LABORATÓRIO TPSE I

Prática 02: Primeiro código para Pisca um LED

Prof. Thiago Werlley

28 de abril de 2025

1 Ativação do clock do GPIO1

Referência: **AM335x TRM, capítulo 8 - Clock Management**.

- A maioria dos módulos do AM335x (UART, GPIO, etc.) não recebem clock automaticamente.
- Antes de usar qualquer periférico, é necessário:
  1. Habilitar o módulo (modulemode = enable).
  2. Ativar os clocks funcionais.
- O endereço base de controle de clock é o CM_PER:

```
#define CM_PER (*(volatile unsigned int *)0x44E00000)
```

- O registrador que controla o clock do GPIO1 é o:

```
CM_PER_GPIO1_CLKCTRL (offset 0xAC)
```

- Ou seja, o endereço completo para controle do clock do GPIO1 é:

```
0x44E00000 + 0xAC = 0x44E000AC
```

- No código:

```
CM_PER |= CM_PER_GPIO1_CLKCTRL |
          CM_PER_GPIO1_CLKCTRL_MODULEMODE_ENABLE |
          CM_PER_GPIO1_CLKCTRL_OPTFCLKEN_GPIO_1_GDBCLK;
```

  - O que isso faz na prática:
    - Seta o Module Mode como Enable (0x2) → ativa o GPIO1.
    - Seta o Optional Functional Clock para Enable no GPIO1.
    - (Nota: aqui você está somando o offset diretamente, o que tecnicamente não é o mais exato, mas para este propósito funciona.)
  - Segundo o manual:
    - O correto para ativar o clock seria:
      - Escrever 0x2 no campo MODULEMODE do registrador CM_PER_GPIO1_CLKCTRL.
      - Esperar IDLEST == 0 para garantir que o módulo saiu do estado de idle.

2 Configuração da Direção do Pino

Referência: **AM335x TRM, capítulo 25 - GPIO**.
Cada GPIO tem um registrador chamado **OE (Output Enable)**:

- Se um bit do `OE` é 1, o pino é **entrada**.
- Se um bit do `OE` é 0, o pino é **saída**.

No código:

```
#define GPIO_1_EO (*(volatile unsigned int *)0x4804C134)
```

`GPIO_1_EO` aponta para o registrador `GPIO1_OE`.
A operação:

```
valor = GPIO_1_EO;
valor &= ~(1 << 21);
GPIO_1_EO = valor;
```

- Lê o valor atual do registrador `GPIO1_OE`.
- Faz um `AND` com a máscara $\sim$`(1 << 21)`, ou seja, zera o bit 21.
- Zerar o bit 21 configura o pino `GPIO1_21` como **saída**.

3 Piscar o LED (Ligar e Desligar GPIO1_21)

Referência: **AM335x TRM, capítulo 25.4.2 - Data Output Control**.
No AM335x, para controlar um pino de saída:

- Usamos `SETDATAOUT` para colocar o pino em `HIGH` (1).
- Usamos `CLEARDATAOUT` para colocar o pino em `LOW` (0).

No seu código:

```
#define GPIO_1_SETDATAOUT (*(volatile unsigned int *)0x4804C194)
#define GPIO_1_CLEARDATAOUT (*(volatile unsigned int *)0x4804C190)
```

Durante o `while(1)`:

```
pisca ^= (0x01u);   // Inverte o valor de pisca (0 -> 1 -> 0 -> 1
   ...)

if (pisca){
        GPIO_1_SETDATAOUT = (1 << 21);  // Liga o LED (coloca o pino
           em HIGH)
}
else{
        GPIO_1_CLEARDATAOUT = (1 << 21); // Desliga o LED (coloca o
           pino em LOW)
}
```

Toda vez que `pisca` troca, o pino 21 do GPIO1 é setado ou limpo.

4 Delay por Software

```
for (i = 0; i < 1000000; i++);
```

- Esse laço vazio serve para criar uma pausa artificial entre ligar e desligar o LED.
- Isso é comum em bare-metal quando ainda não se configurou um timer.

5 Código Pisca LED

main.c

```
// Definindo a ativacao do clock
#define CM_PER (*(volatile unsigned int *)0x44E00000)
#define CM_PER_GPIO1_CLKCTRL                                (0xACu)
#define CM_PER_GPIO1_CLKCTRL_MODULEMODE_ENABLE              (0x2u)
#define CM_PER_GPIO1_CLKCTRL_OPTFCLKEN_GPIO_1_GDBCLK        (0
   x40000u)

// Definicao da direcao do pino
#define GPIO_1_EO (*(volatile unsigned int *)0x4804C134)
#define GPIO_1_CLEARDATAOUT (*(volatile unsigned int *)0
   x4804C190)
#define GPIO_1_SETDATAOUT (*(volatile unsigned int *)0x4804C194)

int _main(void)
{
        unsigned char pisca = 0;
        unsigned int valor;
        volatile unsigned int i;

        // Definindo a ativacao do clock
        CM_PER |= CM_PER_GPIO1_CLKCTRL |
           CM_PER_GPIO1_CLKCTRL_MODULEMODE_ENABLE |
        CM_PER_GPIO1_CLKCTRL_OPTFCLKEN_GPIO_1_GDBCLK;

        valor = GPIO_1_EO;
        valor &= ~(1 << 21);
        GPIO_1_EO = valor;

        while (1)
        {
                pisca ^= (0x01u);
                if (pisca)
                {
                        GPIO_1_SETDATAOUT = (1 << 21);
                }
                else
                {
                        GPIO_1_CLEARDATAOUT = (1 << 21);
                }
                // delay
                for (i = 0; i < 1000000; i++)
                ;
        }

        return (0);
}
```

6 Como funciona o mapeamento de memoria memmap.ld

Este arquivo é um Linker Script usado para controlar como e onde o programa será posicionado na memória da BeagleBone Black (BBB) no ambiente bare-metal.

6.1 Diretiva MEMORY

```
MEMORY{
        ram : ORIGIN = 0x80000000, LENGTH = 0x1B400
}
```

O que significa: define uma área de memória chamada `ram`.
Essa área começa no endereço `0x80000000`, que na BBB (AM335x) é o início da RAM.
O tamanho reservado é `0x1B400` bytes (ou 111.104 bytes ≈ 108 KB).
Detalhes:

Campo	Valor	Significado
ORIGIN	0x80000000	Endereço inicial onde o programa será carregado.
LENGTH	0x1B400 bytes	Tamanho máximo disponível para o programa.

6.2 Diretiva SECTIONS

```
SECTIONS{
        .text : { *(.text*) } > ram
        .data : { *(.data*) } > ram
        .bss  : { *(.bss*)  } > ram
}
```

O significado é: organizar onde cada seção do programa será colocada na memória.
Seções principais:

Seção	Descrição	Ação no script
.text	Código executável (funções e instruções compiladas).	Todos os símbolos `.text*` vão para a RAM.
.data	Dados inicializados (variáveis globais ou estáticas que começam com valor).	Todos os símbolos `.data*` vão para a RAM.
.bss	Dados não inicializados (variáveis globais ou estáticas inicializadas como 0).	Todos os símbolos `.bss*` vão para a RAM.

O `*` no `*(.text*)` diz para pegar todas as seções de todos os arquivos que correspondam.

- **Interpretação do Linker:**
  - Ao encontrar `.text`, coloca todo o código no início da `ram`.
  - Depois posiciona `.data` logo em seguida.
  - Depois `.bss`.

6.3 Observação Prática

Se seu programa crescer muito (mais de 108 KB, que é o que `0x1B400` permite), será necessário:

- Ajustar o `LENGTH`.
- Ou mudar a estratégia para carregar partes em flash, bootloader etc.

Você diz ao linker: **Coloque tudo na RAM começando em 0x80000000**.
O programa inteiro (`.text`, `.data`, `.bss`) vai estar dentro da memória RAM da BBB.
Ideal para bare-metal, onde você carrega o programa via TFTP diretamente na RAM e executa.
Para pequenos programas bare-metal (tipo piscar LED, UART simples),

Endereço Inicial	Seção	Conteúdo
0x80000000	.text	Código das funções
Próximo	.data	Variáveis globais inicializadas
Próximo	.bss	Variáveis globais não inicializadas (zeradas)

7 Código memmap.ld

```
MEMORY{
        ram : ORIGIN = 0x80000000, LENGTH = 0x1B400
}

SECTIONS{
        .text : { *(.text*) } > ram
        .data : { *(.data*) } > ram
        .bss : { *(.bss*) } > ram
}
```

8 Explicação do start.s

Este código é escrito em Assembly ARM e é responsável por inicializar o ambiente de execução bare-metal na BeagleBone Black (AM335x).

```
_start:
mrs r0, cpsr
bic r0, r0, #0x1F @ clear mode bits
orr r0, r0, #0x13 @ set SVC mode
//   orr r0, r0, #0xC0 @ disable FIQ and IRQ
msr cpsr, r0

ldr sp, =0x4030CDFC  @6kB public stack  TMR 26.1.3.2

bl _main

.loop: b .loop
```

8.1 Descrição Linha a Linha

- `_start:` Define o ponto de entrada do programa.
- `mrs r0, cpsr` Lê o registrador CPSR (Current Program Status Register) para r0.
- `bic r0, r0, #0x1F` Limpa os bits de modo do CPSR para preparar a seleção de um novo modo.
- `orr r0, r0, #0x13` Define o modo Supervisor (SVC) no CPSR, permitindo privilégios totais.
- `// orr r0, r0, #0xC0` Comentado. Se usado, desabilitaria FIQ e IRQ.
- `msr cpsr, r0` Atualiza o CPSR com as novas configurações definidas em r0.
- `ldr sp, =0x4030CDFC` Inicializa o Stack Pointer (sp) com um endereço seguro na SRAM pública (6KB), conforme o manual AM335x (seção 26.1.3.2).
- `bl _main` Salta para a função principal `_main` escrita em C.
- `.loop: b .loop` Se `_main` retornar, entra em um loop infinito para evitar comportamento indefinido.
- O código configura o processador para Supervisor Mode.
- Inicializa a pilha para execução de funções C.
- Salta para `_main()` para iniciar o programa principal.
- Em caso de retorno, trava no loop infinito.

Este procedimento é fundamental para garantir a correta execução de programas bare-metal na BBB, onde não há sistema operacional para gerenciar modos de operação ou pilha.