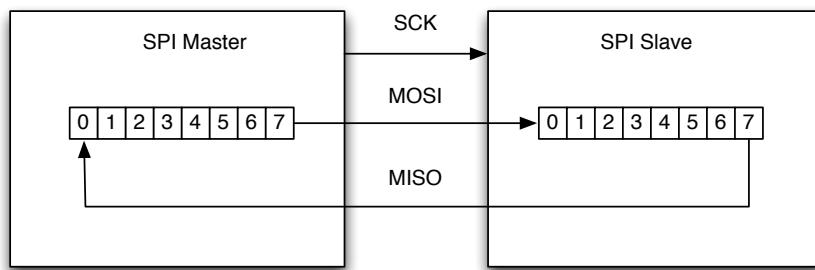


Figura 6.2: Comportamento Logico do Barramento SPI

O mecanismo básico de transferência de dados pode ser visualizado como um par de registradores de deslocamento encadeado como mostrado na Figura 6.2. Nessa figura, os dados são deslocados para fora (in) começando com o bit mais significativo.

O comportamento real do protocolo é mostrado pelo diagrama de temporização na Figura 6.3. Na figura, o mestre inicia a comunicação com um escravo baixando SS. Observe que o sinal MISO se movimenta de um estado de alta impedância (tri-state) uma vez que o escravo é selecionado. O mestre controla a transferência com 8 pulsos no SCLK. Nessa figura, os dados são “temporizados” (“clocked”) no registro de deslocamento mestre/escravo nas bordas crescentes do clock e novos dados são enviados na borda de descida do clock. Infelizmente, a relação específica entre as bordas do clock e os dados de configuração é dependente – existem quatro modos possíveis para o clock; contudo o LCD requer o modo mostrado (comumente referido como CPOL=0,CPHA=0).

6.2. PERIFÉRICOS SPI DO STM32

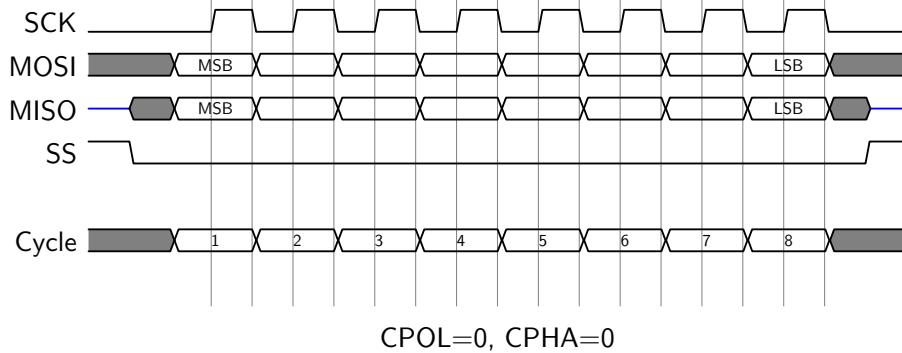


Figura 6.3: Temporização do Protocolo SPI

6.2 Periféricos SPI do STM32

```
enum spiSpeed { SPI_SLOW , SPI_MEDIUM, SPI_FAST };

void spiInit(SPI_TypeDef* SPIx);
int spiReadWrite(SPI_TypeDef* SPIx, uint8_t *rbuf,
                 const uint8_t *tbuf, int cnt,
                 enum spiSpeed speed);
int spiReadWrite16(SPI_TypeDef* SPIx, uint16_t *rbuf,
                   const uint16_t *tbuf, int cnt,
                   enum spiSpeed speed);
```

Listing 6.1: SPI Module Interface

Neste capítulo vamos desenvolver e testar um driver SPI com uma interface simples mostrada na Listagem 6.1. A biblioteca padrão de periféricos define três dispositivos SPI (SPI1, SPI2 e SPI3) e 8 clocks prescalers possíveis no `stm32f10x_spi.h` – para a parte de 24MHz na Discovery Board, um prescaler de 8 resulta no clock 3MHz SPI (24/8). Nós usamos um prescaler (divisores) de 64, 8 e 2 para as velocidades devagar, médio e rápido respectivamente. Esta interface permite inicializar qualquer um destes três dispositivos com uma configuração relativamente genérica. Existe uma operação de transferência de dados que permite a troca de buffers de dados com um dispositivo SPI. Tecnicamente, toda transferência de dado é bidirecional, mas muitos dispositivos não utilizam essa capacidade. Assim, as operações de leitura/escrita

CAPÍTULO 6. SPI

aceitam ponteiros nulos tanto para buffers de recepção quanto de envio. O dispositivo SPI também suporta transferência de 16 bits, daí a nossa interface fornecer uma função de leitura/escrita de 16 bits. Finalmente, a interface permite mudanças on-the-fly na velocidade de transmissão. Tal mudança pode ser necessária se um barramento tem dois escravos com velocidades diferentes.¹

A inicialização dos módulos SPI segue a mesma sequencia geral necessária para qualquer periférico:

1. Ativar o clock para os periféricos e as portas de GPIO associadas.
2. Configurar os pinos de GPIO.
3. Configurar o dispositivo.

Função	SPI1	SPI2	Configuração GPIO
SCK	PA5	PB13	Alternate function push-pull (50MHz)
MISO	PA6	PB14	Input pull-up
MOSI	PA7	PB15	Alternate function push-pull (50MHz)

Tabela 6.1: Configuração dos Pinos de SPI

A configuração necessária para os pinos esta mostrada na Tabela 6.1. Não são mostrados os pinos disponíveis para o controle de hardware da linha de seleção de escravo porque nós implementamos este controle via software. O processo de inicialização esta mostrado na Listagem 6.2. Estamos interessados somente em um modo de operação – o STM32 atuando como um mestre. Em sistemas mais complexos pode ser necessário modificar significativamente este procedimento, por exemplo, passando uma estrutura `SPI_InitStructure`. A rotina de inicialização segue a sequencia mostrada acima e é facilmente estendida para suportar periféricos SPI adicionais.

¹Usamos esta característica no Capítulo 8 onde a velocidade de inicialização e de transferências de blocos são diferentes.

6.2. PERIFÉRICOS SPI DO STM32

```
static const uint16_t speeds[] = {
    [SPI_SLOW] = SPI_BaudRatePrescaler_64,
    [SPI_MEDIUM] = SPI_BaudRatePrescaler_8,
    [SPI_FAST] = SPI_BaudRatePrescaler_2};

void spiInit(SPI_TypeDef *SPIx)
{
    SPI_InitTypeDef SPI_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    GPIO_StructInit(&GPIO_InitStructure);
    SPI_StructInit(&SPI_InitStructure);

    if (SPIx == SPI2) {
        /* Enable clocks, configure pins
         ...
        */
    } else {
        return;
    }

    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = speeds[SPI_SLOW];
    SPI_InitStructure.SPI_CRCPolynomial = 7;
    SPI_Init(SPIx, &SPI_InitStructure);

    SPI_Cmd(SPIx, ENABLE);
}
```

Listing 6.2: Inicialização do SPI

Nosso modulo SPI básico suporta um único tipo de transação – leitura/escrita como mostrado na Listagem 6.3. A rotina de leitura/escrita itera sobre o numero de bytes de dados que devem ser trocados. Cada iteração consiste no envio de um byte, espera a recepção ser concluída, e em seguida o recebimento de um byte. No caso da rotina de somente escrita, um buffer interno é usado para capturar e descartar um byte recebido. Da mesma maneira, a rotina somente leitura transmite uma sequencia de 0xff (efetivamente inativo) bytes enquanto esta recebendo. A rotina 16-bits trabalha modificando temporariamente a configuração para suportar transferência de 16 bits usando

as seguintes chamadas de sistema:

```
SPI_DataSizeConfig(SPIx, SPI_DataSize_16b);
SPI_DataSizeConfig(SPIx, SPI_DataSize_8b);

int spiReadWrite(SPI_TypeDef* SPIx, uint8_t *rbuf,
                 const uint8_t *tbuf, int cnt, enum spiSpeed speed)
{
    int i;

    SPIx->CR1 = (SPIx->CR1 & ~SPI_BaudRatePrescaler_256) |
                  speeds[speed];

    for (i = 0; i < cnt; i++) {
        if (tbuf) {
            SPI_I2S_SendData(SPIx, *tbuf++);
        } else {
            SPI_I2S_SendData(SPIx, 0xff);
        }
        while (SPI_I2S_GetFlagStatus(SPIx, SPI_I2S_FLAG_RXNE) == RESET);
        if (rbuf) {
            *rbuf++ = SPI_I2S_ReceiveData(SPIx);
        } else {
            SPI_I2S_ReceiveData(SPIx);
        }
    }
    return i;
}
```

Listing 6.3: SPI Read/Write

6.3 Testando a Interface SPI

A maneira mais fácil para testar esta interface SPI é conectando-a em modo “loop-back” com MISO diretamente conectado no MOSI e observando a transmissão de dados com o Saleae Logic. Como o Saleae Logic espera um sinal de seleção (SS), o programa de teste deve configurar e controlar o sinal de seleção explicitamente. Na seção seguinte, vamos mostrar como usar a interface SPI para controlar uma EEPROM externa; para esse exemplo usaremos o PC10 como o pino de seleção (para este exemplo de loopback estamos usando o PC3), por isso é conveniente usar o mesmo pino aqui.

A rotina principal de um programa de teste simples é mostrada na Listagem 6.4. Observe que ela testa tanto o modo 8-bits quanto o 16-bits. Um trecho da saída do Saleae Logic para este programa é mostrado na Figura 6.4.

6.3. TESTANDO A INTERFACE SPI

```
uint8_t txbuf[4], rdbuf[4];
uint16_t txbuf16[4], rdbuf16[4];

void main()
{
    int i, j;

    csInit(); // Initialize chip select PC03
    spiInit(SPI2);

    for (i = 0; i < 8; i++) {
        for (j = 0; j < 4; j++)
            txbuf[j] = i*4 + j;
        GPIO_WriteBit(GPIOC, GPIO_Pin_3, 0);
        spiReadWrite(SPI2, rdbuf, txbuf, 4, SPI_SLOW);
        GPIO_WriteBit(GPIOC, GPIO_Pin_3, 1);
        for (j = 0; j < 4; j++)
            if (rdbuf[j] != txbuf[j])
                assert_failed(__FILE__, __LINE__);
    }
    for (i = 0; i < 8; i++) {
        for (j = 0; j < 4; j++)
            txbuf16[j] = i*4 + j + (i << 8);
        GPIO_WriteBit(GPIOC, GPIO_Pin_3, 0);
        spiReadWrite16(SPI2, rdbuf16, txbuf16, 4, SPI_SLOW);
        GPIO_WriteBit(GPIOC, GPIO_Pin_3, 1);
        for (j = 0; j < 4; j++)
            if (rdbuf16[j] != txbuf16[j])
                assert_failed(__FILE__, __LINE__);
    }
}
```

Listing 6.4: Teste de Loopback da SPI

Exercice 6.1 SPI Loopback

Complete o teste loop back e grave a saída resultante com o Saleae Logic. Preste atenção especial para o “byte order” para a transferência de 16-bit. Compare com o byte order para transferência de 8-bit. Você deve usar SPI2 e SPI3 como pino de seleção. Um modelo para este projeto é mostrado na Figura 6.5

CAPÍTULO 6. SPI

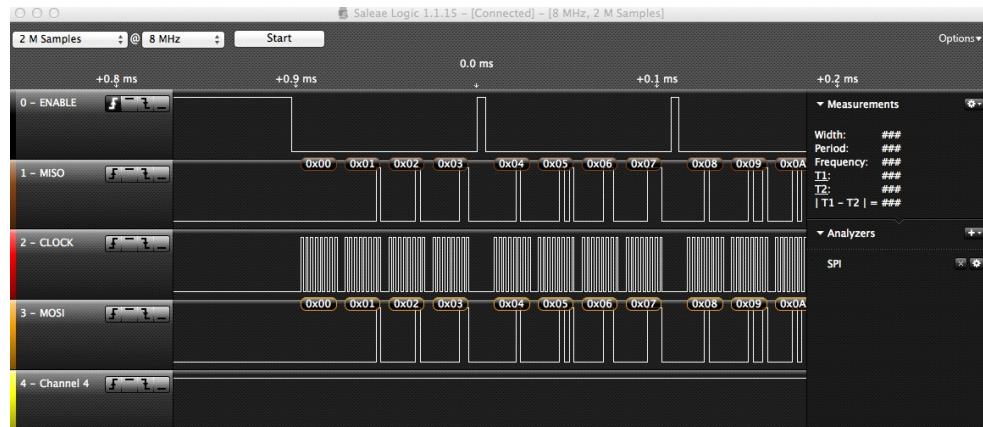


Figura 6.4: Saída do Teste de Loopback da SPI

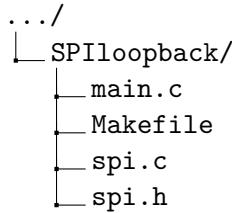


Figura 6.5: Projeto SPI Loopback

6.4 Interface EEPROM

Um dos mais simples e úteis dispositivos SPI é uma EEPROM serial (Electrically Erasable Programmable Memory – memória programável apagável eletricamente) que é usada frequentemente em dispositivos embarcados como um pequeno espaço não volátil para parâmetros de configuração e estado persistente (p. ex.; pontuação mais alta em um jogo). A retenção de dados para EEPROMs é medido em décadas. Uma limitação é que eles se mantêm apenas um numero limitado de ciclos de escrita (talvez um milhão) e, portanto devem ser usados para mudança de informações relativamente lentas. EEPROMs SPI estão disponíveis através de muitos fabricantes com interfaces comuns. As capacidades variam desde $< 1K$ bit a $> 1M$ bit.

Nossa escolha de uma EEPROM para o primeiro exemplo SPI foi feita através de uma consideração – a operação básica é intuitiva (leitura/escrita) e, portanto é relativamente fácil para determinar se o código da interface esta funcionando corretamente. Na seção anterior usamos um circuito de dados

6.4. INTERFACE EEPROM

loopback e o Saleae Logic para garantir que o nosso código da interface SPI estava operando como esperado. É hora de usar o código para fazer uma aplicação simples.

Na discussão a seguir usamos um componente da Microchip 25LC160 que faz parte da série de dispositivos que variam de 1Kbit por \$0.50 (25x010) a 1Mbit por \$3.50 (25x1024). Para este exemplo, o tamanho real é irrelevante. Nós usamos o empacotamento comum PDIP mostrado na Figura 6.6.

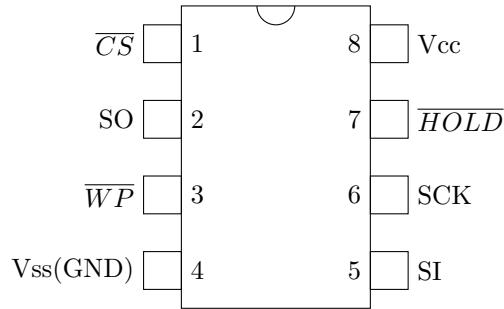


Figura 6.6: Pinagem PDIP

Os dispositivos 25xxxx implementam um protocolo simples de mensagens sob o barramento SPI, onde cada transação consiste de um ou mais envios de bytes de instruções para o dispositivo seguido de um ou mais bytes de dados para ou do dispositivo. [6] Por exemplo, a EEPROM contém um registro de estado que pode ser usado para definir vários modos de proteção bem como determinar se o dispositivo está ocupado – dispositivo EEPROM leva um longo tempo para completar operações de escrita! O status da transação de leitura é mostrada na Figura 6.7 – não é mostrado o clock SPI que completa 16 ciclos durante esta transação. O mestre (STM32) seleciona a EEPROM baixando o SS, que em seguida transmite o código de 8bits do RDSR (0x05) e recebe o valor de status de 8-bits.

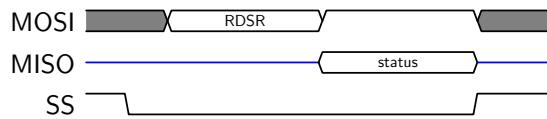


Figura 6.7: Read Status

CAPÍTULO 6. SPI

Um comando para fazer leitura de status pode ser facilmente implementado:

6.4. INTERFACE EEPROM

```

uint8_t eepromReadStatus() {
    uint8_t cmd[] = {cmdRDSR, 0xff};
    uint8_t res[2];
    GPIO_WriteBit(EEPROM_PORT, EEPROM_CS, 0);
    spiReadWrite(EEPROM_SPI, res, cmd, 2, EEPROM_SPEED);
    GPIO_WriteBit(EEPROM_PORT, EEPROM_CS, 1);
    return res[1];
}

```

O formato do registro de status é ilustrado na Figura 6.8. Temos somente dois bits para nos preocupar no momento – WIP (Write In Progress) e WEL (Write Enable Latch). BP[1:0] são blocos de bits de proteção que, se configurados, protegem partes da EEPROM para gravação. O bit WIP é especialmente importante – se for configurado, então uma gravação está em progresso e todos os outros comandos de RDSR vão falhar. Como veremos é importante esperar esse bit ser limpo para fazer qualquer outra ação. O bit WEL deve ser definido em ordem para realizar uma gravação de dados e depois ser resetado automaticamente por qualquer gravação – depois vamos ver como é definido esse bit.

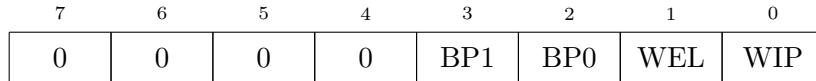


Figura 6.8: EEPROM Status Register

Instrução	Descrição	Código
WRSR	Write Status Register	0x01
WRITE	Data Write	0x02
READ	Data Read	0x03
WRDI	Write Disable	0x04
RDSR	Read Status Register	0x05
WREN	Write Enable	0x06

Tabela 6.2: EEPROM Instructions

Todos os comandos de definição disponíveis estão mostrados na Tabela 6.2. Que codificamos como:

```

enum eepromCMD { cmdREAD = 0x03, cmdWRITE = 0x02
                 cmdWREN = 0x06, cmdWRDI = 0x04,
                 cmdRDSR = 0x05, cmdWRSR = 0x01 };

```

CAPÍTULO 6. SPI

Os comandos mais simples são WRDI (WRIte DIable) e WREN (WRIte ENable) que limpa e define o bit WEL do registrador de estados, respectivamente. Cada um é implementado escrevendo um único byte – o comando – na EEPROM. Nenhum dado é retornado. Lembre-se que ambos os comandos irão falhar se a EEPROM estiver ocupada. Por exemplo, implementamos uma EEPROM enable como mostrado a seguir (observe o pooling do registrador de status!):

```
#define WIP(x) ((x) & 1)

void eepromWriteEnable(){
    uint8_t cmd = cmdWREN;

    while (WIP(eepromReadStatus()));

    GPIO_WriteBit(EEPROM_PORT, EEPROM_CS, 0);
    spiReadWrite(EEPROM_SPI, 0, &cmd, 1, EEPROM_SPEED);
    GPIO_WriteBit(EEPROM_PORT, EEPROM_CS, 1);
}
```

As instruções mais difíceis para implementar são leitura e gravação – ambas podem ler ou gravar mais de um byte. É importante observar que estas operações devem se comportar diferentemente para vários tamanhos de EEPROMs. Para os EEPROMS maiores, tal como o 25LC160, o endereço é transmitido como um par sucessivo de bytes, enquanto que para o 25AA040, o bit mais significativo (8) de endereço é codificado junto com o comando. Aqui assumimos um endereço de 16 bits como mostrado para a operação leitura na Figura 6.9. Uma operação de leitura pode acessar qualquer número de bytes – incluindo toda a EEPROM. A operação começa com o código de instrução, dois bytes de endereço (byte mais significativo primeiro!) e em seguida uma ou mais operações de leitura. A transmissão do endereço é mais convenientemente tratada como uma transferência de 16 bits – se realizada com duas transferências 8 bits do endereço terá que ter seus bytes trocados.

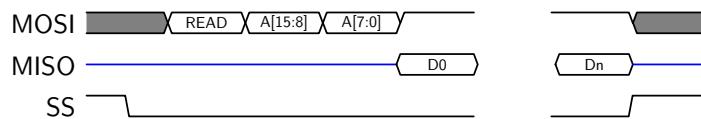


Figura 6.9: EEPROM Read

6.4. INTERFACE EEPROM

A operação gravação é mais complicada por causa da arquitetura de uma EEPROM. A matriz de memoria é organizada em “paginas” e é possível escrever qualquer ou todos os bytes em uma pagina em uma única operação (é mais eficiente gravar em todas as paginas simultaneamente). O tamanho da pagina é específico do dispositivo – o 25LC160 tem paginas de 16 bytes enquanto os maiores podem ter paginas de 64 bytes. Não é possível gravar bytes em mais de uma pagina em uma única operação. Na prática qualquer byte gravado além da pagina selecionada pelos bits endereço A[16:5] envolverá bytes espalhados na pagina. Assim, o código EEPROM de gravação deve verificar os limites da pagina. A implementação mais fácil é retornar um contador de bytes gravados e limitar a escrita de bytes para uma única pagina. Alguns dispositivos limitam leitura para limites de pagina também – é importante a leitura do data sheet do dispositivo que você planeja usar!

```
void eepromInit();
void eepromWriteEnable();
void eepromWriteDisable();
uint8_t eepromReadStatus();
void eepromWriteStatus(uint8_t status);
int eepromWrite(uint8_t *buf, uint8_t cnt, uint16_t offset);
int eepromRead(uint8_t *buf, uint8_t cnt, uint16_t offset);
```

Listing 6.5: EEPROM Module

Exercice 6.2 Write e Teste de um Módulo EEPROM

Implemente um modulo EEPROM com a interface mostrada na Listagem 6.5. A conexão para o chip EEPROM é dada na Tabela 6.3. Você deve usar SPI2 na velocidade lenta (a maioria das EEPROMS é razoavelmente lenta). O uso de um analisador logico será essencial para depurar qualquer erro que surja.

Escreva um programa para realizar um teste de leitura/gravação em locais individuais bem como em um bloco de leitura/gravação.

CAPÍTULO 6. SPI

EEPROM Pin	EEPROM Signal	STM32 Pin
1	\overline{CS}	PC10
2	\overline{SO}	PB14
3	\overline{WP}	3V3
4	VSS	GND
5	SI	PB15
6	SCK	PB13
7	$HOLD$	3V3
8	VCC	3V3

Tabela 6.3: EEPROM Connections

Capítulo 7

SPI : Display LCD

Neste capítulo, consideramos o uso da interface SPI do STM32 para comunicar com um módulo de exibição LCD. O LCD é uma tela de exibição de 1.8" TFT com 128x160 pixels e 18 bits de cores, suportada por um de chip comum (ST7735R).

7.1 Módulo LCD a Cores

O módulo de LCD que consideramos utiliza o controlador ST7735R.

¹. Referimo-nos ao display como 7735 LCD. O 7735 LCD é um display endereçado por pixel; cada pixel requer múltiplos bytes para definir a cor – a memória interna do display 18-bits/pixel (6 bits para cada cor: vermelha, azul e verde). Usaremos este display em modo de 16 bits com 5 bits definindo o vermelho, 6 bits definindo o verde, e 5 bits definindo o azul. O controlador do display extrapola automaticamente de 16 bits para 18 bits quando os pixels são escritos no display. Este modelo de cores é mostrado na Figura 7.1. Existem duas partes nesta figura, o layout de cores separadas dentro de uma palavra de 16-bit e cores resultantes de várias constantes de 16 bits.

Para entender a interface necessária para o 7735 LCD, considere a modelo da Figura 7.1. Existem três componentes principais a considerar – o controlador, o painel LCD e a RAM do display. A RAM do display contém 18 bits de informação de cor para cada um dos (128 x 160) pixels do painel.

² Os dados na RAM do display são continuamente transferidos para o pai-

¹Existem duas variantes do ST7735 – ao inicializar o controlador é importante utilizar o modelo correto!

²Na realidade 132x162, mas configuramos o controlador para um número menor.

CAPÍTULO 7. SPI : DISPLAY LCD

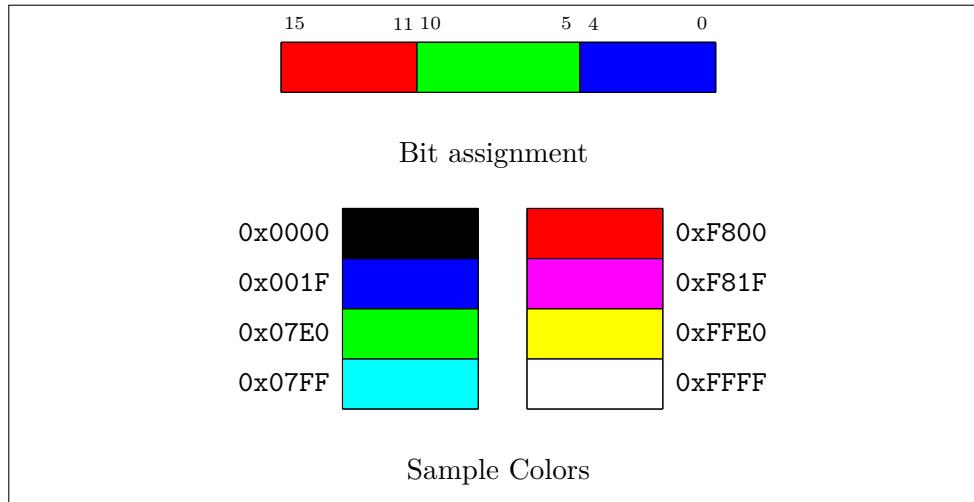


Figura 7.1: Color Encoding

nel – configuramos o dispositivo para varrer de baixo para cima. O modelo usado para registrar os dados do pixel (cor) é um tanto indireta. Primeiro, utilizando uma interface de “controle” separada, é configurado um desenho de um retângulo. Em seguida, os dados de pixel (cor) são gravados. O local onde os dados são gravados é determinado por um par de endereço interno denominados contadores RAC (de Row Address Counter - linha) e CAC (de Column Address Counter - coluna). Cada gravação sucessiva de pixel faz com que esses endereços sejam atualizados. O 7735 pode ser configurado para realizar uma varredura (“sweep”) no retângulo tanto dos lados direito/esquerdo quanto na de baixo/cima. Existem três bits de controle interno (que expomos na interface descrita mais adiante). Além de realizar uma varredura, é possível promover “troca” (“exchange”) de endereço de linha e coluna, e assim suportar o modo “paisagem”. Por default, usamos o modo 0x6 (MY = 1, MX = 1, MV = 0). Estes são descritos integralmente no manual de dados da ST7735. [11]

MY Ordem do endereço da linha: 1 (bottom to top), 0 (top to bottom)

MX Ordem do endereço da coluna: 1 (right to left), 0 (left to right)

MV Troca Coluna/Linha: 1 (landscape mode), 0 (portrait mode)

7.1. MÓDULO LCD A CORES

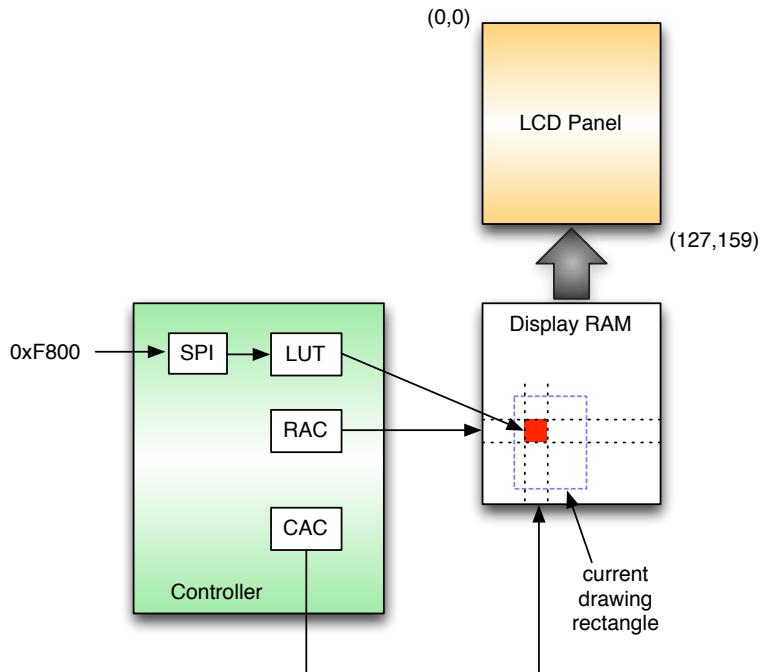


Figura 7.2: Model of 7735 Display

Como mencionado anteriormente, configuramos o painel para aceitar cores de 16 bits, de modo que cada registro de dados consiste em um par de bytes enviado da interface SPI. Estes dados de cor de 16 bits são automaticamente transformados para 18-bits através de uma tabela interna de pesquisa (LUT); os pixels reais da RAM da tela são de 18-bits.

O 7735 tem um sinal de controle separado para diferenciar a informação de “controle” da informação de “data”. O Controle da Informação (a ser discutido mais adiante) é utilizado para configurar o display e para definir o retângulo de desenho. A vantagem desta abordagem é que uma vez que o retângulo é configurado, os dados podem ser enviados de modo rápido sem nenhum controle adicional da informação.

Um “driver” simples para o 7735 LCD requer apenas três rotinas — uma para inicializar o controlador, uma para definir o retângulo de desenho, e uma para escrever dados coloridos para o retângulo atual. Observe que esta interface define a direção de desenho quando é configurado o retângulo (tal como descrito acima, o valor `madctl1` de 0x6 corresponde a uma varre-

CAPÍTULO 7. SPI : DISPLAY LCD

dura top-down/left-right (cima-baixo/esquerda-direita) do retângulo. Um valor `madctl` de 0x02 corresponde a uma varredura bottom-up/left-right (base-topo/esquerda-direita) e é útil para exibir arquivos de imagem BMP, onde os dados são armazenados na ordem base-topo. Uma rotina separada pode ser adicionada para controlar a luminosidade da luz de fundo – por agora vamos implementar isso como on/off. Mais tarde, no Capítulo 10, mostramos como controlar o brilho com um sinal PWM.

7.1. MÓDULO LCD A CORES

```
#define MADCTLGRAPHICS 0x6
#define MADCTLBMP      0x2

#define ST7735_width   128
#define ST7735_height  160

void ST7735_setAddrWindow(uint16_t x0, uint16_t y0,
                           uint16_t x1, uint16_t y1, uint8_t madctl);
void ST7735_pushColor(uint16_t *color, int cnt);
void ST7735_init();
void ST7735_backLight(uint8_t on);
```

O exemplo a seguir preenche a tela 7735 com uma única cor de fundo.

```
void fillScreen(uint16_t color)
{
    uint8_t x,y;
    ST7735_setAddrWindow(0, 0, ST7735_width-1, ST7735_height-1,
                         →MADCTLGRAPHICS);
    for (x=0; x < ST7735_width; x++) {
        for (y=0; y < ST7735_height; y++) {
            ST7735_pushColor(&color,1);
        }
    }
}
```

Embora esta interface possa ser mais sofisticada, é suficiente para a exibição de texto, imagens e gráficos com precisão e com simplicidade de programação.

Uma complicação que deve ser abordado é endianess. A memória STM32 é do tipo little-endian o que significa que os 16-bits são armazenados em locais de memória com o byte de baixa-ordem (bits 0-7) no endereço de memória menor. Em contrapartida, a interface 7735 assume que os dados são recebidos em ordem big-endian (o byte de alta-ordem em primeiro lugar). Portanto, é essencial que os dados de cor sejam transmitidos através das funções de transferência de 16-bit; blocos de transferência de cor com transferências de dados de 8 bits irá resultar na troca dos bytes de alta-ordem e baixa-ordem!

A implementação da funcionalidade básica para o ST7735 depende de um pequeno conjunto interno de primitivas para fornecer o acesso à interface SPI e manusear controle de “chip select”. Eles estão mostrados na Listagem 7.1. Existem dois comandos de dados para transferências de 8-bit e 16-bit, e um comando de controle. Os comandos de interface publica

CAPÍTULO 7. SPI : DISPLAY LCD

são mostrados na Listagem 7.2. Constantes tais como ST7735_CASET (Column Address Set - seta o endereço da coluna) e ST7735_RASET (Row Address Set - seta o endereço da linha), são definidas no datasheet do ST7735 [11], embora o nosso código seja derivado a partir do código disponível em <https://github.com/adafruit/Adafruit-ST7735-Library.git>.

```
static void LcdWrite(char dc, const char *data, int nbytes)
{
    GPIO_WriteBit(LCD_PORT,GPIO_PIN_DC, dc); // dc 1 = data, 0 =
    ↪control
    GPIO_ResetBits(LCD_PORT,GPIO_PIN_SCE);
    spiReadWrite(SPILCD, 0, data, nbytes, LCDSPEED);
    GPIO_SetBits(LCD_PORT,GPIO_PIN_SCE);
}

static void LcdWrite16(char dc, const uint16_t *data, int cnt)
{
    GPIO_WriteBit(LCD_PORT,GPIO_PIN_DC, dc); // dc 1 = data, 0 =
    ↪control
    GPIO_ResetBits(LCD_PORT,GPIO_PIN_SCE);
    spiReadWrite16(SPILCD, 0, data, cnt, LCDSPEED);
    GPIO_SetBits(LCD_PORT,GPIO_PIN_SCE);
}

static void ST7735_writeCmd(uint8_t c)
{
    LcdWrite(LCD_C, &c, 1);
}
```

Listing 7.1: Primitivas Internas do ST7735

A inicialização do ST7735 é um pouco complexa, devido à necessidade para inicializar registradores internos, além de pinos e fontes de clock. É melhor copiar a sequência de comandos de inicialização de um software existente (e ter o cuidado de usar o código de inicialização do ST7735R!). O processo básico de inicialização requer o envio de uma série de comandos intercalada com atrasos. Definimos uma estrutura de dados para realizar essas etapas de inicialização (mostrado na Listagem 7.3). O código de inicialização real é mostrado na Listagem 7.4. Detalhes que faltam podem ser adquiridos a partir do código da biblioteca do qual o nosso código é derivado – os comandos de inicialização suprimidos são mostrados na Listagem 7.5.

A conexão elétrica para o módulo de LCD é relativamente simples, tal como mostrado na Figura 7.1 – o sinal de chip select para o cartão SD é deixado

7.1. MÓDULO LCD A CORES

desconectado por enquanto. A Tabela 7.1 resume as conexões necessárias (os nomes das constantes usados no código de exemplo são mostrados entre parênteses).

Pino TFT	Pino STM32	Função
VCC	5V	Power – 5 Volts
BKL	PA1	Backlight control (GPIO_PIN_BKL)
RESET	PC1	LCD Reset (GPIO_PIN_RST)
RS	PC2	Data/Control (GPIO_PIN_DC)
MISO	PB14	SPI2 MISO
MOSI	PB15	SPI2 MOSI
SCLK	PB13	SPI2 CLK
LCD CS	PC0	LCD select (GPIO_PIN_SCE)
SD_CS	PC6	SD card select
GND	GND	Ground

Figura 7.3: TFT Pin Assignment

Exercise 7.1 Código Completo da Interface

Complete o código para o driver do ST7335 examinando o código de referência em <https://github.com/adafruit/Adafruit-ST7735-Library>. Você vai precisar completar a estrutura de dados do código de inicialização. Para testar o seu código, escreva um programa que mostra as três cores primárias em um ciclo, com um adequado atraso.

Exercise 7.2 Exibindo Textos

Um uso importante para um LCD é exibir mensagens de log do seu código. A primeira exigência é exibir caracteres. O código de referência citado acima inclui uma fonte bit-mapped `glcdfont.c` que define os caracteres ASCII como mapas de bits 5x7 (cada caractere gerado é colocado em um retângulo 6x10 deixando espaço entre as linhas (3 pixels) e entre caracteres (1 pixel). Um fragmento desta fonte é mostrado na Figura 7.1. ASCII 0 é o caractere NULL e, portanto, não é exibido. Muitos dos números iniciais dos caracteres ASCII não são imprimíveis e portanto são deixados em branco ou preenchido com um símbolo padrão. Considere o valor do caractere 'A' (ASCII 65) também mostrado na figura.

Faça uma programa para exibir um único caractere em um local específico definido pelo canto superior esquerdo do caractere – lembre-se que (0,0) é

CAPÍTULO 7. SPI : DISPLAY LCD

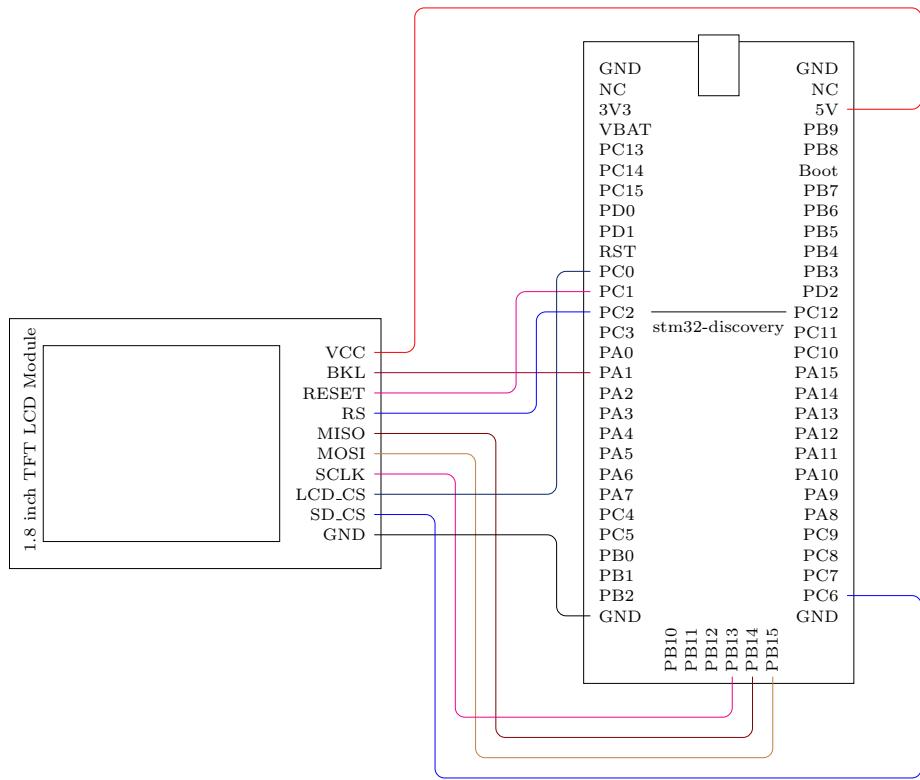


Figura 7.4: Wiring for TFT Module

o canto superior esquerdo da tela e (127,159) é o canto inferior direito. Escrever um caractere requer a escrita de uma cor de primeiro plano (foreground) para cada pixel “on” e uma cor de fundo (background) para cada pixel “off”. Cada caractere deve ocupar uma região de 6x10. É muito mais rápido escrever um bloco para o LCD do que um pixel de cada vez (especialmente quando introduzirmos o DMA).

Estenda seu programa para suportar a escrita de linhas de textos na tela – considere como você vai tratar a sobreposição (wrap).

Exercise 7.3 Graficos

Escreva rotinas para desenhar linhas e círculos de vários tamanhos e cores.

7.1. MÓDULO LCD A CORES

```
#include <stdint.h>

const uint8_t ASCII[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, // 0
    0x3E, 0x5B, 0x4F, 0x5B, 0x3E, // 1
    ...
    0x7C, 0x12, 0x11, 0x12, 0x7C, // 65 A
    0x7F, 0x49, 0x49, 0x49, 0x36, // 66 B
    0x3E, 0x41, 0x41, 0x41, 0x22, // 67 C
    ...
}
```

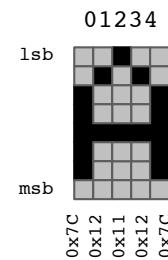


Figura 7.5: Font Fragment

CAPÍTULO 7. SPI : DISPLAY LCD

```
#define LOW 0
#define HIGH 1

#define LCD_C LOW
#define LCD_D HIGH

#define ST7735_CASET 0x2A
#define ST7735_RASET 0x2B
#define ST7735_MADCTL 0x36
#define ST7735_RAMWR 0x2C
#define ST7735_RAMRD 0x2E
#define ST7735_COLMOD 0x3A

#define MADVAL(x) (((x) << 5) | 8)
static uint8_t madctlcurrent = MADVAL(MADCTLGRAPHICS);

void ST7735_setAddrWindow(uint16_t x0, uint16_t y0,
                           uint16_t x1, uint16_t y1, uint8_t madctl)
{
    madctl = MADVAL(madctl);
    if (madctl != madctlcurrent){
        ST7735_writeCmd(ST7735_MADCTL);
        LcdWrite(LCD_D, &madctl, 1);
        madctlcurrent = madctl;
    }
    ST7735_writeCmd(ST7735_CASET);
    LcdWrite16(LCD_D, &x0, 1);
    LcdWrite16(LCD_D, &x1, 1);

    ST7735_writeCmd(ST7735_RASET);
    LcdWrite16(LCD_D, &y0, 1);
    LcdWrite16(LCD_D, &y1, 1);

    ST7735_writeCmd(ST7735_RAMWR);
}

void ST7735_pushColor(uint16_t *color, int cnt)
{
    LcdWrite16(LCD_D, color, cnt);
}

void ST7735_backLight(uint8_t on)
{
    if (on)
        GPIO_WriteBit(LCD_PORT_BKL, GPIO_PIN_BKL, LOW);
    else
        GPIO_WriteBit(LCD_PORT_BKL, GPIO_PIN_BKL, HIGH);
}
```

Listing 7.2: Interface ST7735

7.1. MÓDULO LCD A CORES

```
struct ST7735_cmdBuf {
    uint8_t command;      // ST7735 command byte
    uint8_t delay;        // ms delay after
    uint8_t len;          // length of parameter data
    uint8_t data[16];     // parameter data
};

static const struct ST7735_cmdBuf initializers[] = {
    // SWRESET Software reset
    { 0x01, 150, 0, 0},
    // SLPOUT Leave sleep mode
    { 0x11, 150, 0, 0},
    // FRMCTR1, FRMCTR2 Frame Rate configuration -- Normal mode, idle
    // frame rate = fosc / (1 x 2 + 40) * (LINE + 2C + 2D)
    { 0xB1, 0, 3, { 0x01, 0x2C, 0x2D } },
    { 0xB2, 0, 3, { 0x01, 0x2C, 0x2D } },
    // FRMCTR3 Frame Rate configuration -- partial mode
    { 0xB3, 0, 6, { 0x01, 0x2C, 0x2D, 0x01, 0x2C, 0x2D } },
    // INVCTR Display inversion (no inversion)
    { 0xB4, 0, 1, { 0x07 } },
    /* ... */
}
```

Listing 7.3: ST7735 Initialization Commands (Abbreviated)

CAPÍTULO 7. SPI : DISPLAY LCD

```
void ST7735_init()
{
    GPIO_InitTypeDef GPIO_InitStructure;
    const struct ST7735_cmdBuf *cmd;

    // set up pins
    /* ... */

    // set cs, reset low

    GPIO_WriteBit(LCD_PORT,GPIO_PIN_SCE, HIGH);
    GPIO_WriteBit(LCD_PORT,GPIO_PIN_RST, HIGH);
    Delay(10);
    GPIO_WriteBit(LCD_PORT,GPIO_PIN_RST, LOW);
    Delay(10);
    GPIO_WriteBit(LCD_PORT,GPIO_PIN_RST, HIGH);
    Delay(10);

    // Send initialization commands to ST7735

    for (cmd = initializers; cmd->command; cmd++)
    {
        LcdWrite(LCD_C, &(cmd->command), 1);
        if (cmd->len)
            LcdWrite(LCD_D, cmd->data, cmd->len);
        if (cmd->delay)
            Delay(cmd->delay);
    }
}
```

Listing 7.4: ST7735 Initialization

7.2. INFORMAÇÕES SOBRE COPYRIGHT

7.2 Informações sobre Copyright

Nosso código para o ST7335 é derivado de um módulo disponível em
<https://github.com/adafruit/Adafruit-ST7735-Library.git>

O seguinte copyright se aplica ao código:

```
*****
This is a library for the Adafruit 1.8" SPI display.
This library works with the Adafruit 1.8" TFT Breakout w/SD card
----> http://www.adafruit.com/products/358
as well as Adafruit raw 1.8" TFT display
----> http://www.adafruit.com/products/618

Check out the links above for our tutorials and wiring diagrams
These displays use SPI to communicate, 4 or 5 pins are required to
interface (RST is optional)
Adafruit invests time and resources providing this open source code,
please support Adafruit and open-source hardware by purchasing
products from Adafruit!

Written by Limor Fried/Ladyada for Adafruit Industries.
MIT license, all text above must be included in any redistribution
******/
```

7.3 Comandos de Inicialização (Restantes)

CAPÍTULO 7. SPI : DISPLAY LCD

```
// PWCTR1 Power control -4.6V, Auto mode
{ 0xC0, 0, 3, { 0xA2, 0x02, 0x84 } },
// PWCTR2 Power control VGH25 2.4C, VGSEL -10, VGH = 3 * AVDD
{ 0xC1, 0, 1, { 0xC5 } },
// PWCTR3 Power control, opamp current smal, boost frequency
{ 0xC2, 0, 2, { 0x0A, 0x00 } },
// PWCTR4 Power control, BLK/2, opamp current small and medium low
{ 0xC3, 0, 2, { 0x8A, 0x2A } },
// PWRCTR5, VMCTR1 Power control
{ 0xC4, 0, 2, { 0x8A, 0xEE } },
{ 0xC5, 0, 1, { 0x0E } },
// INVOFF Don't invert display
{ 0x20, 0, 0, 0 },
// Memory access directions. row address/col address, bottom to
// →top refesh (10.1.27)
{ ST7735_MADCTL, 0, 1, { MADVAL(MADCTLGRAPHICS) } },
// Color mode 16 bit (10.1.30
{ ST7735_COLMOD, 0, 1, { 0x05 } },
// Column address set 0..127
{ ST7735_CASET, 0, 4, { 0x00, 0x00, 0x00, 0x7F } },
// Row address set 0..159
{ ST7735_RASET, 0, 4, { 0x00, 0x00, 0x00, 0x9F } },
// GMCTR1 Gamma correction
{ 0xE0, 0, 16, { 0x02, 0x1C, 0x07, 0x12, 0x37, 0x32, 0x29, 0x2D,
                  0x29, 0x25, 0x2B, 0x39, 0x00, 0x01, 0x03, 0x10 } },
// GMCTR2 Gamma Polarity corrction
{ 0xE1, 0, 16, { 0x03, 0x1d, 0x07, 0x06, 0x2E, 0x2C, 0x29, 0x2D,
                  0x2E, 0x2E, 0x37, 0x3F, 0x00, 0x00, 0x02, 0x10 } },
// DISPON Display on
{ 0x29, 100, 0, 0 },
// NORON Normal on
{ 0x13, 10, 0, 0 },
// End
{ 0, 0, 0, 0 }
};
```

Listing 7.5: Comandos de Inicialização do ST7735 (Restantes)

Capítulo 8

Cartões de Memória SD

Neste capítulo vamos mostrar como fazer a interface de um cartão de memória SD comercial com um STM32 VL Discovery Board usando um periférico SPI como discutido no Capítulo 6. Enquanto a comunicação com um cartão de memória SD é uma simples extensão do trabalho anteriormente apresentado, controlar o cartão e interpretar os dados comunicados requer uma quantidade adicional de software. Felizmente, muito do software necessário está disponível no amplamente utilizado módulo FatFS [3]. Apenas uma quantidade modesta de portabilidade é necessária para utilizar este módulo com o nosso driver SPI.

Os processadores STM32 na VL Discovery Board são relativamente limitados em memória – 128K bytes de flash e 8K bytes de RAM – o que limita a capacidade de armazenar grandes quantidades de dados tanto como entrada ou como saída a partir de um programa embarcado. Por exemplo, numa aplicação de jogos pode ser desejável o acesso de arquivos de som e gráficos ou em uma aplicação de data logging, para armazenar extensa quantidade de dados. Além disso, o acesso ao conteúdo da flash do STM32 requer uma interface e software especiais. Em tais aplicações é desejável fornecer armazenamento externo com o qual o STM32 possa acessar durante a execução e que o usuário/programador possa acessar facilmente em outros momentos. Memória comerciais flash (em especial, cartões SD) fornecem uma solução de baixo custo que pode razoavelmente e facilmente serem acessados tanto pelo processador quanto pelo usuário. Na prática, estes cartões têm sistemas de arquivos (tipicamente FAT) e podem ser inseridos em adaptadores comuns e serem acessados por um PC. E também, a interface física tem um modo SPI, que é acessível através do código descrito no Capítulo 6.

Fisicamente, cartões de memória SD consistem em um array (vetor)

CAPÍTULO 8. CARTÕES DE MEMÓRIA SD

de memória flash e um processador de controle que se comunica com um host por um barramento SD (paralelo) ou por um barramento SPI. A comunicação é baseada em transação – o host envia uma mensagem de comando para o cartão SD, e recebe uma resposta. O acesso à memória flash de um cartão SD é realizada através leituras e escritas de blocos de tamanho fixo, e que também são implementadas com o protocolo de comando. Uma visão geral é fornecida em [4].

Os dados em um cartão SD são organizados como um sistema de arquivos – cartões abaixo 2GB são normalmente formatados como sistemas de arquivos FAT16. Em um sistema de arquivos FAT, os primeiros blocos de armazenamento são usados para manter os dados sobre o sistema de arquivos – por exemplo em tabelas de alocação – enquanto que os blocos restantes são usados para armazenar o conteúdo de arquivos e diretórios.

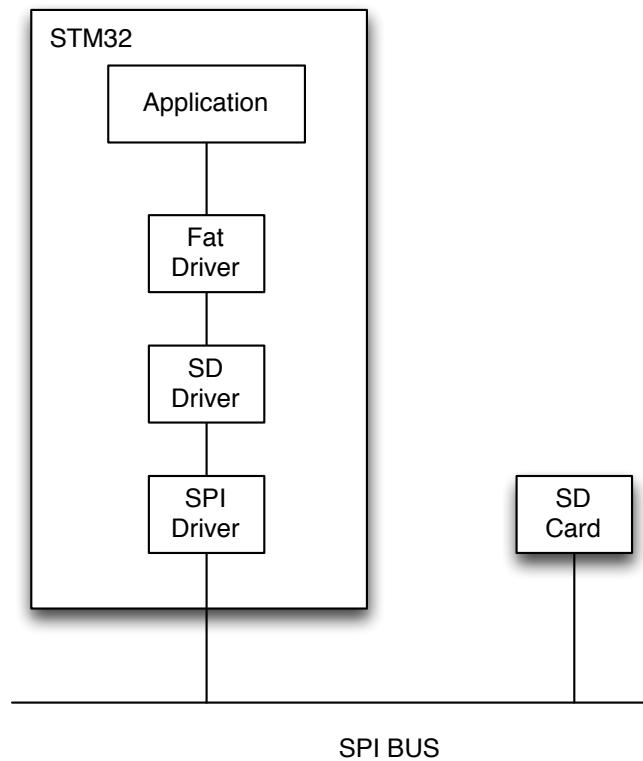


Figura 8.1: SD Card Software Stack

Embora tenhamos previamente desenvolvido um SPI bus driver que é capaz de se comunicar com um cartão SD, estão faltando vários componentes-chave de software. Considere a Figura 8, que mostra a pilha de software necessária. Uma aplicação que deseja acessar dados armazenados em um cartão SD, utiliza comandos de nível de arquivo, como open, read, e write para acessar arquivos específicos dentro do sistema de arquivos do cartão SD. Estes comandos são fornecidos pelo driver do sistema de arquivos FAT. O sistema de arquivos FAT emite comandos ao nível de leitura e escrita de blocos sem nenhum conhecimento de como esses comandos são implementados. Um driver SD separado implementa estes comandos. Finalmente, o driver do SD utiliza a interface SPI para se comunicar com o Cartão SD.

Felizmente, não é necessário escrever todo este software. Neste capítulo iremos descrever o uso do sistema de arquivos genérico FatFs [3]. Este pacote open source fornece a maior parte dos componentes necessários, incluindo um driver SD “genérico” que é relativamente fácil de modificar para utilizar o driver SPI apresentado no Capítulo 6.

Para entender como uma aplicação interage com FatFs, considere o exemplo derivado da distribuição da FatFs, mostrada na Listagem 8.1. Este fragmento de exemplo assume que está se comunicando com um cartão SD, formatado com um sistema de arquivos FAT, que contém um arquivo no diretório raiz chamado **MESSAGE.TXT**. O programa lê esse arquivo, e cria um outro chamado **HELLO.TXT**. Observe o uso de comandos relativamente padronizados do sistema de arquivos.

CAPÍTULO 8. CARTÕES DE MEMÓRIA SD

```
f_mount(0, &Fatfs); /* Register volume work area */

xprintf("\nOpen an existing file (message.txt).\n");
rc = f_open(&Fil, "MESSAGE.TXT", FA_READ);

if (!rc) {
    xprintf("\nType the file content.\n");
    for (;;) {
        /* Read a chunk of file */
        rc = f_read(&Fil, Buff, sizeof Buff, &br);
        if (rc || !br) break; /* Error or end of file */
        for (i = 0; i < br; i++) /* Type the data */
            myputchar(Buff[i]);
    }
    if (rc) die(rc);
    xprintf("\nClose the file.\n");
    rc = f_close(&Fil);
    if (rc) die(rc);
}

xprintf("\nCreate a new file (hello.txt).\n");
rc = f_open(&Fil, "HELLO.TXT", FA_WRITE | FA_CREATE_ALWAYS);
if (rc) die(rc);

xprintf("\nWrite a text data. (Hello world!)\n");
rc = f_write(&Fil, "Hello world!\r\n", 14, &bw);
if (rc) die(rc);
xprintf("%u bytes written.\n", bw);
```

Listing 8.1: FatFs Example

8.1 Organização do FatFs

A discussão a seguir refere-se à versão atual do FatFs (0.9). A distribuição de código está organizada, tal como mostrado na Figura 8.2.

A interface entre o driver do sistema de arquivos FAT e o driver do SD é definido no módulo `diskio.h` mostrado na Listagem 8.2. Modificamos a distribuição default para incluir nomes de parâmetros significativos. Um disco inicializado pode ser lido, escrito e controlado (`ioctl`). Os comandos de leitura/escrita são restritos ao nível de bloco (o tamanho dos blocos depende do cartão SD). A função `ioctl` fornece um meio para determinar a “geometria” do cartão SD (p.ex.: o número e tamanho dos blocos), e adicionalmente pode proporcionar funcionalidades para ativar o controle de energia. A implementação de baixo nível descrita a seguir suporta apenas as funções `ioctl`

8.2. SD DRIVER

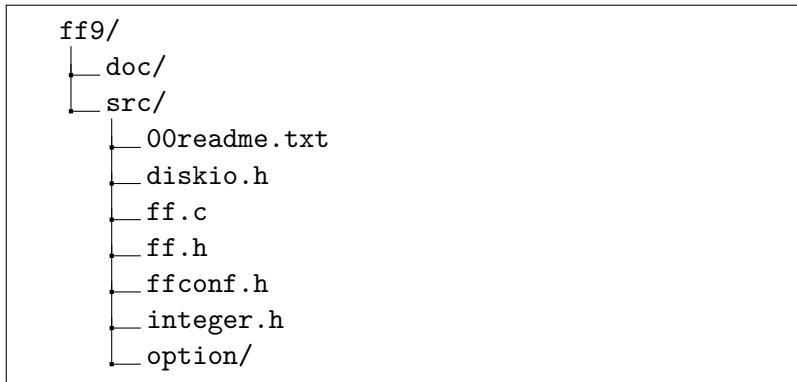


Figura 8.2: Organização da Distribuição do FatFs

para determinar a “geometria” disco e para forçar a conclusão de gravações pendentes.

```
/*-----*/
/* Prototypes for disk control functions */

int assign_drives (int, int);
DSTATUS disk_initialize (BYTE drv);
DSTATUS disk_status (BYTE drv);
DRESULT disk_read (BYTE drv, BYTE* buff, DWORD sector, BYTE count);
#if _READONLY == 0
DRESULT disk_write (BYTE drv, const BYTE* buff, DWORD sector, BYTE
    ↪count);
#endif
DRESULT disk_ioctl (BYTE drv, BYTE ctl, void* buff);
```

Listing 8.2: Low Level Driver Interface

8.2 SD Driver

Como dito anteriormente, o SD driver implementa cinco funções para suportar o FatFs e utiliza o driver SPI para comunicar com o SDCard. O protocolo de comunicação do SDCard é baseado em transações. Cada transação começa com um código de comando de um byte, opcionalmente seguido por parâmetros, e em seguida, a transferência de dados (leitura ou escrita). O protocolo SDCard está bem documentado. [9]. A seguir, apresentamos alguns exemplos para mostrar os conceitos básicos. Felizmente, não é necessário

CAPÍTULO 8. CARTÕES DE MEMÓRIA SD

criar este módulo a partir do zero. Há uma distribuição de exemplos de projetos em <http://elm-chan.org/fsw/ff/ffsample.zip> – usamos o exemplo genérico. Alternativamente, existe uma versão mais sofisticada para o STM32 em (http://www.siwawi.arubi.uni-k1.de/avr_projects/arm_projects/arm_memcards/index.html#stm32_memcard). O código de amostra esta organizado, tal como mostrado na Figura 8.3.

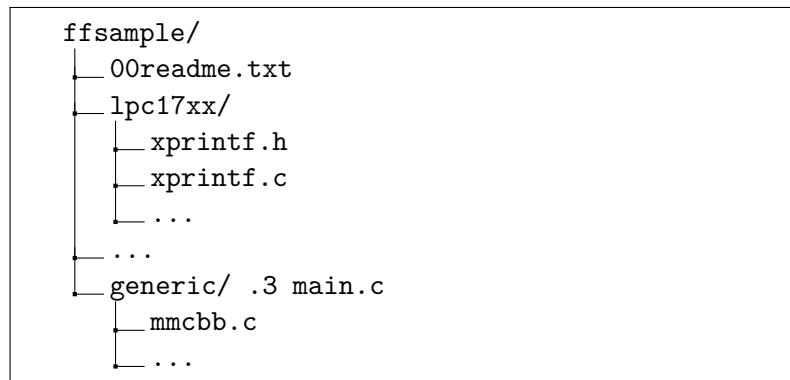


Figura 8.3: Organização do Código de Exemplos de FatFs

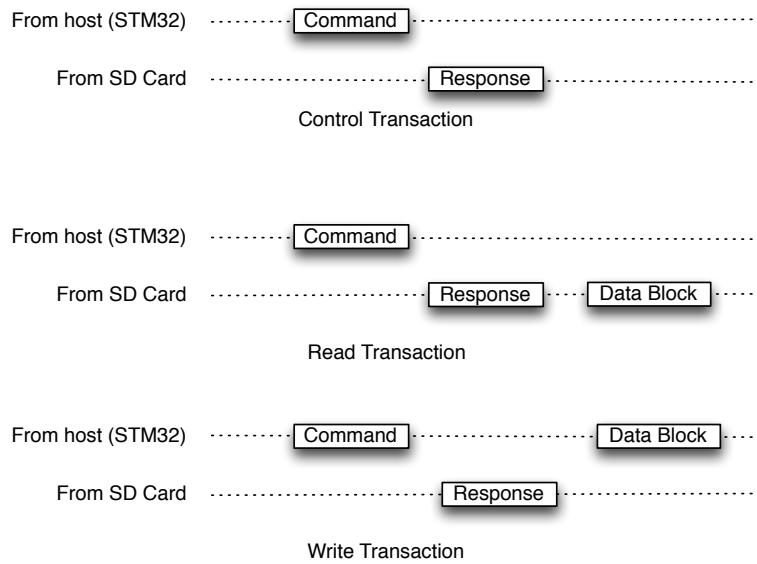


Figura 8.4: Transações SD

8.2. SD DRIVER

Os três tipos de transações básicas que nos interessam são mostradas na Figura 8.2. Cada transação começa com um comando emitido pelo host (neste caso o STM32) seguido por uma resposta do cartão SD. Para operações de controle simples (p. ex.: consultar o status do cartão ou de sua configuração), a resposta do cartão SD termina a transação. Para as operações de leitura ou escrita, a resposta é seguida pela transmissão de um bloco de dados do (escrita) ou para o host (leitura). Há outros casos, incluindo transações de dados multi-bloco e condições de erro que não são mostrados. Deve ficar claro que estes tipos de transação são suficientes para implementar a funcionalidade exigida pelo FatFs.

Nós não vamos nos aprofundar no formato da informação transferida durante as transações, exceto para enumerar alguns dos comandos definidos pelas especificações do Cartão SD e de apontar que todos os campos de uma transação podem, opcionalmente, ser protegido por códigos CRC. Um subconjunto de comandos do cartão SD está mostrado na Tabela 8.2. Observe que há comandos para resetar e inicializar o cartão, parâmetros de leitura/escrita (p.ex.: comprimento do bloco), e leitura/gravação de blocos de dados. Esses comandos são suportados por múltiplos formatos de resposta com comprimentos que variam de 1-5.

Comando	Descrição
CMD0	Reset the SD Card
CMD1	Initialize card
CMD8	Write voltage level
CMD9	Request card-specific data (CSD)
CMD10	Request card identification (CID)
CMD12	Stop transmission
CMD13	Request card status
CMD16	Set transfer block length
CMD17	Read single block
CMD18	Read multiple blocks
CMD24	Write single block
CMD25	Write multiple blocks
CMD58	Read OCR register
ACMD23	Number of blocks to erase
ACMD41	Initialize card

Tabela 8.1: Alguns Comandos SD Card

CAPÍTULO 8. CARTÕES DE MEMÓRIA SD

Nossa implementação do driver SD é um versão simples de `generic/mmbc.c` (veja Figura 8.3). Há apenas um pequeno número de rotinas que devem ser modificadas a fim de utilizar este módulo. Estes são apresentados nas Listagens 8.3 e 8.4. Adicionalmente, o código de exemplo utiliza uma função `wait` (de espera) (`DLY_US`) que conta microssegundos, enquanto o nosso atraso conta milissegundos. É necessário fazer as modificações apropriadas ao longo do código. Não há nada de fundamental sobre a maioria dos períodos de delay (atraso), mas quaisquer mudanças devem tentar um atraso total semelhante. Finalmente, nós modificamos a rotina `disk_initialize` para definir a velocidade do SPI a uma taxa lenta durante a inicialização e uma taxa mais rápida depois que a inicialização for concluída com êxito.

Exercice 8.1 FAT File System

Porte o Driver do FatFs e o programa genérico de exemplo para a Discovery Board. Você pode usar as funções `xprintf` distribuídas com o código de exemplo. FatFs é um sistema altamente configurável. A configuração é controlada através do `ffconf.h`. Para este exercício, você deve usar as configurações default. Uma vez que seu código esteja funcionando, você pode querer experimentar algumas das opções disponíveis. Os passos básicos estão a seguir – um exemplo do Makefile é dado na Listagem 8.5.

1. modificar o `mmcbb.c` do exemplo genérico.
2. criar um projeto que inclui `ff.c` a partir do sistema de arquivos fat, `mmcbb.c`, seu spi driver, e os arquivos de bibliotecas necessárias do STM32.
3. formate seu Cartão SD e crie um arquivo “MESSAGE.TXT”

Para utilizar `xprintf` você irá precisar de incluir algum código no main

```
#include "xprintf.h"

void myputchar(unsigned char c)
{
    uart_putc(c, USART1);
}

unsigned char mygetchar()
{
    return uart_getc(USART1);
}
```

8.2. SD DRIVER

```
#include <stm32f10x.h>
#include <stm32f10x_spi.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_gpio.h>
#include "spi.h"

#define GPIO_Pin_CS GPIO_Pin_6
#define GPIO_CS GPIOC
#define RCC_APB2Periph_GPIO_CS RCC_APB2Periph_GPIOC
#define SD_SPI SPI2

enum spiSpeed Speed = SPI_SLOW;

void Delay(uint32_t);
/* ... */
/*-----*/
/* Transmit bytes to the card */
/*-----*/

static
void xmit_mmc (
    const BYTE* buff, /* Data to be sent */
    UINT bc        /* Number of bytes to send */
)
{
    spiReadWrite(SD_SPI, 0, buff, bc, Speed);
}

/*-----*/
/* Receive bytes from the card */
/*-----*/

static
void rcvr_mmc (
    BYTE *buff, /* Pointer to read buffer */
    UINT bc    /* Number of bytes to receive */
)
{
    spiReadWrite(SD_SPI, buff, 0, bc, Speed);
}
```

Listing 8.3: SD Driver Routines

```
int main(void)
{
    ...
}
```

CAPÍTULO 8. CARTÕES DE MEMÓRIA SD

```
/*-----*/
/* Deselect the card      */
/*-----*/

static
void deselect (void)
{
    BYTE d;

    GPIO_SetBits(GPIO_CS, GPIO_Pin_CS);
    rcvr_mmc(&d, 1); /* Dummy clock (force DO hi-z for multiple
                        →slave SPI) */
}

/*-----*/
/* Select the card       */
/*-----*/

static
int select (void) /* 1:OK, 0:Timeout */
{
    BYTE d;

    GPIO_ResetBits(GPIO_CS, GPIO_Pin_CS);
    rcvr_mmc(&d, 1); /* Dummy clock (force DO enabled) */

    if (wait_ready()) return 1; /* OK */
    deselect();
    return 0;      /* Failed */
}

INIT_PORT()
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_CS, ENABLE);
    /* Configure I/O for Flash Chip select */
    GPIO_InitStructure.GPIO_Pin      = GPIO_Pin_CS;
    GPIO_InitStructure.GPIO_Mode    = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIO_CS, &GPIO_InitStructure);
    deselect();
}
```

Listing 8.4: SD Driver Routines (cont.)

8.3. COPYRIGHT DO FATFS

```
xfunc_in = mygetchar;
xfunc_out = myputchar;
...
}

TEMPLATEROOT = path_to_template_root

# additional compilation flags

CFLAGS += -g -Ipath_to_ff9/src
ASFLAGS += -g
LDLIBS += -lm

# project files

vpath %.c path_to_ff9/src
vpath %.c path_to_ff9/src/option

# ccsbcs.o
OBJS=    $(STARTUP) main.o
OBJS+=   ff.o spi.o uart.o xprintf.o mmcbb.o
OBJS+=   stm32f10x_gpio.o stm32f10x_rcc.o  stm32f10x_usart.o misc.o
OBJS+=   stm32f10x_spi.o core_cm3.o

include $(TEMPLATEROOT)/Makefile.common
```

Listing 8.5: Makefile for SD Card Project

8.3 Copyright do FatFs

```
FatFs module is an open source software to implement FAT file
↳system to
small embedded systems. This is a free software and is opened for
↳education,
research and commercial developments under license policy of
↳following terms.

Copyright (C) 2011, ChaN, all right reserved.

* The FatFs module is a free software and there is NO WARRANTY.
* No restriction on use. You can use, modify and redistribute it
  ↳for
  personal, non-profit or commercial product UNDER YOUR
* RESPONSIBILITY.
* Redistributions of source code must retain the above copyright
  ↳notice.
```

CAPÍTULO 8. CARTÕES DE MEMÓRIA SD

[REDAÇÃO DA PÁGINA VAZIA]

Capítulo 9

I²C – Wii Nunchuk

Neste capítulo apresentamos o I²C, o terceiro maior protocolo que usamos para interagir com módulos externos. I²C é um protocolo de dois fios usados para conectar um ou mais "mestres" com um ou mais "escravos", embora vamos discutir apenas o caso de um único mestre (o STM32) se comunicar com dispositivos escravos. Um exemplo de configuração está mostrado na Figura 9.1. Nesta configuração, um único mestre se comunica com vários escravos com um par de fios de sinal SDA e SCL. Exemplos de dispositivos escravos incluem sensores de temperatura, umidade, e de movimento, bem como EEPROMs seriais.

Como veremos o software necessário para fazer a interface com dispositivos I²C é consideravelmente mais complicado do que com SPI. Por exemplo, I²C tem várias condições de erro que devem ser tratadas, SPI não tem condições de erro no nível físico. Da mesma forma, I²C tem vários tipos de transações, enquanto SPI tem um único tipo de transação básica. Além disso, o SPI é geralmente um barramento muito mais rápido (1-3Mbit/seg vs 100-400Kbit/seg). A maior vantagem de I²C sobre SPI é que o número de fios necessários por I²C é constante (2), independentemente do número de dispositivos conectados, enquanto SPI requer uma linha separada para cada dispositivo. No lugar de selecionar linhas, dispositivos I²C têm endereços internos e são selecionados pelo mestre, através da transmissão desse endereço através do barramento. Esta diferença torna I²C uma boa escolha quando um grande número de dispositivos tem de ser ligado. Finalmente, I²C é um barramento simétrico que pode suportar vários mestres enquanto que SPI é completamente assimétrico.

O restante deste capítulo está organizado da seguinte forma. Começamos com uma introdução ao protocolo I²C na Seção 9.1. Nós, então, discutí-

mos o uso de I^2C para se comunicar com um Wii Nunchuk. O Wii Nunchuk é um dispositivo de entrada de baixo custo que inclui um joystick, dois botões, e um acelerômetro de três eixos (relatado como sendo um LIS302 da ST Microelectronics [24]). Por fim, apresentamos o módulo de comunicação I^2C básico.

9.1 Protocolo I^2C

Nesta seção apresentamos uma visão geral concisa do protocolo de barramento I^2C que abrange apenas os aspectos do protocolo necessário para entender este capítulo. Para uma descrição mais completa, consulte o manual de especificação I^2C [7].

Eletricamente, I^2C é um barramento “wired-or” - o valor dos dois cabos de sinal é “alto” a não ser que um dos dispositivos conectados puxa o sinal para baixo. No lado esquerdo da Figura 9.1 são representados dois resistores que forçam (“pull up”) o valor padrão dos dois fios de barramento para VCC (tipicamente 3.3V). Qualquer dispositivo no barramento com segurança farça o fio para baixo (para GND), a qualquer momento, pois os resistores limitam o consumo de corrente; no entanto, o protocolo de comunicação restringe quando isto pode ocorrer. Os dois fios são chamados SCL (Serial Clock Line - Linha de Clock Serial) e SDA (Serial Data Address - Dados/Endereço Serial). Para se comunicar, um mestre aciona um sinal de clock em SCL durante o acionamento, ou permite que um escravo acione SDA. Assim, a taxa de transferência de bits é determinada pelo mestre.

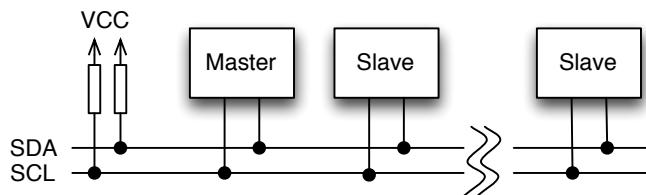


Figura 9.1: Configuração Típica I^2C

A comunicação entre um mestre e um escravo consiste em uma sequência de transações onde o mestre utiliza o SCL como um clock para os dados seriais acionados pelo mestre ou escravo em SDA como mostrado na figura 9.2. Cada transação começa com uma condição Iniciar (S) e termina com a condição de parada (P). A transação consiste em uma sequência de bytes, en-

9.1. PROTOCOLO I²C

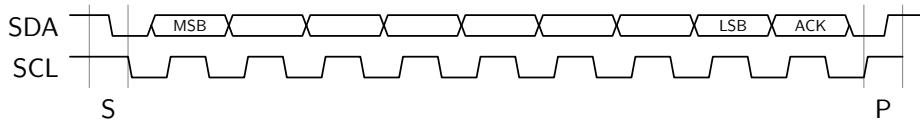


Figura 9.2: Protocolo Físico I²C

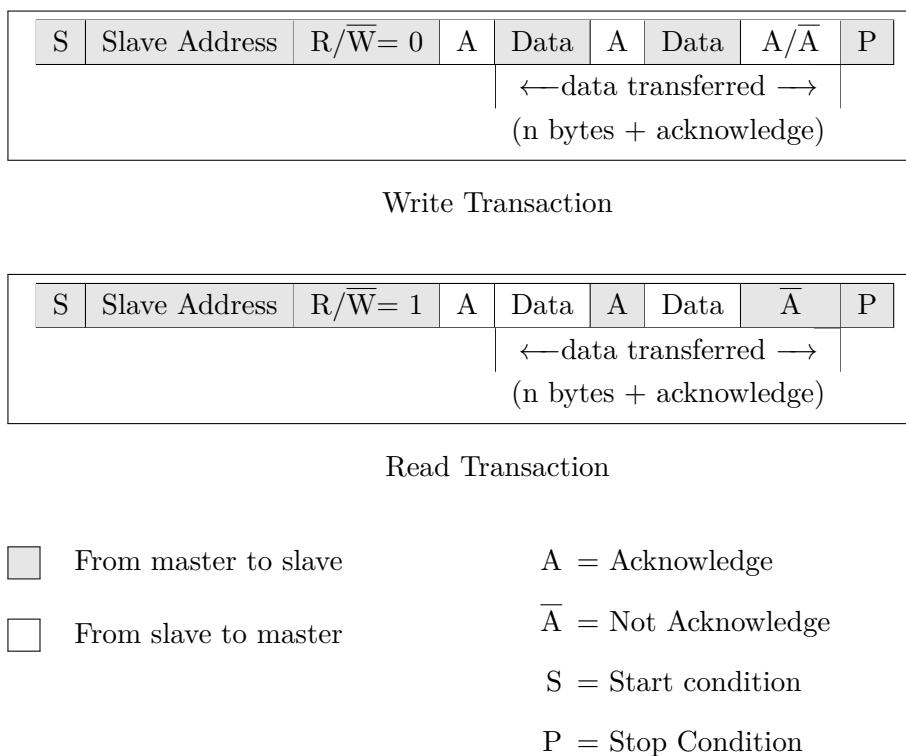


Figura 9.3: I²C Write and Read Transactions

tregando primeiro o bit mais significativo (MSB), cada um deles é terminado por um Acknowledge (ACK), tal como mostrado aqui, ou Not Acknowledge (NACK). Os dados podem ser enviados por qualquer escravo ou pelo mestre, como manda o protocolo, e o ACK ou NACK é gerado pelo receptor dos dados. As condições de Iniciar (S) e Stop (P) são sempre geradas pelo mestre. Uma transição de alto para baixo em SDA enquanto SCL é alta define um

CAPÍTULO 9. I²C – WII NUNCHUK

Start. Uma transição de baixo para alto em SDA enquanto SCL é alta define um Parar.

Existem três tipos de transações no barramento I²C, todos os quais são iniciadas pelo mestre. São elas: escrever, ler, e transações combinadas, onde uma transação combinada concatena uma transação de gravação e leitura. As duas primeiras são mostradas na Figura 9.3 – transações combinadas não são discutidas neste livro. Além disso, existem dois modos de endereçamento: endereçamento de 7-bits, como usado nas transações descritas neste livro, e endereçamento de 10-bits que suporta mais dispositivos em um único barramento em um formato de transações mais complexas e caras.

Todas as transações de endereço de 7 bits começam com um evento de início, a transmissão de um endereço de escravo e um bit que indica se esta é uma gravação ou transação lida pelo mestre. A fase de endereço é terminada por um ACK (0) ou NACK (1) fornecida pelo escravo. Observe que, na hipótese de não haver um escravo correspondente às propriedades elétricas no barramento é garantido o recebimento de um NACK pelo mestre. No caso de transmissão de um endereço seguido por um NACK, o mestre é obrigado a gerar uma condição de parada para terminar a operação, retornando assim o barramento para um estado ocioso – isto é, ambos os sinais altos.

Numa operação de escrita, a fase de endereço é seguida por uma série de bytes de dados (MSB primeiro) transmitida pelo mestre, cada um dos quais é seguido por um ACK ou NACK pelo escravo. A transação é finalizada com uma condição de parada (Stop) após o mestre enviar o máximo de dados da forma que desejar ou o escravo responder com um NACK.

A transação de leitura difere de uma operação de escrita em que os dados são fornecidos pelo escravo e o ACK/NACK pelo mestre. Em uma transação de leitura, o mestre responde ao último byte que pretende obter com um NACK. A transação é finalizada com uma condição de parada (Stop).

A especificação de protocolo descreve transações combinadas, endereçamento de 10-bits, barramentos multi-master, bem como detalhes do barramento físico.

9.2 Wii Nunchuk

Wii Nunchuks são dispositivos de entrada de baixo custo com um joystick, dois botões, e um acelerômetro de três eixos, como mostrado na Figura 9.4. Observe em especial os três eixos: X, Y e Z que correspondem aos dados

9.2. WII NUNCHUK

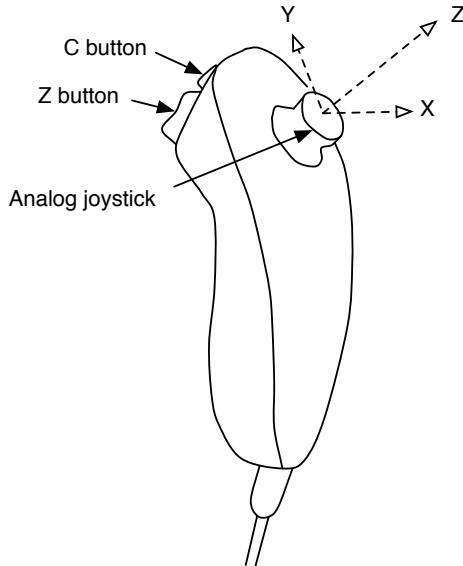


Figura 9.4: Wii Nunchuk

produzidos pelo acelerômetro, joystick. X é direita/esquerda, Y é para frente/-trás, e Z é para cima/baixo (acelerômetro apenas). O barramento I²C é usado para inicializar o Nunchuk para um estado conhecido e, em seguida, para fazer um “poll” regularmente do seu estado. Há uma extensa documentação na web a partir do qual este capítulo é baseado (por exemplo, [8]).

Os dados são lidos a partir do Nunchuck em uma transação de leitura de seis bytes. Estes dados são formatados como mostrado na Figura 9.5 e são lidos iniciando com o byte 0x0 (little-endian). A única complicaçāo com este formato é que os dados do acelerômetro de 10-bits/eixo estão divididos.

A comunicação com o Nunchuk consiste em duas fases – uma fase de inicialização (executada uma vez) em que os dados específicos são escritos para a Nunchuk e uma fase de leitura repetida, sendo que seis bytes de dados são lidos. Cada fase de leitura consiste em duas transações – uma transação de escrita, que define o endereço de leitura para zero, e uma transação de leitura. Os seguintes fragmentos de código mostram a fase de inicialização. O processo de inicialização é descrito em [23]. Essencialmente, a inicialização consiste em duas transações de escrita, cada uma das quais escreve um único byte em um registrador interno para o escravo I²C (reg[0xf0] = 0x55, reg[0xfb] = 0x00). A rotina de escrita tem três parâmetros, a interface I²C a ser usada (I2C1 ou

CAPÍTULO 9. I^2C – WII NUNCHUK

	7	6	5	4	3	2	1	0
0x00								Joystick JX
0x01								Joystick JY
0x02								Accelerometer AX[9:2]
0x03								Accelerometer AY[9:2]
0x04								Accelerometer AZ[9:2]
0x05	AZ[1:0]			AY[1:0]		AX[1:0]	C	Z

Figura 9.5: Nunchuk Data

I^2C2), um buffer de dados, o tamanho do buffer, e o endereço do escravo. As rotinas de transação de I^2C , discutidas na Seção 9.3, retornam as condições de erro que são ignorados neste fragmento.

9.2. WII NUNCHUK

```
// Init

#define NUNCHUK_ADDRESS 0xA4

const uint8_t buf[] = {0xf0, 0x55};
const uint8_t buf2[] = {0xfb, 0x00};

I2C_Write(I2C1, buf, 2, NUNCHUK_ADDRESS);
I2C_Write(I2C1, buf2, 2, NUNCHUK_ADDRESS);
```

O processo de leitura Nunchuk consiste em escrever 0 e, em seguida, ler 6 bytes de dados (como descrito acima). Mais uma vez, ignoramos qualquer retorno de erro.

```
// Read

uint8_t data[6];
const uint8_t buf[] = {0};

I2C_Write(I2C1, buf, 1, NUNCHUK_ADDRESS);
I2C_Read(I2C1, data, 6, NUNCHUK_ADDRESS);
```

A remontagem dos dados é bastante simples; a interpretação pode ser um pouco mais complicada. Os dados do joystick estão na faixa de 0..255, centrados aproximadamente em 128 – descobrimos que é necessário calibra-lo, com base no valor na inicialização. Além disso, o alcance dinâmico foi de pouco menos do que a faixa completa (cerca de 30-220).

Os dados do acelerômetro estão na faixa 0..1023 onde 0 corresponde a -2g e 1023 corresponde a +2g. O acelerômetro pode ser utilizado tanto para detectar o movimento (aceleração), mas também como um “detector de inclinação” (“tilt sensor”) quando não está em movimento, porque podemos usar o campo gravitacional da terra como uma referência. [10] Suponha que tenham valoress medidos de gravidade em três dimensões – G_x, G_y, G_z – sabemos que:

$$G_x^2 + G_y^2 + G_z^2 = 1g^2$$

A partir disto, é possível calcular o “pitch” (rotação em torno do eixo X), o “roll” (rotação em torno do eixo Y) e o “yaw” (rotação em torno do eixo Z). Para o joystick, basta calcular (após a conversão de intervalo para -512..511).

CAPÍTULO 9. I²C – WII NUNCHUK

$$pitch = \text{atan} \left(\frac{AX}{\sqrt{AY^2 + AZ^2}} \right)$$

e

$$roll = \text{atan} \left(\frac{AY}{\sqrt{AX^2 + AZ^2}} \right)$$

Tenha em mente que isto é para 360 graus e uma medida razoável de movimento é, talvez, 90 graus.

Exercise 9.1 Lendo o Wii Nunchuk

Usando o código de interface I²C descrito na Seção 9.3 para escrever um programa que controla o estado das duas interfaces do joysticks (um baseado no acelerômetro) de um Wii Nunchuk e exibir em um LCD 5110. Seu programa deve usar duas letras, X e C, como cursores. Quando o botão c (x) for pressionado, o c (x) cursor deve ser exibido como uma letra maiúscula, caso contrário deve ser exibido como uma letra minúscula. Você provavelmente deve começar com uma velocidade I²C de 10000 – os vários clones Nunchuk parecem não confiáveis a velocidades na faixa de 100.000.

Este é um programa desafiador para escrever. Você precisa primeiro aprender a se comunicar com o Nunchuk. Seu código terá que escalar adequadamente as informações de posição do cursor para exibir cada cursor em um local apropriado. Quando funcionar corretamente, deverá ser possível deslocar cada cursor para a todos os cantos da tela com movimento razoável.

Você terá que modificar seu Makefile para incluir as bibliotecas de matemática padrão, adicionando a seguinte definição (supondo que você está modificando o modelo demo).

```
LDLIBS += -lm
```

A fim de depurar as comunicações I²C é recomendável que user um Saleae Logic para capturar os eventos de comunicação. Um exemplo é mostrado na Figura 9.6. Este exemplo mostra a primeira fase de leitura de dados a partir do Nunchuk – escrevendo um 0 para o endereço do Nunchuk. Observe que a condição de início é indicada por um ponto verde e a condição de término é indicada por um quadrado vermelho. Configurar a Saleae Logic para o

9.3. INTEFACE STM32 I²C

protocolo I²C é semelhante a configuração do protocolo serial, mas utilizamos um analisador de protocolo diferente.

A configuração de hardware necessária é relativamente simples. Você vai precisar de um adaptador Nunchuk como mostrado na Figura 1.7. Existem quatro sinais para conectar, como mostrado na Tabela 9.1.

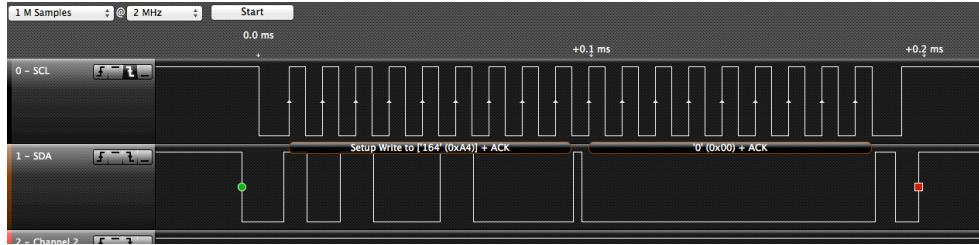


Figura 9.6: Exemplo de Captura do Saleae Logic para I²C

Interface STM32		
	I2C1	I2C2
+	3V3	3V3
-	GND	GND
c	PB6	PB10
d	PB7	PB11

Tabela 9.1: Conexão ao Adaptador Nunchuk

9.3 Inteface STM32 I²C

O dispositivo STM32 I²C é extremamente complicado. Por exemplo, transações de ler com 1, 2, e mais de dois bytes, são tratadas de uma forma significativamente diferente. As melhores referências são o manual do programador ([21, 20]) e ST Nota de Aplicação AN2824 [13]. Esta última descreve alguns exemplos para polling, interrupt-driven, e interfaces de DMA-driven. Infelizmente, o código de exemplo mostra estes três casos juntos e ainda não faz uso da biblioteca de periféricos. Reescrevemos uma solução baseada em polling usando a biblioteca de periféricos.

Tal como acontece com todos os dispositivos STM32, a primeira tarefa é inicializar corretamente o dispositivo, incluindo clocks e pinos. O nosso código de inicialização é mostrado na Listagem 9.1.

CAPÍTULO 9. I^2C – WII NUNCHUK

A implementação de transação de escrita é a mais simples dos dois tipos de transação com poucos casos especiais. Isto é mostrado na Listagem 9.2. Isto segue a Figura 3 do AN2824. Comentários da forma EVn (por exemplo, EV5) se refere a estados, conforme descritos no manual do programador. O código fornecido não tenta se recuperar de erros de interface I^2C . Na verdade, um manipulador (handler) de interrupção é necessário para detectar todos os erros possíveis.

A transação final que vamos considerar é a transação de Leitura. Como observado na AN2824, há casos separados para um byte, 2 bytes, e mais de 2 bytes de leitura. Além disso, existem algumas seções de código de tempo crítico que devem ser executadas sem interrupção! O código comum é mostrado na Listagem 9.3, com os casos de 1, 2, e mais do que dois bytes mostrado nas Listagens de 9.4, 9.5, e 9.6, respectivamente.

9.3. INTEFACE STM32 I²C

```
void I2C_LowLevel_Init(I2C_TypeDef* I2Cx, int ClockSpeed, int
OwnAddress)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    I2C_InitTypeDef  I2C_InitStructure;

    // Enable GPIOB clocks

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

    // Configure I2C clock and GPIO

    GPIO_StructInit(&GPIO_InitStructure);

    if (I2Cx == I2C1){

        /* I2C1 clock enable */

        RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);

        /* I2C1 SDA and SCL configuration */

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
        GPIO_Init(GPIOB, &GPIO_InitStructure);

        /* I2C1 Reset */

        RCC_APB1PeriphResetCmd(RCC_APB1Periph_I2C1, ENABLE);
        RCC_APB1PeriphResetCmd(RCC_APB1Periph_I2C1, DISABLE);

    }
    else { // I2C2 ...}

    /* Configure I2Cx */
    I2C_StructInit(&I2C_InitStructure);
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
    I2C_InitStructure.I2C_OwnAddress1 = OwnAddress;
    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
    I2C_InitStructure.I2C_AcknowledgedAddress =
    I2C_AcknowledgedAddress_7bit;
    I2C_InitStructure.I2C_ClockSpeed = ClockSpeed;

    I2C_Init(I2Cx, &I2C_InitStructure);
    I2C_Cmd(I2Cx, ENABLE);
}
```

Listing 9.1: Initializing I²C Device

Revision: (None) ((None))

139

CAPÍTULO 9. I²C – WII NUNCHUK

```
#define Timed(x) Timeout = 0xFFFF; while (x) \
{ if (Timeout-- == 0) goto errReturn;}

Status I2C_Write(I2C_TypeDef* I2Cx, const uint8_t* buf,
                  uint32_t nbytes, uint8_t SlaveAddress) {

    __IO uint32_t Timeout = 0;

    if (nbytes)
    {
        Timed(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));

        // Initiate Start Sequence

        I2C_GenerateSTART(I2Cx, ENABLE);
        Timed(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

        // Send Address EV5

        I2C_Send7bitAddress(I2Cx, SlaveAddress,
                            I2C_Direction_Transmitter);
        Timed(!I2C_CheckEvent(I2Cx,
                            I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

        // EV6 Write first byte EV8_1

        I2C_SendData(I2Cx, *buf++);
        while (--nbytes) {

            // wait on BTF

            Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));
            I2C_SendData(I2Cx, *buf++);
        }

        Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));
        I2C_GenerateSTOP(I2Cx, ENABLE);
        Timed(I2C_GetFlagStatus(I2Cx, I2C_FLAG_STOPF));
    }
    return Success;
errReturn:
    return Error;
}
```

Listing 9.2: I²C Write Transaction

9.3. INTEFACE STM32 I²C

```
 Status I2C_Read(I2C_TypeDef* I2Cx, uint8_t *buf,
                  uint32_t nbytes, uint8_t SlaveAddress) {

    __IO uint32_t Timeout = 0;

    if (!nbytes)
        return Success;

    // Wait for idle I2C interface
    Timed(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));

    // Enable Acknowledgment, clear POS flag
    I2C_AcknowledgeConfig(I2Cx, ENABLE);
    I2C_NACKPositionConfig(I2Cx, I2C_NACKPosition_Current);

    // Initiate Start Sequence (wait for EV5)
    I2C_GenerateSTART(I2Cx, ENABLE);
    Timed(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

    // Send Address
    I2C_Send7bitAddress(I2Cx, SlaveAddress, I2C_Direction_Receiver);

    // EV6
    Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_ADDR));

    if (nbytes == 1) { /* read 1 byte */ ... }
    else if (nbytes == 2) { /* read 2 bytes */ ... }
    else { /* read 3 or more bytes */ ... }

    // Wait for stop
    Timed(I2C_GetFlagStatus(I2Cx, I2C_FLAG_STOPF));
    return Success;

errReturn:
    return Error;
}
```

Listing 9.3: I²C Read Transaction

CAPÍTULO 9. I²C – WII NUNCHUK

```
if (nbyte == 1) {
    // Clear Ack bit

    I2C_AcknowledgeConfig(I2Cx, DISABLE);

    // EV6_1 -- must be atomic -- Clear ADDR, generate STOP

    __disable_irq();
    (void) I2Cx->SR2;
    I2C_GenerateSTOP(I2Cx, ENABLE);
    __enable_irq();

    // Receive data   EV7

    Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_RXNE));
    *buf++ = I2C_ReceiveData(I2Cx);

}
```

Listing 9.4: I²C Read 1 Byte

```
else if (nbyte == 2) {
    // Set POS flag

    I2C_NACKPositionConfig(I2Cx, I2C_NACKPosition_Next);

    // EV6_1 -- must be atomic and in this order

    __disable_irq();
    (void) I2Cx->SR2;                                // Clear ADDR flag
    I2C_AcknowledgeConfig(I2Cx, DISABLE); // Clear Ack bit
    __enable_irq();

    // EV7_3 -- Wait for BTF, program stop, read data twice

    Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));

    __disable_irq();
    I2C_GenerateSTOP(I2Cx, ENABLE);
    *buf++ = I2Cx->DR;
    __enable_irq();

    *buf++ = I2Cx->DR;

}
```

Listing 9.5: I²C Read 2 Bytes

9.3. INTEFACE STM32 I²C

```
else {
    (void) I2Cx->SR2;      // Clear ADDR flag
    while (nbyte-- != 3)
    {
        // EV7 -- cannot guarantee 1 transfer completion time,
        // wait for BTF instead of RXNE

        Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));
        *buf++ = I2C_ReceiveData(I2Cx);
    }

    Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));

    // EV7_2 -- Figure 1 has an error, doesn't read N-2 !

    I2C_AcknowledgeConfig(I2Cx, DISABLE); // clear ack bit

    __disable_irq();
    *buf++ = I2C_ReceiveData(I2Cx); // receive byte N-2
    I2C_GenerateSTOP(I2Cx,ENABLE); // program stop
    __enable_irq();

    *buf++ = I2C_ReceiveData(I2Cx); // receive byte N-1

    // wait for byte N

    Timed(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_RECEIVED));
    *buf++ = I2C_ReceiveData(I2Cx);

    nbyte = 0;
}
```

Listing 9.6: I²C Read 3 or More Bytes

Capítulo 10

Temporizadores ou *Timers*

Microcontroladores, como o STM32, utilizam temporizadores em hardware para gerar sinais em várias frequências, gerar sinais de saída modulados por largura de pulso (PWM), medir pulsos de entrada, e ativar eventos a frequências ou atrasos conhecidos. A família STM32 tem diferentes tipos de periféricos temporizadores que variam em sua capacidade de configuração. Os temporizadores mais simples (TIM6 e TIM7) são limitados primariamente para gerar sinais a frequências conhecidas ou pulsos de largura fixa. Enquanto temporizadores mais sofisticados possuem hardware adicional para utilizar uma frequência gerada para gerar independentemente sinais com larguras de pulso específicas ou medir esse sinal. Neste capítulo mostramos como os temporizadores podem ser usados para controlar a intensidade da luz de fundo (*backlight*) do ST7735 (modulando seu sinal de ativação) e controlar servos comuns.

Um exemplo de um temporizador básico é mostrado na Figura 10.1. Esse temporizador tem 4 componentes – um controlador, um prescaler (PSC), um registrador de “auto-reload” (auto carregamento) (ARR) e um contador (CNT). A função do *prescaler* é dividir o clock de referência a fim de diminuir a frequência para frequencias mais baixas. Os temporizadores do STM32 tem registradores prescaler de 16 bits e podem dividir o clock de referência por qualquer valor entre 1 e 65535. Por exemplo, o clock do sistema de 24MHz do STM32 VL Discovery poderia ser usado para gerar uma frequência de contagem de 1MHz com um divisor (prescaler) de 23 ($0..23 == 24$ valores). O registrador de contagem pode ser configurado para contar para cima (Up), para baixo (Down), ou para cima/baixo (Up/Down) e ser recarregado pelo registrador de auto carregamento sempre que “estoura” (um “update event” ou “evento de atualização”) ou para parar. O temporizador básico gera um

CAPÍTULO 10. TEMPORIZADORES OU TIMERS

evento de saída (TGRO) que pode ser configurado para ocorrer em um evento de atualização ou quando o contador é ativado (por exemplo, em uma entrada GPIO).

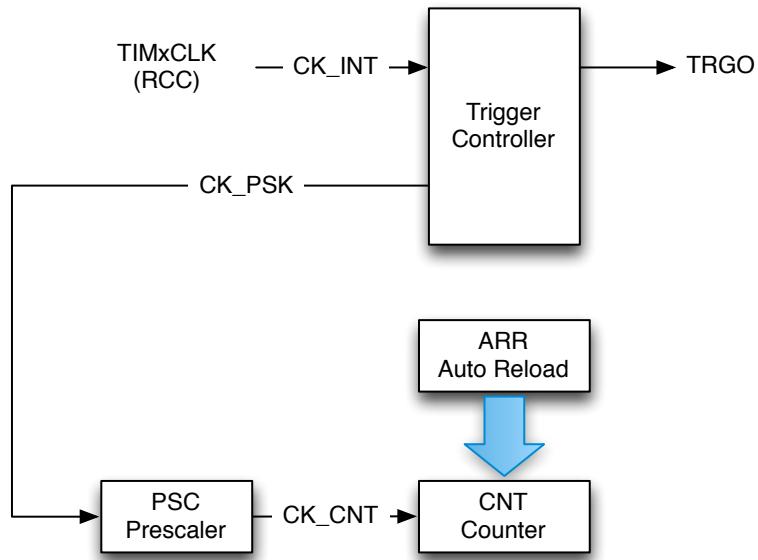


Figura 10.1: Timer Basico

Para entender os três modos do contador, considere a Figura 10.2. Nestes exemplos, assumimos um prescaler de 1 (contador de clock é metade do clock interno), e um valor de *auto reload* de 3. Note que no modo “Up”, o contador incrementa de 0 até 3 (ARR) e então reseta para 0. Quando o reset ocorre, um “update event” (“evento de atualização”) é gerado. Esse evento pode estar amarrado ao TRGO, ou em temporizadores mais complexos com canais de captura/comparação que podem ter efeitos adicionais (descritos abaixo). Da mesma maneira, no modo “Down” (para baixo), o contador decrementa de 3 para 0 e então reseta para 3 (ARR). No modo “Down”, um “evento” de atualização (UEV - Update Event) é gerado quando o contador é resetado para ARR. Finalmente, no modo Up/Down (para cima/para baixo), o contador incrementa o ARR, então decrementa para 0, e repete. Um UEV é gerado antes de cada inversão com o efeito que o período no modo Up/Down é menor que nos modos Up ou Down.

Muitos temporizadores estendem o módulo básico com a adição de ca-

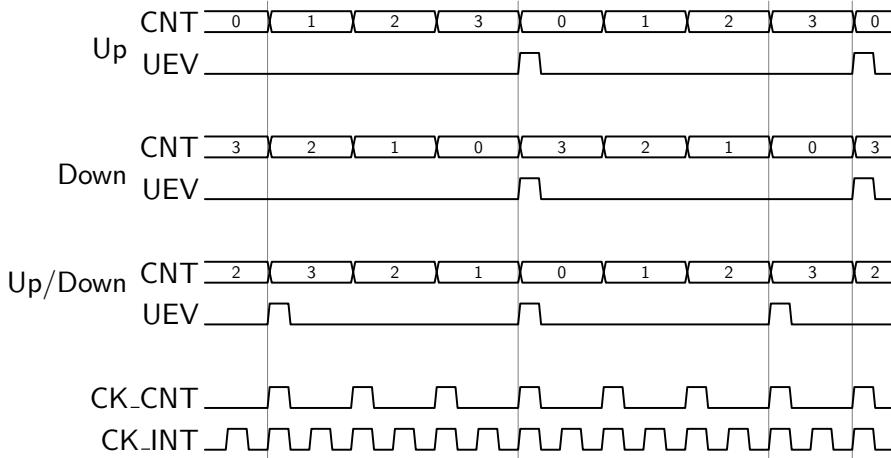


Figura 10.2: Counter Modes (ARR=3, PSC=1)

nais contadores como os mostrados na Figura 10.3. O “x” refere-se ao número do canal — frequentemente, temporizadores suportam múltiplos canais. Com esse hardware adicional modesto, uma saída pode ser gerada sempre que o registrador de contagem alcança um valor específico ou o registrador de contagem pode ser capturado quando um evento de entrada específico ocorrer (possivelmente um clock de entrada prescalado).

Um uso importante para canais de contagem é a geração de pulsos temporizados precisos. Há duas variações desse uso — pulso “one-pulse” (pulso único), no qual um único pulso é gerado, e modulação por largura de pulso, no qual uma série de pulsos é gerada com o período do contador UEV. A largura de pulso é controlada pelo registrador de captura/comparação (CCR - Capture/Compare Register). Por exemplo, o canal de saída (OCxREF) pode estar ligado ao registrador CNT se é maior (ou menor) que o registrador de comparação. Na Figura 10.4 mostramos o uso de dois canais para saídas “one-pulse” e PWM. Aqui assumimos que o ARR é 7 e o CCR é 3. No modo PWM, ARR controla o período e CCR controla a largura de pulso (e portanto o ciclo de trabalho - *duty cycle*). No modo *one-pulse*, o pulso começa ciclos CCR depois de um evento inicial de disparo, e tem a largura de ARR-CCR. É possível o uso de canais múltiplos para criar um conjunto de pulsos sincronizados começando com atrasos precisos entre eles.

CAPÍTULO 10. TEMPORIZADORES OU TIMERS

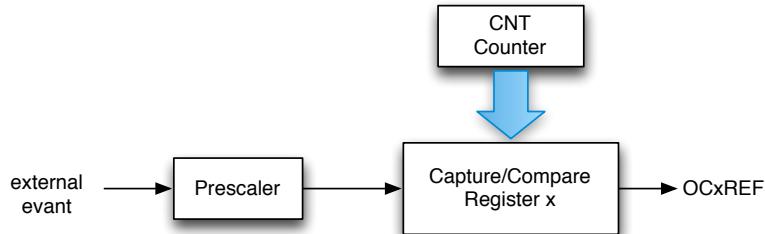


Figura 10.3: Timer Channel

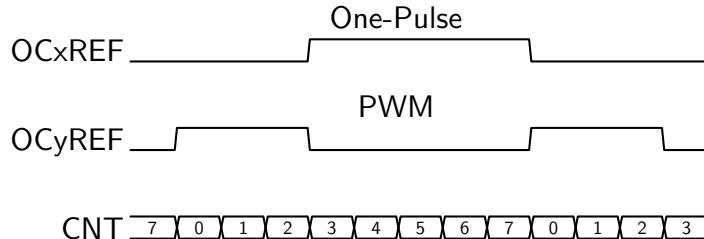


Figura 10.4: Pulse Width Modulation (PWM)

Um canal temporizador pode também ser usado para medir largura de pulsos — na verdade decodificando sinais PWM. Há muitas outras opções de configurações para os temporizadores do STM32 incluindo mecanismos para sincronizar temporizadores múltiplos, tanto para si como para sinais externos.

No restante deste capítulo consideraremos 2 temporizadores de aplicação incluindo saída PWM (Seção 10.1), e medição de entrada de pulso (Seção 10.2). No Capítulo 13 mostramos como usar um temporizador para controlar transferências DMA para um player de áudio e no Capítulo 14 usamos um temporizador para amostrar uma entrada analógica em intervalos regulares.

10.1 Saída PWM

Nesta seção consideraremos 2 exemplos utilizando saída PWM — o controle da luz de fundo para o 7735 LCD e o controle de um servo motor.

10.1. SAÍDA PWM

7735 Backlight

O 7735 backlight consiste em um número de LEDs que são ligados quando se ativa o pino de controle de luz de fundo (backlight) (PA1) em estado baixo (low) e desliga os LEDs colocando em estado alto (high). É possível “escurecer” (“dim”) os LEDs ao aplicar um sinal PWM modulando seu ciclo de trabalho (*duty cycle*). Nessa seção, mostramos como configurar um temporizador para permitir que intensidade de luz de fundo do 7735 seja modificada pelo controle do programa. O código biblioteca para configuração dos temporizadores está em `stm32f10x_tim.[ch]`.

Ao consultar o Manual do Usuário do STM32 VL Discovery [14], encontramos que PA1 é “convenientemente” associada ao TIM2_CH2 — isto é, canal 2 do temporizador TIM2 pode “drive” o pino.¹

A Listagem 10.1 mostra os passos necessários para configurar o Timer 2 para operar com um período de 100 Hz e 1000 passos do timer clock. Isso nos permite definir uma saída com largura de pulso de 0-100% com uma precisão de 0.1%. Os principais parâmetros de configuração são os prescaler, período e o modo de contagem (Up!). O canal de saída é configurado no modo PWM (repetitivo) (há na verdade 2 variações — alinhamento de borda e alinhamento de centro — aqui escolhemos alinhamento de borda). Não foi mostrado código de reconfiguração de PA1 para modo de “função alternativa de push-pull”. Note que, quando configurando o canal de saída, o “número” é embutido no nome da chamada — nesse caso `TIM_OC2Init` inicializa o canal 2, da mesma maneira `TIM_OC4Init` inicializa o canal 4. A largura de pulso `pw` (0..999) pode ser “setada” com o seguinte comando. (De novo, o número do canal é embutido no nome do processo — `TIM_SetCompare2`.)

```
TIM_SetCompare2(TIM2, pw);
```

Exercise 10.1 Ramping LED

Escreva uma aplicação que exibe uma única cor no 7735 LCD e repetidamente pisque (“ramps”) a luz de fundo a uma taxa de 2ms por passo (2ms para aparecer, e 2ms para desaparecer).

Exercise 10.2 Controle de um Servo Hobby

¹Um dos problemas mais difíceis de otimização quando projetamos sistemas baseados em micro-controladores é escolher os pinos de maneira que permita acesso a todos os periféricos de hardware necessários. Planejamento cuidadoso pode evitar um bocado de sofrimento!

CAPÍTULO 10. TEMPORIZADORES OU TIMERS

```
TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
TIM_OCInitTypeDef  TIM_OCInitStructure;

// enable timer clock
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

// configure timer
// PWM frequency = 100 hz with 24,000,000 hz system clock
// 24,000,000/240 = 100,000
// 100,000/1000 = 100

TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
TIM_TimeBaseStructure.TIM_Prescaler
    = SystemCoreClock/100000 - 1; // 0..239
TIM_TimeBaseStructure.TIM_Period = 1000 - 1; // 0..999
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

// PWM1 Mode configuration: Channel2
// Edge-aligned; not single pulse mode

TIM_OCStructInit(&TIM_OCInitStructure);
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OC2Init(TIM2, &TIM_OCInitStructure);

// Enable Timer

TIM_Cmd(TIM2, ENABLE);
```

Listing 10.1: Configuração do Timer para PWM

Os servos comumente usados em veículos controlados por rádio são facilmente controlados usando os temporizadores do STM32. Um servo típico é mostrado na Figura Figura 10.5. O servo consiste em um motor com um braço de alavanca móvel (ou roda) e um controle eletrônico simples usado para definir o ângulo da alavanca. Estes servos têm tipicamente 3 conexões — alimentação (4-6 volts) (normalmente o fio do meio), terra (marrom ou preto) e um sinal de controle. Servos estão disponíveis em uma larga faixa de tamanhos, desde algumas gramas até mais de 20 gramas dependendo dos requisitos de energia. Entretanto, todos trabalham de uma maneira similar.

O protocolo de controle é muito simples — a cada 20ms um pulso é enviado ao servo. A largura do pulso determina a posição do servo. Isso é

10.1. SAÍDA PWM

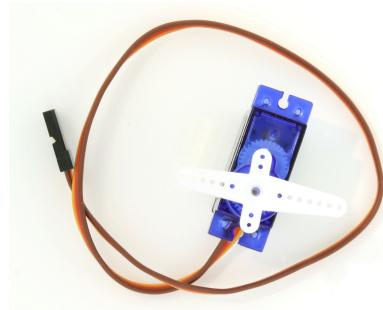


Figura 10.5: Servo Típico de Hobby

ilustrado na Figura 10.6. Um pulso de 1.5ms corresponde ao “centro” (45°), 1ms corresponde a “totalmente a esquerda” (“full left”) (0°) e 2ms corresponde a “totalmente a direita” (“full right”) (90°). Pulsos entre 1ms e 2ms podem ser usados para definir qualquer ângulo entre 0° e 90° .

Server tem uma configuração elétrica simples -- terra, alimentação e sinal. Mesmo que a *maioria* dos servos usem o conector do meio como alimentação, é importante checar o servo que será usado. Há muitas referências na internet. O fio de sinal pode ser conectado diretamente ao “pino do canal de saída do temporizador” (timer output channel pin). O sinal de alimentação deve ser 5V ou menos.

Configure um temporizador para controlar um par de servos usando 2 canais temporizadores (pinos PB8 e PB9 são boas escolhas). Use o controle Nunchuck para controlar a posição. Para esse exercício, não há problemas em usar a alimentação pela USB se os servos são pequenos e se há um diodo flyback para proteger a Discovery Board. Entretanto, seria melhor se houvesse uma fonte de alimentação separada (uma bateria estaria ok). Só precisa lembrar de conectar o terra da bateria com o terra do STM32.

CAPÍTULO 10. TEMPORIZADORES OU TIMERS

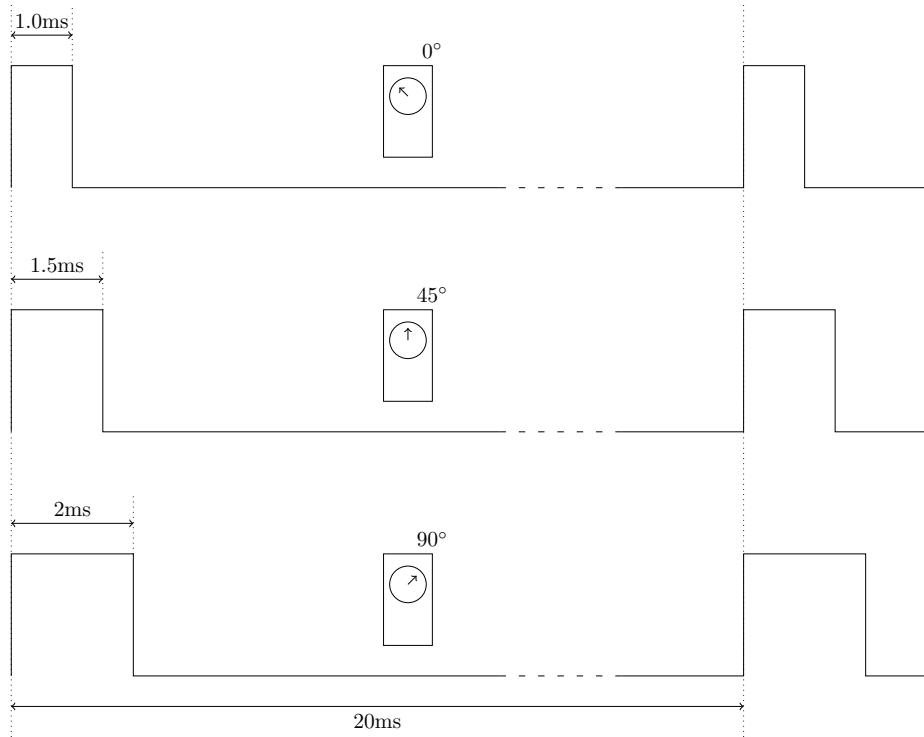


Figura 10.6: Pulses de Controle de Servos

10.2 Captura de Entrada (Input)

Ao gerar sinais de saída PWM nós usamos o recurso de “comparação” do registrador de captura/comparação. Nesta seção nós discutimos sobre a “captura”. O registrador de captura fornece um mecanismo para monitorar um pino de entrada e, baseado no tipo de borda programada, capturar o valor atual do temporizador contador correspondente. O propósito principal dos registradores de captura é possibilitar medições, em relação a uma referência de tempo, quando eventos acontecem. É possível juntar múltiplos registradores de captura em uma única entrada, capturando os tempos das bordas de subida e descida, para a medição da largura de pulso. É possível usar eventos de entrada para “resetar” o contador de tempo, para usar valores de entrada para habilitar o contador de tempo e para sincronizar múltiplos temporizadores. Nessa seção mostraremos como uma captura de entrada pode ser usada em

10.2. CAPTURA DE ENTRADA (INPUT)

conjunto com uma saída PWM para controlar um sensor ultrassônico como o mostrado na Figura 10.7.

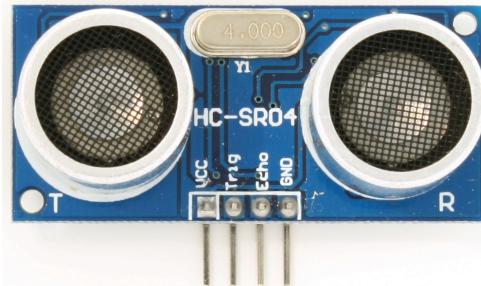


Figura 10.7: Sensor Ultrassônico HC-SR04

O módulo ultrassônico HC-SR04 tem uma resolução de 3mm numa faixa de 20-500mm. É necessário 5V para sua alimentação. Em operação, o módulo é ativado ao fornecer um pulso de $10\mu s$. Algum tempo depois um pulso de “echo” é gerado cuja largura é proporcional à distância medida como mostrado na Figura 10.8 e definida pela seguinte fórmula onde pw é o pulso de “eco”.

$$distancia = pw * \frac{cm}{58\mu s}$$

Se a distância é menor que 20mm ou maior que 500mm, um pulso de 38ms é retornado. Internamente, o circuito controlador ultrassônico gera um sinal de 8 pulsos de 40kHz que aciona o transdutor.

No restante dessa seção descrevemos como usar dois temporizadores -- um para saída e outro para entrada para controlar o sensor autonomamente. Isso significa que depois de configurar os temporizadores, um aplicativo só precisa ler o registrador de captura para saber a distância medida mais recente. Com a adição de interrupções (Capítulo 11, é possível notificar a aplicação sempre que uma nova medição está disponível. Aqui nós usamos um processo de medidas contínuas. Note que, quando usado múltiplos módulos do sensor, é aconselhável eles não estarem ativos continuamente. Com uma pequena quantidade de hardware externo e alguns pinos GPIO, é possível multiplexar o hardware de temporização para controlar um número arbitrário de módulos com apenas 2 temporizadores.

Como mencionado, usamos dois temporizadores -- um para gerar o pulso de ativação e um para medir o pulso de eco. Nós discutimos como gerar um sinal de saída PWM na Seção 10.1 e agora deixamos isso como um

CAPÍTULO 10. TEMPORIZADORES OU TIMERS

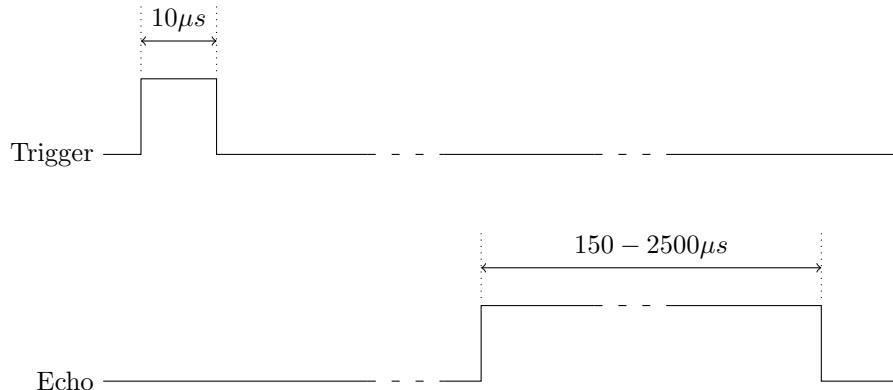


Figura 10.8: Protocolo do Sensor Ultrassônico

exercício ao leitor para utilizar o temporizador TIM4 e o pino PB9 para gerar pulsos de ativação a uma taxa de 10Hz.

Para medir o pulso de eco, iremos usar o temporizador TIM1 conectado a PA8. A arquitetura de vários temporizadores é bem complexa. Exploramos dois conceitos chaves:

1. Pares de registradores de captura (1,2) e (3,4) podem ser “casados” para habilitar em bordas opostas de uma única entrada.
2. Contadores de tempo podem ser configurados como escravos para capturar entradas 1 e 2, por exemplo, para resetar o contador em um evento específico de entrada.

Os detalhes da criação de registradores de captura podem ser bastante complexos, no que se segue, nos referimos ao modelo simplificado de acordo com a Figura 10.9. Na figura dois registradores de captura são mostrados juntamente com o caminho funcional a partir das entradas (t_1 e t_2). As entradas são (opcionalmente) filtradas e sinais gerados nas bordas de subida e descida. Qualquer um dos quatro sinais de bordas podem ser selecionados e, depois (opcionalmente) divididos, usados para ativar o evento de captura. Assim, o registrador de captura (canal) 1 ou 2 pode ser carregado nas bordas de subida ou descida de cada entrada t_1 e t_2 . Além disso, dois sinais TI1FP1

10.2. CAPTURA DE ENTRADA (INPUT)

e TI2FP2 podem ser usados para controlar o contador de tempo, por exemplo, causando assim o reset na borda de entrada selecionada.

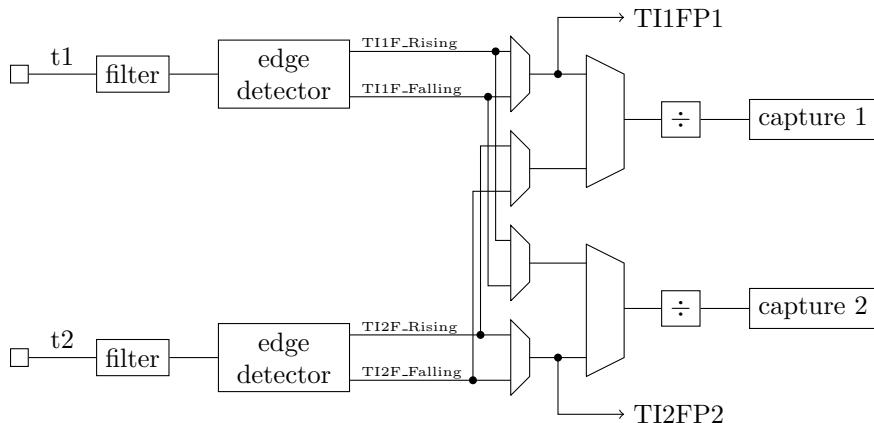


Figura 10.9: Circuito de Captura

Para configurar um TIM1 para medir o pulso do eco ultrassônico precisamos fazer o seguinte (além do pino, da configuração e da distribuição do clock!):

1. Configurar o TIM1 prescaler e o período.
2. Configurar o canal 1 para travar o temporizador em uma entrada de subida em t1.
3. Configurar o canal 2 para travar o temporizador em uma entrada de descida em t2.
4. Configurar TIM1 no modo escravo para resetar na captura do evento 1.

A tarefa (1) é idêntica à da geração da ativação; embora você prefira usar um período mais longo. Segue a configuração para o canal 1:

```

TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;
TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
TIM_ICInitStructure.TIM_ICPrescaler = 0;
TIM_ICInitStructure.TIM_ICFilter = 0;
TIM_ICInit(TIM1, &TIM_ICInitStructure);
    
```

CAPÍTULO 10. TEMPORIZADORES OU TIMERS

Não exigimos nenhuma filtragem ou prescaling (divisão) do sinal de entrada. Queremos capturar na borda de subida usando a entrada t1. A configuração do canal 2 tem as seguintes diferenças:

```
TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Falling;  
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_IndirectTI;
```

Finalmente nós precisamos configurar o modo de temporizador escravo com TI1FP1 como sinal de reset:

```
TIM_SelectInputTrigger(TIM1, TIM_TS_TI1FP1);  
TIM_SelectSlaveMode(TIM1, TIM_SlaveMode_Reset);  
TIM_SelectMasterSlaveMode(TIM1, TIM_MasterSlaveMode_Enable);
```

Exercise 10.3 Sensor Ultrassônico

Escreva uma aplicação que monitora, imprimindo através da USART, a distância, em centímetros, medida pelo sensor ultrassônico a cada 100ms.

Capítulo 11

Interrupções

Interrupções são mecanismos de hardware fundamentais que permitem que os periféricos notifiquem o software de eventos críticos.¹ Por exemplo, podemos querer gerar um sinal analógico de saída em intervalos precisos a fim de tocar um arquivo de áudio. Um modo de alcançar isso é configurar um *timer* para gerar interrupções em intervalos precisos. Quando a interrupção configurada acontece, o processador comuta a execução do programa de aplicação para um manipulador de interrupção especial que pode então “atender” (“service”) o evento de interrupção transferindo a amostra de dado para a saída analógica. Uma vez que o manipulador de interrupção completou sua tarefa, o processador retoma a execução do programa de aplicação. Interrupções são também importantes na comunicação – por exemplo, notificando o processador quando um caractere chegou a uma UART. Neste capítulo nós discutimos como interrupções funcionam na família do micro-controlador STM32 (mais precisamente no Cortex-M3) e apresentamos alguns exemplos concretos para demonstrar seu uso.

Nós temos usado interrupções ao longo desse livro para implementar nossa função de delay (atraso) como mostrado pelo fragmento de código na Listagem 11.1. Dentro da função `main`, nós configuramos o “SysTick” do Cortex-M3 para ativar uma interrupção a cada milissegundo. Além disso, nós definimos um manipulador de interrupção, `SysTick_Handler`, para ser executado quando a interrupção SysTick ocorrer. Este manipulador decremente a variável `TimingDelay` e, em seguida, retorna o controle para o programa de aplicação. O programa de aplicação pode esperar por um período preciso de

¹Interrupções são um caso especial de exceções, que podem incluir eventos internos tais como violações de acesso.

CAPÍTULO 11. INTERRUPÇÕES

```
main(){
    ...
    if (SysTick_Config(SystemCoreClock / 1000))
        while (1);
    ...
}

static __IO uint32_t TimingDelay;

void Delay(uint32_t nTime){
    TimingDelay = nTime;
    while(TimingDelay != 0);
}

void SysTick_Handler(void){
    if (TimingDelay != 0x00)
        TimingDelay--;
}
```

Listing 11.1: SysTick Interrupt

tempo chamando o procedimento `Delay` com um intervalo e então esperando por esse intervalo acabar (literalmente ser decrementado pelo manipulador).

Interrupções podem ser disparadas por uma grande variedade de eventos incluindo o temporizador do sistema, falhas de acesso a memória, resets externos e por todos os vários periféricos fornecidos no processador STM32.² É mais fácil visualizar cada possível fonte de interrupção como um sinal separado que é monitorado pelo núcleo do processador durante a execução do programa. Se o núcleo detecta um sinal de interrupção válido e está configurado para aceitar pedidos de interrupção, ele reage salvando o estado do programa atual em execução na pilha do programa e executando um manipulador (algumas vezes chamado de rotina de serviço de interrupção) correspondente a interrupção aceita. Quando o manipulador termina, o estado do programa salvo é restaurado e a execução normal do programa é restaurada.

O conceito de interrupções, e de modo mais geral, as exceções podem ser relativamente difíceis de entender. Lembre-se que programas em linguagens como C são compilados em código assembly (a representação simbólica de instruções de máquina) de onde o código de máquina é gerado. No STM32,

²Falhas de acesso a memória e outras violações de permissões são frequentemente chamadas de “exceções” ao invés de interrupções porque elas são o resultado da execução do programa e não de eventos externos – de modo geral elas são tratadas de maneira idêntica.

esse código de máquina é copiado para a memória flash ao programar o dispositivo (nós fazemos isso com o gdb) e executado quando o processador é resetado. O modelo básico de execução é:

```
while (1){  
    inst = *pc++;  
    eval(inst);  
}
```

O processador pode ser visto como um interpretador de código de máquina que lê instruções da memória e as avalia. O contador de programa, pc, contém o endereço da próxima instrução a ser executada. Considere a seguinte declaração em 'C' do manipulador de SysTick:

```
TimingDelay--;
```

O compilador traduz esse comando em 3 passos em linguagem assembly que carrega (load (ldr)) o valor de TimingDelay, decrementa (subs) o valor e escreve (str) o valor decrementado de volta. Esta listagem inclui os 6 bytes de código de máquina (0x68a1, 0x3a01, 0x601a) gerado pelo assembler. Embora fora do escopo desse livro, é importante perceber que linguagem assembly é fundamentalmente uma forma legível do código de máquina binário. O processo de vinculação (linking) atribui estas instruções para endereços fixos de memória.

```
681a  ldr   r2, [r3, #0]  
3a01  subs  r2, #1  
601a  str   r2, [r3, #0]
```

Implementar interrupções no processador requer estender este modelo ligeiramente:

```
while (1) {  
    if (interrupt_pending()) {  
        save_state();  
        pc = find_handler();  
    } else {  
        inst = *pc++;  
        eval(inst);  
    }  
}
```

A cada “ciclo” um teste é feito para determinar se a interrupção está pendente – isto literalmente corresponde a checar se a entrada de hardware

CAPÍTULO 11. INTERRUPÇÕES

é 1 ou 0. Se sim, o atual estado do programa (incluindo o contador de programa) é salvo, um endereço do manipulador de interrupção é encontrado e a execução continua com esse manipulador. Quando o manipulador completa a execução, o estado salvo é restaurado e a execução da aplicação continua do ponto quando a interrupção foi ativada. Note que as interrupções acontecem entre as instruções de máquina e não entre comandos em “C”. Como iremos ver, esta compreensão é fundamental para escrever código confiável de interrupção.

No caso do SysTick observe que o manipulador somente modifica o dado compartilhado (`TimingDelay`) quando não é zero e a aplicação somente modifica o dado compartilhado quando é zero. Garantias como esta são importantes porque manipuladores de interrupção forçam o código de aplicação a ser suspenso em locais imprevisíveis e portanto as estruturas de dados que são usadas para comunicação entre manipuladores e o código de aplicação precisam ser construídas cuidadosamente para garantir que elas funcionem de forma correta. Considere o caos que pode resultar se ambos, um programa de aplicação e um manipulador de interrupção, acessarem uma *linked list* – é melhor o manipulador de interrupção não acessar os *links* que estão no processo de serem movidos pelo código do aplicativo. Este tipo de conflito é chamado de “race condition”. Nós apresentamos um exemplo do uso de um manipulador de interrupção para atender a uma UART onde a aplicação e o manipulador compartilham duas filas (queues) de dados – o código é cuidadosamente trabalhado para evitar possíveis “race conditions”.

O exemplo do SysTick demonstra algumas interações típicas entre o manipulador de interrupção e aplicações. O manipulador executa muito brevemente – somente algumas instruções de máquina – a fim de atualizar algumas informações compartilhadas com o programa de aplicação. Para alcançar a confiabilidade os manipuladores de interrupção devem satisfazer três propriedades:

1. Eles precisam executar rapidamente
2. Seu tempo de execução deve ser previsível (por exemplo, eles nunca devem esperar)
3. O uso de “dados compartilhados” (“shared data”) deve ser cuidadosamente gerenciado.

Esse modelo é, com certeza, uma super simplificação. Entretanto, para as discussões subsequentes de interrupções, é um modelo suficiente. Questões

11.1. MODELO DE EXCEÇÃO DO CORTEX-M3

chaves que iremos abordar são – como nós habilitamos periféricos para gerar interrupções pendentes, como nós asseguramos que nossos manipuladores são executados e como nós escrevemos manipuladores confiáveis? Não está ilustrado nesse modelo o conceito de prioridade de interrupção – quando mais de duas interrupções são requisitadas, como é feita a seleção? Também não tratamos do problema de preempção de interrupções – se um manipulador de interrupção está em execução, é possível outros manipuladores serem chamados recursivamente?

O restante desse capítulo está organizado como segue. Nós começamos com uma discussão do modelo de interrupção (exceção) do Cortex-M3 incluindo os vários tipos de exceções, pilhas e privilégios. Nós então discutimos a tabela de vetores de interrupção – o mecanismo pelo qual o núcleo do Cortex-M3 associa manipuladores com as causas específicas de interrupção e o *Nested Vector Interrupt Controller* (NVIC) que o Cortex-M3 usa para selecionar dentre as requisições de interrupções concorrentes.

11.1 Modelo de Exceção do Cortex-M3

Os processadores Cortex-M3 suportam dois modos de operação, modo *Thread* e modo *Handler*. O modo *Thread* é iniciado em um Reset e pode ser iniciado através de um retorno de exceção. O modo *Handler* é iniciado como o resultado de uma exceção sendo usada. Os processadores Cortex-M3 suportam 2 ponteiros distintos para pilha – como mostrado na Figura 11.1. Exceto o reset, todo código utiliza a pilha principal; entretanto, o processador pode ser configurado para que a aplicação utilize uma “pilha de processo” (“process stack”) separada. Sempre que o processador chama uma exceção, o manipulador é executado usando a pilha principal, e quando a execução retorna da exceção, o processador retorna usando a pilha usada anteriormente à exceção. Quando se chama uma exceção, o estado atual do programa precisa ser salvo na pilha principal para ser restaurado quando o manipulador de exceção terminar. Com aplicações de threads simples, como os exemplos que consideramos até aqui, não há vantagem significativa em utilizar pilhas separadas; entretanto, quando se usa um sistema operacional que suporta threads, é geralmente considerado desejável usar pilhas separadas para manipuladores de exceção e execução dentro do kernel do OS.

Salvar e restaurar o estado é uma colaboração entre o hardware do processador e o manipulador de exceção. A arquitetura Cortex-M3 assume que o código do manipulador de exceção vai obedecer a *Arm Architecture Procedure Call Standard* [2] define que todos os procedimentos (inclusive os manipula-

CAPÍTULO 11. INTERRUPÇÕES

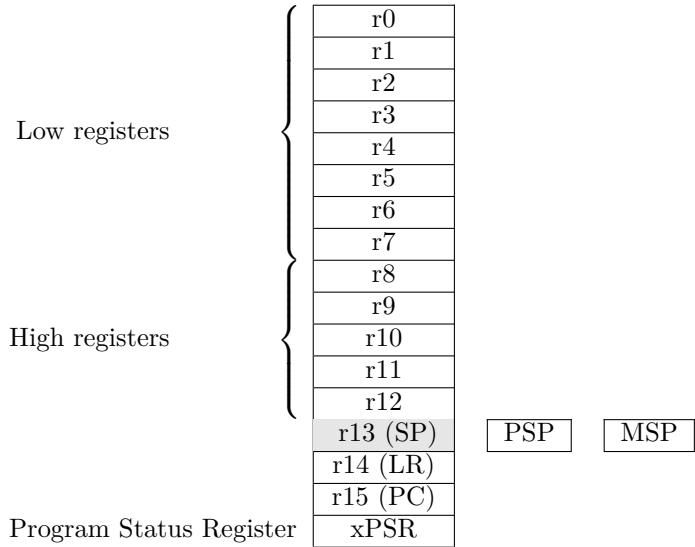


Figura 11.1: Processor Register Set

dores de exceção) salvam e restauram registradores específicos caso eles sejam modificados. O hardware de exceção do Cortex-M3 toma a responsabilidade por salvar qualquer outro registrador. Especificamente quando uma exceção é feita, o processador coloca (“pushes”) oito registradores – xPSR, PC, LR, r12, r3, r2, r1, e r0 – na pilha principal como mostrado na Figura 11.1. Quando retorna de um manipulador de exceção, o processador automaticamente “pops” estes valores da pilha principal. Durante a entrada para o manipulador de exceção, o Link Register (LR) contém um valor especial que controla o retorno da exceção.

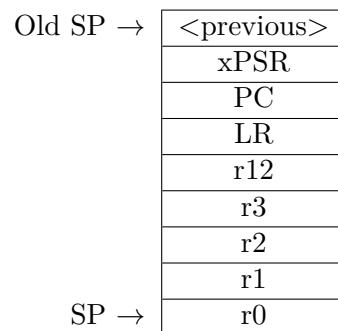


Figura 11.2: Conteúdo da Pilha principal após uma preempção

11.1. MODELO DE EXCEÇÃO DO CORTEX-M3

O processador Cortex-M3 suporta múltiplas prioridades de exceção. Mesmo se um manipulador de exceção está sendo executado, uma exceção de maior prioridade pode ser chamada fazendo preempção do manipulador atual (prioridades são definidas como inteiros com os menores valores tendo prioridades maiores). Neste caso, o estado do manipulador anterior será salvo na pilha principal antes de executar o próximo manipulador. A exceção de maior prioridade é a Reset (-3), que pode fazer preempção de todas as outras. A prioridade da maioria das exceções pode ser definida no programa de controle.

Nome	Descrição	Endereço
-	Initial Stack Pointer	0x0000_0000
Reset_Handler	Executed at reset	0x0000_0004
NMI_Handler	Non-maskable interrupt	0x0000_0008
	...	
SysTick_Handler	System Tick Timer	0x0000_0010
	...	
EXTI0_IRQHandler	External interrupt line 0	0x0000_0058
	...	
TIM3_IRQHandler	Timer 3 interrupt	0x0000_00B4
	...	
USART1_IRQHandler	USART1 global interrupt	0x0000_00D4

Tabela 11.1: STM32 F100xx Interrupt Vector Table

O núcleo do processador Cortex-M3 localiza os manipuladores de interrupção (exceção) através de uma tabela de vetores. Cada entrada na tabela consiste no endereço de um manipulador de interrupção. A tabela é indexada por um único número de cada fonte de interrupção. O formato da tabela de vetores do STM32 esta definida tanto no manual de referência do Cortex-M3 [1] quanto no manual de referência do STM32 [21, 20]. Um fragmento dessa tabela para os dispositivos STM32 F100xx é mostrada na Tabela 11.1.

As 16 primeiras entradas (através do SysTick_Handler) são definidas pelas especificações do Cortex-M3 e o restante são específicos do processador. A primeira entrada na tabela não é o endereço do manipulador de interrupção, mas o endereço da pilha inicial. Ao resetar, o núcleo carrega o ponteiro da pilha do endereço de memória 0x0000_0000 e começa a execução no local 0x0000_0004 onde o Reset_Handler está armazenado. Esse fragmento também inclui as entradas para TIM3, USART1 e EXTI0, que iremos considerar nesse capítulo.

CAPÍTULO 11. INTERRUPÇÕES

O Cortex-M3 especifica que a tabela de vetores esteja localizada começando na posição de memória 0x0000_0000. Há algumas exceções importantes. O STM32, quando faz um boot da memória flash, equaliza (“aliases”) a memória flash, que começa na posição 0x0800_0000 a 0x0000_0000 – assim leituras de memória da localização 0x0000_0000 retornam o valor armazenado em 0x0800_0000 (esse e outros aliases são descritos na seção 2.4 da Referência [20]). É também possível mover a localização da atual tabela de vetores em tempo de execução (veja [19]) – uma importante característica para suporte de bootloaders alternativos que podem colocar as tabelas de vetores em outros locais. A tabela de vetor interrupção é definida no código de inicialização e sua localização definida pelo linker, ambos descritos no Capítulo 3.

Reset_Handler	DMA1_Channel16_IRQHandler
NMI_Handler	DMA1_Channel17_IRQHandler
HardFault_Handler	ADC1_IRQHandler
MemManage_Handler	EXTI9_5_IRQHandler
BusFault_Handler	TIM1_BRK_TIM15_IRQHandler
UsageFault_Handler	TIM1_UP_TIM16_IRQHandler
SVC_Handler	TIM1_TRG_COM_TIM17_IRQHandler
DebugMon_Handler	TIM1_CC_IRQHandler
PendSV_Handler	TIM2_IRQHandler
SysTick_Handler	TIM3_IRQHandler
WWDG_IRQHandler	TIM4_IRQHandler
PVD_IRQHandler	I2C1_EV_IRQHandler
TAMPER_IRQHandler	I2C1_ER_IRQHandler
RTC_IRQHandler	I2C2_EV_IRQHandler
FLASH_IRQHandler	I2C2_ER_IRQHandler
RCC_IRQHandler	SPI1_IRQHandler
EXTIO_IRQHandler	SPI2_IRQHandler
EXTI1_IRQHandler	USART1_IRQHandler
EXTI2_IRQHandler	USART2_IRQHandler
EXTI3_IRQHandler	USART3_IRQHandler
EXTI4_IRQHandler	EXTI15_10_IRQHandler
DMA1_Channel1_IRQHandler	RTCAlarm_IRQHandler
DMA1_Channel2_IRQHandler	CEC_IRQHandler
DMA1_Channel3_IRQHandler	TIM6_DAC_IRQHandler
DMA1_Channel4_IRQHandler	TIM7_IRQHandler
DMA1_Channel5_IRQHandler	

Figura 11.3: Nomes de Vetores Definidos para a Família Medium Density Value Line

O código de inicialização discutido no Capítulo 3 foi projetado para simplificar modificações na tabela de vetores. Cada vetor de interrupção é

11.2. HABILITANDO INTERRUPÇÕES E DEFININDO SUAS PRIORIDADE

fornecido com uma definição inicial “fraca”. Para substituir essa definição, basta definir um processo com o nome correto. Por exemplo, anteriormente definimos um manipulador timer do sistema:

```
void SysTick_Handler(void){  
    if (TimingDelay != 0x00)  
        TimingDelay--;  
}
```

Da mesma maneira, definimos um manipulador para USART1 como

```
void USART1_IRQHandler(void) {  
    // Check interrupt cause  
    ...  
    // Clear interrupt cause  
}
```

Observe que ao contrário do manipulador SysTick, a maioria dos manipuladores devem, no mínimo, determinar a causa da interrupção – com a USART isso pode ser um buffer de transmissão vazio ou um buffer de transmissão cheio – e limpar a causa da interrupção. Determinar a causa é geralmente feita lendo um registrador específico do periférico. Limpar a interrupção é feita realizando uma ação necessária (por exemplo: ler um registrador de dados) ou resetando diretamente o bit de status correspondente.

Os nomes dos manipuladores necessários definidos no código de inicialização são mostrados na Figura 11.1. Observe que diferentes membros da família STM32 suportam diferentes periféricos e portanto tem nomes de manipuladores diferentes. Além disso, estes nomes são definidos no código de inicialização e não por funções de bibliotecas. Em caso de dúvida, você precisa olhar nos códigos de inicialização! Além disso, tenha cuidado com erros de digitação em nomes de vetores, isso pode ser difícil de diagnosticar.

11.2 Habilitando Interrupções e Definindo suas Prioridade

O núcleo do Cortex-M3 define um mecanismo de prioridade sofisticado que permite fontes de interrupções serem atribuídas tanto a uma prioridade quanto a uma sub-prioridade. Em um determinado nível de prioridade, duas fontes de interrupções são atendidas por ordem de sub-prioridade (números menores tem precedência). Se um manipulador de interrupção está ativo e outra interrupção chega com um número de prioridade menor, o manipulador

CAPÍTULO 11. INTERRUPÇÕES

ativo sofrerá uma preempção. O Cortex-M3 define até 8 bits de níveis de prioridade que podem ser divididos entre prioridades e sub-prioridades. O processador STM32 implementa somente 4 desses bits. Ao longo desse livro utilizamos a configuração onde 0 bits são alocados como prioridade e 4 bits são alocados como sub-prioridade. Em outras palavras, nós escolhemos não habilitar a preempção de interrupção.

O mecanismo de prioridade de interrupção é gerenciado pelo NVIC (Nested Vectored Interrupt Controller) que é um periférico padrão para todos os processadores baseados no Cortex-M3.³ O NVIC fornece as seguintes funções para cada fonte de interrupção:

- Configuração de prioridade e sub-prioridade.
- Habilita (desabilita) a interrupção.
- Seta/Limpa bit pendente de interrupção.

11.3 Configuração do NVIC

O NVIC do STM32 suporta 16 níveis de prioridades e subprioridades distintos suportados por 4 bits que são divididos entre essas duas funções. Interrupções com diferentes prioridades podem preemptar umas as outras (prioridades de menor valor tem precedência) enquanto sub-prioridades dentro de uma única prioridade só afetam a escolha da interrupção tomada quando duas ou mais estão pendentes. Por exemplo, neste capítulo nós usamos 0 bits para prioridade. No Capítulo 16 iremos discutir o “FreeRTOS”, sistema operacional de tempo real, que requer o uso de 4 bits para prioridade.

```
#include <misc.h>

/*
 * NVIC_PriorityGroup_0  0 bits priority, 4 bits subgroup
 * NVIC_PriorityGroup_1  1 bits priority, 3 bits subgroup
 * NVIC_PriorityGroup_2  2 bits priority, 2 bits subgroup
 * NVIC_PriorityGroup_3  3 bits priority, 1 bits subgroup
 * NVIC_PriorityGroup_4  4 bits priority, 0 bits subgroup
*/
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
```

³Além da prioridade de interrupção, O NVIC também controla a configuração do Timer SysTick, bem como outras funções de propósitos especiais.

11.4. EXEMPLO: INTERRUPÇÕES DO TIMER

O seguinte fragmento de código configura e habilita a interrupção do TIM2. Observe que isso deve seguir qualquer configuração de dispositivo!!!

```
NVIC_InitTypeDef NVIC_InitStructure;  
  
// No StructInit call in API  
  
NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;  
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;  
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;  
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;  
NVIC_Init(&NVIC_InitStructure);
```

O número do IRQChannel pode ser específico do dispositivo. Estes são definidos no `stm32f10x.h` no qual você pode consultar o nome da constante correto – não copie o valor da constante no seu código! Observe que esse arquivo de cabeçalho depende de definições específicas do dispositivo – em nosso arquivo nós definimos `STM32F10X_MD_VL` nos nossos flags de compilação como discutido no Capítulo 3.

11.4 Exemplo: Interrupções do Timer

Na Seção 10.1 mostramos como configurar o timer TIM2 para controlar a luz de fundo do LCD. Neste exemplo, mostramos como habilitar a interrupção TIM2. Uma vez que este se baseia no trabalho que você já viu anteriormente, apresentamos um esquema básico na Listagem 11.2. Além disso, para configurar o timer, é necessário configurar o NVIC para o vetor de interrupção apropriado e habilitar o timer para gerar interrupções. Os Timers podem gerar interrupções em condições múltiplas – aqui escolhemos ativar interrupções sempre que o contador é atualizado. Finalmente, nós precisamos de um manipulador que, ao menos, limpe as interrupções pendentes.

Exercise 11.1 Timer Interrupt – Blinking LED

Complete o programa de interrupção de Timer para que o LED verde (PA9) pisque a uma taxa de 1hz (metade do tempo apagado e a outra metade aceso).

CAPÍTULO 11. INTERRUPÇÕES

```
// Configure clocks for GPIOA and TIM2
...
// Configure NVIC -- see preceding section
...
// Configure Timer
...
// Enable Timer Interrupt, enable timer

TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
TIM_Cmd(TIM2, ENABLE);

while(1) { /* do nothing */ }

void TIM2_IRQHandler(void)
{
    /* do something */
    TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
}
```

Listing 11.2: Timer Tick Interrupt

11.5 Exemplo: Comunicação Serial com Interrupção

Um ponto fraco do código de comunicação serial apresentado no Capítulo 5 é que, a menos que o código esteja constantemente sondando a USART e esteja preparado para receber caracteres logo quando eles chegarem, existe uma grande probabilidade que o buffer de recebimento estoure e caracteres serão perdidos. Idealmente, nós queremos uma solução que não requeira um grau elevado de atenção do código e que também garanta que caracteres não serão perdidos. Similarmente, quando o código deseja transmitir uma string, é preciso esperar pelo buffer transmissor esvaziar cada caractere mandado.

Uma solução parcial é criar grandes buffers de transmissão e recebimento no software e utilizar um manipulador de interrupção para gerenciar os detalhes do recebimento e transmissão de caracteres. O manipulador de interrupção é uma rotina criada pelo usuário que é executada, assincronamente com o código de usuário, sempre que o buffer de transmissão USART se torna vazio e o buffer de recebimento USART se torna cheio. Isso alivia o programa do usuário do ônus da monitoração dos buffers USART e, fornecendo espaço de buffer adicional, ajuda a desacoplar o programa do usuário do processo de comunicação. É apenas uma solução parcial porque, uma vez o buffer esteja

11.5. EXEMPLO: COMUNICAÇÃO SERIAL COM INTERRUPÇÃO

cheio, qualquer caractere adicional recebido será perdido. Uma solução completa, descrita na Seção 11.5, utiliza um sinal de controle adicional da USART para bloquear a comunicação quando não existe espaços disponíveis.

Como discutido no Capítulo 5, implementações com pooling baseadas em `getchar` e `putchar` sofrem problemas significativos de performance. No caso do `putchar`, o código de aplicação é retardado para a taxa de transmissão sempre que ele tenta transmitir caracteres de volta para trás. A situação para `getchar` é ainda mais terrível – se o código de aplicação não monitorar constantemente o registrador de recebimento de dados da USART há um significativo risco de perda de dados. Nessa seção mostramos como interrupções em conjunto com buffers de software podem ser usados para aliviar, mas não eliminar, o problema. Uma solução completa, como discutido na Seção 11.5 requer uso de controle de fluxo por hardware em conjunto com interrupções.

Em uma USART dirigida por interrupção a responsabilidade por recebimento e transmissão de dados é dividida entre o código de aplicação e o código de interrupção. O código de interrupção é chamado sempre que um evento configurado ocorre; por exemplo, quando o registrador de transmissão de dado está vazio ou o registrador de recebimento de dados está cheio. O código de interrupção é responsável pela remoção da condição de interrupção. No caso de um registrador de recebimentos de dados cheio, o manipulador de interrupção remove a condição de interrupção através da leitura dos dados recebidos. O código de aplicação e o manipulador de interrupção se comunicam por um par de buffers de software implementados como queues (filas) como mostrado na Figura 11.4.

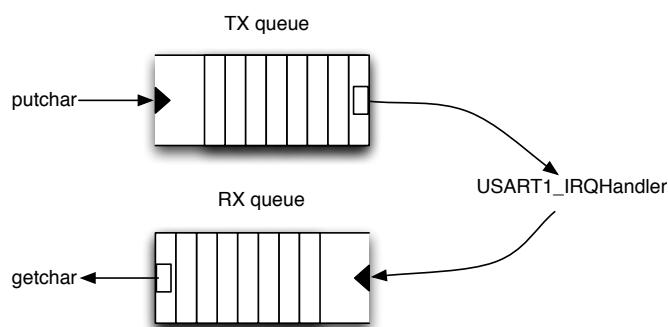


Figura 11.4: Interrupt Driven USART Software

CAPÍTULO 11. INTERRUPÇÕES

A ideia básica é simples. Sempre que o código de aplicação executa `putchar`, um caractere é adicionado na fila TX e sempre que o código de aplicação executa `getchar`, um caractere é removido da fila RX. Da mesma maneira, sempre que o manipulador de interrupção, `USART1_IRQHandler`, é chamado, o manipulador remove a condição de interrupção movendo um caractere da fila TX para o registrador de transmissão de dado ou movendo um caractere do registrador de recebimento de dado para a fila RX. Todas as dificuldades de implementação surgem de lidar com os casos extremos onde as duas filas estão vazias ou cheias.

As decisões de projeto para os casos extremos diferem para o código da aplicação e da interrupção. O requisito mais importante para o código da interrupção é que ele nunca deve travar. Por exemplo, se o registrador de dados recebido esta cheio e a fila RX esta também cheia, a única forma de remover a condição de interrupção é ler o registrador de dados recebido e descartar o dado. Assim, uma USART dirigida a interrupções não pode eliminar completamente o problema de dados perdidos da solução baseada em pooling – que irá surgir com a adição do fluxo de controle. Ao contrário, o código da aplicação pode travar. Por exemplo, se a aplicação executa um `putchar` e a fila TX esta cheia, então ela poderá fazer um “poll” para esperar pela condição completa para ser removido (pelo manipulador de interrupções). Neste caso, o código da aplicação será novamente atrasado na taxa de transmissão, mas somente após a fila TX estar cheia. Uma decisão de implementação importante é quão grande as filas devem ser para prevenir que a aplicação atrase.

Vamos assumir a estrutura de dados do tipo `queue` (fila) com duas operações

```
struct Queue;
struct Queue USART1_TXq, USART1_RXq;

int Enqueue(struct Queue *q, uint8_t data);
int Dequeue(struct Queue *q, uint8_t *data);
```

A operação `Enqueue` pega como parâmetros uma fila e um byte de dados; sua função é adicionar o dado na fila. Se a fila está cheia, então `Enqueue` deverá retornar um erro (0). Assim, o uso confiável da operação `Enqueue` pode exigir chamadas repetidas. A operação `Dequeue` é similar, porém a operação remove o byte de dados da fila. Assim como `Enqueue`, `Dequeue` retorna um indicador de sucesso e sua confiabilidade pode exigir múltiplas chamadas.

Nós ainda assumimos que o hardware de interrupção necessário está habilitado para duas condições – o registrador de recebimento de dados não

11.5. EXEMPLO: COMUNICAÇÃO SERIAL COM INTERRUPÇÃO

está vazio e o registrador de transmissão de dado está vazio. O código do manipulador de interrupção deve, então, lidar com essas duas possibilidades de condição:

CAPÍTULO 11. INTERRUPÇÕES

```
static int TxPrimed = 0;
int RxOverflow = 0;
void USART1_IRQHandler(void)
{
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        uint8_t data;

        // buffer the data (or toss it if there's no room
        // Flow control will prevent this

        data = USART_ReceiveData(USART1) & 0xff;
        if (!Enqueue(&UART1_RXq, data))
            RxOverflow = 1;
    }

    if(USART_GetITStatus(USART1, USART_IT_TXE) != RESET)
    {
        uint8_t data;

        /* Write one byte to the transmit data register */

        if (Dequeue(&UART1_TXq, &data)){
            USART_SendData(USART1, data);
        } else {
            // if we have nothing to send, disable the interrupt
            // and wait for a kick

            USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
            TxPrimed = 0;
        }
    }
}
```

Observe que ambos, os códigos de recebimento e transmissão, contém os elementos essenciais das implementações de pooling de `putchar` e `getchar`, eles lidam com os casos extremos de modo diferente. Se não há espaço na fila RX, o manipulador de interrupção recebe, mas descarta qualquer dado no registrador de recebimento. Sempre que o manipulador de interrupção descarta um dado, ele seta uma variável global, `RxOverflow` para 1; cabe ao código da aplicação monitorar essa variável e decidir como lidar com a perda de dados. Se a fila TX está vazia, o manipulador de interrupção não pode resolver a condição de interrupção (uma vez que não há nada a escrever para o registrador de transmissão), então ele desativa a condição `USART_IT_TXE`. A variável `TxPrimed` é usada para comunicar com a aplicação (especificamente `putchar`)

11.5. EXEMPLO: COMUNICAÇÃO SERIAL COM INTERRUPÇÃO

que a interrupção precisa ser reabilitada quando dados forem adicionados à fila TX.

A nova implementação para `getchar` é bem simples (isso pode ser generalizado para substituir a implementação `uart_getc` que você desenvolveu no Exercício 5.2).

```
int getchar(void)
{
    uint8_t data;
    while (!Dequeue(&UART1_RXq, &data));
    return data;
}
```

A nova implementação para `putchar` requer um teste extra para determinar se a condição de reabilitação da interrupção de transmissão é necessária:

```
int putchar(int c)
{
    while (!Enqueue(&UART1_TXq, c))
        if (!TxPrimed) {
            TxPrimed = 1;
            USART_ITConfig(USART1, USART_IT_TXE, ENABLE);
        }
}
```

A interação assíncrona entre o código do manipulador de interrupção e o código da aplicação pode ser bastante sutil. Observe que nós setamos `TxPrimed` antes de reabilitar a interrupção. Se a ordem fosse invertida seria possível para o manipulador de interrupção recém habilitado esvaziar a fila de transmissão e limpar `TxPrimed` antes da aplicação setar `TxPrimed`, perdendo, assim, o sinal do manipulador para o código de aplicação. É certo que, a USART é suficientemente lenta e que este cenário é improvável, mas com o código de interrupção vale a pena programar defensivamente.

Ainda restam duas tarefas – implementação da estrutura de dados queue (fila) necessária e habilitação de interrupções.

Filas de Interrupções Seguras

A peça final de nosso código de interrupção USART é a implementação da fila. A abordagem mais comum é fornecer um buffer circular de tamanho fixo com pontos de leitura e escrita separados. Isso é mostrado na Figura 11.5.

Há várias ideias-chave por trás do buffer circular. O ponteiro de leitura “persegue” o ponteiro de escrita ao redor do buffer; assim, as funções de

CAPÍTULO 11. INTERRUPÇÕES

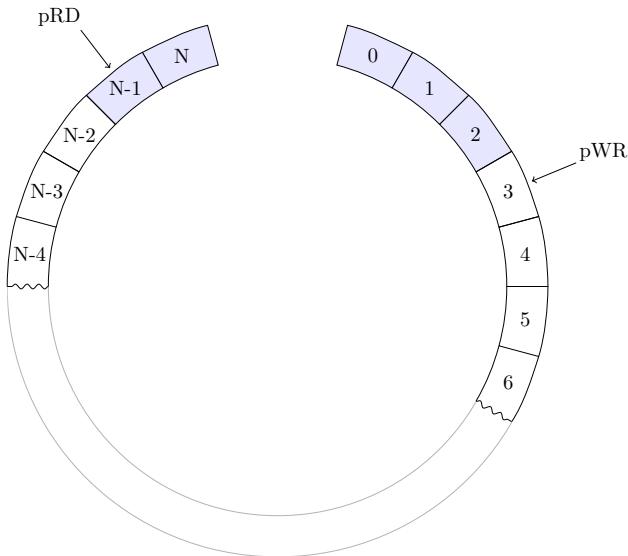


Figura 11.5: Buffer Circular

incremento devem se sobrepor. Além disso, o ponteiro de escrita sempre faz referência a um local “vazio”. Assim, um buffer com $N+1$ locais pode conter, no máximo, N elementos de dados. Assim, nós definimos a estrutura de dados básica como segue:

```

struct Queue {
    uint16_t pRD, pWR;
    uint8_t q[QUEUE_SIZE];
};

static int QueueFull(struct Queue *q)
{
    return (((q->pWR + 1) % QUEUE_SIZE) == q->pRD);
}

static int QueueEmpty(struct Queue *q)
{
    return (q->pWR == q->pRD);
}

static int Enqueue(struct Queue *q, uint8_t data)
{

```

11.5. EXEMPLO: COMUNICAÇÃO SERIAL COM INTERRUPÇÃO

```
if (QueueFull(q))
    return 0;
else {
    q->q[q->pWR] = data;
    q->pWR = ((q->pWR + 1) == QUEUE_SIZE) ? 0 : q->pWR + 1;
}
return 1;
}

static int Dequeue(struct Queue *q, uint8_t *data)
{
    if (QueueEmpty(q))
        return 0;
    else {
        *data = q->q[q->pRD];
        q->pRD = ((q->pRD + 1) == QUEUE_SIZE) ? 0 : q->pRD + 1;
    }
    return 1;
}
```

Apesar de sua simplicidade, uma estrutura de dados compartilhados, como essa, oferece amplas oportunidades para “race conditions” de dados. Observe que a operação `Enqueue` (`Dequeue`) escreve (lê) o local da fila referenciada antes de atualizar o ponteiro `qWR` (`qRD`). A ordem das operações é importante! Como apresentado, a implementação assume um único leitor e um único escritor. Numa situação com múltiplos leitores ou escritores (por exemplo: múltiplos threads) uma variável de bloqueio é necessária para prevenir “race conditions” de dados. Nessa situação, o manipulador de interrupção não pode utilizar um bloqueio e, portanto, precisa ter acesso exclusivo a uma extremidade da fila.

Controle de Fluxo por Hardware

Enquanto adicionar buffers dirigidos a interrupção melhora tanto a performance como a confiabilidade da transmissão de dados, isso não é suficiente para evitar prevenção de estouro de buffer e, portanto, perda de dados. Nessa seção discutimos o uso de dois sinais adicionais, RTS (Request To Send) e CTS (Clear To Send), que podem ser usados para habilitar o controle de “fluxo por hardware” (“hardware flow”). Esses sinais estão presentes tanto na ponte `usb-uart` (assumindo que você está usando um que os expõe) e o STM32. Na prática, nRTS (nCTS) do STM32 está conectado com nCTS (nRTS) da ponte `usb-uart`.⁴ Quando o STM32 é preparado para receber dados, ele ativa

⁴Usamos o nRTS para indicar que o sinal RTS é “active low”.

CAPÍTULO 11. INTERRUPÇÕES

(iguala a 0) o nRTS e, quando ele quer parar de receber dados, levanta (1) o nRTS. Da mesma maneira, a ponte usb-uart usará seu sinal nRTS (conectado com o nCTS do STM32) para indicar sua disponibilidade para receber dados. Esses dois pinos são descritos na documentação do STM32 [20] como segue:

Se o controle de fluxo RTS está habilitado..., então nRTS é afirmado (sinal baixo) enquanto o receptor USART estiver preparado para receber novos dados. Quando o registrador receptor está cheio, nRTS é desafirmado, indicando que é esperado que a transmissão pare no final do quadro (frame) atual...

Se o controle de fluxo CTS está habilitado..., então o transmissor verifica a entrada nCTS antes de transmitir o próximo quadro (frame). Se o nCTS é afirmado (sinal baixo), então o próximo dado é transmitido (supondo que um dado é transmitido ...), se não a transmissão não ocorre. Quando nCTS é desafirmado durante a transmissão, a transmissão atual é completada antes da transmissão parar.

Essa abordagem é chamada de controle de fluxo por hardware (“hardware flow control”) porque a sinalização é realizada no hardware em oposição aos mecanismos de software que dependem da inserção de caracteres de controle especiais no fluxo de dados. A solução que apresentamos não é totalmente voltada ao hardware – nós conduzimos o pino nRTS através do software (por exemplo: o manipulador de interrupção) para indicar para o dispositivo remoto que deve parar de transmitir e deixar o hardware USART parar o transmissor sempre que o dispositivo remoto desafirmar o nCTS.

Infelizmente, a abordagem totalmente automatizada está fadada ao fracasso. Para entender, considere essa declaração da faq da FTDI (FTDI é um importante produtor de chips ponte USB-UART):

Se [nCTS] tem nível lógico 1, está indicando que o dispositivo externo não aceita mais dados. O FTxxx vai parar de transmitir dentro de 0-3 caracteres, dependendo do que está no buffer.

Essa potencial saturação de 3 caracteres ocasionalmente apresenta problemas. Clientes devem ser informados que o FTxxx é um dispositivo USB e não um dispositivo “normal” RS232 como encontrado nos PCs. Como tal, o dispositivo opera a base de pacotes e não a base de bytes.”

11.5. EXEMPLO: COMUNICAÇÃO SERIAL COM INTERRUPÇÃO

Assim, o comportamento de dispositivos reais não está sempre em conformidade com as expectativas do STM32. A única solução viável é o software “desafirmar” nRTS enquanto ainda é capaz de receber dados adicionais de entrada. O problema é fundamentalmente insolúvel como não parece ser uma clara especificação que define a quantidade permitida de saturação de caracteres.

As mudanças reais necessárias ao código apresentado na Seção 11.5 são modestas. Precisamos configurar dois pinos adicionais para nCTS e nRTS, modificar a inicialização do USART para permitir o nCTS travar o transmissor e modificar tanto a rotina do manipulador de interrupção quanto a rotina `getchar`. Nossa estratégia é definir uma marca de “água alta” (“high water”) na entrada do buffer abaixo do qual o software afirma nRTS e acima do qual desafirma nRTS. A “high water” fornece espaço para caracteres adicionais chegarem depois da desafirmação do nRTS.

A Tabela 5.2 fornece as definições dos pinos para USART1. Além do tx e rx que já foram configurados previamente nós devemos configurar nRTS (PA12) e nCTS (PA11):

```
// Configure CTS pin

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOA, &GPIO_InitStructure);

// Configure RTS pin -- software controlled

GPIO_WriteBit(GPIOA, GPIO_Pin_12, 1);           // nRTS disabled
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

Nós modificamos ligeiramente a configuração da USART:

```
USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_CTS;
```

e habilitamos nRTS

```
// nRTS enabled

GPIO_WriteBit(GPIOA, GPIO_Pin_12, 0);
```

CAPÍTULO 11. INTERRUPÇÕES

Como discutido acima, nós adicionamos código a rotina do `getchar`

11.5. EXEMPLO: COMUNICAÇÃO SERIAL COM INTERRUPÇÃO

```
char getchar (void)
{
    uint8_t data;
    while (Dequeue(&UART1_RXq, &data, 1) != 1);

    // If the queue has fallen below high water mark, enable nRTS

    if (QueueAvail(&UART1_RXq) <= HIGH_WATER)
        GPIO_WriteBit(GPIOA, GPIO_Pin_12, 0);

    return data;
}
```

e à parte de recebimento do manipulador de interrupção:

```
if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
{
    uint8_t data;

    // clear the interrupt

    USART_ClearITPendingBit(USART1, USART_IT_RXNE);

    // buffer the data (or toss it if there's no room
    // Flow control is supposed to prevent this

    data = USART_ReceiveData(USART1) & 0xff;
    if (!Enqueue(&UART1_RXq, &data, 1))
        RxOverflow = 1;

    // If queue is above high water mark, disable nRTS

    if (QueueAvail(&UART1_RXq) > HIGH_WATER)
        GPIO_WriteBit(GPIOA, GPIO_Pin_12, 1);
}
```

Finalmente, nós definimos a high water mark in `uart.h`

```
#define HIGH_WATER (QUEUE_SIZE - 6)
```

Exercise 11.2 Comunicação Serial dirigida a Interrupções

- Complete a implementação de uma interrupção dirigida a UART com controle de fluxo.

CAPÍTULO 11. INTERRUPÇÕES

- Escreva uma aplicação simples que lê caracteres e os ecoa.

Você deve testar a aplicação canalizando um arquivo grande através da serial, capturando os resultados e comparando os arquivos de entrada e saída.

- Reescreva o programa acima para que ele leia/escreva linhas inteiras (até o limite de buffer de 80 caracteres.)
- Tente os mesmos testes com pooling baseado na uart.

Por agora, sua biblioteca de módulos deve parecer como:

```
Library
└── ff/
└── i2c.c
└── i2c.h
└── lcd.c
└── lcd.h
└── mmcbb.c
└── spi.c
└── spi.c
└── uart.c
└── uart.h
└── uart-fc.c
└── xprintf.c
└── xprintf.h
```

11.6 Interrupções Externas

O microcontrolador STM32 F1xx fornece até 20 possibilidades de fontes EXTI (EXternal Interrupt - interrupções externas); embora em muitos casos as várias fontes compartilham um único vetor de interrupção. As fontes possíveis e seus correspondentes eventos/vetores (para o STM32 F100) são:

11.6. INTERRUPÇÕES EXTERNAS

Evento	Origem	Vetor
EXT0	PA0-PG0	EXT0_IRQHandler
EXT1	PA1-PG1	EXT1_IRQHandler
EXT2	PA2-PG2	EXT2_IRQHandler
EXT3	PA3-PG3	EXT3_IRQHandler
EXT4	PA4-PG4	EXT4_IRQHandler
EXT5	PA5-PG5	EXT9_5_IRQHandler
...
EXT15	PA15-PG15	EXT1_10_IRQHandler
EXT16	PVD	PVD_IRQHandler
EXT17	RTC Alarm	RTC_WKUP
EXT18	USB Wakeup	not on STM32 F100
EXT19	Ethernet Wakeup	not on STM32 F100

Observe que somente um dos PAx-PGx, onde x varia de 1 a 15, pode ser configurado como uma fonte EXTI a qualquer momento. No caso de várias fontes EXTI compartilharem o mesmo manipulador, interrupções pendentes podem ser determinadas pela leitura o registrador pendente EXTI_PR. Também, qualquer fonte EXTI pode ser “disparada” (“triggered”) através do software definindo o bit apropriado no “software interrupt event register” EXTI_SWIER.

Configurar uma interrupção externa consiste em alguns passos:

1. Configurar o NVIC para o vetor correspondente.
2. Para pinos GPIO, configurar o registrador apropriado AFIO_EXTICR_x para selecionar o pino correto (por exemplo: PA0).
3. Definir a condição de disparo (borda de subida/descida ou ambos).
4. Definir o bit apropriado no registrador de máscara do evento.

Isso tudo pode ser feito usando os módulos da biblioteca padrão de periféricos `stm32f10x_nvic.[ch]` e `stm32f10x_exti.[ch]`. Por exemplo, configurar PA0 (conectada ao “user push button” da Discovery Board) para disparar uma interrupção na borda de subida:

```
// Connect EXTI0 to PA0
GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);

// Configure EXTI0 line // see stm32f10x_exti.h
```

CAPÍTULO 11. INTERRUPÇÕES

```
EXTI_InitStructure.EXTI_Line = EXTI_Line0;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);

// Configure NVIC EXTI0_IRQHandler ...
```

Um manipulador básico deve checar o status da interrupção e limpar qualquer bit pendente;

```
void EXTI0_IRQHandler(void){ if (EXTI_GetITStatus(EXTI_Line0) != RESET){ ... EXTI_ClearITPendingBit(EXTI_Line0); } }
```

Fontes EXTI também podem ser configuradas no “modo evento” (“event mode”). Nesse modo, elas não geram uma interrupção, mas ao invés, geram um evento interno que pode ser usado para acordar o processador do modo sleep. Por exemplo, a instrução WFE pode ser usada pelo software para entrar no modo Sleep de onde pode sair por um evento ou qualquer linha EXTI no modo evento.

Exercise 11.3 Interrupção Externa

Usando os fragmentos de código apresentados, escreva um programa simples que responde a eventos de bordas de subida (soltar o botão) no PA0 alternando o led3.

Ao escrever essa aplicação você provavelmente notará que, dependendo de como você solta o botão, muitas vezes, você não irá observar o comportamento esperado.

A seguinte citação do manual de referência é uma pista para entender o que está acontecendo:

Nota: As linhas externas de wakeup (despertar) são disparadas por bordas, nenhuma falha deverá ser gerada nessas linhas. Se uma borda de descida numa linha de interrupção externa acontecer durante a escrita do registrador EXTI_FTSR, o bit pendente não será setado.

Botões e interruptores são famosos por “bounce” – quando um botão é apertado, os contatos não se separam corretamente levando a vários picos. Isso pode ser visto na Figura 11.6 que mostra o comportamento real capturado

11.6. INTERRUPÇÕES EXTERNAS

em um osciloscópio – o traço no topo mostra a tensão no PA0 quando o botão é pressionado e o traço de baixo quando o botão é solto.

A solução é adicionar um circuito de de-bouncing que serve como um filtro “passa-baixa” (“low-pass”) para remover as mudanças rápidas do sinal. Um circuito simples de “de-bounce” está mostrado na Figura 11.6 que consiste em um resistor e um capacitor. Como a Discovery Board tem um resistor de pull-down conectado ao botão, o circuito sugerido será subaproveitado, porém irá eliminar as falhas mostradas na Figura 11.6.

Tente usar seu código com o circuito de de-bounce adicionado ao PA0.

CAPÍTULO 11. INTERRUPÇÕES

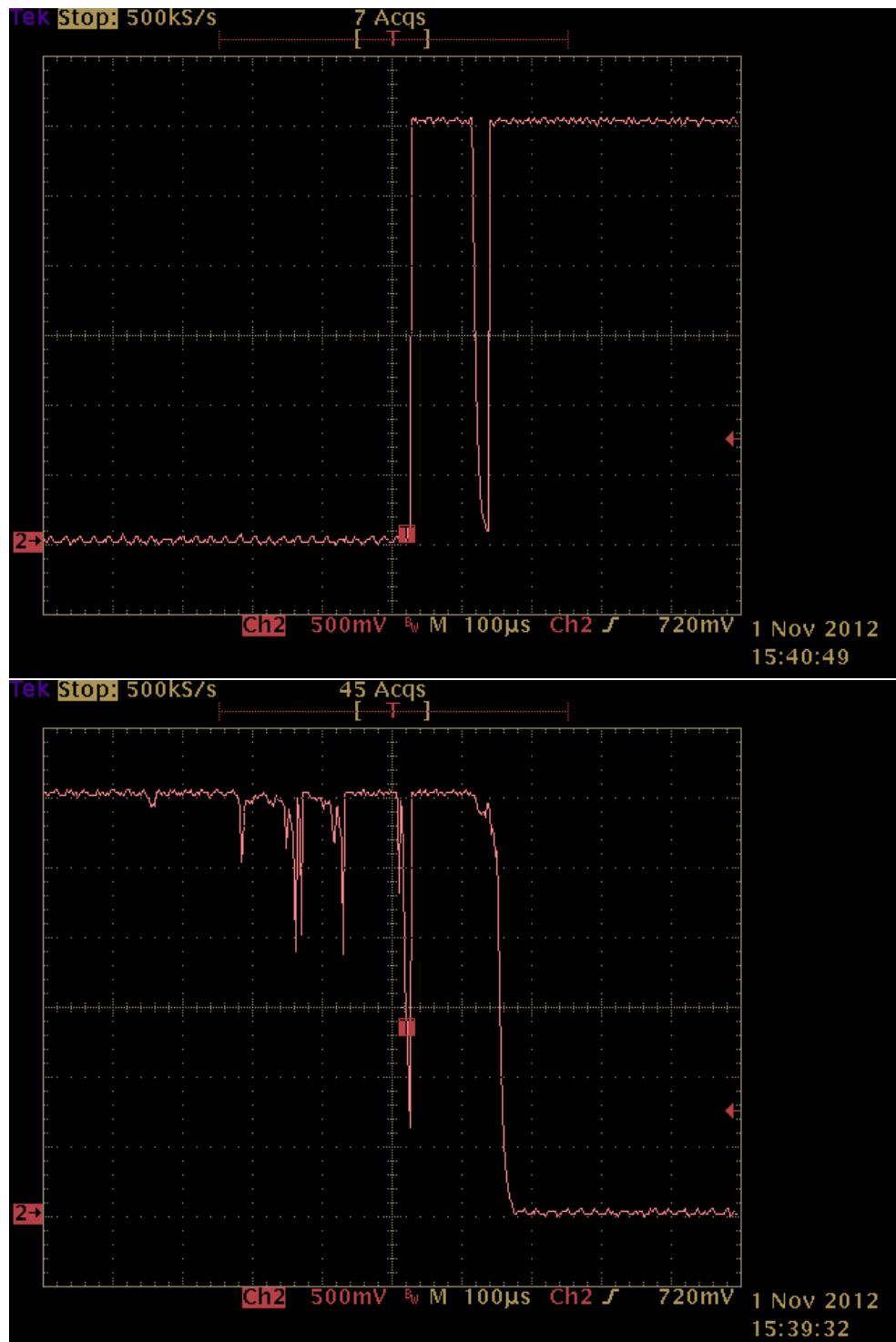


Figura 11.6: Button press/release

11.6. INTERRUPÇÕES EXTERNAS

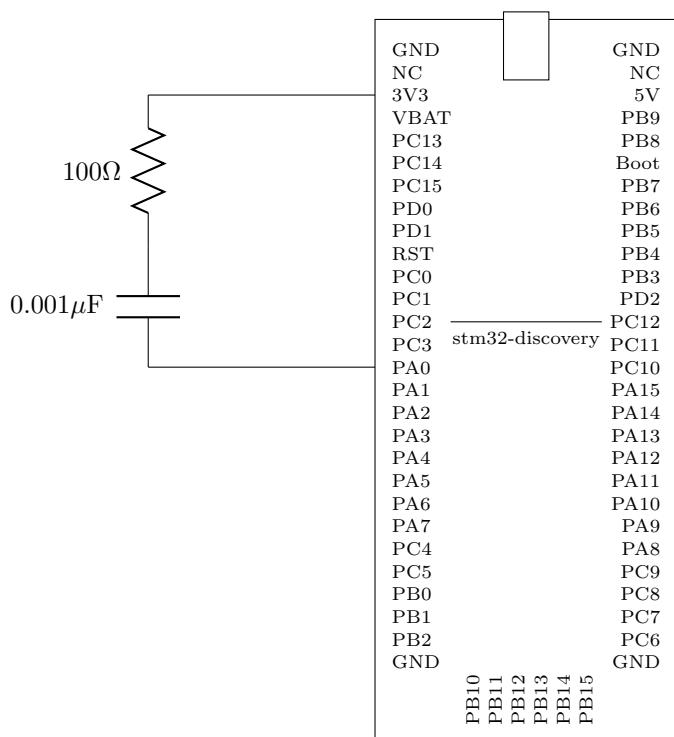


Figura 11.7: De-bounce Circuit

CAPÍTULO 11. INTERRUPÇÕES

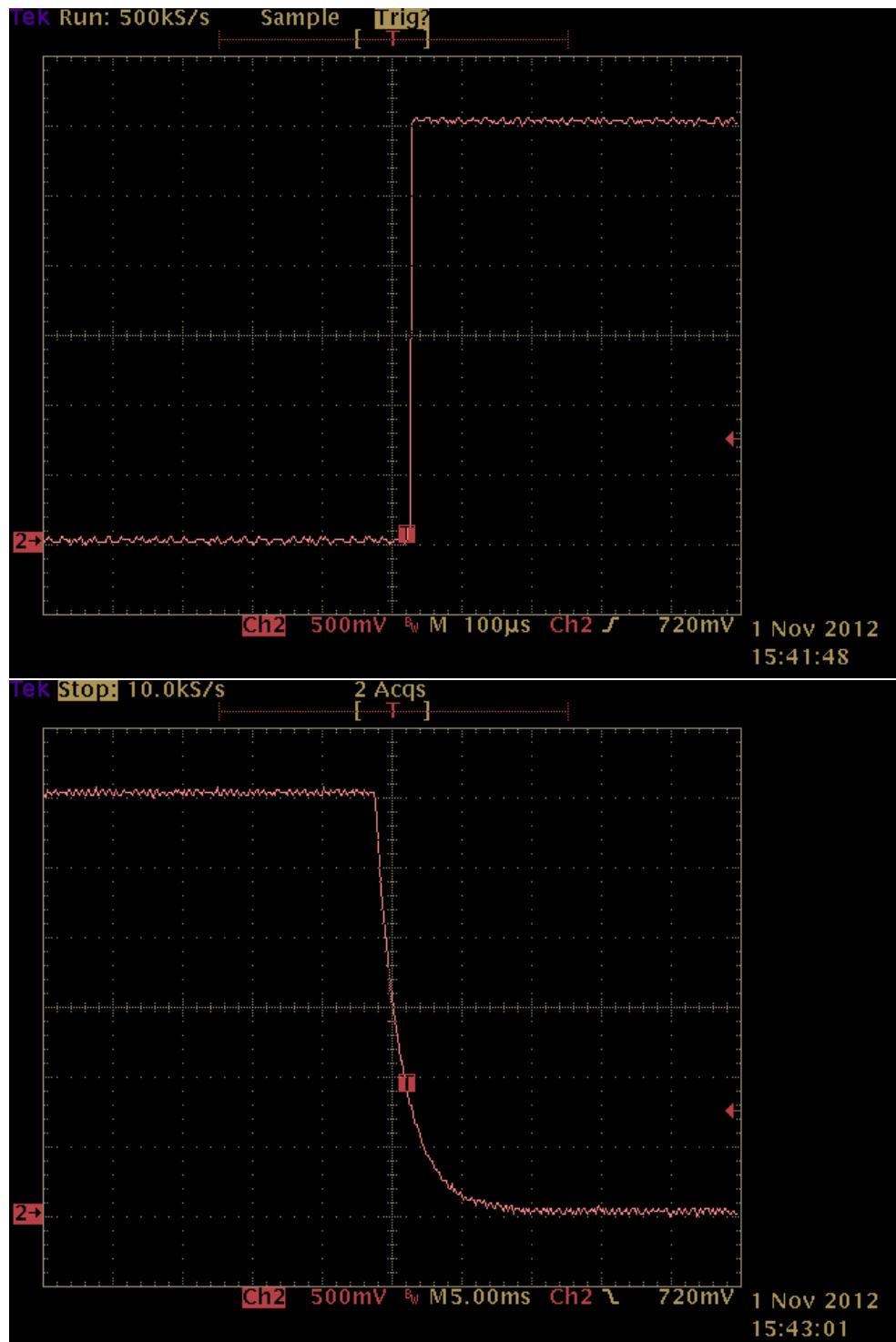


Figura 11.8: Button press/release (filtered)

Capítulo 12

DMA: Direct Memory Access

Neste capítulo, discutiremos o uso do acesso direto à memória (DMA - direct memory access) para aliviar o processador dos custos de transferência de blocos de dados entre a memória e os periféricos. Considere o seguinte código típico em que um bloco de dados é lido de um periférico repetidamente esperando por um flag de status para então ler um item do periférico.

```
for (i = 0; i < N; i++) {  
    while(flagBusy);  
    buf[i] = peripheralRegister;  
}
```

Vimos isso com a comunicação serial (Seção 5.1), a comunicação SPI (Listagem 6.3), e novamente com a comunicação I2C (Figura 9.3). Esta abordagem, chamada de polling de software, tem três limitações. Em primeiro lugar, o processador fica preso durante a transferência e não pode executar outras tarefas; se temos uma grande transferência de dados (por exemplo, a leitura de um setor de dados a partir de um cartão SD), a transferência poderia ser pausada e o processador ficaria livre para executar outras tarefas enquanto a transferência é realizada. Em segundo lugar, a taxa de transferência real é menor do que o hardware subjacente pode permitir. Finalmente, é difícil de alcançar limites de tempo restritos, por exemplo, streaming de áudio depende de transferência de dados a uma taxa constante.

Para ver as diferenças dramáticas de desempenho, considere duas telas de captura do Saleae Logic mostrando as transferências SPI, preenchendo um LCD 7735 com uma cor sólida, como apresentado na Figura 12. A captura de cima mede o tempo de transferência de 2 pixels, enquanto que a captura de baixo mede o tempo para a transferência de 128 pixels. O pico teórico é

CAPÍTULO 12. DMA: DIRECT MEMORY ACCESS

$12 \times 10^6 / 16 = 750,000$ pixels/segundo para um clock SPI de 12 MHz. Sem DMA a taxa de transferência é de $1/20 \times 10^{-6} = 50,000$ pixels/segundo, enquanto que com o uso de DMA a taxa de transferência é de $128/17 \times 10^{-4} = 735,000$ pixels/segundo. É claro que, com DMA, há uma inevitável sobrecarga entre os blocos. Assim, existe um tradeoff entre de taxa de transferência e o espaço de memória (blocos maiores produzem um throughput maior).

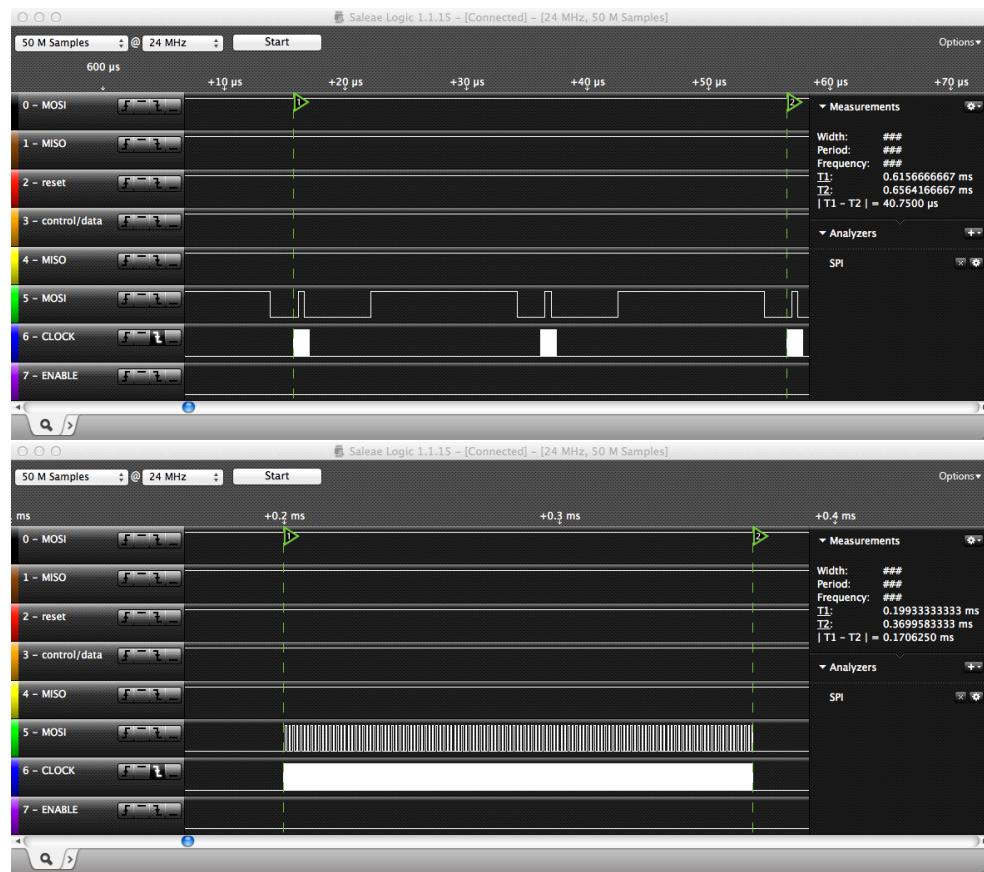


Figura 12.1: Color Fill com (em baixo) e sem (em cima) DMA

DMA é implementada em processadores com dispositivos de hardware dedicados. Estes dispositivos compartilham o barramento de memória e os barramentos de periféricos com o processador (CPU), tal como mostrado na Figura 12. Nesse diagrama, o dispositivo DMA lê a memória a partir do barramento da memória e grava em um periférico através do barramento do periférico. A situação com o STM32 é um pouco mais complicada, porque

12.1. ARQUITETURA DMA DO STM32

existem vários barramentos de periféricos, mas o princípio é o mesmo.

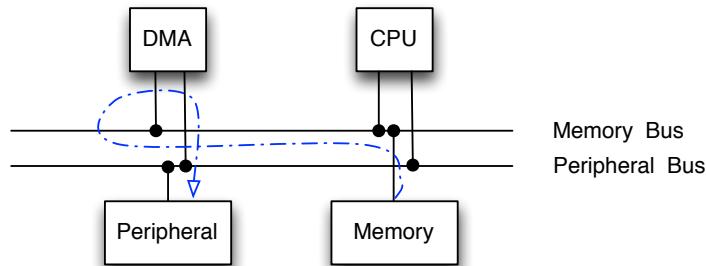


Figura 12.2: DMA Transfer

Somente um dispositivo pode utilizar o barramento de cada vez, então as transferências via DMA causam algum impacto na CPU. Entretanto, considere que o pico da taxa transferência para o dispositivo SPI seja de 750,000 transferências/segundo enquanto o barramento de memória da linha STM32 pode suportar 24,000,000/5 transferências da RAM/segundo (cada transferência compreende 5 ciclos do barramento). Assim, a nossa transferência por bloco consome aproximadamente cerca de 15% dos ciclos do barramento de memória. A arquitetura STM32 garante que o processador não ficará ocioso. Além disso, apenas uma fração das instruções do STM32 acessa diretamente a memória RAM – o resto simplesmente puxa instruções da FLASH que utiliza um barramento diferente.

12.1 Arquitetura DMA do STM32

O STM32 tem dois periféricos DMA e cada um possui vários “canais” independentes e configuráveis (7 para o DMA1 e 5 para DMA2). Um canal pode ser considerado como a realização em hardware de uma transação. Para inicializar o DMA entre um periférico e uma memória, é necessário configurar o canal apropriadamente. Por exemplo, o DMA1 canal 2 (3) pode ser usado para receber (transmitir) dados a partir de (para) o SPI1.

Antes de utilizar os periféricos DMA, lembre-se de habilitar os seus clocks! Por exemplo,

```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);+
```

CAPÍTULO 12. DMA: DIRECT MEMORY ACCESS

Configurar um canal consiste no estabelecimento de parâmetros adequados através de uma estrutura `DMA_InitTypeDef`:

```
typedef struct
{
    uint32_t DMA_PeripheralBaseAddr;
    uint32_t DMA_MemoryBaseAddr;
    uint32_t DMA_DIR;
    uint32_t DMA_BufferSize;
    uint32_t DMA_PeripheralInc;
    uint32_t DMA_MemoryInc;
    uint32_t DMA_PeripheralDataSize;
    uint32_t DMA_MemoryDataSize;
    uint32_t DMA_Mode;
    uint32_t DMA_Priority;
    uint32_t DMA_M2M;
}DMA_InitTypeDef;
```

Os parâmetros incluem o endereço base do periférico (por exemplo: `SPI1->DR`), o endereço de memória do buffer, a direção da transferência, o tamanho do buffer, etc. Uma vez que o canal do DMA foi inicializado, ele deve ser habilitado (`enabled`). O firmware STM32 fornece comandos específicos do periférico para habilitar uma transação DMA. O fragmento de código a seguir, mostra um exemplo para a transmissão de dados via SPI utilizando DMA.

```
DMA_Init(txChan, &DMA_InitStructure);
DMA_Cmd(txChan, ENABLE);
SPI_I2S_DMACmd(SPIx, SPI_I2S_DMAReq_Tx, ENABLE);
```

Múltiplos canais DMA podem ser inicializados simultaneamente. Com efeito, para o SPI a configuração natural utiliza o DMA tanto para transmissão quanto para recepção simultaneamente. A conclusão de um pedido DMA é detectada através de um flag apropriado. Vamos examinar mais esses detalhes na Seção 12.2.

Os canais DMA fornecidos pelo STM32 estão cada um associado com periféricos específicos (ver [20] para obter a documentação completa). Por exemplo DMA1 canal 1 suporta ADC1, TIM2_CH3, e TIM4_CH1. Na concepção de um sistema com o STM32, é importante estar ciente do potencial conflito de recursos. Por exemplo o DAC_Channel1 (conversor analógico digital) requer acesso para DMA1 Canal 3, que também é usado para o SPI1_TX. Assim, não é possível transmitir simultaneamente do SPI1 e receber em DAC_Channel1 usando DMA. Eu havia desenvolvido inicialmente exemplos SPI usando SPI1; no entanto, mais tarde eu descobri que eu precisava utilizar o DAC para saída de áudio que levou a uma troca pelo SPI2.

12.2. SUPORTE DMA PARA O SPI

12.2 Suporte DMA para o SPI

```
static int dmaRcvBytes(void *rbuf, unsigned count)
{
    DMA_InitTypeDef DMA_InitStructure;
    uint16_t dummy[] = {0xffff};

    DMA_DeInit(DMA1_Channel2);
    DMA_DeInit(DMA1_Channel3);

    // Common to both channels

    DMA_InitStructure.DMA_PeripheralBaseAddr =
        →(uint32_t)(&(SPI1->DR));
    DMA_InitStructure.DMA_PeripheralDataSize =
        →DMA_PeripheralDataSize_Byte;
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
    DMA_InitStructure.DMA_BufferSize = count;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
    DMA_InitStructure.DMA_Priority = DMA_Priority_VeryHigh;
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;

    // Rx Channel

    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)rbuf;
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;

    DMA_Init(DMA1_Channel2, &DMA_InitStructure);

    // Tx channel

    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) dummy;
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;

    DMA_Init(DMA1_Channel3, &DMA_InitStructure);
    /* ... */
}
```

Listing 12.1: SPI DMA Receive (Parte 1)

Uma rotina completa de recepção de dados em 8-bits utilizando DMA é mostrada nas Listagens 12.1 e 12.2. A estrutura de inicialização do DMA é configurada do seguinte modo:

CAPÍTULO 12. DMA: DIRECT MEMORY ACCESS

```
// Enable channels

DMA_Cmd(rxChan, ENABLE);
DMA_Cmd(txChan, ENABLE);

// Enable SPI TX/RX request

SPI_I2S_DMACmd(SPIx, SPI_I2S_DMAReq_Rx | SPI_I2S_DMAReq_Tx,
    →ENABLE);

// Wait for completion

while (DMA_GetFlagStatus(DMA1_FLAG_TC2) == RESET);

// Disable channels

DMA_Cmd(rxChan, DISABLE);
DMA_Cmd(txChan, DISABLE);

SPI_I2S_DMACmd(SPIx, SPI_I2S_DMAReq_Rx | SPI_I2S_DMAReq_Tx,
    →DISABLE);

return count;
}
```

Listing 12.2: SPI DMA Receive (Parte 2)

- Endereço base do periférico é setado para o registrador de dados do SPI.
- O tamanho de dados do periférico e da memória é setado para 8-bits (1 Byte).
- O incremento do periférico é desabilitado – todos os dados são lidos/-gravados de um único registrador DR.
- O tamanho do buffer é setado para o número de dados a serem transferidos (contador).
- Modo DMA Normal
- Prioridade High (alta)

O código em seguida, configura o lado da memória e inicializa o canal de recebimento seguido da reconfiguração do lado da memória e configuração

12.2. SUPORTE DMA PARA O SPI

do canal de transmissão. Para operações de leitura unidirecionais, as transmissões de dados são fornecidas através de um buffer “dummy” que é lido repetidamente durante a transferência (DMA_MemoryInc_Disable).

Uma vez que ambos os canais estão configurados, eles são habilitados, é feito um pedido SPI DMA, e finalmente, o código aguarda um flag de conclusão. Observe que nesta implementação, o processador estabelece a transferência e aguarda a conclusão, mas isso ainda envolve uma espera ocupada (busy waiting). Uma abordagem alternativa (não considerada aqui) é permitir uma interrupção do DMA e definir um manipulador apropriado (por exemplo: DM1_Channel1_IRQHandler). O código principal poderia, então, retornar e aguardar um evento de conclusão apropriada. Eu adiarei a discussão de tal arquitetura até que eu tenha a oportunidade de introduzir um RTOS.

Exercice 12.1 Módulo SPI DMA

Implemente uma rotina genérica de troca com SPI:

```
static int xchng_datablock(SPI_TypeDef *SPIx, int half, const void
    *tbuf, void *rbuf, unsigned count)
```

Sua configuração deve utilizar os canais de DMA para tanto o SPI1 ou o SPI2 como definido no momento da chamada. O tamanho da transferência deve ser configurado como 8 bits ou 16-bits, conforme definido pelo parâmetro `half`. Deve ser possível utilizar essa rotina como leitura/gravação/troca conforme necessário.

Você deve, então, criar um novo módulo SPI (o chamaremos de `spidma.c`) que implementa a interface da Figura 6.1 e utiliza o DMA para todas as transferências mais longas que 4 itens de dados (transferências mais curtas devem ser realizadas sem o DMA). Este módulo deve ser derivado seu módulo `spi.c` baseado em polling.

Exercice 12.2 Mostrando imagens BMP a partir do Fat File System

Escreva um aplicativo que lê imagens BMP (ver http://en.wikipedia.org/wiki/BMP_file_format) a partir de um cartão SD e os exibe na tela do 7735. Observe que as imagens BMP são normalmente armazenados de baixo para cima, para isso você deverá necessariamente definir adequadamente a direção ao usar a interface 7735. Para ajudar você a começar, um programa Linux simples que analisa arquivos BMP é mostrado nas Listagens 12.3 e 12.4. Você pode manipular imagens em diversos formatos (por exemplo: redimensionar, girar) com o programa `convert` do Linux. Seu programa provavelmente

CAPÍTULO 12. DMA: DIRECT MEMORY ACCESS

deve verificar o tamanho de imagens BMP que ele tenta mostrar e rejeitar aqueles que não são 128x160 e com cor de 24 bits. Seu programa terá que converter de cores de 24 bits para cores de 16 bits. A conversão é relativamente fácil – para cada byte RGB, selecione os 5 ou 6 bits mais baixos (RGB = 565) e agrupe-os em um único byte de cor de 16 bits. Finalmente, o programa deve percorrer o conjunto de arquivos BMP no diretório raiz de um cartão SD. Observação – lembre-se que o sistema de arquivos FAT16 usa nomes 8.3 – que é de 8 caracteres seguidos por um sufixo de 3 caracteres!

Para obter os benefícios do DMA você deve ler um bloco de pixels em um buffer, convertê-los e agrupa-los em um segundo buffer, e depois gravá-los. Experimente com diferentes tamanhos de bloco, 16, 32, 64 e 128 pixels, e calcule quanto tempo leva para carregar e exibir um arquivo. Você pode facilmente utilizar o temporizador do sistema de interrupção para medir os atrasos. Por exemplo, adicione uma variável que é incrementada em cada interrupção; zere essa variável antes de ler e de escrever uma imagem. Compare seus resultados com e sem DMA.

O Linux fornece várias ferramentas para a manipulação de imagens. Para redimensionar uma imagem JPEG para a geometria do LCD com 128x160:

```
convert -resize 128x160! input.jpg output.jpg
```

Conversão de um arquivo BMP pode ser feita com um par de ferramentas

```
jpegtopnm output.jpg | ppmtobmp > output.bmp
```

12.2. SUPORTE DMA PARA O SPI

```
#include <stdint.h>

struct bmpfile_magic {
    unsigned char magic[2];
};

struct bmpfile_header {
    uint32_t filesz;
    uint16_t creator1;
    uint16_t creator2;
    uint32_t bmp_offset;
};

typedef struct {
    uint32_t header_sz;
    int32_t width;
    int32_t height;
    uint16_t nplanes;
    uint16_t bitspp;
    uint32_t compress_type;
    uint32_t bmp_bytesz;
    int32_t hres;
    int32_t vres;
    uint32_t ncolors;
    uint32_t nimpcolors;
} BITMAPINFOHEADER;

struct bmppixel { // little endian byte order
    uint8_t b;
    uint8_t g;
    uint8_t r;
};
```

Listing 12.3: BMP File Structures

CAPÍTULO 12. DMA: DIRECT MEMORY ACCESS

```
#include <fcntl.h>
#include <stdio.h>
#include ``bmp.h''

struct bmpfile_magic magic;
struct bmpfile_header header;
BITMAPINFOHEADER info;

main(int argc, char *argv[])
{
    int f;
    if (argc > 1){
        if ((f = open(argv[1], O_RDONLY)) == -1)
        {
            perror(``problem opening file '');
            return 1;
        }
        read(f, (void *) &magic, 2);
        printf(``Magic %c%c\n'', magic.magic[0], magic.magic[1]);
        read(f, (void *) &header, sizeof(header));
        printf(``file size %d offset %d\n'', header.filesz,
               header.bmp_offset);
        read(f, (void *) &info, sizeof(info));
        printf(``Width %d Height %d, bitspp %d\n'', info.width,
               info.height, info.bitspp);
        close(f);
    }
}
```

Listing 12.4: Parsing BMP Files

Capítulo 13

DAC : Conversão Digital Analógica

A família STM32 F1xx tem um módulo de conversão digital analógico de 12 bits – (DAC) com dois canais de saída independentes – DAC1 (pino PA4) e DAC2 (pino PA5). Os canais podem ser configurados em modo de 8 bits ou de 12 bits e as conversões podem ser feitas de forma independente ou simultâneas. O modo simultâneo pode ser utilizado onde dois sinais independentes, mas sincronizados devem ser gerados – por exemplo, os canais esquerdo e direito de áudio estéreo. Muitas vezes é importante que a saída analógica seja atualizada em instantes precisos, às vezes controlada por hardware externo – assim, o processo de conversão pode ser desencadeado por temporizadores ou sinais externos. Finalmente, uma utilização comum para o hardware DAC é a geração de um sinal variável no tempo. Onde a taxa de amostragem é alta, é impraticável controlar o processo de conversão inteiramente através do software aplicativo ou mesmo com manipuladores de interrupção. Assim, cada canal DAC tem capacidade DMA que pode ser controlada pelo sinal de um trigger.

Neste capítulo, vamos explorar o uso de um único canal de DAC. Vamos começar com o caso mais simples – software de controle direto de uma saída analógica – o que é útil para a criação de sinais lentos que não têm que ser sincronizados no tempo. Nós, então, iremos explorar a geração de sinais de sincronização de tempo; primeiro com o módulo DAC fornecendo um gerador de ondas triângulares, e depois o código “interrupt driven” para produzir uma onda senoidal. Operações “interrupt driven” não são escaláveis para altas taxas de atualizações – eventualmente, todos os ciclos de processador serão exigidos apenas para atender a interrupção. Assim, vamos examinar o uso

CAPÍTULO 13. DAC : CONVERSÃO DIGITAL ANALÓGICA

de DMA para “alimentar” o DAC, com uma interrupção de frequência menor usada para atualizar o buffer DMA. Finalmente, vamos definir um exercício para ler arquivos de áudio a partir de um cartão SD e reproduzi-los, via DMA, a um amplificador de áudio externo impulsionado pelo DAC.

Advertência: Os exercícios preliminares deste capítulo não podem ser facilmente completados sem o acesso a um osciloscópio para exibir as formas de ondas. O exercício final, reproduutor de áudio pode ser concluído sem um osciloscópio, mas pode ser mais difícil de debugar.

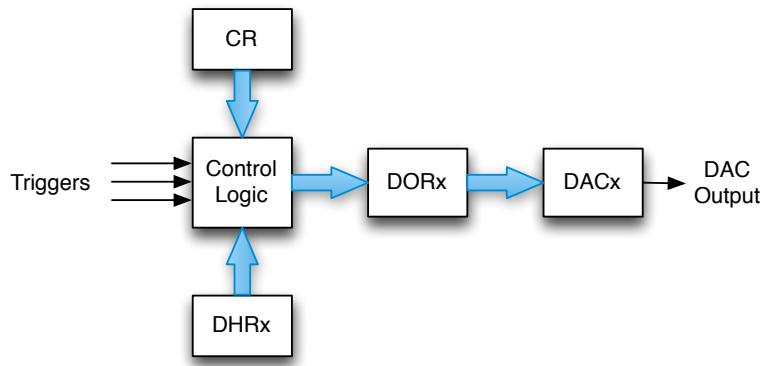


Figura 13.1: DAC Channel x

Um diagrama de blocos simplificado de um canal de DAC é mostrado na Figura 13. Cada canal tem uma lógica de controle separada que é configurada através de um único registo de controle (CR). Dados a serem convertidos pelo canal “x” são gravados nos “Data Holding Register” (DHRx) (registradores de armazenamento de dados). Em resposta a um evento de disparo, DHRx é transferido para o “Data Output Register” (DORx) (registador de saída de dados) e, depois de um tempo de repouso, o valor analógico correspondente aparece na saída. A tensão de saída do DAC é linear entre 0 e V_{REF+} (3,3 V na Discovery Board) e é definido pela seguinte equação:

$$DACoutput = V_{REF+} \times \frac{DOR}{4095}$$

Os eventos de trigger podem incluir os sinais de trigger de vários temporizadores (TIM2-TIM7, TIM15), uma interrupção externa (EXTI linha 9),

e de software. Também é possível configurar o DAC sem um trigger, no qual DHRx é automaticamente copiado para o DORx após um único ciclo de clock de atraso.

Exercice 13.1 Gerador de Formas de Onda

Escreva um programa em C para gerar uma onda senoidal com os valores mínimos e máximos de 512 e 1536, onde um ciclo completo é composto de 100 amostras. A saída de seu programa deve ter a forma:

```
int16_t a441[] = {  
    ...  
};
```

A fim de simplificar o alinhamento de dados e saída síncrona, a interface de software para os dois registradores que contém os dados é feita por meio de um conjunto de registradores cada um dos quais aborda um problema de alinhamento específico. Os vários exemplos são mostrados na Figura 13. Para cada um dos dois canais do DAC existem registradores alinhados para o lado esquerdo e direito dos dados de 12 bits bem como dados de 8 bits. Escrever nesses registradores afetam os registros correspondentes de DHRx como mostrado. Para habilitar a saída síncrona, existem três registradores de interface (DHR12RD, DHR12LD, e DHR8RD) para cada um dos três casos de alinhamento de dados. Observe que para o caso de dados de 8 bits, os bits de dados preenchem os bits 11..4 do DHR (os bits de ordem mais alta), o que preserva o intervalo completo da saída analógica em 0..VREF.

Além de converter os dados fornecidos pelo usuário, os canais DAC suportam independentemente a geração de ondas triangulares e de ruído. Neste modo, dados a serem convertidos são geradas em resposta a um evento de trigger. As formas de onda de saída podem ser úteis tanto para testar o DAC quanto para testar hardware externo.

Exercice 13.2 Application Software Driven Conversion

Uma utilização de hardware DAC comum é a de gerar uma saída analógica que muda com pouca frequência, e que não necessita ser sincronizada com qualquer sinal. Essa configuração segue o padrão para todos os dispositivos anteriores – habilitar os clocks, configurar os pinos, configurar os dispositivo, habilitar os dispositivos. No uso básico, a configuração padrão para a DAC é adequada, embora nós escolhemos habilitar um buffer de saída que fornece um pouco mais de acionamento elétrico em detrimento da velocidade, como mostrado na Listagem 13.1.

CAPÍTULO 13. DAC : CONVERSÃO DIGITAL ANALÓGICA

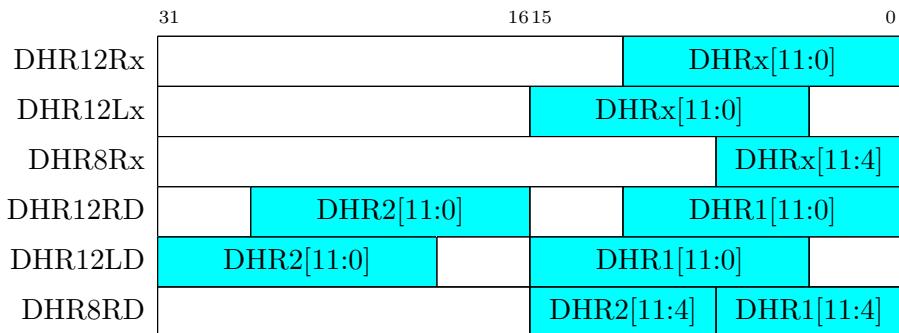


Figura 13.2: Data Holding Registers

De modo um pouco confuso, o pino é configurado como “analog in” (“entrada analógica”). O nome é um pouco enganador porque o módulo DAC impulsiona o pino quando configurado. No entanto, essa configuração realiza a importante função de desligar qualquer buffer de entrada digital! Uma vez que o DAC é inicializado, os dados podem ser escritos usando os seguintes comandos (definidos no `stm32f10x_dac.[ch]` onde “align” é um dos `DAC_Align_12b_R`, `DAC_Align_12b_L`, `DAC_Align_8b_R`.

```
DAC_SetChannel1Data(align,value);
DAC_SetChannel2Data(align,value);
```

Agora escreva um programa que, em um intervalo de 1 ms, grava os valores de saída de 12-bits correspondente a onda senoidal criada acima (um ciclo completo é composto de 100 amostras). Teste a saída examinando em um osciloscópio.

Exercise 13.3 Conversão Interrupt Driven

Forçar o programa principal do aplicativo a manter uma temporização precisa não é uma solução robusta. Primeiro, ele prende o programa de aplicação em um loop interno apertado, e em segundo lugar, na presença de interrupções, não podemos garantir que os dados serão enviados precisamente com o atraso desejado. O “jitter” de temporização introduzido pode ser um sério problema em algumas aplicações. Por exemplo, com áudio, jitter pode ser ouvido na forma de distorção.

Uma abordagem alternativa é configurar um timer para gerar interrupções na frequência de conversão desejada e usar um manipulador de interrupção para escrever o registo DHR. Uma vez que um temporizador é introduzido

```

GPIO_InitTypeDef          GPIO_InitStructure;
DAC_InitTypeDef          DAC_InitStructure;

RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);

GPIO_StructInit(&GPIO_InitStructure);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
GPIO_Init(GPIOA, &GPIO_InitStructure);

// DAC channel1 Configuration

DAC_StructInit(&DAC_InitStructure);
DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
DAC_Init(DAC_Channel_1, &DAC_InitStructure);

// Enable DAC

DAC_Cmd(DAC_Channel_1, ENABLE);

```

Listing 13.1: Inicialização do DAC

na solução, também podemos usar a interrupção de temporizador para provocar a transferência do DHR para o DOR. Isto protege o temporizador DAC de quaisquer atrasos possíveis devido a outras interrupções. Contanto que o manipulador de interrupção seja permitido para executar antes do próximo “tick” do temporizador não haverá jitter (instabilidade) no sinal de saída.

Neste exercício, iremos utilizar TIM3 tanto para acionar a transferência DAC quanto para chamar um manipulador de interrupção que será responsável por recarregar a DHR. Você pode modificar o exercício anterior, como segue.

- Configure o TIM3 para atualizar a 44.1 kHz – veja o Capítulo 10 para rever como isso é feito.
- Configure o trigger da saída do TIM3 para que ele ocorra nos eventos de atualização:

```
TIM_SelectOutputTrigger(TIM3, TIM_TriggerSource_Update);
```

- Configure o NVIC para habilitar o canal de interrupção TIM3_IRQn com a prioridade mais alta (0 tanto para a sub-prioridade quando a prioridade de preempção) – veja Capítulo 11 para ver como isso é feito.

CAPÍTULO 13. DAC : CONVERSÃO DIGITAL ANALÓGICA

- Mudar o trigger no código de inicialização do DAC:

```
DAC_InitStructure.DAC_Trigger = DAC_Trigger_T3_TRGO;
```

- Escreva um manipulador de interrupção adequado.

```
void TIM3_IRQHandler(void){  
}
```

Teste o seu programa primeiro usando um osciloscópio e, em seguida, conecte-o ao módulo alto-falante externo. Você deve ouvir um “A” (“Lá”) (na verdade, 441Hz) se tudo está funcionando corretamente.

Observe que para a configuração do timer, você não precisa definir o prescaler (divisor) para garantir que o período do timer seja um determinado número de ciclos, como fizemos para o PWM. Neste exemplo, estamos apenas preocupados com a frequência dos eventos de atualização – não em quantos ticks do clock ocorreram entre os eventos de atualização.

13.1 Exemplo: DMA Driven DAC

Enquanto um interrupt driven DAC tem menos problemas de jitter, em última análise, o manipulador de interrupção irá consumir todos os ciclos disponíveis. Com o uso de DMA, podemos reduzir a sobrecarga de software por um fator proporcional ao tamanho do buffer que estamos dispostos a alocar. O hardware DMA lida com os detalhes de escrever amostras individuais ao DAC, à taxa do temporizador enquanto um manipulador de interrupção é responsável por atualizar o buffer. Como veremos, com um buffer de N-itens, podemos reduzir a nossa taxa de interrupção por um fator de $N/2$.

Este exemplo se baseia no Exercício 13.3. As principais mudanças são:

- Inicializar um canal DMA (DMA1 Channel 3)
- Inicializar o NVIC para um manipulador de interrupções
- Criar um manipulador de interrupções

Continuamos a usar TIM3 para conduzir o processo de conversão; no entanto, ao invés de chamar um manipulador de interrupção para gerar o próximo ponto de referência, o dispositivo DMA fornece este dado. Nós assumimos que você gerou um array de 100 itens com o tom A440. Para este

13.1. EXEMPLO: DMA DRIVEN DAC

exemplo, vamos criar um buffer do mesmo tamanho – inicialmente uma cópia perfeita.

```
for (i = 0; i < A440LEN; i++)
    outbuf[i] = a440[i];
```

O manipulador de interrupção DAC é responsável pela recarga deste buffer em metades alternada (Listagem 13.2). Neste exemplo, a ação de recarga não é particularmente interessante, pois estamos simplesmente recopilando o tom A440 em nosso buffer de saída. No entanto, a ideia fundamental é que o manipulador de interrupção seja chamado com a transferência DMA estando concluída (DMA1_IT_TC3) e “half completed” (DMA_IT_HC3) – cada caso preenche uma metade diferente do buffer. Esta abordagem fornece a maior capacidade de resiliência a problemas potenciais de temporização.

```
int completed; // count cycles

void DMA1_Channel3_IRQHandler(void)
{
    int i;
    if (DMA_GetITStatus(DMA1_IT_TC3)){      // Transfer complete
        for (i = A440LEN/2; i < A440LEN; i++)
            outbuf[i] = a440[i];
        completed++;
        DMA_ClearITPendingBit(DMA1_IT_TC3);
    }
    else if (DMA_GetITStatus(DMA1_IT_HT3)){ // Half Transfer complete
        for (i = 0; i < A440LEN/2; i++)
            outbuf[i] = a440[i];
        DMA_ClearITPendingBit(DMA1_IT_HT3);
    }
}
```

Listing 13.2: DMA Interrupt Handler

A configuração é um pouco mais complicada por causa da necessidade de inicializar um canal DMA (Listagem 13.3). Observe que o registrador do periférico é o DHR12R1, e que o DMA está funcionando no “modo circular” o que significa que ele puxa continuamente a partir de um único buffer de memória. O canal de DMA está configurado para gerar interrupções nas metades e em todas as marcas de conclusão. Outro passo notável de configuração é a configuração do DAC para utilizar DMA DAC_DMACmd(DAC_Channel_1, ENABLE).

Exercice 13.4 Audio Player

CAPÍTULO 13. DAC : CONVERSÃO DIGITAL ANALÓGICA

```
// DMA Channel 3 Configuration

RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
DMA_DeInit(DMA1_Channel3);

DMA_StructInit(&DMA_InitStructure);
DMA_InitStructure.DMA_PeripheralBaseAddr = &DAC->DHR12R1;
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
DMA_InitStructure.DMA_PeripheralDataSize =
    →DMA_PeripheralDataSize_HalfWord;
DMA_InitStructure.DMA_BufferSize = A440LEN;
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) outbuf;
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
DMA_InitStructure.DMA_MemoryDataSize =
    →DMA_MemoryDataSize_HalfWord;
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
DMA_InitStructure.DMA_Priority = DMA_Priority_High;
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;

DMA_Init(DMA1_Channel3, &DMA_InitStructure);

// Enable Interrupts for complete and half transfers

DMA_ITConfig(DMA1_Channel3, DMA_IT_TC | DMA_IT_HT, ENABLE);

NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel3_IRQn;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    →//HIGHEST_PRIORITY;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

// Enable everything

DMA_Cmd(DMA1_Channel3, ENABLE);
DAC_Cmd(DAC_Channel_1, ENABLE);
DAC_DMACmd(DAC_Channel_1, ENABLE);
TIM_Cmd(TIM3, ENABLE);
```

Listing 13.3: Configuração DMA

Criar um reproduutor de áudio com a interface mostrada na Listagem 13.4. O reproduutor deve ser baseado em seu código anterior e no exemplo da Seção 13.1, mas com as seguintes diferenças: o tamanho dos dados deve ser de 8-bits, a taxa de dados deve ser configurável e o manipulador de interrup-

13.1. EXEMPLO: DMA DRIVEN DAC

```
#ifndef PLAYER_H
#define PLAYER_H

#define AUDIOBUFSIZE 128

extern uint8_t Audiobuf[];
extern int audioplayerHalf;
extern int audioplayerWhole;

audioplayerInit(unsigned int sampleRate);
audioplayerStart();
audioplayerStop();

#endif
```

Listing 13.4: Audio Player Interface

ção deve simplesmente setar os dois flags `audioplayerHalf` e `audioplayerWhole` quando as transferências da DMA estão completadas pela metade e completa, respectivamente. O trabalho de manter o buffer de áudio `Audiobuf` completo é empurrado para a aplicação que tem de consultar esses dois flags, reabastecer a metade inferior ou superior do buffer conforme o caso, e redefinir o flag; mais tarde, quando examinarmos um sistema operacional em tempo real, esta interface vai se adaptar facilmente a um sistema baseado em threads. A função “init” deve configurar o timer, canal de DMA, DAC, e a interrupção; mas não deve habilitar o hardware. A função “start” deverá habilitar o hardware – antes de chamar “start”, o aplicativo deve primeiro preencher o buffer de dados. A função “stop” deve desabilitar o hardware. Uma vez que um som é reproduzido integralmente o aplicativo precisa parar o reproduutor. Teste esta interface com uma onda senoidal. A seguir, vamos querer ler arquivos de áudio a partir de um cartão SD.

Um reproduutor de áudio fica muito mais interessante se ele puder ler arquivos “standard” de áudio. A seguir, vamos descrever um subconjunto do formato de arquivo WAV que pode ser gerado pela maioria dos software de áudio. Nós só consideraremos o caso para codificação PCM com um único canal de áudio (mono). Se você tem arquivos de áudio estéreo, em outros formatos ou com diferentes codificações, o programa “sox” do Linux fornece um utilitário poderoso de conversão. Você deve escrever um parser (analisador) para este subconjunto de WAV para testar em uma máquina desktop – basta imprimir os campos chave ao fazer o parsing (análise) de um arquivo.

Há muitas referências a arquivos WAV disponíveis na Internet. Uma

CAPÍTULO 13. DAC : CONVERSÃO DIGITAL ANALÓGICA

boa pode ser encontrada em <http://www-mmsp.ece.mcgill.ca/documents/audioformats/wave/wave.html>. Um arquivo WAV consiste de uma sequência (recursiva) de pedaços. Cada pedaço exceto o primeiro tem a seguinte forma:

Campo	Tamanho (bytes)	Conteúdo
ckID	4	Chunk ID (e.g. “RIFF”)
cksize	4	Chunk size n
body	n	Chunk contents

Com exceção do campo ckID, que é uma sequência de quatro caracteres em ordem big-endian, todos os outros campos estão em ordem little-endian. Os arquivos WAV que consideramos tem a seguinte sequência: pedaço Master (RIFF), pedaço formato, 0 ou mais pedaços, pedaços de dados.

Para fazer o parsing de arquivos WAV precisamos ser capazes de ler o pedaço do cabeçalho, fazer o parsing dos três tipos de pedaço de interesse, e pular os pedaços que não interessam. O pedaço mestre RIFF tem a seguinte forma:

Campo	Tamanho (bytes)	Conteúdo
ckID	4	Chunk ID: “RIFF”
cksize	4	Chunk size 4+n
WAVEID	4	WAVE ID: “WAVE”
WAVE chunks	n	Chunk contents

O pedaço Format tem a seguinte forma. Existem três variações para o formato, mas nós somente precisamos fazer o parsing da parte comum; entretanto o seu código deve saltar qualquer byte ignorado!

Campo	Tamanho (bytes)	Conteúdo
ckID	4	Chunk ID: “fmt ”
cksize	4	Chunk size 16 or 18 or 40
wFormatTag	2	0x0001 for PCM data
nChannels	2	Number of channels – 1 for mono
nSamplesPerSec	4	Sampling rate (per second)
nAvgBytesPerSec	4	Data rate
nBlockAlign	2	Data block size (bytes)
wBitsPerSample	2	Bits per sample – 8
	0-24	ignore

As peças-chave da informação são o formato (deve ser de 1 para PCM), o número de canais (deve ser de 1 para mono), a taxa de amostragem (a ser configurado no reproduutor), os bits por amostra (deve ser 8).

13.1. EXEMPLO: DMA DRIVEN DAC

O pedaço Format pode ser seguido por 0 ou mais pedaços de não-dados que o seu parser deve saltar. O pedaço final a considerar é o pedaço de dados:

Campo	Tamanho (bytes)	Conteúdo
ckID	4	Chunk ID: “data”
cksize	4	n
sampled data	n	Samples
pad byte	0 or 1	Pad if n is odd

Para escrever o seu parser utilize essas estruturas de dados úteis:

```
#define RIFF 'FFIR'
#define WAVE 'EVAW'
#define fmt 'tmf'
#define data 'atad'

struct ckhd {
    uint32_t ckID;
    uint32_t cksize;
};

struct fmtck {
    uint16_t wFormatTag;
    uint16_t nChannels;
    uint32_t nSamplesPerSec;
    uint32_t nAvgBytesPerSec;
    uint16_t nBlockAlign;
    uint16_t wBitsPerSample;
};
```

É conveniente comparar os IDs do pedaços com cadeias de caracteres; no entanto, uma string convencional em C tem um byte de terminação nula. Uma alternativa, mostrado no fragmento de código anterior é um caractere multi-byte (definido pela ordem inversa para lidar com endianess!).

O seu parser deve tomar um nome de arquivo WAV como argumento, abrir e analisar o arquivo. Como mencionado, ao fazer o parsing do arquivo, o parser deve imprimir os campos relevantes. Isso pode ser feito facilmente se você usar os comandos do sistema “open”, “close”, e “read”, e em seguida, pular os pedaços que não interessam:

```
lseek(fid, skip_amount, SEEK_CUR);
```

Uma vez que seu analisador esteja funcionando, é hora de criar um player de áudio que pode ler um arquivo WAV de um cartão SD e reproduzi-lo. O sistema de arquivos FAT descrito no Capítulo 8 possui os comandos para

CAPÍTULO 13. DAC : CONVERSÃO DIGITAL ANALÓGICA

ler **f_read**), e buscar (**f_lseek**) com interfaces ligeiramente diferentes do que o equivalente em Linux. Você deve modificar o seu reprodutor de áudio para abrir um determinado arquivo WAV (você precisa copiar um arquivo adequado para um cartão SD) a partir de um cartão SD, fazer o parsing do cabeçalho, inicializar o reprodutor, e depois “play” (“tocar”) o arquivo – aqui está um exemplo de como eu tratei o evento **audioplayerHalf** (o próximo é o número de bytes a serem copiados que é geralmente a metade do tamanho do buffer, exceto no final do arquivo).

```
if (audioplayerHalf) {
    if (next < AUDIOBUFSIZE/2)
        bzero(Audiobuf, AUDIOBUFSIZE/2);
    f_read(&fid, Audiobuf, next, &ret);
    hd.cksize -= ret;
    audioplayerHalf = 0;
}
```

Capítulo 14

ADC : Conversão Analógica Digital

O oposto de um DAC é um ADC (conversor analógico-digital) que converte sinais analógicos para valores digitais. Os processadores STM32 incluem um ou mais periféricos ADC – há somente um na família Medium Density Value Line. O STM32 ADC usa aproximação sucessiva – o ADC tem a capacidade para gerar um conjunto discreto de tensões e compará-los contra uma amostra de tensão de entrada; ele essencialmente realiza uma pesquisa binária para encontrar a melhor aproximação. Para 12 bits de precisão, o STM32 leva pelo menos 14 ciclos do clock ADC (um múltiplo do clock do sistema) – os dois ciclos extras são uma sobrecarga devida à amostragem. Assim, com um clock de ADC 12 MHz, o STM32 ADC pode executar uma amostra em pouco mais de $1\mu\text{seg}$.

Embora o componente STM32 VL tenha um único ADC, este pode suportar múltiplas entradas analógicas. A arquitetura básica está mostrada na Figura 14. Na Discovery Board, PA0-PA7, PB0-PB1 e PC0-PC5 podem todos ser usados como entradas analógicas que podem ser multiplexados para o ADC. O ADC pode ser configurado para amostrar qualquer subconjunto dessas entradas em sucessão. Existem dois modos básicos de operação – conversão única e contínua. Com a conversão única, uma vez que o ADC é acionado, ele converte uma única entrada e armazena o resultado no seu registro de dados (DR). O acionamento (trigger) pode vir do software ou de um sinal, tal como um temporizador. No modo contínuo, o ADC começa outra conversão logo que termina uma. O ADC também pode operar em modo de varredura, onde um conjunto de entradas a serem digitalizados está configurado. Uma conversão única é realizada para cada entrada configurada

CAPÍTULO 14. ADC : CONVERSÃO ANALÓGICA DIGITAL

sucessivamente. Varreduras podem também ser contínuas no sentido de que uma nova digitalização começa assim que uma outra é concluída.

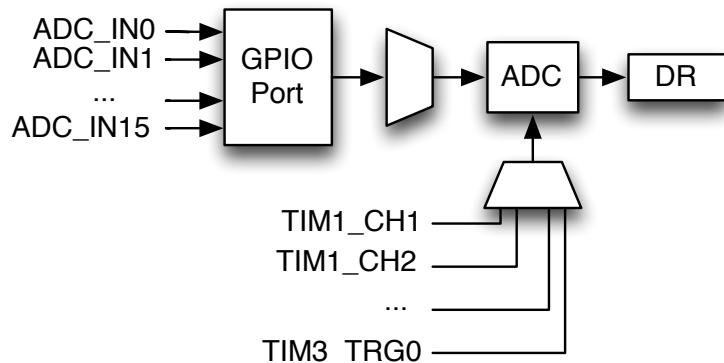


Figura 14.1: Analog Digital Converter

O ADC tem um único registo de dados, por isso, quando se digitaliza múltiplas entradas analógicas, é essencial que os dados sejam lidos entre as duas amostras. Isto pode ser realizado através de pooling – o software pode monitorar um flag, interrupção, ou DMA. Interrupções e DMA pode ser disparadas ao final de cada conversão. Neste capítulo mostraremos como usar o ADC para amostrar continuamente uma única entrada, para utilizar um temporizador para provocar a conversão de um único canal e uma interrupção de ADC para ler o resultado, e como usar o DMA para lidar com as conversões no caso de gatilhos temporizados. Apresentamos, também, um exercício para criar um gravador de voz que salva os arquivos WAV resultantes em um cartão SD.

É importante notar que a nossa apresentação ignora várias características adicionais do STM32 ADC (por exemplo, “canais injetados” (“injected channels”)), que são melhores compreendidas através da leitura do manual de referência.

14.1 Sobre ADCs de Aproximação Sucessivas

Para entender completamente o ADC do STM32 é importante ter alguma noção de como um ADC de aproximação sucessiva opera. Considere a Figura 14.1 que mostra um ADC típico. Na parte central encontra-se um conversor digital analógico (DAC). Para iniciar uma conversão, um controle

14.1. SOBRE ADCS DE APROXIMAÇÃO SUCESSIVAS

de hardware “captura” uma amostra de tensão de entrada (V_{ain}) – mostrado aqui com alguma resistência externa R_{ain} a ser discutido em seguida. Uma vez que a entrada (V_{samp}) é capturada, o controlador gera uma sequência de aproximações digitais $D(v_{est})$ e verifica cada uma, convertendo a aproximação de um sinal analógico, que é então comparado com a entrada amostrada. A sequência de aproximações corresponde a uma busca binária – de MSB e LSB. Uma vez que a melhor aproximação é encontrada, o valor digital é carregado em um registrador de dados (DR). Para uma aproximação de N-bits, o processo de aproximação requer N iterações.

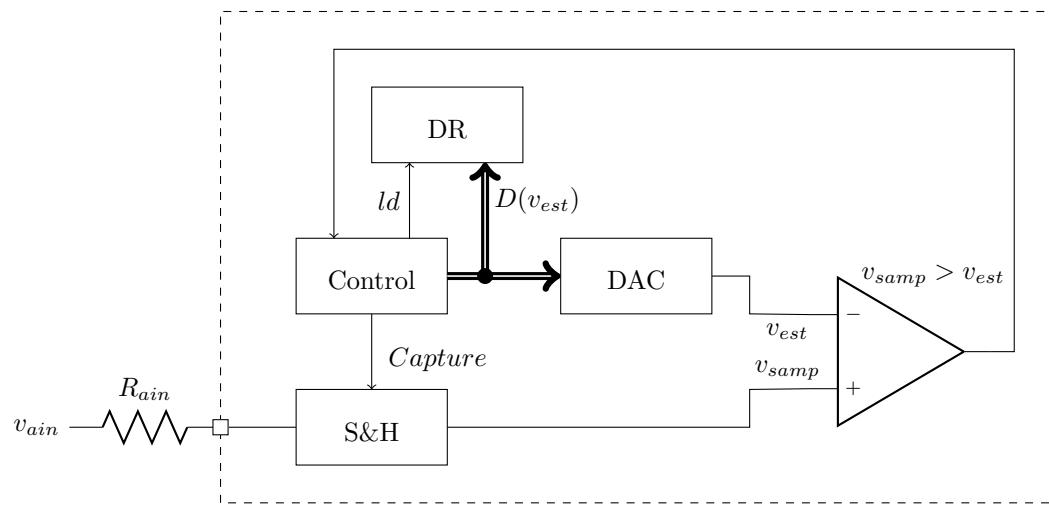


Figura 14.2: Successive Approximation ADC

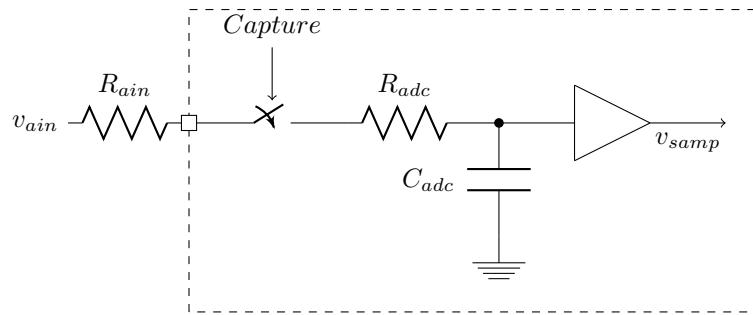


Figura 14.3: Sample and Hold

CAPÍTULO 14. ADC : CONVERSÃO ANALÓGICA DIGITAL

O circuito de amostragem e retenção opera através da carga de um capacitor interno mostrado na Figura 14.1. Durante a fase de captura, uma chave (transistor) é fechada permitindo carregar/descarregar o capacitor interno C_{adc} para a tensão v_{ain} . Este é o lugar onde a resistência externa entra em jogo. Se a fonte de tensão externa tem “alta impedância”, isso pode não produzir uma grande corrente atrasando o carregamento do capacitor. Em contraste, uma fonte de baixa impedância pode carregar o capacitor rapidamente. Assim, o tempo necessário para capturar uma tensão de entrada é dependente do circuito externo; no STM32, o tempo de captura é configurável de 1,5 até 239,5 ciclos para acomodar diferenças da entrada. Para a conversão de alta velocidade, pode ser necessária a utilização de um amplificador operacional externo para buferizar a entrada, a fim de converter um fonte de alta impedância em uma entrada de baixa impedância. Quando a velocidade não é um problema, o melhor é escolher o tempo mais longo de amostra que seja viável.

Para entender o impacto do tempo de amostra na precisão, considere o gráfico da Figura 14.1. O tempo necessário para carregar o capacitor interno C_{adc} é dependente das impedâncias internas e externas –

$$t_c = C_{adc} * (R_{adc} + R_{ain})$$

com a tensão do capacitor no instante t igual a:

$$v_{samp} = v_{ain} \times (1 - e^{-t/t_c})$$

Em geral, é necessário esperar um múltiplo de t_c para alcançar uma precisão razoável. Por exemplo dado t_c podemos calcular o tempo de amostra necessária para atingir 12 bits de precisão:

$$\begin{aligned} \frac{v_{samp}}{v_{ain}} &> 1 - \frac{1}{2^{12}} \\ 1 - e^{-t/t_c} &> 1 - 2^{-12} \\ e^{-t/t_c} &\leq 2^{-12} \\ \frac{-t}{t_c} &\leq \ln(2^{-12}) \\ \frac{-t}{t_c} &\leq -12 \times \ln(2) \\ t &> 8.32 \times t_c \end{aligned}$$

14.1. SOBRE ADCS DE APROXIMAÇÃO SUCESSIVAS

O modelo aqui apresentado é bastante simplista, mas ilustra algumas das principais ideias. Outros fatores a serem considerados são layout da placa (capacitância parasita), tensões de referência, o ruído do sinal, etc. Tudo isso é descrito em detalhes em [12]. Isso é suficiente para concluir dizendo que uma conversão analógica-digital rápida e precisa é complexa; no entanto, muitas aplicações não exigem tanto alta velocidade ou alta precisão. Em tais aplicações, apenas uma atenção modesta aos detalhes é necessária.

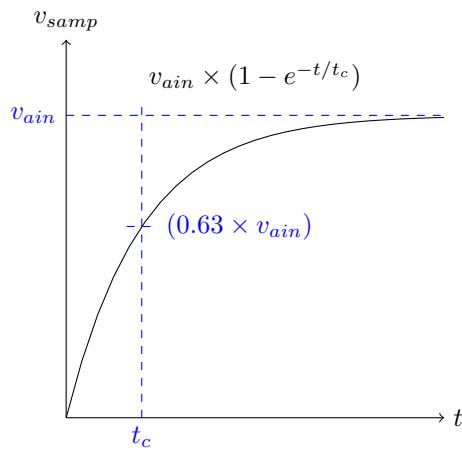


Figura 14.4: Impact of Sample Time on Accuracy

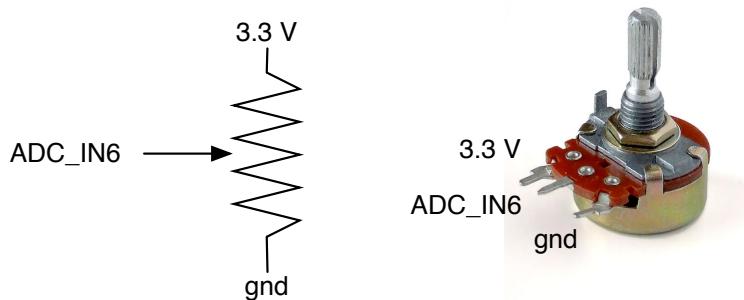


Figura 14.5: Simple Analog Input

Exercise 14.1 Amostragem Continua

CAPÍTULO 14. ADC : CONVERSÃO ANALÓGICA DIGITAL

Neste exemplo, uma fonte analógica simples – um potenciômetro será ligado a PA6 (ADC_IN6). Tal como mostrado na Figura 14.1 dois terminais do potenciômetro devem ser conectados a 3,3 V e GND, enquanto o centro é ligado a ADC_IN6. No lado esquerdo da figura está o símbolo elétrico de um potenciômetro e à direita uma fotografia de um potenciômetro típico. Um potenciômetro é um resistor variável; nesta aplicação usamos como um divisor de tensão que pode produzir qualquer entrada entre 0 e 3,3 V.

A aplicação irá ler continuamente a entrada analógica e, se for acima de 3,3/2 Volts, acender o LED verde, ou desligar o LED caso contrário. Até agora você já deve saber como fazer o seguinte:

- Configurar os clocks para GPIOA, GPIOC, e o ADC.
- Configurar a porta PA6 como entrada analógica.
- Configure a porta PC9 como saída push/pull.

Configurar o ADC segue um padrão familiar:

```
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
ADC_InitStructure.ADC_ScanConvMode = DISABLE;
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
ADC_InitStructure.ADC_ExternalTrigConv =
    →ADC_ExternalTrigConv_None;
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStructure.ADC_NbrOfChannel = 1;

ADC_Init(ADC1, &ADC_InitStructure);

// Configure ADC_IN6

ADC-RegularChannelConfig(ADC1, ADC_Channel_6, 1,
    →ADC_SampleTime_55Cycles5);

// Enable ADC

ADC_Cmd(ADC1, ENABLE);
```

Observe que ao configurar o canal, temos que selecionar um “tempo de amostragem”. O mínimo para 12 bits de precisão é um tempo de amostragem de 13,5 ciclos. Escolhendo o tempo de amostragem “certo” pode ser complicado se é desejável combinar velocidade e precisão. Nossas exigências para este exemplo são pequenas.

14.1. SOBRE ADCS DE APROXIMAÇÃO SUCESSIVAS

Ao contrário de outros periféricos, o ADC precisa ser calibrado. O hardware suporta calibração automática como mostrado no código a seguir. Novamente para este exemplo, nós provavelmente podemos dispensar calibração.

CAPÍTULO 14. ADC : CONVERSÃO ANALÓGICA DIGITAL

```
// Check the end of ADC1 reset calibration register  
  
while(ADC_GetResetCalibrationStatus(ADC1));  
  
// Start ADC1 calibration  
  
ADC_StartCalibration(ADC1);  
  
// Check the end of ADC1 calibration  
  
while(ADC_GetCalibrationStatus(ADC1));
```

O que resta é ler o resultado da conversão:

```
ain = ADC_GetConversionValue(ADC1);
```

Só resta então criar um aplicativo funcional.

Exercise 14.2 Conversão Dirigida pelo Timer

Neste exercício, iremos converter o exercício anterior, para ser dirigido por um temporizador com um manipulador de interrupção para executar sempre que a conversão for concluída. Nós teremos que alterar um pouco o código de inicialização do ADC, configurar um temporizador, e configurar o NVIC.

Você deve configurar o NVIC para habilitar `ADC1_IRQHandler` – a prioridade específica é relativamente pouco importante. Configure TIM3 para gerar um evento TRG0 a cada 1ms. Configure o ADC como se segue:

14.1. SOBRE ADCS DE APROXIMAÇÃO SUCESSIVAS

```
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
ADC_InitStructure.ADC_ScanConvMode = DISABLE;
ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
ADC_InitStructure.ADC_ExternalTrigConv =
    ADC_ExternalTrigConv_T3_TRGO;
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStructure.ADC_NbrOfChannel = 1;

ADC_Init(ADC1, &ADC_InitStructure);

ADC-RegularChannelConfig(ADC1, ADC_Channel_6, 1,
                        ADC_SampleTime_55Cycles5);

ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);

ADC_ExternalTrigConvCmd(ADC1, ENABLE);

ADC_Cmd(ADC1, ENABLE);
```

Mova o código que alterna a LED para um manipulador de interrupção com a seguinte estrutura:

```
void ADC1_IRQHandler(void)
{
    // read ADC DR and set LED accordingly
    ADC_ClearITPendingBit(ADC1, ADC_IT_EOC);
}
```

Exercise 14.3 Gravador de Voz

TBD

Capítulo 15

NewLib

Ao longo deste livro temos trabalhado sem as bibliotecas-padrão C (`libc`), apesar de uma implementação dela ser distribuída com as ferramentas Sourcery que utilizamos para a construção dos projetos. A principal razão para termos evitado a biblioteca (`libc`), até agora, é manter a clareza sobre qual código estamos realmente executando – como veremos, o uso da `libc` pode rapidamente levar a ocultação dos códigos. Uma segunda razão é que a `libc` é uma devoradora de memória – com apenas 8k de SRAM disponível no processador da Discovery Board, não há muito espaço de sobra. No entanto, a primeira consideração é agora discutível – você teve uma introdução bastante completa sobre como escrever códigos para o STM32, e a segunda consideração é muito menos importante, se você passar para os membros maiores da família de processadores STM32. Há muitas vantagens em usar a `libc` incluindo o acesso às funções da biblioteca padrão.

A implementação da `libc` distribuída com as ferramentas Sourcery e amplamente utilizada para sistemas embarcados é a “newlib”. A newlib inclui as `stdlib` (p. ex.: `abs`, `atoi`, `atoll`, `malloc`, e suporte a `unicode`), `stdio` (`printf`, `sprintf`, `scanf`, ...), suporte a `string`, e muitos outras. A newlib também inclui uma implementação da biblioteca matemática padrão, `libm`. Infelizmente, as funções `stdio` são devoradoras de memória visto que elas alocam grandes buffers de memória para suportar os sistemas de I/O. Além disso, por padrão (por exemplo, a distribuição Sourcery) a newlib não está configurada para minimizar o uso de memória. Assim, também mostraremos como compilar a newlib a partir do código fonte de maneira a otimizar o uso da memória.

A arquitetura da newlib requer uma implementação externa das principais funções do sistema, incluindo – `open`, `close`, `read`, `write`, e `sbrk` – o

último deles é o alicerce sobre o qual a `malloc` é construída. A fim de usar a `libc`, é necessário fornecer pelo menos um esboço para estas e outras funções. Vamos mostrar como projetar esses esboços para suportar o I/O padrão utilizando a USART da STM32.

Este capítulo está organizado da seguinte forma. Vamos começar com um exemplo simples – “hello world” (surpresa!) para mostrar os problemas que precisam ser resolvidos, a fim de usar a newlib, e os problemas de memória que ele suscita. Lembre-se que, no Capítulo 3, salientou-se que “hello world” é na verdade um programa complexo, embora a maioria da complexidade este “sob o capô”. Finalmente, vamos mostrar como compilar a `libc` de uma maneira que minimize os requisitos de memória (especialmente SRAM). Se você for usar a `libc` em uma variante do STM32 com SRAM substancial (pelo menos 20k), este passo é desnecessário – você pode usar as bibliotecas distribuídas com as ferramentas Sourcery.

15.1 Hello World

“Hello World” é o exemplo de programação mais famoso no universo C. Como dissemos no Capítulo 3, é realmente muito complexo. Considere o programa:

```
#include <stdio.h>
main() {
    printf("hello world\n");
}
```

`printf` é uma função da biblioteca que recebe uma string de formato e uma lista opcional de parâmetros. Deve transformar esses parâmetros em uma única string fundindo com a string de formato, e escrever a string resultante para a `stdout`. O que é `stdout`? É uma “buffered stream” – as bibliotecas `stdio` gerenciam buffered streams (fluxos de buffers). Os dados são bufferizados a medida em que são escritos, e então os dados no buffer são gravados em um arquivo ou dispositivo. Neste caso, a nossa intuição é que este fluxo deve estar de alguma forma ligado a uma UART para imprimir em uma tela em algum lugar. Enquanto o nosso uso de `printf` é trivial, invocações mais complexas alocam memória, a fim de ter espaço para a realização de todas as conversões.

Se tentarmos compilar “hello world” (crie um projeto e tente isso!), saberemos de imediato que há uma série de funções indefinidas (Listagem 15.1).

15.1. HELLO WORLD

```
undefined reference to `_sbrk'  
undefined reference to `_write'  
undefined reference to `_close'  
undefined reference to `_fstat'  
undefined reference to `_isatty'  
undefined reference to `_lseek'  
undefined reference to `_read'
```

Listing 15.1: Undefined Functions in newlib

Em um ambiente desktop, todos eles correspondem às chamadas do sistema operacional – `libc` é apenas uma biblioteca de código, em última análise, ela precisa de acesso a uma API do sistema operacional. A fim de usar a `newlib`, nós devemos fornecer a funcionalidade ausente na forma de procedimentos. Algumas delas nós substituiremos com esboços simples, outros (`read` e `write`) com o código que acessa uma UART.

Como mencionamos, a `libc` aloca memória – um monte dela – a partir do “heap”. A `libc` fornece funções para gerenciar o heap (`malloc` e `free`), mas depende de uma função do sistema `_sbrk` para alocar a memória que ela gerencia. Considere a Figura 15.1 repetida do Capítulo 2, que mostra a utilização da RAM por um programa de execução. A heap cresce para cima a partir dos dados alocados pelo compilador na direção da pilha que cresce para baixo. Dentro do heap, a memória é alocada pela `malloc` e desalocada pela `free`; no entanto, sempre que `malloc` tem espaço livre insuficiente, ele pede mais memória via `_sbrk`, que tem o efeito de mover o heap para cima (ou para baixo para um pedido de negativo). Em um sistema desktop, isso é implementado alocando-se mais páginas de memória, de fato `malloc` geralmente pede blocos de memória que são números inteiros de páginas (p. ex.: 4096 bytes), o que é um problema em um sistema limitado de memória, como a placa Discovery.

A fim de implementar `_sbrk` precisamos saber onde o heap começa (isto é, o final da memória alocada pelo compilador/linkeditor) e qual é o seu limite físico (ou seja, o valor máximo para o final do heap). No script do linker, definimos dois valores `_end` – o fim do segmento de dados – e `_stackend` que é o limite do espaço reservado para a pilha.¹

Antes de desenvolver uma implementação de `_sbrk`, é esclarecedora a leitura da página manual do Linux para `sbrk` (que chama a função do sistema

¹Calcular o tamanho da stack (pilha) pode ser desafiador – especialmente na presença de uma biblioteca de código. É melhor ser relativamente conservador!

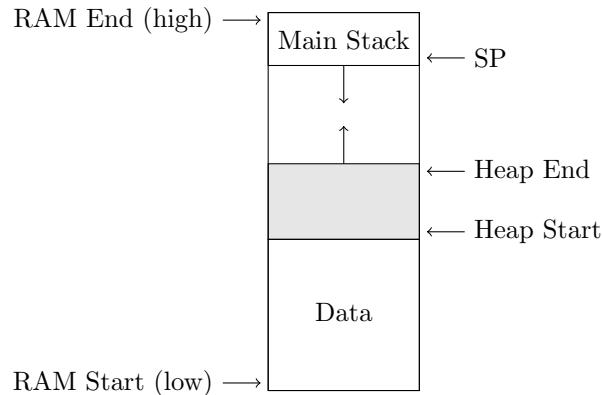


Figura 15.1: Program Memory Model

`_sbrk).`

```
void *sbrk(intptr_t increment);
```

...

As funções `brk()` e `sbrk()` alteram o local de término do programa, o qual define o fim do segmento de dados do processo (ou seja, o ponto de término do programa é o primeiro local após o final do segmento de dados não inicializado). Aumentar o local de término de um programa tem o efeito de atribuição de memória para o processo; diminuir o local término desaloca memória.

Alguns outros pontos-chave serão vistos a seguir. A função `_sbrk(0)` retorna o atual “término do programa” (“program break”). Se a chamada não puder ser satisfeita, `_sbrk` retorna -1 e define o `errno` atual para um código de erro apropriado. Na `newlib`, `errno` é parte de uma estrutura usada para fazer a biblioteca reentrante – esta estrutura “reent” é outra grande fonte de uso de memória que deve ser replicada em um ambiente multi-threaded. Uma implementação da `_sbrk` é mostrada na Listagem 15.2

Os esboços (stubs) restantes são fornecidos na Listagem 15.3. Operações de `read` e `write` utilizam a interface `putc` e `getc` descritas no Capítulo 5; em ambos os casos, restringimos nossas transferências para um único caractere. Observe que a maioria dos esboços restantes simplesmente retornam um código de erro; uma exceção é a `_isatty` que retorna 1 uma vez que estamos usando a UART como um terminal. A `_fstat` que fornece meta-dados para

15.1. HELLO WORLD

```
#include <errno.h>

// defined in linker script

extern caddr_t _end, _stackend;

caddr_t _sbrk(int nbytes){
    static caddr_t heap_ptr = NULL;
    caddr_t base;

    if (heap_ptr == NULL) {
        heap_ptr = (caddr_t)&_end;
    }

    if ((caddr_t) &_stackend > heap_ptr + nbytes) {
        base = heap_ptr;
        heap_ptr += nbytes;
        return (base);
    } else {
        errno = ENOMEM;
        return ((caddr_t)-1);
    }
}
```

Listing 15.2: Implementação de `_sbrk`

arquivos abertos sempre define o modo do arquivo para a `S_IFCHR` que indica um dispositivo orientado a caractere.

Há um detalhe final requerido para utilizar a newlib. O código de inicialização deve chamar a `_libc_init_array(void)` antes do `main()`. Isso garante que todas as estruturas de dados necessárias estarão corretamente inicializadas. Em nosso código de inicialização, nós fornecemos uma implementação padrão “fraca” para o caso de se compilar sem a `libc`.

```
void __attribute__((weak)) __libc_init_array (void){}

void Reset_Handler(void) {
    ...
    __libc_init_array();
    main();
    ...
}
```

Exercice 15.1 Hello World

CAPÍTULO 15. NEWLIB

Complete e teste o exemplo ”Olá mundo.” Você deve colocar todo o código stub em um único arquivo – **syscalls.c** e compilar este com o seu código.

15.1. HELLO WORLD

```
int _close(int file) {
    errno = ENOTSUP;
    return -1;
}

int _fstat(int file, struct stat *st) {
    st->st_mode = S_IFCHR; // character device
    return 0;
}

int _isatty(int file) {
    return 1;
}

int _link(char *old, char *new) {
    errno = EMLINK;
    return -1;
}

int _lseek(int file, int ptr, int dir) {
    errno = ENOTSUP;
    return -1;
}

int _open(const char *name, int flags, int mode) {
    errno = ENOTSUP;
    return -1;
}

int _read(int file, char *ptr, int len) {
    if (len){
        *ptr = (char) uart_getc(USART1);
        return 1;
    }
    return 0;
}

int _unlink(char *name) {
    errno = ENOENT;
    return -1;
}

int _write(int file, char *ptr, int len) {
    if (len) {
        uart_putc(*ptr, USART1);
        return 1;
    }
    return 0;
}
```

Listing 15.3: Stubs Básicos de Syscall
Revision: (None) ((None))

15.2 Construindo a newlib

A distribuição da newlib com a ferramenta Sourcery não foi compilada para o uso mínimo de memória. Você pode construir sua própria versão baixando as fontes newlib e usando o seguinte processo de compilação:

```
mkdir newlib-build
cd newlib-build
export CFLAGS_FOR_TARGET=''-g -O2 -DSMALL_MEMORY''
/path_to_newlib_source/configure --target=arm-none-eabi
    ↪--prefix=/target-directory --disable-newlib-supplied-syscalls
    ↪--disable-libgloss --disable-nls
```

Capítulo 16

Sistemas Operacionais de Tempo-Real

O hardware do STM32 é capaz de realizar ações simultaneamente em todos os seus barramentos de comunicação – p.ex.: ler arquivos de áudio de um cartão SD no barramento SPI, enquanto toca esses arquivos de áudio através do DAC, monitorando Nunchuks sobre o barramento I2C e registrar mensagens através da UART. No entanto, a coordenação dessas atividades paralelas através de software pode ser um verdadeiro desafio – especialmente quando restrições de tempo devem ser atendidas. Uma estratégia comum é particionar a responsabilidade para tarefas múltiplas entre threads separados – cada um dos quais atuando de forma autônoma – e que estão escalonados (scheduled) com base em prioridades. Por exemplo, thread com grandes restrições de tempo recebem prioridades mais elevadas do que outros threads.

Os thread fornecem uma maneira de dividir a lógica de um programa em tarefas separadas. Cada thread tem seu próprio estado e aparentemente executa como um programa autônomo enquanto compartilha dados com outros threads. Em um uni-processador, como o STM32, threads são executados de forma intercalada com acesso ao processador controlado por um escalonador (scheduler). Sempre que ocorre uma interrupção, há uma oportunidade de suspender o thread atual e retomar um thread interrompido. A interrupção por temporizador fornece o mecanismo para “time-slice” (“fatiamento do tempo”) do processador, permitindo que cada thread pronto para executar possa ser executado.

A coordenação das tarefas de hardware por vários threads é ativado através objetos de sincronização. Por exemplo, um thread que está tentando

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

transmitir um pacote de dados através de uma UART não pode avançar quando o buffer de saída está cheio. Neste caso, o thread deve “esperar”, permitindo que outros threads sejam executados. Mais tarde, quando o espaço é libertado no buffer de transmissão, por exemplo por um manipulador de interrupção, o thread que espera pode ser “sinalizado” para retomar. Este padrão espera/sinal é implementado usando um objeto de sincronização chamado um “semáforo” (“semaphore”).

A decisão para estruturar um programa em torno de threads não deve ser tomada sem o devido cuidado, porque há muitas armadilhas potenciais. Threads exigem memória RAM adicional pois cada thread requer uma pilha separada; em dispositivos de memória restrita, como o processador na Discovery Board, este pode ser um problema real. Threads podem “estourar” as suas pilhas se espaço insuficiente é alocado – pode ser difícil de estimar com precisão o espaço necessário. Threads oferecem ampla oportunidade para erros sutis na forma de condições de corrida sempre que os threads compartilham dados. Finalmente, threads são extremamente difíceis de debugar. Enquanto alguém poderia querer fazer um “trace” de execução de um único thread, os “breakpoints” são feitos normalmente a nível de instruções e com código compartilhado, será interrompido com qualquer thread que esteja executando a instrução. Além disso, o GDB não é integrado com a maioria dos pacotes de threads e, portanto, não é fácil ver o estado de outros threads que não o atualmente parado.

Existem muitos sistemas operacionais em tempo-real disponíveis para o STM32. Neste capítulo vamos usar o FreeRTOS porque é relativamente simples e está disponível na forma de código fonte. FreeRTOS fornece um núcleo (kernel) básico com um pequeno conjunto de primitivas de sincronização. Em contraste, Chibios fornece uma camada de abstração de hardware completa com drivers para muitos dos dispositivos STM32. FreeRTOS serve os nossos propósitos pedagógicos de modo melhor porque é mais fácil de “olhar sob o capô”, mas Chibios fornece uma base significativamente mais rica para a construção de grandes projetos. A escolha de utilizar FreeRTOS tem duas consequências negativas -- documentos principais estão disponíveis apenas para compra, e o núcleo requer alocação dinâmica de memória que não é desejável em situações em que memória é restrita. Em contraste, o núcleo Chibios é alocada estaticamente e todos os documentos estão disponíveis livremente.

O restante deste capítulo está organizado da seguinte forma. Começamos com uma discussão de threads, a sua implementação e a API FreeRTOS para gestão de threads. Em seguida, discutimos as primitivas de sincronização principais e mostramos como eles podem ser usados para controlar o acesso ao

16.1. THREADS

hardware compartilhado, os dados compartilhados (por exemplo, um sistema de arquivos FatFS) e coordenar o escalonamento (scheduling) com manipuladores de interrupção (filas (queues) da UART de recepção e transmissão).

Observação: A STM32VL Discovery Board não possui memória suficiente para suportar libc (newlib) e o FreeRTOS simultaneamente. Assim, nós não trataremos de questões adicionais que surgem quando os dois são combinados (a distribuição do FreeRTOS fornece mais informações sobre este tópico).

16.1 Threads

Os programas com um único thread que desenvolvemos ao longo deste livro organizam a memória RAM com todos os dados definidos estaticamente em espaços alocados na parte baixa de endereços da memória RAM e a pilha do programa é iniciada no topo (endereço mais alto) na RAM. A medida que o programa é executado, a pilha cresce para baixo com a entrada de procedimentos e encolhe para cima a saída de procedimentos. Quando as interrupções ocorrerem, partes importantes do contexto de execução são empurrados (pushed) para a pilha principal – consulte o Capítulo 11 para obter mais explicações. Em um ambiente multi-thread, a pilha principal é utilizada durante a inicialização e então, primariamente como uma pilha de interrupção. A cada thread ativo é alocado a sua própria pilha dentro da RAM como mostrado na Figura 16.1.

A área de dados é alocada estaticamente em tempo de linkedição e inclui todas variáveis globais e estáticas tanto inicializadas e não inicializadas. A área acima dos dados é utilizada para alocação dinâmica de memória (por exemplo, malloc). Esta área, chamada heap, é expandida pelo alocador de memória conforme necessário. A pilha principal é colocada na extremidade mais alta da memória RAM pelo linker e cresce para baixo. No FreeRTOS, as pilhas dos threads são blocos alocados dentro do heap. O espaço é alocado dentro de cada pilha pelo código em execução (por exemplo, na entrada do procedimento). A execução correta do programa exige que nenhuma pilha cresça para além da sua região previamente definida. Embora existam testes em tempo de execução que podem ajudar a detectar um acontecimento tão indesejável, é difícil garantir que um overflow nunca ocorrerá.

O FreeRTOS suporta threads (chamados de “tasks” no FreeRTOS) com um núcleo preemptivo priorizado – a qualquer momento, o thread com a maior prioridade tem permissão para executar. Threads com prioridades

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

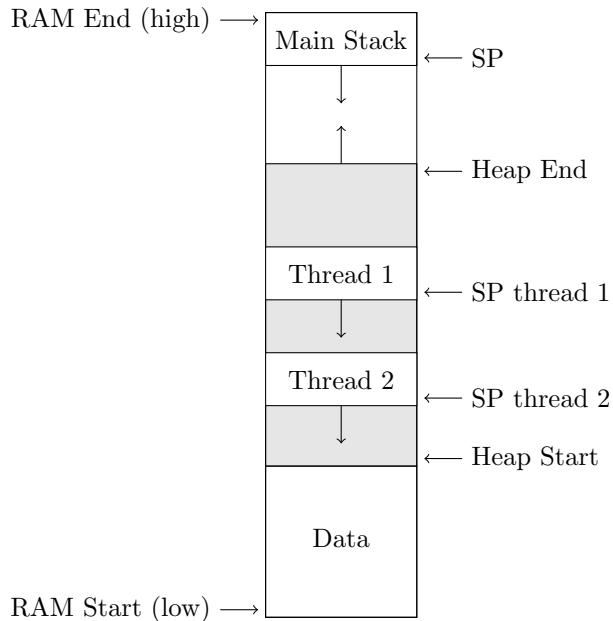


Figura 16.1: Layout da RAM com Threads em Execução

idênticas são “time-sliced” com cada um sendo permitido executar por um período fixo antes de ser preemptado. Para compreender as principais ideias do kernel (núcleo), considere os estados dos threads na Figura 16.1. Cada thread (tarefa) está em um dos quatro estados – Pronto, Executando, Bloqueado ou Suspenso. Quando um thread é criado (0) ele é colocado em estado Pronto. O (único) thread executando está no estado Executando. Um thread em execução pode ser preemptado e retornado ao estado Pronto (1) neste caso um thread pronto é movido para o estado de execução (2). Um thread também pode estar bloqueado (3) chamando uma função da API de bloqueio (é como esperar em um semáforo). Um thread bloqueado pode ficar pronto (4), quando desbloqueado pelas ações de outro thread ou manipulador de interrupção. O FreeRTOS tem um estado adicional, Suspenso, que iremos ignorar por enquanto.

O código para um thread (task) é definido por uma função em C que nunca retorna, como em:

```
void threadFunction(void *params){
    while(1) {
        // do something
    }
}
```

16.1. THREADS

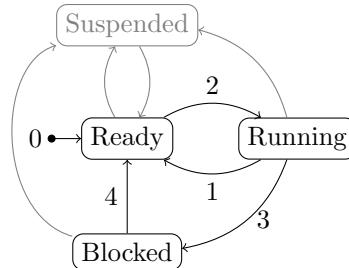


Figura 16.2: Estados dos Threads (Tasks) no FreeRTOS

}

Quando “criado” um thread é passado como um ponteiro para uma estrutura de parâmetros -- muitos threads podem dividir uma única função, mas podem ser especializados pelos parâmetros. Um thread é criado com uma função, um nome, um tamanho da pilha, e um ponteiro de parâmetro opcional, uma prioridade, e (opcionalmente) um local para armazenar um único “handle” especial para o thread.

```
portBASE_TYPE xTaskCreate(
    pdTASK_CODE pvTaskCode,
    const portCHAR * const pcName,
    unsigned portSHORT usStackDepth,
    void *pvParameters,
    unsigned portBASE_TYPE uxPriority,
    xTaskHandle *pvCreatedTask
);
```

O núcleo do FreeRTOS aloca memória para a pilha e cria as estruturas de dados necessária para o thread.¹

O tamanho da pilha é um parâmetro importante – muito pequeno e o thread vai gerar um overflow na pilha e corromper a memória, muito grande e a memória é desperdiçada. Como o FreeRTOS usa a pilha para manter o contexto para tarefas não-executantes (68 bytes para 17 registradores!) o tamanho mínimo da pilha é na verdade relativamente grande. Além disso, se o código da task exige quaisquer funções da biblioteca, a pilha precisa

¹FreeRTOS necessita de um alocador de memória para alocar dinamicamente as estruturas de dados principais.

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

ser grande o suficiente para armazenar todos os dados alocados na pilha e quaisquer registros salvos. A maioria dos threads irá precisar de 128-256 bytes.

O kernel do FreeRTOS também cria um “idle thread”, que tem a menor prioridade e é executado sempre que nenhum outro thread pode executar. (O “idle thread” (thread ocioso) tem uma pilha também!). O fluxo básico de um programa multi-thread segue:

```
main() {  
    // include misc.h  
  
    NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );  
  
    // Initialize hardware  
    ...  
    // Create tasks  
  
    xTaskCreate( ... );  
    ...  
    // Start Scheduler  
  
    vTaskStartScheduler();  
}
```

Tal como acontece com todos os programas considerados até aqui, o código começa com a inicialização do hardware. Uma inicialização do hardware que chamamos a atenção é a configuração NVIC. O FreeRTOS requer prioridades de thread preemptiva; aqui vamos ativar o NVIC para suportar 16 prioridades (e nenhuma subprioridades). Esta é a configuração assumida pela versão do FreeRTOS para o STM32. Uma vez que o hardware é inicializado, o conjunto inicial de threads é criado. Enquanto FreeRTOS permite alocação dinâmica de threads, a RAM limitada (8K) disponível na Discovery Board exige que nós criemos os threads inicialmente. O escalonador (scheduler) nunca retorna nada (há uma chamada da API para sair do scheduler, mas isso não é particularmente útil para as nossas aplicações). Um exemplo mais completo, com dois threads que piscam dois LEDs é ilustrado na Listagem 16.1. Observe que os threads utilizam vTaskDelay para dormir.

16.1. THREADS

```
static void Thread1(void *arg) {
    int dir = 0;
    while (1) {
        vTaskDelay(300/portTICK_RATE_MS);
        GPIO_WriteBit(GPIOC, GPIO_Pin_9, dir ? Bit_SET : Bit_RESET);
        dir = 1 - dir;
    }
}

static void Thread2(void *arg) {
    int dir = 0;
    while (1) {
        vTaskDelay(500/portTICK_RATE_MS);
        GPIO_WriteBit(GPIOC, GPIO_Pin_8, dir ? Bit_SET : Bit_RESET);
        dir = 1 - dir;
    }
}

int main(void)
{
    // set up interrupt priorities for FreeRTOS !!

    NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );

    // initialize hardware

    init_hardware();

    // Create tasks

    xTaskCreate(Thread1,                      // Function to execute
                "Thread 1",                  // Name
                128,                         // Stack size
                NULL,                        // Parameter (none)
                tskIDLE_PRIORITY + 1,        // Scheduling priority
                NULL);                      // Storage for handle (none)
    );
    xTaskCreate(Thread2, "Thread 2", 128,
                NULL, tskIDLE_PRIORITY + 1, NULL);

    // Start scheduler

    vTaskStartScheduler();

    // Schedule never ends
}
```

Listing 16.1: Exemplo FreeRTOS

16.2 Configuração do FreeRTOS

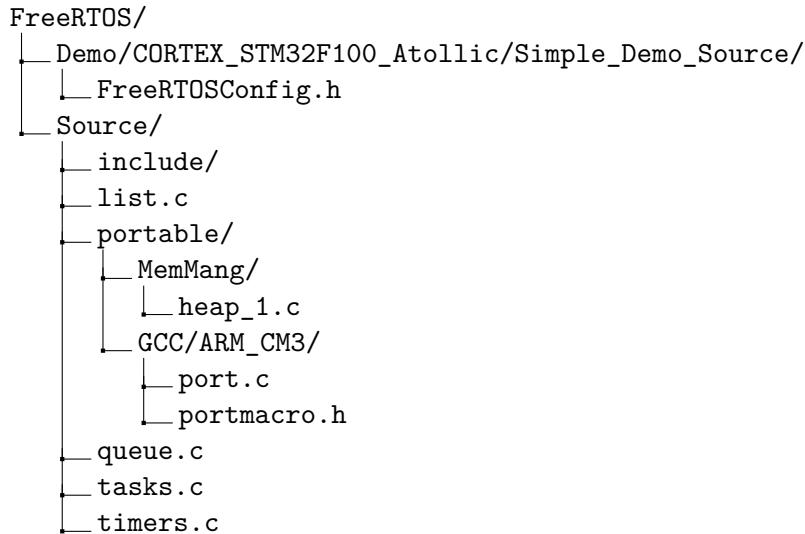


Figura 16.3: Partes Principais da Distribuição do FreeRTOS

O código-fonte do FreeRTOS está disponível gratuitamente em <http://sourceforge.net/projects/freertos/files/>. A distribuição pode ser um pouco confusa. Os arquivos-chave que usamos são mostrados na Figura 16.3. Ao criar uma aplicação de exemplo, será necessário ampliar seu makefile para incluir os recursos necessários, como mostrado na Listagem 16.2.

```

FreeRTOS = ... path_to_FreeRTOS ...
CFLAGS += -I$(FreeRTOS)/include -DGCC_ARMCM3
...
vpath %.c $(FreeRTOS)-
vpath %.c $(FreeRTOS)/portable/MemMang
vpath %.c $(FreeRTOS)/portable/GCC/ARM_CM3
...
OBJS+= tasks.o queue.o list.o timers.o heap_1.o port.o

```

Listagem 16.2: Build Paths para o FreeRTOS

Cada projeto FreeRTOS requer um arquivo de configuração. Ele é usado para definir os parâmetros-chave do projeto; por exemplo, ativar ou desativar recursos do núcleo – `FreeRTOSConfig.h`. Baseamos nossos projetos no arquivo de configuração mostrado na Figura 16.3. O arquivo de configuração é:

16.3. SINCRONIZAÇÃO

ração serve a vários propósitos. Primeiro ele define parâmetros-chave – por exemplo frequência de clock, tamanho do heap, número de prioridades, etc. Segundo, ele define recursos a serem ativados numa compilação -- recursos são habilitados (1) ou desabilitados (0) por definições da seguinte forma:

```
#define INCLUDE_xxx 0+
```

Exemplos das características chave incluem

```
#define configUSE_MUTEXES          1
#define configCHECK_FOR_STACK_OVERFLOW 1
#define configUSE_RECURSIVE_MUTEXES    0
#define configUSE_COUNTING_SEMAPHORES  0
```

Finalmente, o FreeRTOS também fornece a oportunidade para o código do usuário de fazer um “hook” (“gancho”) no kernel em lugares chave:

```
#define configUSE_MALLOC_FAILED_HOOK 0
#define configUSE_IDLE_HOOK           0
#define configUSE_TICK_HOOK           0
```

Se você compilar um projeto com FreeRTOS e tem erros de linkedição para funções não encontradas, então você deve verificar para ver se um gancho correspondente foi definido. Depois, você tem a opção de fornecer a função exigida ou desativar o gancho.

Exercise 16.1 RTOS – Blinking Lights

Complete o programa demo Pisca Led. Você deve desabilitar qualquer recurso que seu projeto não precise fazendo uma copia local do arquivo de configuração. Você achará necessário reduzir o tamanho do heap (de 7k para 5k) para seu projeto compilar.

16.3 Sincronização

Threads não podem compartilhar de modo seguro as estruturas de dados e nem hardware sem um mecanismo de sincronização. Considere a seguinte implementação de `getchar`, discutida no Capítulo 5:

```
int getchar(void){
    while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
    return USARTx->DR & 0xff;
}
```

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

Lembre-se que isso funciona através da leitura do registrador de status da UART até que registrador de dados recebidos esteja “not empty” (“não vazio”) e então lendo o registrador. Supondo que dois threads acessem a UART – se thread 1 lê o registrador de status e encontra “not empty”, mas em seguida é preemptado pelo thread 2 que também testa o status e lê o registro de dados, e então quando o thread 1 é retomado, seu conhecimento do registrador de status será incorreto e pode ler lixo do registrador de dados.

A idéia-chave para fazer isso funcionar é encontrar um método para garantir o acesso exclusivo pelo thread 1 ao registrador de dados da UART. Uma abordagem é a de impedir a preempção desabilitando as interrupções, no entanto, considere o que aconteceria se nenhum caractere chegar ou simplesmente chegar depois de um atraso substancial. O que é necessário é um mecanismo para bloquear apenas os threads que estão disputando o acesso ao hardware de recepção da UART. A solução é a utilização de um semáforo.

```
xSemaphoreHandle uartRcvMutex;

uart_init(...){
    ...
    uartRcvMutex = xSemaphoreCreateMutex();
}

int getchar(void){
    int c;
    if (xSemaphoreTake(uartRcvMutex, portMAX_DELAY) == pdTRUE)
    {
        while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
        c = USARTx->DR & 0xff;
        xSemaphoreGive(uartRcvMutex);
    }
    return c;
}
```

Um semáforo é uma primitiva de sincronização padrão (na verdade, a primeira descrita) para a estruturação do código de sistemas operacionais. A ideia é proporcionar a um thread uma interface segura para gerenciar recursos compartilhados. A um thread que solicita um recurso ou é concedido ou bloqueado pelo escalonador (scheduler) até que o recurso se torne disponível. Um thread liberando um recurso pode, como efeito colateral, desbloquear um thread em espera.

FreeRTOS suporta três tipos de semáforos – semáforos binários, exclusões mútuas (Mutex) e semáforos contáveis. Mutex e semáforos binários são semelhantes, mas se comportam de forma diferente em relação a prioridade

16.4. MANIPULADORES DE INTERRUPÇÃO

do thread. Aqui usamos Mutex. Há duas operações a se notar – *take* e *give*. Quando o mutex é inicializado ele têm um único “token”. *Take* remove o token se disponível, ou bloqueia o thread que chama e põe o thread em uma lista associada ao Mutex se nenhum token estiver disponível. *Give* restaura o token. A maior diferença entre mutexes (semáforos binários) e semáforos contáveis é que o último pode ter vários tokens.

Outro Mutex pode ser adicionado para proteger da mesma forma `putchar`. Proteger `getchar` e `putchar` com mutexes impedirão “data races” (corridas de dados), tais como a mostrada acima; no entanto, isso pode não produzir o comportamento esperado. Suponha múltiplos threads chamando `putchar` através de um procedimento `putstring`:

```
void putstring(char *s){  
    while (*s && *s)  
        putchar(*s++);  
}
```

Neste caso, a saída de dois threads escrevendo simultaneamente duas strings pode ser intercalada!

Exercise 16.2 Multiplos Threads

Escreva um programa com dois threads que imprimem continuamente strings para a UART1. Um terceiro thread deve piscar um dos Leds a 2Hz. Você deve usar os parâmetros do thread para especializar uma função comum aos threads para os dois threads que imprimem. Teste seu código usando apenas um Mutex em um `putchar`. Então ache uma solução que previna a intercalação de strings.

16.4 Manipuladores de Interrupção

O código de `getchar` acima tem uma grande falha – a thread que tem o mutex continuará a rodar sobre o teste do flag até que tenha êxito. Precisamos de um mecanismo para permitir que o thread em espera possa dormir até que haja espaço disponível. Com periféricos como UARPs já vimos que o código de interrupção pode lidar com as mudanças reais no status do hardware. Na Seção 11.5, mostramos como isso pode funcionar. A questão remanescente é como comunicar esses eventos de hardware com os manipuladores de interrupção. Como mostramos anteriormente, gostaríamos de associar um par de queues com as interfaces de transmissão e recepção da UART. Onde antes o nosso código de `putchar` falhou quando a fila de transmissão estava vazia e

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

nosso código `getchar` falhou quando a fila de recebimento estava vazia, em um ambiente com múltiplos threads, gostaríamos de bloquear um thread em qualquer um destes casos e ter o manipulador de interrupção acordar o thread.

FreeRTOS fornece suas próprias primitivas de bloqueio de queues (na verdade, semáforos e mutexes são casos especiais de queues):

```
xQueueHandle xQueueCreate(
    unsigned portBASE_TYPE uxQueueLength,
    unsigned portBASE_TYPE uxItemSize);
```

Uma queue (fila) é criada com tanto um comprimento quanto um tamanho de dados. Os itens são enfileirados, copiando-os e não por referência (pode-se sempre enfileirar um ponteiro, mas pense com cuidado antes de fazer isso!). Os dois procedimentos de interface que necessitamos são “send” e “receive”. Observe que cada um tem um “timeout” (“tempo de espera”), que pode variar de 0 (não espere se a operação falhar) até `portMAX_DELAY` que espera para sempre.

```
portBASE_TYPE xQueueSend( xQueueHandle xQueue,
    const void * pvItemToQueue,
    portTickType xTicksToWait );

portBASE_TYPE xQueueReceive(xQueueHandle xQueue,
    void *pvBuffer,
    portTickType xTicksToWait );
```

Ambas interfaces podem ser utilizadas por meio de threads, mas não por manipuladores de interrupção, que deve usar versões especiais destas:

```
portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue,
    const void *pvItemToQueue,
    portBASE_TYPE *pxHigherPriorityTaskWoken);

portBASE_TYPE xQueueReceiveFromISR( xQueueHandle pxQueue,
    void *pvBuffer,
    portBASE_TYPE *pxTaskWoken);
```

Observe que essas “ISR” (Interrupt Service Routine) (Rotinas de Interrupção do Serviço) têm um terceiro parâmetro diferente — um flag é retornado se uma tarefa foi “woken” (“acordada”) por uma chamada. Nesta situação, o manipulador de interrupção deve notificar o kernel como nós em breve iremos demonstrar.

Observação: É muito importante que os manipuladores de interrupção que acessam a API do FreeRTOS tenham prioridades mais baixas (altos

16.4. MANIPULADORES DE INTERRUPÇÃO

números) que o valor

`LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY` definido no arquivo de configuração o FreeRTOS.

Nós agora construiremos uma versão com interrupção (driven) da UART como mostrado nas Listagens 16.3 e 16.4

```
int uart_putc(int c){
    xQueueSend(UART1_TXq, &c, portMAX_DELAY);
    // kick the transmitter interrupt
    USART_ITConfig(USART1, USART_IT_TXE, ENABLE);
    return 0;
}

int uart_getc (){
    int8_t buf;
    xQueueReceive(UART1_RXq, &buf, portMAX_DELAY);
    return buf;
}
```

Listing 16.3: Interrupt Driven UART

Exercise 16.3 Multithreaded Queues

Complete a UART com interrupção (driven) com controle de fluxo usando filas junto com um programa multi-thread que exercita tanto o envio quanto a recepção.

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

```
void USART1_IRQHandler(void)
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET){

        uint8_t data;

        USART_ClearITPendingBit(USART1, USART_IT_RXNE);

        data = USART_ReceiveData(USART1) & 0xff;
        if (xQueueSendFromISR(UART1_RXq, &data,
            ↪&xHigherPriorityTaskWoken) != pdTRUE)
            RxOverflow = 1;
    }

    if(USART_GetITStatus(USART1, USART_IT_TXE) != RESET) {
        uint8_t data;

        if (xQueueReceiveFromISR(UART1_TXq, &data,
            ↪&xHigherPriorityTaskWoken) == pdTRUE){
            USART_SendData(USART1, data);
        }
        else {

            // turn off interrupt

            USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
        }
    }
    // Cause a scheduling operation if necessary
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Listing 16.4: Interrupt Driven UART (Handler)

16.5 SPI

As outras interfaces de comunicação que introduzimos neste livro também precisam ser reconsideradas em face de multi-thread. A interface I2C é muito parecida com a interface UART na qual um semáforo pode ser adicionado diretamente ao código da interface para proteger as interfaces de leitura/escrita (apenas um semáforo é necessário). SPI é um pouco diferente – enquanto que o protocolo I2C inclui endereçamento de dispositivos nos da-

16.5. SPI

dos transmitidos, SPI usa linhas de seleção separadas para cada dispositivo. Nossa uso da interface SPI é algo como:

```
select();  
spi_operation();  
deselect();
```

onde marcar e desmarcar são específicos do dispositivo. Se seguimos o padrão UART de acessar um único semáforo dentro das operações do SPI, então seria possível ter um thread interferindo com outro. Nós temos um par de opções de projeto. Poderíamos simplesmente fazer o semáforo para a interface SPI como um objeto global e exigir que todos os usuários da interface primeiro requisitem o semáforo. Isto tem a desvantagem significativa de exigir que os escritores do código do aplicativo sejam muito familiarizados com os pressupostos de sincronização de baixo nível para a interface e corre um risco significativo de originar código falho. Poderíamos acrescentar o pino selecionado GPIO como um (par) de parâmetros para a interface SPI como em:

```
void spiReadWrite(SPI_TypeDef* SPIx, uint8_t *buf, uint8_t *buf,  
                  →int cnt,  
                  uint16_t pin, GPIO_TypeDef* GPIOx);
```

Ou, poderíamos passar uma função de retorno de chamada (callback) em que a interface SPI poderia usar para selecionar/desmarcar o dispositivo depois de pegar o semáforo.

```
typedef void selectCB_t(int sel);  
void spiReadWrite(SPI_TypeDef* SPIx, uint8_t *buf, uint8_t *buf,  
                  →int cnt,  
                  selectCB_t selectCB);
```

Esta abordagem final parece preferível, uma vez que não carrega a interface da SPI com conhecimento detalhado de como funciona o selecionar/-desmarcar.

A interface SPI também é usada para transferências de dados bastante longos utilizando DMA. Por exemplo, podemos usá-lo para transferir blocos de dados para a FatFS. Em um ambiente multi-thread, é desejável que DMA continue em segundo plano. Na nossa implementação atual usamos “busy waiting” (espera ocupada) para a DMA completar.

```
while (DMA_GetFlagStatus(dmaflag) == RESET) { ; }
```

Tal como acontece com o código UART, esta busy waiting (espera ocupada) pode ser substituída por um bloqueio do thread iniciando a operação de DMA sobre um objeto de sincronização (neste caso um semáforo binário

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

é adequado) e utilizando um manipulador de interrupção para desbloquear o thread (como mostrado a seguir).

16.6. FATFS

```
static void vExampleInterruptHandler( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    // handle interrupt

    ...

    // release semaphore

    xSemaphoreGiveFromISR( xBinarySemaphore,
                           &xHigherPriorityTaskWoken );
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

No Capítulo 13, mostramos como criar manipuladores de interrupção DMA. Para a interface SPI precisamos habilitar as interrupções para o caso de “transferencia completa”.

Exercise 16.4 Multithreaded SPI

Crie uma interface SPI que suporte multi-threading e DMA. Você deve usar retornos de chamada (callbacks) para a seleção de dispositivo. Teste a sua interface com o LCD com duas threads concorrentes para usar a interface para desenhar retângulos coloridos em locais aleatórios (cores diferentes para os dois threads).

16.6 FatFS

O código FatFS tem algum suporte nativo para multi-thread, que deve ser ativado no `ffconf.h`.

```
#define _FS_REENTRANT 0      /* 0:Disable or 1:Enable           */
#define _FS_TIMEOUT    1000   /* Timeout period in unit of ticks  */
#define _SYNC_t        HANDLE /* O/S dependent type of sync object */
```

Cabe ao usuário definir um tipo de `_SYNC_t` adequada e fornecer implementações para `ff_req_grant`, `ff_rel_grant`, `ff_del_syncobj` e `ff_cre_syncobj`. Também é necessário modificar a interface do dispositivo de baixo nível para utilizar uma interface SPI modificada e para substituir a busy waiting com funções de atraso do RTOS adequadas.

Exercise 16.5 Multithreaded FatFS

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

Reescreva a interface de baixo nível da FatFS para usar a nova interface SPI e remover a busy waiting – suas alterações devem funcionar tanto em um único ambiente de thread quanto em multi-threaded (use o flag `_FS_REENTRANT` adequadamente!) Teste o seu código com um programa multi-thread que reproduz arquivos de áudio e mostra imagens a partir de um cartão SD. Um thread deve reproduzir um arquivo de áudio, enquanto outro exibe uma imagem.

16.7 FreeRTOS API

A API do FreeRTOS está documentado em www.freertos.org. Aqui nós identificamos as principais funções da API utilizadas ao longo deste capítulo.

16.8. DISCUSSÃO

Task Creation	
xTaskCreate()	Create task
xTaskDelete()	Delete task
Task Utilities	
vTaskDelay()	Delay from now
vTaskDelayUntil()	Delay from previous wake time
Kernel Utilities	
taskYIELD()	Force a context switch
vTaskSuspendAll()	Prevent current task from being preempted
vTaskResumeAll()	Resume normal scheduling
Queues	
uxQueueMessagesWaiting()	Number of messages in queue
xQueueCreate()	Create queue
xQueueDelete()	Create queue
xQueueSend()	Send to queue
xQueueReceive()	Receive from queue
xQueueSendFromISR()	ISR send to queue
xQueueReceiveFromISR()	ISR Receive from queue
Semaphore	
vSemaphoreCreateBinary()	Create a binary semaphore
vSemaphoreCreateCounting()	Create a counting semaphore
vSemaphoreCreateMutex()	Create a mutex
vSemaphoreCreateTake()	Take from semaphore
vSemaphoreCreateGive()	Give to semaphore
vSemaphoreCreateGiveFromISR()	Give from ISR to semaphore

Tabela 16.1: FreeRTOS API – Key Calls

16.8 Discussão

O desenvolvimento de um projeto com FreeRTOS é muito trabalhoso! Todas as interfaces de dispositivo têm que ser escritas para suportar multi-thread e grande cuidado tem que ser tomado para assegurar a ausência de condições de corrida. Uma alternativa é usar um RTOS com uma camada de abstração de hardware, como Chibios onde grande parte do trabalho pesado já foi feito para você.

Você provavelmente também descobriu que a depuração do código multi-thread é muito desafiador. A mensagem é clara -- use threads somente

CAPÍTULO 16. SISTEMAS OPERACIONAIS DE TEMPO-REAL

quando realmente necessário.

Capítulo 17

Próximos Passos

Nos capítulos anteriores você aprendeu muitas das habilidades básicas necessárias para construir projetos interessantes que utilizam processadores STM32 e módulos comerciais. Neste capítulo, eu forneço referências para diversos módulos e orientações adicionais para saber como as habilidades que você aprendeu podem ser aplicadas para desenvolver o código para fazer interface com eles. Até agora, deveria ser evidente que a SRAM é uma limitação importante do processador da STM32 VL Discovery. Ambos newlib e FreeRTOS precisam de muito mais memória para operar e projetos grandes podem se beneficiar muito com a camada adicional de abstração fornecida por essas bibliotecas. Assim, eu começo com uma discussão de placas que fornecem os processadores STM32 com memórias maiores. Deve também ser evidente agora que a utilização de um LCD com base em SPI, ao mesmo tempo suficiente para displays básicos não atendem as necessidades de performance de gráficos complexos; desempenhos superiores são conseguidos à custa de uma nova interface.

As interfaces que você aprendeu podem ser aplicadas a uma ampla variedade de novo sensores incluindo climáticos (temperatura/pressão/umidade), posição e inercial (acelerômetro/giroscópio/magnetômetro/GPS), força (sensores de flexibilidade) – vou discutir alguns deles, mas um passo útil é examinar as ofertas de empresas como a SparkFun embora salientando as interfaces que seus módulos exigem. A interface serial básica permite a utilização de vários dispositivos de comunicação sem fios, incluindo bluetooth, wifi, e GSM (telefone celular) enquanto a interface SPI permite o uso de dispositivos de rádio de baixa potência. Finalmente os temporizadores fornecem a chave para o controle de movimento, incluindo motor de passos e controle DC com o feedback de posição.

CAPÍTULO 17. PRÓXIMOS PASSOS

Este capítulo destina-se principalmente para mostrar o mundo de novas habilidades adquiridas – não como um guia definitivo para interfaceamento e utilização de novos módulos; esse trabalho é com você.

17.1 Processadores

O micro-controlador STM32 F100 usado na VL Discovery Board é um membro de uma série de componentes STM32 F1xx que inclui:

- Value line STM32 F100 – 24 MHz CPU com controle de motor e funções CEC
- Access line STM32 F101 – 36 MHz CPU, até 1 MByte Flash
- USB Access line STM32 F102 – 48 MHz com sistema de arquivos USB
- Performance line STM32 F103 – 72 MHz, até 1 Mbyte Flash com controle de motor, USB e CAN
- Connectivity line STM32 F105/STM32 F107 – 72 MHz CPU com Ethernet MAC, CAN e USB 2.0 OTG

Todas estas famílias de componentes utilizam as mesmas bibliotecas padrão de periféricos. Elas exigem modificações no script do linker (particularmente as definições de regiões de memória) e código de inicialização (as definições dos vetores de exceções); entretanto nenhuma das tarefas é difícil. Eles também incluem periféricos que você ainda não viu, incluindo suporte para memória externa (FSMC), uma interface de alto nível para SD (SDIO) e interfaces de comunicação com USB, Ethernet e CAN. As interfaces de comunicação em geral, exigem significativas bibliotecas de software adicionais e podem ser muito difíceis (por exemplo, USB) para depurar.

Todos os exemplos deste livro devem funcionar sem modificação em qualquer processador das famílias STM32F1xx. Algumas mudanças serão necessárias para os arquivos no diretório template fornecido com este livro. O script do linker de STM32-Template precisa da seguinte modificação:

```
MEMORY
{
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 8K
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 128K
}
```

17.1. PROCESSADORES

[frame=none]

Os tamanhos das duas regiões de memória devem serem modificadas de modo a refletir o componente que está sendo usado. `Makefile.common` precisa ter suas variáveis `LDSCRIPT`, `PTYPE`, `STARTUP` modificadas conforme o caso. Finalmente, a tabela de vetor no `startup_stm32f10x.c` precisará ser atualizada para refletir os vetores do componente da família que está sendo usado. A biblioteca padrão de periféricos inclui arquivos de inicialização em linguagem assembly que podem servir como um guia. Observe que a tabela de vetores termina com um número “mágico” que é necessário se você construir projetos que farão boot da SRAM.

Há famílias STM32 adicionais, incluindo STM32 F0 (baseado no Cortex-M0), STM32 L1 (Cortex-M3 de Ultra baixa potência), STM32 F2 (Cortex-M3 de alto desempenho – 120MHz), e STM32 F4 / F3 (baseado no Cortex-M4). Cada destes exige a utilização de uma biblioteca padrão de periféricos diferente. Embora muitos dos periféricos sejam os mesmos ou melhorias daqueles na série STM32 F1, alguns cuidados serão necessários para portar o código deste documento para estes dispositivos. Existem “discovery boards” extremamente atraentes para o STM32 L1, STM32 F0, e STM32 F4 que valem a pena investigar. Neste momento, o código da interface `gdb` usados neste documento suporta todos menos o dispositivo STM32 F0; no entanto, parece que o `openocd openocd.sourceforge.net` pode suportar todos.

Pelas razões descritas acima, a atualização mais simples do STM32F100 é um outro membro da família STM32F1xx. A SparkFun vende uma header board para o STM32F103 com 20K bytes de RAM e 128K bytes de Flash por aproximadamente US\$40. – a mesma placa também é vendida pela Mouser. Esta placa, produzida pela Olimex, também tem um conector USB como o STM32F103 tem um periférico USB escravo (não para os fracos de coração!). Várias placas com desenvolvimento semelhante estão disponíveis de vários fornecedores no Ebay por US\$ 15-25. Por cerca de US\$30 é possível comprar uma placa STM32F103 com um touch-screen de 2,8”TFT no Ebay. Esse é um excelente valor e suporta uma alta performance gráfica através do FSMC paralela interfaceada com o periférico do STM32F103.

A maioria das placas STM32 atualmente disponíveis fornecem um conector padrão JTAG para a interface do depurador. Embora seja possível utilizar a STM32VL Discovery Board para se comunicar com esta interface, os problemas de conexão podem se tornar irritantes. A ST vende um módulo depurador separado ST-LINK/V2 que faz a interface direta com o conector JTAG. Este módulo está disponível a partir de tanto na Mouser (www.mouser.com)

CAPÍTULO 17. PRÓXIMOS PASSOS

como na Digikey (www.digikey.com) por aproximadamente US\$ 22 e seria um bom investimento para trabalhar com placas STM32. O protocolo ST-LINK/V2 é suportado pelo driver gbd utilizado neste documento – na verdade, causa menos problemas em um ambiente OS X ou Linux do que o protocolo V1.

Como mencionado anteriormente, displays mais poderosos geralmente requerem o uso de uma interface paralela, como a FSMC fornecida pelo STM32F103. Tomar partido das capacidades avançadas requer uma biblioteca de gráficos mais sofisticada. Há um bom número de bibliotecas comerciais e de código aberto disponíveis. A ST fornece uma biblioteca STM32 [22]. A biblioteca STM32 inclui uma camada de abstração de hardware (HAL - Hardware Abstraction Layer), que pode ser modificada para suportar vários dispositivos, incluindo LCDs, touch-screens e joysticks.

Recentemente a ST Microelectronics lançou uma Discovery Board com microcontroladores STM32F4 e STM32F3. Trata-se de um produto incrível. Por exemplo, a placa STM32F3 Discovery inclui uma unidade de movimento inercial de 9 graus (acelerômetro, giroscópio, bússola) por menos de US\$ 15. A maior barreira para começar a utilizá-la será a necessidade de se adaptar a uma nova biblioteca para periféricos. Embora esteja confiante de que os exemplos deste livro podem ser utilizados com um trabalho modesto, alguns periféricos podem ter mudado consideravelmente – por exemplo, a inicialização dos pinos do GPIO tem parâmetros adicionais.

17.2 Sensores

Existem muitos sensores disponíveis que possuem interfaces SPI, I2C, ou analógicas. Nesta seção, vamos discutir alguns deles. Os leitores interessados podem folhear o site SparkFun para ideias.

Medições de Posição/Inercial

A ST, a Analog Devices e a Freescale fazem dispositivos acelerômetros – muitos deles são suportados por placas de break-out da Sparkfun.com que fornece um guia de compra acessível em <http://www.sparkfun.com/tutorials/167>. Em geral, estes dispositivos utilizam interfaces I2C, SPI, ou analógicas. Uma “unidade de medida inercial” completa geralmente contém um giroscópio de 3 eixos, um acelerômetro de 3 eixos e um magnetômetro de 3 eixos; a ST produz um módulo que contém todos os três dispositivos em um fator de forma DIP (STEVAL-MKI108V2), que está disponível na Mouser por US\$ 29.

17.2. SENsores

Um outro dispositivo de detecção de posição útil é um receptor GPS. A SparkFun vende uma boa unidade com base no Venus638FLPx por US\$ 50. Como a maioria de tais unidades, ela se comunica via serial assíncrona – basta ligá-la a uma UART.

Sensores Ambientais

Sensores ambientais incluem sensores de temperatura, umidade e pressão. Estes são amplamente disponíveis com interfaces I2C, embora alguns, como os dispositivos 1-Wire da Maxim/Dallas exigem um protocolo especial. O protocolo 1-wire não é suportado diretamente nos periféricos STM32, mas podem ser implementados com temporizadores e GPIO. O Saleae Logic inclui um analisador de protocolo para dispositivos 1-wire. Observe que existem alguns muito semelhantes, mas incompatíveis com o estilo do barramento 1-wire; no entanto, a maioria é relativamente simples.

Alem dos protocolos digitais, sensores ambientais frequentemente produzem um sinal analógico. Exemplos incluem os vários sensores de gás (álcool, de monóxido de carbono, hidrogênio, ...) vendidos pela SparkFun e outros.

Sensores de Força e Movimento

Existem muitos sensores de força/posição cuja resistência varia com a entrada. A medição destes requer uma tensão de referência e resistência para traduzir a resistência do sensor para uma tensão analógica.

Identificação – Código de Barras/RFID

A SparkFun e outros vendem leitores de RFID e de código de barras com interface serial e I2C.

Proximidade

Os sensores de proximidade incluem tanto os módulos de distância por ultra-som e infravermelho. Nós demonstramos o uso de sensores de ultra-som no Capítulo 10. Sensores por reflexão infravermelha frequentemente necessitam de uma saída digital para alimentar um LED e uma entrada analógica para medir a intensidade da luz refletida.

17.3 Comunicação

Muitos projetos interessantes exigem uma comunicação melhorada seja com wireless ou com fio. No domínio wireless, módulos de baixo custo de bluetooth, wifi, e até mesmo celulares GSM estão disponíveis com interfaces seriais. Para aplicações de baixa potência, links de rádio dedicados, como o Nordic nRF24L01 utilizam a interface SPI. Finalmente, é relativamente fácil fazer o interfaceamento de controles remotos infravermelho padrão com um detector de infravermelho simples.

17.4 Discussão

Espero que as idéias discutidas neste capítulo lhe deem uma ideia das possibilidades. Há uma comunidade vibrante que pode fornecer orientação prática – você pode buscar exemplos desenvolvidos para a plataforma Arduino. Eu acredito que as técnicas apresentadas neste livro vão lhe dar uma base sólida para a experimentação.

Atribuições

A Figura 1.10 foi obtida em <http://commons.wikimedia.org/wiki/File:Potentiometer.jpg> sob Creative Commons Attribution-Share Alike 3.0 Unported license (<http://creativecommons.org/licenses/by-sa/3.0/deed.en>). Essa imagem foi também utilizada na Figura 14.1

A Figura 1.6 é coberta pela licença Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported. A imagem foi obtida em www.sparkfun.com.

Referências Bibliográficas

- [1] ARM Limited. Cortex-M3 technical reference manual, 2006.
- [2] ARM Limited. Procedure call standard for the ARM® architecture, October 2009. IHI 0042D.
- [3] ChaN. FatFs generic FAT file system module, 2011. http://elm-chan.org/fsw/ff/00index_e.html. Accessed April 2012.
- [4] F. Foust. Secure digital card interface for the MSP430, 2004. http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf. Accessed July 2012.
- [5] J. Jiu. *The Definitive guide to the ARM Cortex-M3*. Newnes, 2010.
- [6] Microchip. 25AA160A/25LC160A 16K SPI bus serial EEPROM, 2004.
- [7] NXP. i2c bus specification and user manual (rev 03), June 2007. UM10204.
- [8] C. Philips. Read wii nunchuck data into arduino. <http://www.windmeadow.com/node/42>. Accessed February 4, 2012.
- [9] SD specifications, part 1, physcial layer simplified specification, version 3.01, May 2010.
- [10] F. Semiconductor. Tilt sensing using linear accelerometers, 2012.
- [11] Sitronix Technology Corporation. Sitronix st7735r 262k color single-chip TFT controller/driver, 2009.
- [12] STMicroelectronics. How to get the best ADC accuracy in the STM32F10xxx devices, November 2008.
- [13] STMicroelectronics. STM32F10xxx i2c optimized examples, 2010.

REFERÊNCIAS BIBLIOGRÁFICAS

- [14] STMicroelectronics. User manual STM32 value line discovery, 2010. UM0919.
- [15] STMicroelectronics. Low & medium-density value line, advanced ARM-based 32-bit MCU with 16 to 128 kb flash, 12 timers, ADC, DAC & 8 comm interfaces, 2011. Doc ID 16455.
- [16] STMicroelectronics. Migrating a microcontroller application from STMF1 to STM32F2 series, 2011. AN3427: Doc ID 019001.
- [17] STMicroelectronics. Migrating a microcontroller application from STMF1 to STM32L1 series, 2011. AN3422: Doc ID 018976.
- [18] STMicroelectronics. Migration and compatibility guideslines for STM32 microcontroller applications, 2011. AN3364: Doc ID 018608.
- [19] STMicroelectronics. Programming manual: Stm32f10xxx/ 20xxx/ 21xxx/ l1xxxx cortex-m3 programming manual, March 2011. PM0056.
- [20] STMicroelectronics. Reference manual stm32f100xx advanced ARM-based 32-bit MCUs rev. 4, 2011. RM0041.
- [21] STMicroelectronics. Reference manual stm32f101xx stm32f102xx stm32f103xx stm32f105xx and stm32f107xx advanced ARM-based 32-bit MCUs rev. 14, 2011.
- [22] STMicroelectronics. STM32 embedded graphic objects/touchscreen library, 2011.
- [23] wiibrew. Wiimote/Extension Controllers. <http://wiibrew.org/wiki/Wiimote/Extension.Controllers>. Accessed February 6, 2012.
- [24] wikipedia. Wii remote. http://en.wikipedia.org/wiki/Wii_Remote. Accessed February 2, 2012.