PWM com STM32 Bluepill

Marcos G. T. Souza - 344303

22 de Setembro de 2024

Conteúdo

Capítulo 1

Identificando os componentes.

1.1 Identificando o microcontrolador

No Microcontrolador se encontra escrito: "STM32F103C6T6".

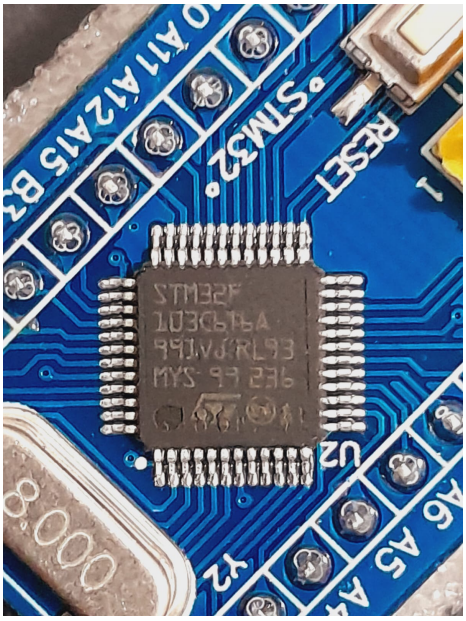


Figura 1.1: Imagem capturada com o celular

Conforme o datasheet desse microcontrolador [1] podemos verificar os seguin-

[1] STM32F103C6 Product Specfications, pg. 95

tes atributos:

- STM32F103 **C** 6T, o "C"indica conter 48 pinos.
- STM32F103C **6** T, o "6"indica ter 32kBytes de memória Flash.
- STM32F103C6 **T**, o "T"indica ter um encapsulamento LQFP.

  Assim sabemos se tratar de um LQFP48, conforme indica o datasheet[2].

  Também podemos verificar[3] outros atributos como que ele tem 10kBytes de memória SRAM, 2 Timers de propósito geral, entre outras coisas.

Podemos ver também que ele é o LQFP48, por ter 48 pinos, 12 em cada lateral, como se pode ver na Figura 1.1

1.2 Identificando a placa de desenvolvimento

A placa é chamada "Blue Pill". Podemos perceber alguns detalhes dessa placa, connforme o datasheet[4] ela apresenta dois relógios externos:

- Externo de Alta Velocidade: 8MHz.
- Externo de Baixa Velocidade: 32.768kHz.

Conforme o esquemático[5] podemos verificar como os pinos se ligam, e como funcionam as partes a mais da placa:

- O relógio externo de alta velocidade (HSE) se conecta com os pinos OSCIN e OSCOUT da placa, definidos do datasheet do microcontrolador[6] como PDO-OSC_IN, PD1-OSC_OUT, para representar sua função alternativa como PD, portas genéricas.
- O relógio externo de baixa velocidade (LSE) se conecta com os pinos PC14, PC15, definidos no datasheet como $PC14 - OSC32_IN$, $PC15 - OSC32_OUT$, representando sua função alternativa como PC, portas genéricas
- Os pinos BOOT0 e BOOT1 estão conectados com Ground e 3V3, e ajustando manualmente podemos alterar as entradas de BOOT0 e BOOT1, que no datasheet são chamados de BOOT0 e PB2, mostrando a função alternativa do pino PB2, que é ser BOOT1.

[2]STM32F103C6 Product Specfications, pg. 1
[3]STM32F103C6 Product Specfications, pg. 11
[4]STM32F103C8T6 Bluepill, apesar de se referir ao "C8" os pinos são os mesmos, pois o encapsulamento é o LQFP48.
[5]STM32F103C6T6 Blue Pill: high-resolution pinout and specs
[6]STM32F103C6 Product Specfications, pg. 24

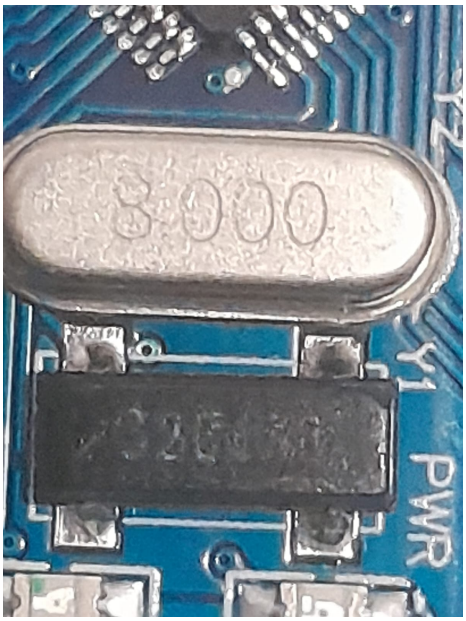


Figura 1.2: Imagem capturada com o celular

- O botom de RESET é feito da seguinte forma: o botão fica em paralelo com um capacitor, então quando está aberto, o capacitor fica carregado positivamente, gerando HIGH para o pino de reset quando o botom é apertado, cria-se um curto com o ground, descarregando o capacitor, e gerando LOW para o pino de reset. O pino de reset é chamado NRST no datasheet, para representar que ele é negado, isto é $RESET = HIGH$, não reseta, $RESET = LOW$, reseta.

- Há dois LEDs na placa, um fica entre 3V3 e GND, portanto está aceso sempre que a placa está devidamente alimentada. O outro fica entre 3V3 e PC13, portanto está acesa quando PC13 é LOW, e está apagada com PC13 é HIGH.

- Há também uma conexão USB, ligando D- com PA11, e D+ com PA12.

Capítulo 2

Problema 1: Piscar um LED

2.1 Saída

Se desejamos piscar um LED, é de imaginar que devemos ter uma saída que alterne entre HIGH e LOW, de maneira a fazer o LED piscar, uma possibilidade é o uso das GPIOs, que podem alternar com até 18MHz (apesar de que nosso LED não vai piscar tão rápido assim, usaremos frequências muito, mas muito menores.)[1].

As GPIOs podem ser configuradas como saída ou entrada, no nosso caso seria como saída, e a saída por sua vez pode ter duas configurações: push-pull ou open-drain.

- Push-pull: Usa um CMOS de tal maneira que a saída sempre será HIGH ou LOW.
- Open-drain: A saída será LOW ou ficará em alta impedância, flutuando. [2].

A lógica é usar então um GPIO configurado como saída em push-pull.

2.2 Registradores GPIO

Podemos ver no Manual de Referência dessa família de microprocessadores[3] as possíveis configurações de cada porta GPIO.

Cada porta GPIO possui dois registradores de configuração 32-bit, dois registradores de dados 32-bit, dois registradores set/reset de 32-bit, um registrador reset 16-bit, um registrador de travamento 32-bit.

[1] STM32F103C6 Product Specfications, pg. 21
[2] Open Drain Output vs Push-Pull Output
[3] RM0008 Cap. 9

Percebe-se que cada coisa está associada a memória, por isso precisamos entender anteriormente como funciona a memória.

2.3 Mapeamento da memória

Existem duas formas de trabalhar com periféricos em um microcontrolador, mapeamento por memória, mapeamento por porta, o primeiro tipo associa periféricos com endereços de memória, o segundo tipo não tem um mapeamento conjunto na memória, separando por portas, assim antes de trabalhar é preciso se direcionar a porta desejada[4].

Essa família de STM32 trabalha com mapeamento por memória, e como podemos ver no datasheet[5], temos as portas GPIO entre os endereços:

- Port A: 0x4001 0800 - 0x4001 0BFF
- Port B: 0x4001 0C00 - 0x4001 0FFF
- Port C: 0x4001 1000 - 0x4001 13FF
- Port D: 0x4001 1400 - 0x4001 17FF

Assim os tais registradores se encontram na memória referente a esses periféricos, vamos configurar tudo utilizando a porta A1.

Com base no RM0008 temos que a configuração para saída push-pull é a seguinte:

- MODE[1:0] = 01b ou 10b ou 11b, isso é para indicar que a porta servirá para saída, e indicará a frequência máxima dessa saída. 01b é para 10 MHz, 10b é para 2 Mhz, 11b é para 50 MHz. Como nosso objetivo é só piscar um LED escolheremos a menor frequência logo nosso MODE[1:0] = 10b.
- CNF1 = 0, isso indica que usaremos output de propósito geral, não pretendemos usar nenhuma função alternativa da porta.
- CNF0 = 0, isso indica que usaremos Push-pull ao invés de Open-drain.
- PxODR = 0 ou 1, nossa saída pode ser então zero ou um, LOW ou HIGH.

2.4 Valores dos Registradores

2.4.1 GPIOx_CRL

As portas A são divididas em duas faixas, LOW e HIGH, sendo a faixa LOW as portas A[0:7].

Como usaremos a porta A1, então estamos na faixa LOW.

[4] Bare -Metal STM32: Exploring memory-mapped i/o and linker scripts
[5] STM32F103C6 Product Specfications, pg. 29

Address offset: 0x00

Reset value: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 2.1: Imagem retirada da página 171 do RM0008

Conforme podemos ver na Figura 2.1, sabemos que o valor após RESET de é $0100b$, isto é $MODE[1:0] = 00b$, que é modo de input, e $CNF[1:0] = 01b$ que deixa o input aberto, flutuando.

Deixaremos todos no seu valor de RESET, exceto A1. Sabendo que A1 fica no último byte, podemos manter os restantes no seu valor de reset.

A1 vai com $MODE[1:0] = 10b$, para mostrar que usaremos baixa frequência em output. $CNF[1:0] = 00b$ para mostrar que é um output push-pull com propósito geral. Logo ele fica $0010b$, em hexadecimal $0x2$, podemos alterar simplesmente o segundo digito menos significativo, devido a propriedade de conversão de binário para hexadecimal, que permite alterar de 4 em 4 digitos.

O registrador 32-bit fica portanto: $0x44444424$

Mas onde exatamente ele vai ficar? Sabemos que o offset é 0x00, portanto ele fica exatamente no começo da memória referente a Porta A, logo $[0x40010400 : 0x40010403] = 0x44444424$.

Perceba que o sistema é little-endian, dígitos menos significativos ficam nas memórias mais altas.

2.4.2 GPIOx_CRH

Refere-se as portas altas, será deixado como está.

Outro detalhe importante, é que as portas PA[13:15] não está configuradas como PA[13:15], elas por default estão configuradas como SWDIO, SWCLK, JTDI[6].

Mas estão por default como função alternativa de input flutuando, isso parece uma contradição, porque o RM0008 diz que para SW elas devem estar configuradas como função alternativa de output push-pull, porém não estão, mas isso se explica porque nas configurações de AFIO, o padrão é Full SWJ, que vai configurar as portas para essa funcionalidade assim que PA14 receber uma determinada sequência específicaque determinará se o debug será por SWD ou JTAG. Já que na configuração de input floating, o que realmente está em alta impedância é o output.

[6]STM32F103C6 Product Specfications, pg. 27

2.4.3 GPIOx_IDR

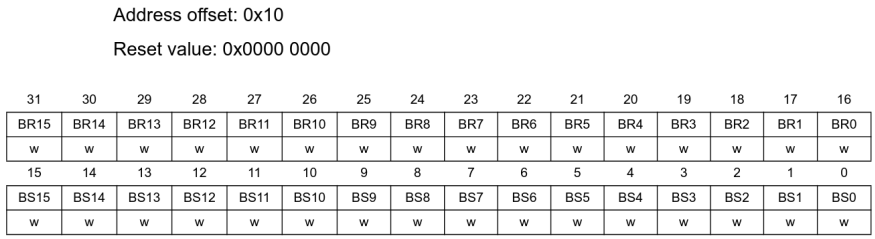
Refere-se a leitura do input dessas portas, no nosso caso não importa, portanto será deixado como está, afinal esses registradores foram feitos apenas para leitura ou são reservados.

2.4.4 GPIOx_ODR

Refere-se ao output, que será LOW ou HIGH, entretanto o RM0008 recomenda que para set/reset atômicos, que é a nossa intenção, mudar especificamente algum bit, não devemos usar esse registrador, mas sim o GPIOx_BSRR, que será explicado na sequência.

2.4.5 GPIOx_BSRR

O registrador serve para mudanças atômicas, os primeiros 16 bits dão reset, os outros dão set. É possível que esses bits se contradigam, como por exemplo dar set e reset junto, porém nesse caso, set tem preferência.



Address offset: 0x10

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Figura 2.2: Imagem retirada da página 173 do RM0008

Temos dois desejos, deixar o resto como está e dar set no A1, e deixar o resto como está e dar reset no A1, alternadamente.

Para deixar como está devemos deixar 0 no bit do registrador, para dar set colocamos 1 no set e 0 no reset, para dar reset colocamos 0 no set e 1 no reset.

Portanto evidentemente para dar set o registrador fica como: $00000000000000000000000000000010b$, ou em hexadecimal: $0x00000002$.

Para dar reset o registrador fica como: $00000000000000100000000000000000b$, ou em hexadecimal: $0x00020000$

O seu endereço de memória pode ser determinado pelo offset de 0x10, $[0x40010410 : 0x40010413] = 0x000[2, 0]000[0, 2]$.

2.4.6 GPIOx_BRR

É uma outra opção para resetar os bits, porém não será usada, e será deixada no seu valor padrão, que é sem efeito.

2.4.7 GPIOx_LCKR

Serve para travar mudanças na configuração da porta até um próximo reset, não será usado, pois não é necessário no nosso caso, aliás é proibido no nosso caso, pois essa configuração proibirá qualquer alteração de configuração nas portas, impedindo que alternemos seus valores.

2.5 Programando o piscar do LED

Precisamos saber de onde o microcontrolador executa código, e sabemos pelo RM0008[7] que ele pode ler instruções de três locais:

- Flash
- System Memory
- SRAM

Não devemos mexer na System Memory, escrever na SRAM não é prático, pois ela se apaga com o desligamento, resta então escrever código na memória Flash.

O RM0008 apresenta como devemos fazer para executar código da memória Flash como vemos na Figura 2.3.

3.4 Boot configuration

In the STM32F10xxx, 3 different boot modes can be selected through BOOT[1:0] pins as shown in *Table 9*.

Table 9. Boot modes

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as boot space
0	1	System memory	System memory is selected as boot space
1	1	Embedded SRAM	Embedded SRAM is selected as boot space

Figura 2.3: Imagem retirada da página 60 do RM0008

2.6 Escrevendo na memória Flash

Como escrever na memória Flash? Nosso único acesso é via SWD, então como podemos usar o SWD para escrever na memória Flash?

[7] RM0008 Cap. 3.4

Podemos encontrar nos pacotes do Debian uma ferramenta chamada `stlink-tools`[8], que permite conectar-se com o ST-LINK, e escrever na memória Flash, será então essa a ferramenta a ser usada, ela permite ler, escrever, e depurar com o `gdb`.

Como o SWD é ativo ativo nativamente nas entradas, como já vimos antes em 2.4.2, não temos que nos preocupar em ativar coisa alguma. Basta escrever no Flash, e dar boot com essa configuração.

O conjunto de ferramentas nos dá o `st-flash`, que nos permite escrever na memória flash, ou ler algo de qualquer canto da memória, mas para escrever, precisamos do endereço da memória flash.

Pelo datasheet podemos saber que é `0x0800 0000`[9].

2.7 A escrita do código

Como programar? Existe um manual[10] de programação para a família `STM32F10xxx`, que usa o processador Cortex-M3. Outra ferramenta importante é a Manual de Referência Técnica[11] do Cortex-M3, citado no Datasheet[12].

2.7.1 Pseudo-código, v1.

Precisamos do seguinte:

1. Escrevemos nos respectivos registradores as configurações necessárias para a porta A1.
2. Damos set na saída da porta A1.
3. Esperamos um tempo.
4. Damos reset na saída da porta A1.
5. Esperamos um tempo.
6. Reiniciamos o ciclo pelo set em A1.

Daí concluímos que temos três problemas para transmitir ao Cortex M3:

- Armazenar e copiar valores na memória.
- Contar o tempo.
- Repetir um ciclo.

Vamos abordá-lo gradualmente.

[8] Package: stlink-tools, Debian.
[9] STM32F103C6 Product Specfications, pg. 29
[10] PM0056, Programming manual
[11] Cortex-M3 Technical Reference Manual
[12] STM32F103C6 Product Specfications, pg. 9

2.7.2 Registradores

Para armazenar e copiar valores temos os registradores.

Podemos verificar no manual de programação do Cortex M3[13] que o Cortex M3 usa registradores como memória interna, temos que nos ocupar aqui com diferentes tipos de registradores, mas por enquanto veremos os que aparentemente usaremos:

- Registradores de propósito geral: eles existem em dois tipos, os baixos e os altos, os baixos vão do R0 até R7, os altos do R8 até o R12, a diferença central entre eles é que os baixos podem trabalhar com 16-bits ou 32-bits, enquanto os altos apenas com 32-bits.

2.7.3 MOVW/MOVT

Com MOVW e MOVT é possível armazenar qualquer constante na memória[14]. MOVW pode ser usado para copiar uma constante de 16 bits para dentro de um registrador, mas na metade menos significante, MOVT copia para a metade mais significante.

Atenção: A documentação **NÃO** diz que MOVW não altera a parte superior do registrador, apenas afirma que MOVT não altera a inferior, e de fato é isso MOVW zera a parte superior. Portanto deve-se usar MOVW antes de MOVT.

Assim podemos armazenar o endereço de configuração do GPIO-A.

```
MOVW R0, #0x0800
MOVT R0, #0x4001
```

Podemos também armazenar o valor das configurações:

```
MOVW R1, #0x4424
MOVT R1, #0x4444
```

2.7.4 LDR/STR

Com LDR podemos pegar o valor de um registrador, para o endereço de memória presente em outro[15].

Já o STR faz o contrário e joga o valor do registrador para a memória que está em outro registrador. No nosso caso queremos jogar os valores de R1 para o endereço presente em R0.

```
STR R1, [R0]
```

[13] PM0056, pg.14
[14] PM0056, pg.80
[15] PM0056, pg. 64

2.7.5 Criando os valores de set e reset

Já vimos anteriormente como devem ser os valores de set e reset, portanto basta apenas colocá-los na memória:

```
MOVW R1, #0x0002
MOVT R1, #0x0000
MOVW R2, #0x0000
MOVT R2, #0x0002
```

Assim R1, que antes armazena a configuração da porta, foi trocado pela configuração de set, e R2 configuração de reset.

Para ligar o LED basta então usar:

```
MOVW R0, #0x0810
MOVT R0, #0x4001
LDR R1, [R0]
```

2.7.6 Como contar o tempo?

Não se encontra funções no conjunto de instruções que façam ele parar por um tempo, existem no máximo funções que esperam por eventos externos ou interrupções externas, mas não temos nenhum equipamento externo para provocar essas alterações, então devemos usar o seguinte:

Cada comando exige um determinado número de ciclos, e cada ciclo tem um tempo aproximadamente preciso, conforme a determinação do relógio, sendo assim, podemos contar o tempo, mas antes precisamos configurar os relógios.

2.8 Configurando relógios

No RM0008[16] podemos ver uma árvore dos relógios:

[16] RM0008 pg. 93

Figure 8. Clock tree

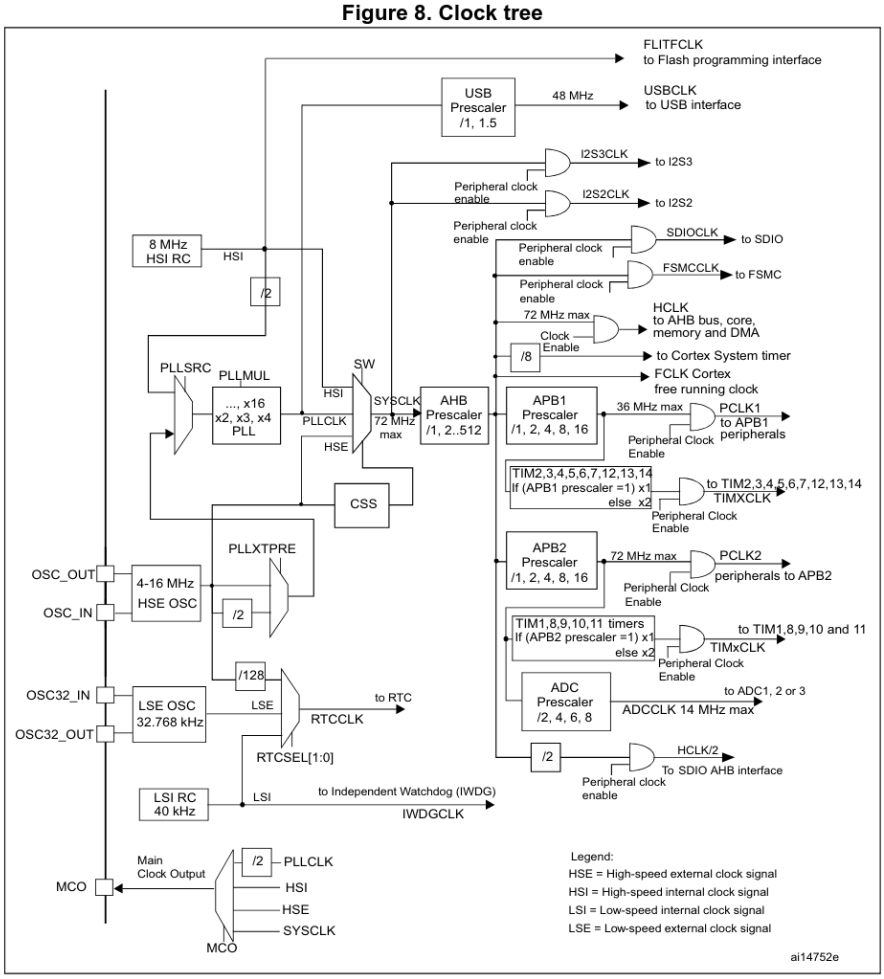


Figura 2.4: Imagem retirada da página 93 do RM0008

Sabemos que GPIO-A usa APB2, portanto devemos ligar esse relógio, e sabemos que estamos limitados para 2MHz nas configurações do GPIO, portanto não convém superar essa velocidade. Nosso HSI é de 8MHz, portanto basta fazermos o seguinte: SYSCLK é HSI, mas APB2 prescaler divide por 8.

2.8.1 RCC_CFGR

Temos algo importante na Figura 2.5, PPRE2 precisa ser colocado para `110b` para provocar a divisão da frequência por 8.

Assim sabemos que $[0x40021004 : 0x40021007] = 0x00003000$

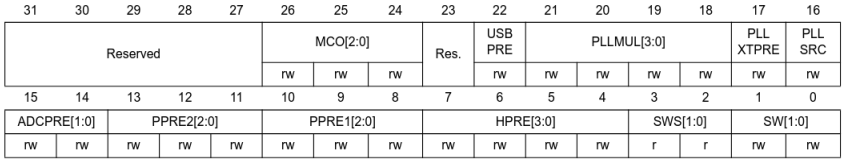


Figura 2.5: Imagem retirada da página 101 do RM0008

2.8.2 RCC_APB2ENR

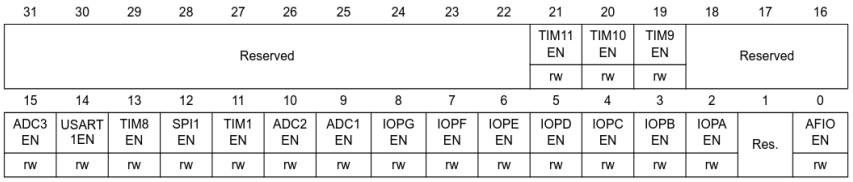


Figura 2.6: Imagem retirada da página 113 do RM0008

Podemos verificar que na Figura 2.6 que devemos ativar o relógio para os periféricos, no caso ativaremos para IOPA_EN. Assim sabemos que $[0x40021018 : 0x4002101B] = 0x00000004$.

Assim temos os relógios como são exigidos, temos então 1MHz na GPIO-A, portanto basta configurar o número de ciclos conforme a frequência que colocamos, mas não é a frequência de GPIO-A, essa frequência só foi reduzido para economizar energia, mas não é ela quem calcula as coisas, ela apenas transfere e administra os dados, quem calcula é a CPU, e a CPU recebe a frequência de HCLK, que é SYSCLK no nosso caso, pois não dividimos para AHB.

2.9 Como fazer o ciclo se repetir?

Precisamos encontrar um comando que repita sempre esse processo, esse comando é o B e seus derivados[17].

Ele trabalha com uma label, uma label é uma abstração no assembly para representar um determinado endereço de memória onde fica uma instrução. Assim o B retorna para essa label.

Há também B com condições, como por exemplo BEQ, que será útil no caso, ele apenas vai para a label se uma determinada comparação der um resultado correto.

[17] PM0056, Programming manual pg. 92

Mas como fazer comparações? Podemos usar o comando CMP[18]. Ele permite comparar dois valores, e ver se é igual, o resultado da comparação é armazenado em uma flag, um outro registrador interno da CPU, assim ele registra se o valor é negativo ou se é zero. BEQ vai procurar por um flag zero 1.

2.10 Desenvolvendo o código

Vamos primeiro relembrar os valores que devemos colocar nas memórias:

Registrador	Endereço	Valor
RCC_CFGR	0x40021004	0x00003000
RCC_APB2ENR	0x40021018	0x00000004
GPIOA_CRL	0x40010800	0x44444414
GPIOA_BSRR	0x40010810	0x00020000
GPIOA_BSRR	0x40010810	0x00000002

A primeira etapa é configurar os relógios:

```
movw r0, 0x1004
movt r0, 0x4002
movw r1, 0x3000
str r1, [r0]
movw r0, 0x1018
movt r0, 0x4002
movw r1, 0x0004
str r1, [r0]
```

Perceba que não fizemos o "movt"para o r1, pois o "movw"já zero em cima.
Agora vamos configurar a saída GPIOA_CRL.

```
movw r0, 0x0800
movt r0, 0x4001
movw r1, 0x4414
movt r1, 0x4444
str r1, [r0]
```

Agora temos que pensar em um ciclo, o ciclo será simplesmente um conjunto de operações que se repetirá até que passe o tempo, e como se passará o tempo? Pela contagem de ciclos das operações.

```
movw r0, 0x0810
movt r0, 0x4001
movw r1, 0x0002
movw r2, 0x0000
```

[18]PM0056, Programming manual pg. 78

```
movt r2, 0x0002

ciclo1:
movw r3, 0xXXXX @contador
movt r3, 0xXXXX
mov r4, r1 @ alternando r1 e r2 para provocar a mudanca
mov r1, r2
mov r2, r4
movw r4, 0x0000 @ zero, o valor a ser comparado.
str r1, [r0]

ciclo2:
sub r3, #0x0001
cmp r3, r4
beq ciclo1 @ quando o contador for zero, ciclo1
b ciclo2 @ se nao for repete ciclo2
```

Para calcular o valor exato que colocaremos no contador, vamos contar quanto duram os ciclos. O Manual de Referência Técnica do Cortex-M3[19] nos dá o número de ciclos de cada instrução, para as que utilizamos temos o seguinte:

MOVW	1
MOVT	1
MOV	1
SUB	1
CMP	1
STR	2
B{cc}	1 ou 1 + P

B condicionado será 1 apenas no caso em que ele não efetue o branch.

Mas precisamos determinar o P, esse P depende de 3 fatores: alinhamento das palavras, o comprimento da instrução, e se o processador pode prever o endereço para onde ele vai. Como sempre usaremos instruções alinhadas, ou seja com memória múltipla, e o endereço para onde vamos é fixo, resta então supor que P seja em média 1, logo nossos branches demoraram tempo 1 se não forem executados, e tempo 2 se forem executados.

Vamos calcular então o tempo do ciclo 1:

$1 + 1 + 1 + 1 + 1 + 1 + 2 + R \cdot (1 + 1 + 1 + 2) =$
$8 + 5R$

Assim temos 8 + 5R ciclos em média até trocar o LED.

A frequência da nossa CPU é 8MHz, sendo assim cada ciclo dura 125ns.

Se quisermos que cada ciclo dure 500ms, precisamos então de $500ms/125ns =$ 4000000 instruções.

Eu vou desprezar o 8 do $8 + 5R$ pois ele é imperceptivel, logo teremos $5R =$ $4000000 \rightarrow R = 800000$, logo sabemos o valor do nosso registrador $0xc3500$

[19] Technical Reference Manual - Cortex-M3

2.11 Versão final do código

```
movw r0, 0x1004
movt r0, 0x4002
movw r1, 0x3000
str r1, [r0]
movw r0, 0x1018
movt r0, 0x4002
movw r1, 0x0004
str r1, [r0]

movw r0, 0x0800
movt r0, 0x4001
movw r1, 0x4414
movt r1, 0x4444
str r1, [r0]

movw r0, 0x0810
movt r0, 0x4001
movw r1, 0x0002
movw r2, 0x0000
movt r2, 0x0002

ciclo1:
movw r3, 0x3500 @contador
movt r3, 0x000c
mov r4, r1 @ alternando r1 e r2 para provocar a mudanca
mov r1, r2
mov r2, r4
movw r4, 0x0000 @ zero, o valor a ser comparado.
str r1, [r0]

ciclo2:
sub r3, #0x0001
cmp r3, r4
beq ciclo1 @ quando o contador for zero, ciclo1
b ciclo2 @ se nao for repete ciclo2
```

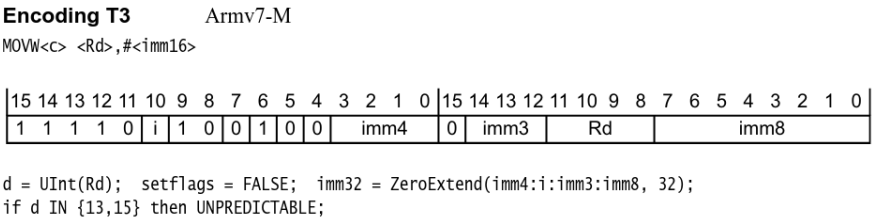
2.12 Jogando o código para o STM32

É preciso jogar um arquivo binário para que o código seja executado corretamente. O Manual de Referência Técnica do Cortex-M3[20] nos afirma que o processador usa instruções ARMv7-M Thumb[21].

[20] Technical Reference Manual - Cortex-M3
[21] ARMv7-M Architecture Reference Manual

2.12.1 Comando MOVW

Estamos usando MOVW com um valor imediato de 16-bits, portanto usaremos o Enconding T3 em Thumb do MOV immediate.



Encoding T3 Armv7-M

MOVW<c> <Rd>,#<imm16>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	1	0	0	imm4				0	imm3			Rd				imm8							

d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;

Figura 2.7: Imagem retirada da página 291 do Manual de Referência da Arquitetura ARMv7-M.

Vamos pegar um comando de exemplo: `movw r0, 0x1004`.

Primeiro precisamos identificar os valores de Rd, e os valores imediatos.

Nosso Rd é R0, o valor é então $0000b$.

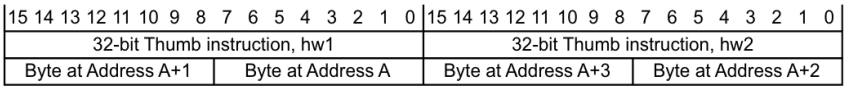
Os valores de imm4, i, imm3, imm8 são $imm16 = imm4 : i : imm3 : imm8$.

Sabemos que $imm16$ é $0x1004 = 0001000000000100b$, fazendo a separação desejada temos: $0001 : 0b : 000b : 00000100b$, portanto resta colocar cada valor no seu lugar.

$11110010010000010000000000000100b$

Em hexadecimal temos: $F2410004$

Resta ver como devemos jogar essa sequência na memória, é importante se atentar para o fato de que o sistema é little endian.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
32-bit Thumb instruction, hw1	32-bit Thumb instruction, hw2
Byte at Address A+1 / Byte at Address A	Byte at Address A+3 / Byte at Address A+2

Figure A3-5 Instruction byte order in memory

Figura 2.8: Imagem retirada da página 68 do Manual de Referência da Arquitetura ARMv7-M.

Pode-se perceber que as half-words são invertidas, logo isso entrará na memória como:

$41F20400$

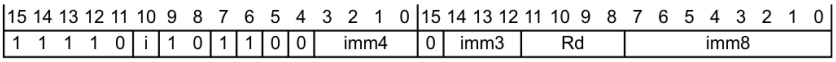
2.13 Assembler

Vamos produzir um assembler para o código que fizemos, para isso precisamos ver o Encoding de outros comandos que usamos.

2.13.1 MOVT

Encoding T1 Armv7-M

```
MOVT<c> <Rd>,#<imm16>
```




```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d IN {13,15} then UNPREDICTABLE;
```

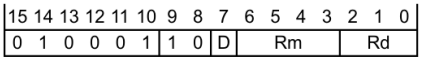
Figura 2.9: Imagem retirada da página 296 do Manual de Referência da Arquitetura ARMv7-M.

2.13.2 MOV (dois registradores)

Encoding T1 Armv6-M, Armv7-M

If <Rd> and <Rm> both from R0-R7,
otherwise all versions of the Thumb instruction set.
If <Rd> is the PC, must be outside or last in IT block

```
MOV<c> <Rd>,<Rm>
```




```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Figura 2.10: Imagem retirada da página 293 do Manual de Referência da Arquitetura ARMv7-M.

Perceba que ele só permite usar os registradores baixos, não os altos.

2.13.3 SUB

Encoding T2 All versions of the Thumb instruction set.

```
SUBS <Rdn>,#<imm8>
```
Outside IT block.

```
SUB<c> <Rdn>,#<imm8>
```
Inside IT block.

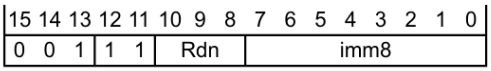


Figura 2.11: Imagem retirada da página 402 do Manual de Referência da Arquitetura ARMv7-M.

Também aceita apenas registradores baixos.

2.13.4 STR (2 registradores)

Encoding T1 All versions of the Thumb instruction set.

```
STR<c> <Rt>, [<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Figura 2.12: Imagem retirada da página 386 do Manual de Referência da Arquitetura ARMv7-M.

2.13.5 CMP (2 registradores)

Encoding T1 All versions of the Thumb instruction set.

```
CMP<c> <Rn>,<Rm>
```

<Rn> and <Rm> both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm			Rn		

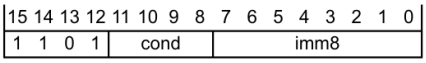
```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Figura 2.13: Imagem retirada da página 224 do Manual de Referência da Arquitetura ARMv7-M.

2.13.6 B{cc}

Encoding T1 All versions of the Thumb instruction set.

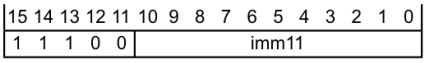
`B<c> <label>` Not permitted in IT block.




```
if cond == '1110' then SEE UDF;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb instruction set.

`B<c> <label>` Outside or last in IT block




```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Figura 2.14: Imagem retirada da página 205 do Manual de Referência da Arquitetura ARMv7-M.

Primeiro veremos o que exatamente é `cond`:

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS[b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC[c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL)[d]	Always (unconditional)	Always (unconditional)	Any

Figura 2.15: Imagem retirada da página 178 do Manual de Referência da Arquitetura ARMv7-M.

Esse comando requer uma atenção especial. Esses imediatos se referem na verdade a um offset em relação a PC, o que é PC? PC é o program counter, ele indica o próximo endereço (de palavra, ou seja + 4 bytes) a ser executado no caso de instruções de branch[22].

Mas esse valor tem sinal, ou seja, pode ser negativo, como representar números negativos para o processador? Segundo o manual da arquitetura[23] os números negativos são representados em complemento de 2, a não ser que explicitamente se diga outra coisa.

Além do mais, há um detalhe, antes de tudo offset sofre o seguinte trabalho: ele recebe um zero no final antes de sofrer o offset. Assim apenas branches múltiplos de 2 bytes são permitidos.

2.13.7 Produzindo o Assembler

O assembler a ser produzido não será um assembler completo, mas um assembler apenas das instruções.

[22] PM0056, Programming manual pg 56
[23] ARMv7-M Architecture Reference Manual pg 855

E ele funcionará da seguinte forma, escreve-se o comando, e ele retornará o hexadecimal. Após isso o conteúdo poderá simplesmente ser copiado num editor hexadecimal.

Sua produção não será abordada aqui, pois é trivial, mas o código estará no Github [24]

2.13.8 Código binário

Eu os escrevi com um endereço de referência para conseguir calcular os offsets do branches.

```
00: 41f20400
04: c4f20200
08: 43f20001
0c: 0160
0e: 41f21800
12: c4f20200
16: 40f20401
1a: 0160
1c: 40f60000
20: c4f20100
24: 44f21441
28: c4f24441
2c: 0160
2e: 40f61000
32: c4f20100
36: 40f20201
3a: 40f20002
3e: c0f20202
42: 43f20053
46: c0f20c03
4a: 0c46
4c: 1146
4e: 2246
50: 40f20004
54: 0160
56: 013b
58: a342
5a: f2d0
5c: fbe7
```

2.13.9 NVIC e Stack - Vetores

Há porém um detalhe, segundo o RM0008, página 61, durante o BOOT, o endereço 0x0 serve para definir o topo do stack, e o endereço 0x4 serve para

[24] ArmV7-M Assembler

definir o endereço da primeira instrução, porém há ainda mais um detalhe, segundo o Manual de Referência Técnica do Cortex-M3, em Programmers Model - Exceptions, todo vetor de ponto de entrada deve ter bit[0] igual a 1.

A Stack do Cortex M3 é descendente, portanto quanto maior o endereço, maior será a Stack, como não usamos a Stack, ela será seu endereço de início, o início da memória RAM que é 0x20000000, como está no Memory Mapping do Datasheet.

Todo endereço ocupa uma word, 4 bytes, portanto seja lá qual for o endereço da primeira instrução, ele ocupa 4 bytes, portanto a primeira instrução fica em 0x0800008, pois começa no nono byte da memória Flash, como o bit[0] precisa ser 1, temos que a primeira instrução deve ser colocada como 0x08000009.

Porém como as words são registradas em little-endian, é preciso inverter todos os bytes, conforme especifica o PM0056 na página 30.

Assim teremos antes de tudo:

```
00000020
09000008
```

2.13.10 wxHexEditor

Usei o wxHexEditor para montar o binário conforme tudo que foi colocado:

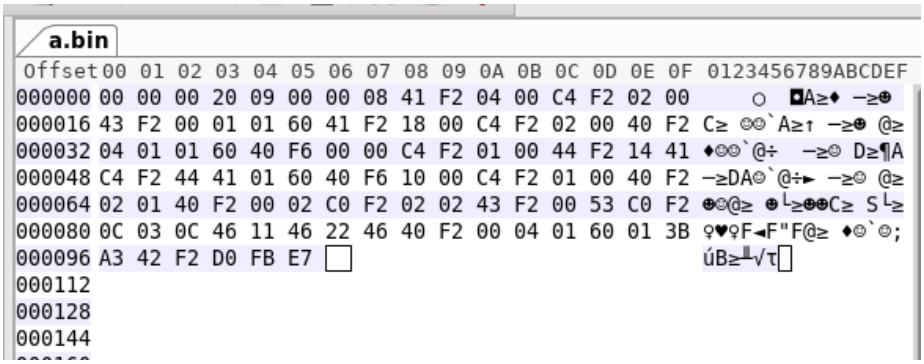


Figura 2.16: Print de trecho da tela do wxHexEditor 0.24 Beta for Linux.

2.14 st-flash e Resultados

Para colocar o programa, basta jogá-lo para memória flash como o st-flash, garantindo que o boot esteja configurado para memória flash com BOOT0 = BOOT1 = 0.

```
st-flash write a.bin 0x08000000
```

Percebe-se que funcionou, como ficou no vídeo postado no youtube em meu perfil[25]

[25]LED Piscando - STM32 Blue Pill.

Capítulo 3

Problema 2: PWM

Um PWM (Pulse Width Modulatation) é uma técnica usada em eletrônica para controlar a potência média alterando o tamanho do pulso. [1]

O objetivo é então controlar duas entradas por meio de potenciômetros, uma entrada o período, a outra controla o tamanho do pulso. Um detalhe importante é que essas entradas não serão simplesmente True/False, elas terão valores intermediários, e esses valores intermediários precisam ser levados em conta. Tudo indica que será necessária uma conversão AD, vamos ver isso então.

3.1 Configurando ADC

3.1.1 Alimentação

Para alimentar é necessário ter o seguinte[2]: VDD, VSS, VDDA, VSSA, VREF+, VREF-. Porém no nosso caso não será necessário se preocupar com nada disso, pois como podemos ver no esquemático do Bluepill[3], essas coisas já estão conectadas com 3V3 ou GND.

Há outro detalhes importante: os inputs são feitos pelos GPIOS configurados como funções alternativas.

3.1.2 Configurando os IOs

Vamos utilizar PA1 e PB1 para receberem as entradas.

Para usar ADC é necessário configurar os GPIOs na função Analog[4].

Analog mode é uma função disponível apenas em Input mode, portanto nos registradores do GPIO devemos colocar MODE = 00, para que entre em Input

[1] Byjus, Pulse Width Modulation
[2] RM0008 Pg 218
[3] Esquemático
[4] RM0008 pg. 169

Mode, e CNF = 00, para que entre em Analog Mode.[5].

Essas configurações tem valor de reset `0x4444 4444`, portanto para colocar 0 nas portas 1, temos `4444 4404`. Esse registrador tem offset `0x00` nas porta GPIO. O endereço de GPIOA[6] é `0x4001 0800`, e de GPIOB é `0x4001 0C00`.

Portanto já temos os valores corretos para colocar em cada lugar.

3.1.3 Ligando o relógio

É necessário ligar o relógio do ADC para que ele funcione, e também dos GPIOs, pois vamos utilizá-los. Como podemos ver no datasheet[7], ADC1 está conectado com APB2, portanto devemos ativar o relógio ali,assim como GPIOA e GPIOB.

Portanto conforme o RM0008[8] devemos ativar os bits de ADC1EN, IOPAEN, IOPBEN.

O que nos dá um registrador com o seguinte: `0x0000 020C`

A memória do controle de relógio fica em `0x4002 1000` como já vimos, e esse registrador tem offset `0x18`.

3.1.4 Ligando o ADC

É necesário ativar o ADC conforme explica o RM0008[9] no registrador ADC_CR2.

Mas antes de ligá-lo precisamos entender melhor como ele funciona, ele apresenta três modos: Conversão única, contínua e scan. O modo de conversão única funciona da seguitne maneira: eu configuro ele para fazer uma leitura, ele faz e acaba, o modo de leitura contínua funciona para ele fazer uma leitura, e refazê-la várias vezes, o modo de scan permite que façamos um grupo de leitura, enquanto armazenamos essas informações na memória SRAM. Por praticidade, eu vou usar o modo de leitura única, e vou alternando a configuração entre os canais, já que o meu problema não exige uma velocidade muito alta, e armazenar valores na SRAM exige que eu ligue o DMA, o que seria um gasto a mais de energia.

Como ligar o modo de conversão única? Basta que no registrador ADC_CR2, eu ligue ADON, porém com CONT desligado. Sabemos pelo datasheet que ADC1 fca em: `0x4001 2400`. O registrador conform página 240 do RM0008 tem offset 0x08, e o valor que queremos colocar será `0x0000 0001`.

3.1.5 Sequência Regular

É necessário configurar a sequência regular, queremos definir que desejamos ler PA1, e PB1, mas como podemos fazer isso?

Basta configurar a sequência, mas como nesse modo de ADC se lê apenas um de cada vez, devemos fazer os ajustes necessários. Como podemos ver na

[5] RM0008 pg. 171
[6] Datasheet, pg 29
[7] Datasheet, pg 12
[8] RM0008 pg. 113
[9] RM0008 pg. 218

seção 11.12.9 do RM0008, podemos ver que o valor padrão da sequência de conversão é um, como é nosso interesse, agora basta ajusta o primeiro valor a ser convertido que fica no registrador ADC_SQR3. Vamos ver no datasheet qual o valor dos registradores:

Na tabela de Pinouts and Pin Description do Datasheet nós podemos ver quais as portas ADC de cada um, PA1 é ADC12_IN1. PB1 é ADC12_IN9

Portanto no caso de PA1 queremos que o valor do registrador seja `0x0000 0001` e de PB1 seja `0x0000 0009`.

3.1.6 Dando o Start da conversão

É preciso que após ter iniciado tudo se espere um tempo para que o ADC inicialize, podemos usar um loop como já fizemos antes.

Depois de configurado tudo, é só reescrever o bit, se eu escrever 1 no bit ADON que já é 1, ele inicia a conversão, conforme nos diz o RM0008 na página 243.

3.1.7 Lendo os dados

Antes de ler é preciso ver se EOC está ativado, ou seja se a conversão terminou.

Os dados podem ser obtidos nos bits de baixo do ADC_DR, que tem offset $0x4C$. É importante definir o alinhamento dos 12 bits, se serão para esquerda ou para direita, por padrão eles são para direita.

3.2 Teste: Acendendo um LED com condição da leitura de uma tensão.

Como teste do nosso ADC vamos fazer o seguinte, vamos controlar um LED verificando se a tensão obtida em uma determinada porta é maior ou menos que um determinado valor.

3.2.1 Projeto base

Vamos usar PA1 para leitura do input como analógico, e vamos usar PA2 para saída do LED como output.

3.2.2 Tabela de registradores

RCCAPB2ENR	0x40021018	0x00000204
GPIOACRL	0x40010800	0x44444104
GPIOABSRR	0x40010810	0x00000004
GPIOABSRR	0x40010810	0x00040000
ADCCR2	0x40012408	0x00000001
ADCSQR3	0x40012434	0x00000001

3.2.3 Código Pseudo-Assembly

Primeiro vamos ligar o relógio:

```
movw r0, 0x0204
movw r1, 0x1018
movt r1, 0x4002
str r0, [r1]
```

Agora vamos configurar as portas:

```
movw r0, 0x4104
movt r0, 0x4444
movw r1, 0x0800
movt r1, 0x4001
str r0, [r1]
```

Ligar ADC e esperar um pouco:

```
movw r0, 0x0001
movw r1, 0x2408
movt r1, 0x4001
str r0, [r1]
movw r1, 0xffff
movw r0, 0x0000
loop:
sub r1, 0x0001
cmp r1, r0
bne loop
```

Configurar ADC:

```
movw r0, 0x0001
movw r1, 0x2434
movt r1, 0x4001
str r0, [r1]
```

Iniciar conversão e esperar EOC, código reaproveitado para agilizar.

```
movw r0, 0x0001
movw r1, 0x2408
movt r1, 0x4001
str r0, [r1]
movw r1, 0xffff
movw r0, 0x0000
loop:
sub r1, 0x0001
cmp r1, r0
bne loop
```

Ler a entrada e decidir:

```
movw r1, 0x244c
movt r1, 0x4001
movw r2, 0x0810
movt r2, 0x4001
movw r3, 0x0004
movw r4, 0x0000
movt r4, 0x0004
ldr r0, [r1]
cmp r0, 0x0800 @ metade
bge acende
b apaga
acende:
str r3, [r2]
b loop3
apaga:
str r4, [r2]
loop3:
b loop3
```

Compilando, temos o código funcionando perfeitamente, quando A1 é HIGH, o LED acende, quando A1 é LOW, o LED apaga. Não consigo testar muitos valores intermediários devido a falta de potenciômetros, mas sem dúvida funcionaria. Pode-se ver seu funcionamento no vídeo do youtube[10]

3.3 Configurando o Contador

De acordo com o datasheet, podemos configurar um PWM usando os Timers, usaremos um Advanced-control Timer.

O TIM1 tem 4 registradores importantes para marcar o tempo, o TIM1_CNT, TIM1_PSC, TIM1_ARR, TIM1_RCR.

O TIM1_CNT conta, ele basicamente conta até um determinado número, marcando assim o tempo.

O TIM1_PSC é um prescaler, ele altera a frequência do clock do TIM1 no caso para frequencia do contador, de quanto em quanto tempo ele atualiza.

O TIM1_RCR é usado quando desejamos alterar as coisas apenas após um determinado número de repetições da contagem inteira.

O TIM1_ARR vai determinar quando as configurações vão se atualizar, o valor máximo do contador.

Mas antes devemos configurar sempre o TIM1_CR1.

[10]LED Controlado por Input Analógico

3.3.1 TIM1_CR1

As configurações se encontram na página 338 do RM0008.

Queremos configurar para a configuração só alterar após completar uma contagem, portanto nosso bit ARPE deve ser positivo.

Nossa contagem será para cima, portanto CMS = 00.

A direção sendo para cima será 0, como já é o valor padrão.

As outras configurações não nos interessam, exceto CEN, que ativa o relógio, e que será um para nós.

Portanto a configuração ficará: `0x0000 0001`

TIM1 fica na memória `0x4001 2C00` e tem offset 0x00.

3.3.2 TIM1_PSC

Vamos usar os 8MHz padrão que chegam no Timer, porém queremos uma frequência bem baixa, queremos que cada número da contagem dure 1ms, ou seja nossa frequência deve ser 1KHz, logo nosso prescaler deve ser 7999 + 1. Que em hexadecimal é 0x1F3F.

Esse registrador tem offset 0x28.

3.3.3 TIM1_ARR

Aqui deve entrar o tempo máximo em ms que obteremos de uma porta analógica.

Esse registrador tem offset 0x2C.

3.4 Configurando o PWM

Este conteúdo se encontra na página 317 do RM0008.

Precisamos configurar ele para modo de PWM, usaremos a porta 1 do PWM que conforme o datasheet pode ser utilizada na PA8.

Colocaremos OC1M para 110, que é PWM edge-aligned em TIMx_CCMRx.

Devemos ativar seu preloader em OCxPE no mesmo registrador.

As configurações do output se encontram em TIMx_CCER.

3.4.1 TIMx_CCMR1

Pode-se perceber que esse registrador apresenta múltiplos significados, e exige uma configuração prévia, nós queremos usá-lo como output, portanto devemos obrigatoriamente configurar CC1S como output, ou seja modo de comparação, para isso CC1S deve ser 00, é necessário porém que ele esteja desativado para conseguirmos escrever, mas ele vem por padrão desativado e depois devemos ativar no TIM1_CCER colocando 0x1 como valor.

OC1PE deve ser 1, para ativar seu preload, e permitir que seja alterado a cada ciclo completo do relógio. OC1M como vimos será 110.

Assim esse registrador que se encontra no endereço `0x4001 2C00`, com offset 0x18 deve ter valor: `0x0068`.

3.4.2 TIMx_CCR1

Esse é o valor de comparação, a saída será high sempre que o valor do contador for menor do que esse comparado, e low caso contrário.

Esse valor será tirado de uma entrada analógica.

Esse registrador tem offset 0x34.

Se ele for maior que o ARR, a saída será sempre high, mas atualizando conforme ARR.

Está então tudo perfeito, como deve estar, vamos escrever o código.

3.4.3 Portas

Conforme o datasheet, a porta que tem função alternativa de canal 1 do TIM1 é PA8, portanto devemos configurar PA8.

Segundo a página 166 do RM0008 para ser output de TIM1, a porta deve estar configurada como Alternate function push-pull.

PA8 é uma porta alta, portanto no registrador GPIOA_CRH que tem endereço `0x4001 0800` com offset `0x4` devemos configurar a porta 8 com MODE 01, pois é output mode, e CNF como 10, para função alternativa com push-pull na saída.

3.5 Código

Vamos começar ligando os relógios:

Nós usaremos PA1, PB1 como entradas analógicas ambas em ADC1 com single conversion mode, e usaremos PA8 como saída do TIM1, portanto devemos ligar o relógio para os seguinte periféricos: GPIOA, ADC1, TIM1.

Como podemos ver no Perfomance line block diagram do Datasheet, GPIOA, GPIOB, TIM1, ADC1 estão todos alimentados por APB2.

Os relógios serão deixados como estão, ou seja todo mundo trabalhando a 8 MHz conforme HSI.

```
movw r0, 0x1018 @ endereço RCC
movt r0, 0x4002
movw r1, 0x0a0c @ ativa tim1, adc1, gpioa, gpiob
movt r1, 0x0000
str r1, [r0]
```

Configurando GPIO:

```
movw r0, 0x0800
movt r0, 0x4001
movw r1, 0x4404 @ analog mode
movt r1, 0x4444
str r1, [r0]
```

```
movw r0, 0x0804
movt r0, 0x4001
movw r1, 0x4446 @ output pwm
movt r1, 0x4444
str r1, [r0]

movw r0, 0x0C00
movt r0, 0x4001
movw r1, 0x4404 @ analog mode
movt r1, 0x4444
str r1, [r0]
```

Configurando ADC:

```
movw r0, 0x2400 @ adc1
movt r0, 0x4001
movw r1, 0x0001 @ liga adc1 em single conversion mode.
str r1, [r0]

movw r0, 0xFFFF @ esperar ligar
movw r1, 0x0000
loop1:
sub r0, 0x0001
cmp r0, r1
bge loop1
```

Ligando o TIM1:

```
movw r0, 0x0081 @ arpe, upcounting, clock enable
movw r1, 0x2c00
movt r1, 0x4001 @ tim1 cr1
str r0, [r1]

movw r0, 0x0068 @ pwm mode, preloaded, output
movw r1, 0x2c18 @ ccmr1
movt r1, 0x4001 @ tim1
str r0, [r1]

movw r0, 0x0001 @ liga canal 1
movw r1, 0x2c20 @ tim ccer
movt r1, 0x4001
str r0, [r1]

movw r0, 0x1f3f @ 8000 - 1 = 7999
movw r1, 0x2c28 @ tim_psc
```

```
movt r1, 0x4001
str r0, [r1]
```

Perceba que esses registradores só usam halfword, e que o comando de str é little-endian, mas tudo fica certo, porque na memória fica: segundo byte do primeiro registrador, primeiro byte do primeiro registrador, segundo byte do segundo registrador, primeiro byte do segundo registrador. Mas o primeiro byte a entrar é exatamente o menor, isto é o segundo byte do primeiro registrador, depois o primeiro, assim em diante, então está tudo certo.

Fazendo a leitura de ARR:

```
leitura:
movw r0, 0x0001 @ PA1 = IN1 do ADC
movw r1, 0x2434 @ adc_sqr3
movt r1, 0x4001
str r0, [r1]

@ codigo reaproveitado para poupar tempo.
movw r0, 0x2408 @ adc1 cr2
movt r0, 0x4001
movw r1, 0x0001 @ inicia conversao
str r1, [r0]
movw r0, 0xFFFF
movw r1, 0x0000
loop2:
sub r0, 0x0001 @ espera para termino de conversao, poderia so esperar EOC, porem isso é mais
cmp r0, r1
bge loop2

movw r1, 0x244c @ adc dr
movt r1, 0x4001
ldr r0, [r1] @ le a entrada.
movt r0, 0x0000 @excluindo o halfword de cima.

movw r1, 0x2c2c @arr
movw r1, 0x4001
str r0, [r1] @ joga a leitura em arr
```

Depois a leitura de CCR1, e por fim repete as leituras:

```
movw r0, 0x0009 @ PB1 = IN9 do ADC
movw r1, 0x2434 @ adc_sqr3
movt r1, 0x4001
str r0, [r1]

movw r0, 0x2408 @ adc1 cr2
```

```
movt r0, 0x4001
movw r1, 0x0001 @ inicia conversao
str r1, [r0]
movw r0, 0xFFFF
movw r1, 0x0000
loop3:
sub r0, 0x0001 @ espera para termino de conversao, poderia so esperar EOC, porem isso é mais
cmp r0, r1
bge loop3

movw r1, 0x244c @ adc dr
movt r1, 0x4001
ldr r0, [r1] @ le a entrada.
movt r0, 0x0000 @excluindo o halfword de cima.

movw r1, 0x2c34 @ccr1
movw r1, 0x4001
str r0, [r1] @ joga a leitura em cc1
b leitura
```

E é isto.