Exemplos para o Tiva Launchpad usando CMSIS

Índice

Introdução

Cortex Microcontroller Software Interface Standard (CMSIS) é uma iniciativa da ARM de padronizar o acesso ao hardware dos microcontroladores baseados no ARM Cortex-M. Existem algumas dezenas de fabricantes deste microcontroladores e cada um, usa uma sistemática diferente (em arquivos cabeçalho) para acessar o hardware. A migração de um software para um microcontrolador para outro de outro fabricante se revelou muito dificil.

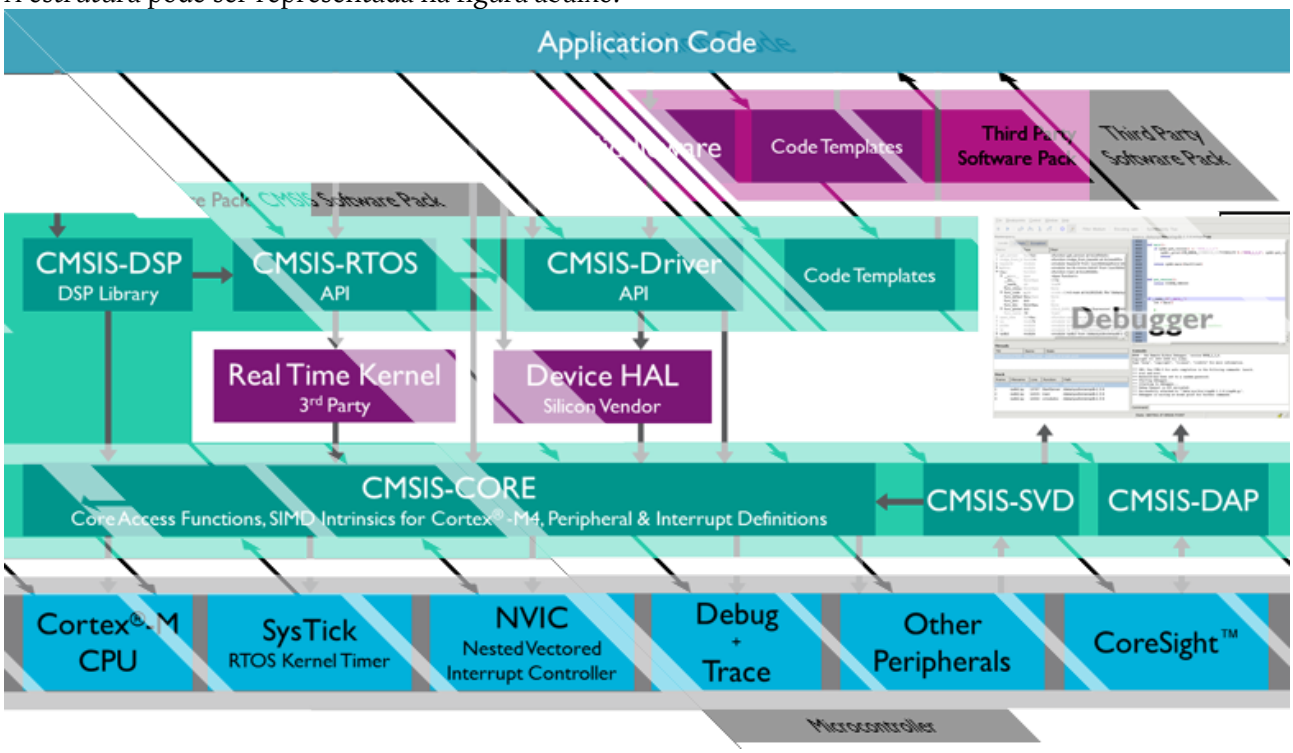
O CMSIS padroniza:
1 – A estrutura do código fonte;
2 – O modo de acessar os registradores;
3 – O conteúdo de alguns arquivos fonte com codigo comum a várias aplicações;
4 – O conteúdo dos arquivos cabeçalho;

Além disso, provê, entre outros, suporte para uso de instruções DSP através de uma biblioteca e uma abstração para usar um sistema de tempo real.

Os componentes do CMSIS são:

CMSIS-CORE	API para processador e periféricos
CMSIS-Driver	Interfaces genéricas para periféricos
CMSIS-DSP	Biblioteca com rotinas para processamento de sinais usando instruções específicas
CMSIS-RTOS-API	API Padrão para RTOS
CMSIS-Pack	Uma maneira padrão para gerar arquivos cabeçalho, etc.
CMSIS-SVD	Descreve os periféricos de um microcontrolador de uma forma padrão.
CMSIS-DAP	Firmware padrão para depuração.

A estrutura pode ser representada na figura abaixo.

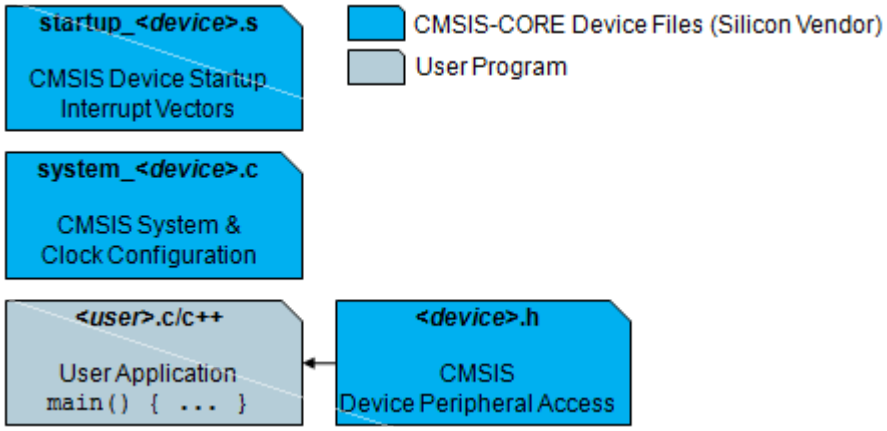


Um projeto deve conter os arquivos específicos da aplicação, entre os quais, o main.c., e os seguintes arquivos fornecidos pelo fabricante do microcontrolador ou da ferramenta (Figura 2) ,alé

startup_DEVICE.[cs]	código de inicialização do processador, inclusive vetor de interrupção
system_DEVICE.c	rotinas SystemInit, SystemCoreUpdate e a variável global SystemCoreClock.

system_DEVICE.h	Arquivo cabeçalho para acesso as rotinas do system_DEVICE.c
DEVICE.h	Arquivo cabeçalho para o processador.

Todos estes arquivos, exceto o DEVICE .h, devem ficar na pasta do aplicativo, pois pode ser necessários adaptá-las. O arquivo DEVICE.h é usado em praticamente todos os arquivos fonte, e ele além das definições de registradores e campos, inclui cabeçalhos padrão fornecidos pela ARM. Para o Cortex M4, eles são ARMCM4.h, core_cm4.h, arm_math.h, core_cm4_simd.h, core_cmFunc.h e core_cmInstr.h, entre outros.



Existem diversas versões do CMSIS (a atual é 4.3).

Uma das características do CMSIS é o uso de estruturas (struct) para acessar periféricos. Por exemplo, para acessar os registradores do SysTick é definida uma struct como abaixo.

```
typedef struct
{
  uint32_t CTRL;      /* Offset: 0 (R/W)  SysTick Control and Status Register */
  uint32_t LOAD;      /* Offset: 4 (R/W)  SysTick Reload Value Register       */
  uint32_t VAL;       /* Offset: 8 (R/W)  SysTick Current Value Register      */
  uint32_t CALIB;     /* Offset: C (R/ )  SysTick Calibration Register        */
} SysTick_Type;
```

e um simbolo do preprocessador

```
#define SCS_BASE      (0xE000E000UL) /*!< System Control Space Base Address  */
#define SysTick_BASE  (SCS_BASE +  0x0010UL) /*!< SysTick Base Address      */
#define SysTick       ((SysTick_Type *) SysTick_BASE)
```

Assim é possível se acessar o registrador VAL do SysTick usando

```
SysTick->VAL
```

Isto é bastante diferente das bibliotecas antigas, como por exemplo, as fornecidas pela Texas, que definem diversos símbolos, um para cada registrador.

```
#define NVIC_ST_CTRL_R          (*((volatile uint32_t *)0xE000E010))
#define NVIC_ST_RELOAD_R        (*((volatile uint32_t *)0xE000E014))
#define NVIC_ST_CURRENT_R       (*((volatile uint32_t *)0xE000E018))
```

O arquivo startup_DEVICE.c contém o vetor de interrupções, rotinas default para tratamento de interrupções e a rotina Reset_Handler, que tem o código que é executado após um reset ou energização do processador. O vetor de interrupções usa mecanismos do gcc, que permite definir um simbolo fraco, ou seja, que pode ser redefinido, e valores default para símbolos. Assim, o vetor de interrupções já está montado apontado para rotinas com nomes já definidos. Se no projeto, em qualquer outro arquivo fonte, for necessário implementar uma rotina de interrupção, basta definir uma rotina com o nome correspondente. Por exemplo, no vetor de interrupções na posição 15 contém o nome SysTick_Handler,

que está definido como weak e, na ausência de definição, aponta para Default_Handler. Se uma rotina com teste nome for definida, a substituição é automática, sem a necessidade de modificação do código neste arquivo.

O vetor de interrupções deve ser armazenado na posição 0 da memória Flash. O endereço 0 deve conter o valor do apontador de pilha (SP) e o endereço 4, o do contador de programa (PC) após o reset. Para isto, foi usada uma extensão não padrão do C que permite dizer em qual seção um código ou dado será armazenado. O atributo section(".nvictable") informa ao compilador que a tabela será armazenada na seção nvictable. Durante a ligação, o ligador é instruído para montar esta seção logo no inicio da memória flash.

Na rotina Reset_Handler:

- É feita a inicialização da área de dados inicializados (seção DATA). Para isto, são copiados os valores iniciais gravados em ROM para a RAM.
- A área de dados não inicializada é zerada.
- É chamada a rotina SystemInit, para executar o mais rápido possível a inicialização do sistema
- É chamada a rotina _main, que é usada para inicialização da alguma biblioteca usada
- Finalmente, é chamada a rotina main, que é a rotina principal da aplicação.

A rotina Default_Handler simplesmente trava o microcontrolador num ciclo eterno.

O arquivo system_DEVICE.c contém uma implementação do SystemInit, que permite a inicialização de periféricos antes mesmo da execução do código na rotina main. Contém também uma variável global chamada SystemCoreClock, que contem o valor da frequencia em MHz do relógio do processador e a rotina SystemCoreClockUpdate, que deve ser chamada toda vez que o relógio do processador é modificado para atualizar a variável SystemCoreClock. Este arquivo pode conter outras rotinas e variáveis específicas para uma implementação ou dispositivo.

O arquivo tm4c123.ld contém as instruções para o ligador construir o arquivo executável. Inicialmente são definidos os tamanhos das diferentes memórias. Depois são mostradas as regras que definem qual seção de código será armazenado em qual memória e em qual ordem.

O arquivo Makefile define uma série de ações para gerar o código, gravar no microcontrolador automaticamente cuidando das dependências.

make all	gera código objeto e a imagem da memória
make flash	grava no microcontrolador
make edit	abre os arquivos em um editor
make debug	inicia uma sessão de depuração
make openocd	ativa um programa que interfacea o depurador com o microcontrolador na placa
make size	mostra tamanho do código
make nm	lista simbolos
make dump	disassembla o programado

A Texas Instruments fornece arquivos cabeçalho no padrão CMSIS[1] somente para versões muito antigas do processador[2]. Existe na web um repositório com arquivos cabeçalho para os processadores mais modernos[3]. Outra alternativa é usar um pacote postado em um fórum da Texas[4].

1 http://www.ti.com/tool/cmsis_device_headers

2 Stellaris era uma linha de microcontroladores fabricada pela Luminary Micro (que foi comprada pela Texas em 2009), foi discontinuada em 2013 e deu origem a linha Tiva.

3 github.com/speters/CMSIS

4 http://e2e.ti.com/cfs-file.ashx/__key/communityserver-discussions-components-files/908/4212.svd_2D00_tiva_2D00_11073.zip.

Ferramentas de software

São necessárias duas ferramentas de desenvolvimento para a placa Tiva TM4C123GH6PM Launchpad:

- Compilador. A Texas disponibiliza uma versão gratuita mas limitada chamada Code Composer. A ARM mantêm uma distribuição especifica do Gnu C Compiler com licença de código aberto[5]. Em ambos os casos há versões para Windows e Linux.
- Gravador. O fabricante disponibiliza a ferramenta Uniflash[6] para Windows e Linux e uma ferramenta mais antiga para Windows chamada Flash Programmer[7]. Para Linux existe ainda o software de código aberto lm4flash[8]. Pode ser usando ainda o OpenOCD para gravar.

Para a edição, controle do processo de compilação e depuração pode ser usado:

- um ambiente integrado de desenvolvimento como o Eclipse[9] ou o do Code Composer ou
- ferramentas separadas como editores (existem vários) e automatizadores de compilação como o make.

Os projetos a seguir foram feitos usando uma plataforma Linux com o compilador gcc-arm-embedded e o GNU make padrão para Linux.

Para o depurador foi usado o gdb

5 http://launchpad.net/gcc-arm-embedded
6 http://www.ti.com/tool/uniflash
7 http://www.ti.com/tool/lmflashprogrammer
8 http://github.com/utzig/lm4tools
9 http://www.eclipse.org

Exemplo 1 – Blink muito simples

O programa abaixo mostra como se fazer piscar os LEDs em sequencia acessando-se diretamente os registradores conforme definidos no padrão CMSIS. O simbolo do preprocessador USE_AHB defini qual o tipo de barramento usado para acessar o módulo GPIO. Quando se usa o barramento AHB (Advanced High-Performance Bus) o acesso é feito em um ciclo de relógio, ou seja, sem espera. No caso do APB (Advanced Peripheral Bus) são necessários dois ciclos de relógio para um acesso. A diferença é o endereço base dos registradores do GPIO (definidos pelos símbolos GPIOF_AHB ou GPIOF). Não se deve misturar os dois tipos de acesso!

Para se evitar constantes numéricas difíceis de entender em uma primeira leitura, foi definida uma macro BIT, que gera uma expressão com o valor correspondente ao inteiro com o bit especificado igual a 1 e todos os outros iguais a zero.

```
#define BIT(N)       (1U<<(N))
```

Desta maneira pode-se definir as constantes para cada LED e também para o conjunto dos LEDs.

```
#define LED_RED      BIT(1)
#define LED_BLUE     BIT(2)
#define LED_GREEN    BIT(3)
#define LED_ALL     (LED_RED|LED_BLUE|LED_GREEN)
```

Uma rotina simples é usada para fazer passar o tempo que o LED deve ficar aceso. A constante abaixo foi determinada por tentativa e erro, e deve ser ajustada caso a frequencia do relógio do processador seja alterada.

```
#define INTERVAL     800000

void xdelay(unsigned volatile v) {

    while( --v ) {}
}



int main(void) {
GPIOA_Type *gpio;

#ifdef USE_AHB
    gpio = GPIOF_AHB;
    /* Enable GPIO access using AHB */
    SYSCTL->GPIOHBCTL |= BIT(5);
#else
    gpio = GPIOF;
#endif

    /* Enable clock for Port F */
    SYSCTL->RCGCGPIO  |= BIT(5);

    /* Pins for led are digital output */
    gpio->DIR    = LED_RED|LED_GREEN|LED_BLUE;
    gpio->DEN    = LED_RED|LED_GREEN|LED_BLUE;
```

O ciclo principal é simples. Apaga-se todos os LEDs (se já estiver apagado, não faz diferença) e acende-se o LED desejado.

```
    while(1)
    {
        gpio->DATA &= ~LED_ALL;
        gpio->DATA |= LED_ALL;
        xdelay(INTERVAL);

        gpio->DATA &= ~ LED_ALL
        gpio->DATA |= LED_RED;
        xdelay(INTERVAL);

        gpio->DATA &= ~LED_ALL;
        gpio->DATA |= LED_GREEN;
        xdelay(INTERVAL);

        gpio->DATA &= ~LED_ALL;
        gpio->DATA |= LED_BLUE;
        xdelay(INTERVAL);

        gpio->DATA &= ~LED_ALL;
        xdelay(INTERVAL);
    }
    return 0; /* NEVER */
}
```

Exemplo 2 – Blink com abstração para GPIO

Neste exemplo, foi usada uma abstração para a Porta F do GPIO. Foram definidas as rotinas GPIO_Init e GPIO_Write. Todo o acesso a porta é feita através destas rotinas.

O programa principal fica então bem mais legível (mas ainda pode melhorar).

```
int main(void) {

    GPIO_Init(LED_ALL);

    while(1) {
        /* White */
        GPIO_WritePin(LED_ALL, LED_ALL);
        Delay(1000);

        /* Red */
        GPIO_WritePin(LED_ALL, LED_RED);
        Delay(1000);

        /* Green */
        GPIO_WritePin(LED_ALL, LED_GREEN);
        Delay(1000);

        /* Blue */
        GPIO_WritePin(LED_ALL, LED_BLUE);
        Delay(1000);

        /* Off */
        GPIO_WritePin(LED_ALL, LED_NONE);
        Delay(1000);
    }
    return 0;
}
```

A rotina GPIO_Init precisa saber quais pinos são de entrada e quais são de saída. Nesta versão, um parâmetro fornece a informação sobre quais pinos serão de saída e somente estes são configurados. A implementação é idêntica a do exemplo anterior. Quanto a rotina GPIO_Write ,existem diferentes abordagens para a sua interface. Uma seria o parâmetro especificar o valor que os pinos teriam que assumir. Mas neste caso, em muitas aplicações, o programa principal teria que guardar a informação sobre quais os valores atuais dos pinos. A abordagem usada é a de se especificar quais os pinos que serão zerados e quais que serão setados. No caso de se especificar o mesmo pino para as duas ações, o pino será setado.

```
static inline void
GPIO_WritePin(uint32_t zeroes, uint32_t ones) {
GPIOA_Type *gpio;

#ifdef USE_AHB
    gpio = GPIOF_AHB;
    SYSCTL->GPIOHBCTL |= BIT(5);
#else
    gpio = GPIOF;
#endif

    gpio->DATA = (gpio->DATA&-zeroes)|ones;
}
```

No código acima, a rotina foi especificada como inline (e também como estática, isto é, com alcance local). Assim, o tamanho do programa será muito próximo ao do exemplo anterior.

Exemplo 3 – Blick com *bit banding*

Este exemplo é identico ao anterior, mas se usará o *bit-banding*., que é um mecanismo que permite a modificação da saída usando apenas um ciclo.

No exemplo anterior, a modificação dos pinos é feita através da instrução.

```
    gpio->DATA = (gpio->DATA&~zeroes)|ones;
```

Isto significa que há um acesso de leitura e outro de escrita. O mecanismo de bit-banding possibilita a modificação de um registrador, setando-se alguns bits e apagando-se outros, em apenas uma operação de escrita. Para isto, a informação sobre quais bits serão afetados em uma operação é codificada em uma máscara nos bits 9 a 2 do endereço. Os bits 0 e 1 do endereço devem ser sempre zero para garantir o alinhamento.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Endereço base																						Máscara								00	

Um bit 0 na posição N da mascara impedirá que o bit N da porta seja modificado. Onde o bit for 1, o bit correspondente da porta será modificado de acordo com o dado armazenado.

```
uint32_t *base = base = (uint32_t *) GPIOF_AHB_BASE;
...
    *(base+(zeroes|ones)) = ones;
```

Não é necessário acrescentar os zeros finais pois o ponteiro é para um dado de 32 bits. A mascara é um OU entre os bits a serem apagados ou setados (ou seja, todos os bits envolvidos). O dado a ser armazenado tem bits 1 nas posições onde os bits que serão setados. Os outros serão apagados.

Exemplo 4 – Blink usando SysTick

A grande diferença deste código em relação aos anteriores é a maneira de se gerar os atrasos. Neste se usará o temporizador SysTick para gerar uma interrupção a cada 1 ms. Isto é feito chamando-se a rotina SysTickConfig com o valor da frequência de relógio dividido por 1000. A cada interrupção é incrementada a variável tick.

```
volatile uint32_t tick = 0;

void SysTick_Handler(void) {
    tick++;
}
```

O atraso é controlado guardando-se o valor da variável tick no inicio da contagem de tempo, e então lendo-se a variável tick e calculando-se se a diferença de tempo.

```
void Delay(uint32_t delay) {
uint32_t begin = tick;

    while( (tick-begin) < delay ) {}
}
```

Para se garantir que o compilador não otimize o acesso a memória necessário para o acesso a variável tick, esta é definida como volatile. Neste caso, a cada referencia é buscado o valor na memória.

O resultado é que a contagem de tempo é muito mais precisa. Mas é necessário se ter o valor da frequencia do relógio. O CMSIS especifica que este valor deve estar na variável global SystemCoreClock e que, sempre que for modificada a configuração do relógio, deve ser chamada a rotina SystemCoreClockUpdate para atualizar esta informação. A implementação (ainda incompleta) está em system_tm4c123.c.

Exemplo 5 – Blink com máquina de estados

O uso do conceito de máquinas de estado simplifica a implementação de sistemas embarcados complexos. Neste exemplo, todo o processamento é feito como parte do processo de tratamento da interrupção gerada pelo temporizador SysTick a cada 1 ms.

Na rotina SysTick_Handler, a cada 1000 interrupções é chamada a rotina StateMachine.

```
void SysTick_Handler(void) {
    if( tick == 0 ) {
        StateMachine();
        tick = 1000;
    }
    tick--;
}
```

A rotina StateMachine tem uma variável state, que deve ter seu valor preservado entre os chamados e por isso, é declarada static. A implementação é padrão para máquinas de estado, onde para cada estado, é especificada uma ação e também, qual o próximo estado.

```
void StateMachine(void) {
static int state = 0;

    switch(state) {
    case 0:
        GPIO_WritePin(LED_ALL, LED_RED);
        state = 1;
        break;
    case 1:
        GPIO_WritePin(LED_ALL, LED_GREEN);
        state = 2;
        break;
    case 2:
        GPIO_WritePin(LED_ALL, LED_BLUE);
        state = 0;
        break;
    }
}
```

Exemplo 6 – Blink com API para GPIO genérica

Este exemplo apresenta uma camada de abstração para a GPIO mais genérica. Nos exemplos anteriores somente era usada a Porta F e não havia operações de leitura.

Neste caso, pode-se especificar qual a porta que quer inicializar e quais os pinos são de entrada ou de saída.

```
GPIOA_Type *
GPIO_Init(uint32_t gpiobaseaddr,uint32_t inputbits, uint32_t outputbits) {
GPIOA_Type *gpio = (GPIOA_Type *) gpiobaseaddr;

    // Search in table info about GPIO
    struct gpioinfo *p = gpiotab;

    while( 1 ) {
        if( ! p->address )
            return 0;
        if( p->address == gpiobaseaddr )
            break;
        p++;
    }

    if( p->address == 0 )
        return 0;                   // Not found in table

    // If AHB address specified (AHBBIT is set), AHB access must be enabled
    if( p->bitmask & AHBBIT ) {
        SYSCTL->GPIOHBCTL |= p->bitmask&0x0FF;
    }

    SYSCTL->RCGCGPIO  |= p->bitmask&0x0FF;  // Enable clock for Port F

    // Only necessary for reconfiguration
    gpio->LOCK   = 0x4C4F434BU;     // Unlock with preprogrammed password
    gpio->AMSEL  = 0x00U;           // Disable analog functions
    gpio->PCTL   = 0x00U;           // GPIO
    gpio->AFSEL  = 0x00U;           // GPIO

    // Configure digital I/O and direction
    gpio->DIR    = outputbits;
    gpio->DEN    = outputbits|inputbits;

    xdelay(10);

    return gpio;
}
```

A rotina retorna o endereço para acesso ao módulo ou zero, se não conseguir identificar o módulo. O arquivo cabeçalho define os simbolos GPIOx_BASE e GPIOx_AHB_BASE, que serão usados para especificar o módulo.

Existem duas implementações para o GPIO_WritePort (o nome foi modificado para consistencia com a rotina de leitura). Uma usando funções inline como abaixo.

```
static inline void
GPIO_WritePort(GPIOA_Type *gpio, uint32_t zeroes, uint32_t ones) {
    gpio->DATA = (gpio->DATA&(~zeroes))|ones;
}
```

Como inline é uma sugestão para o compilador, a outra implementação usa macros, que como trabalha com substituição de texto, garante a inserção do código no ponto.

```
#define GPIO_WritePort(GPIO,ZEROES,ONES) do { \
             (GPIO)->DATA = (gpio->DATA&(~ZEROES))|ONES; \
              } while(0);
```

O uso do do {....} while(0) garante que a substituição funcione mesmo em um if como abaixo.

```
      If( condicao )
            GPIO_WritePort(gpio,LED_ALL,LED_RED);
      else
            GPIO_WritePort(gpio,LED_ALL,LED_RED);
```

Se tivesse sido delimitado por chaves o código resultado estaria errado.

Finalmente, a implementação da rotina GPIO_Init foi separada. Assim são possíveis o teste separado e o reuso de código já estado. A implementação está no arquivo gpio.c. A interface, no arquivo gpio.h. Como se deseja fazer com que o código das funções GPIO_WritePort e GPIO_ReadPort seja inserido no ponto de chamado, este código deve estar definidas no arquivo cabeçalho.

Exemplo 7 – Blink usando abstração para LEDs

Uma outra abstração é possível. Neste caso, o foco são os LEDs, que passam a ser controlados por duas rotinas:

```
LED_Init
LED_Write
```

Em muitos casos, uma camadad de abstração pode ser construída usando outras. Neste caso, poderia ter sido usado a implementação do exemplo anterior constituída do gpio.c e gpio.h. Mas considerando a simplicidade e o aumento do overhead causado por um nível adicional de abstração, tanto LED_Init como LED_Write são implementados acessando diretamente os registradores.

LED_Init faz toda a inicialização e é muito semelhante a GPIO_Init dos exemplos anteriores.

```
void LED_Init(void) {
#ifdef AHB
GPIOA_Type *gpio = GPIOF_AHB;
#else
GPIOA_Type *gpio = GPIOF;
#endif

#ifdef AHB
    SYSCTL->GPIOHBCTL |= 0x20;
#endif

    SYSCTL->RCGCGPIO  |= 0x20;        /* Enable clock for Port F */

    /* Pins for led are digital output */
    gpio->DIR    |= LED_ALL;
    gpio->DEN    |= LED_ALL;

    xdelay(10);

}
```

De forma idêntica, foi implementada a rotina LED_Write.

```
void LED_Write(uint32_t zeroes, uint32_t ones) {
#ifdef AHB
GPIOA_Type *gpio = GPIOF_AHB;
#else
GPIOA_Type *gpio = GPIOF;
#endif

    gpio->DATA = (gpio->DATA&(~zeroes))|ones;

}
```

A máquina de estados é implementada de forma similar.

```
void StateMachine(void) {
static int state = 0;

    switch(state) {
    case 0:
```

```
        LED_Write(LED_ALL, LED_RED);
        state = 1;
        break;
    case 1:
        LED_Write(LED_ALL, LED_GREEN);
        state = 2;
        break;
    case 2:
        LED_Write(LED_ALL, LED_BLUE);
        state = 0;
        break;
    }

}
```

Ambas rotinas foram implementadas em um arquivo led.c, possibilitando assim o seu reuso. A interface está definida no arquivo led.h, que define não só as rotinas como também simbolos para os LEDs. Deve ser observado que o macro BIT foi renomeada LEDBIT para evitar conflitos.

Exemplo 8 – Usando UART

- Uma UART, interface para comunicação serial assíncrona, é muito útil para a comunicação simples com outros equipamentos. Ao contrario de uma interface USB que é complexa e exige hardware especifico, uma UART pode até mesmo ser emulada por software (*bit banging*). Computadores PC até recentemente tinham portas serial com conectores DB-9. Embora este conector tenha 9 pinos, somente três (um deles, o terra) é necessário para uma comunicação bidirecional de até 115 Kbps. O padrão RS-232 especifica forma do sinal e os níveis de tensão, mas não o conector. Devido ao fato do padrão ser muito antigo, os níveis nominais de tensão usados são -12 V para o sinal 1 e +12 para o sinal 0. É necessário um circuito para conversão de níveis de tensão entre o microcontrolador e a interface externa. No entanto, é cada vez mais comum, a comunicação serial ser feita diretamente usando níveis lógicos compatíveis com TTL (5 ou 3 V).
- Uma interface serial assíncrona também é útil para a depuração do sistema. Na placa Tiva Launchpad existe uma comunicação serial assíncrona entre o processador alvo e processador com suporte para depuração. Esta canal de comunicação funciona a 115200 Kbps, com caracteres de 8 bits e sem paridade. O processador de depuração implementa uma interface USB e usando ela, uma interface serial virtual (Windows: COMx, Linux: /dev/ttyACMx).
- Uma interface para se usar a UART apenas com saída pode ter as rotinas, como está definido no arquivo uart.h.

```
void UART_Init(void)
void UART_SendChar(char ch)
void UART_SendString(char *p )
void UART_SendEOL(void)
```

- A implementação em uart.c é simples. Infelizmente os arquivos cabeçalhos CMSIS achados em github.com/speters/CMSIS contém somente as informações sobre registradores e os endereços dos periféricos. Ao contrario dos arquivos fornecidos por outros fabricantes, não há simbolos para os os campos nem para seus valores.
- As alternativas são:
- Usar expressões do tipo UART0->CTL &= ~0x00000100;
- Usar expressões do tipo UART0→CTL &= ~BIT(8);
- Definir simbolo e usar uma expressão do tipo UART0→CTL &= ~UART_CTL_TXE_B
- O arquivo tm4c123-fields.h foi criado a partir da documentação e tem símbolos para os campos e seus valores. É usada um convenção para o nome dos campos e símbolos:
- Nome é composto por MODULO_REGISTRADOR_CAMPO.
- Campo de um bit é definido como MODULO_REGISTRADOR_CAMPO_B

- Campo de mais do que um bit é definido como MODULO_REGISTRADOR_CAMPO_M e seus valores como MODULO_CAMPO_VALUE_V.
- Uma outra maneira de se ter um arquivo com símbolos para os campos seria filtrá-los do arquivo do dispositivo usando no TivaWare[10].
- Também é mostrada (em conv.c) a implementação de rotinas que convertem um inteiro na sua representação decimal em ASCII (itoa ou utoa) ou na sua representação hexadecimal em ASCII (itohex). Nas rotinas que convertem para decimal, é feito uso do operador de divisão (/) e de resto (%), que podem não estar presentes em alguns microcontroladores.
-
-

10 O comando egrep -v "_R |volatile" tm4c123gh6pm.h > tm4c123gh6pm-fields.h gera o arquivo somente com as definições de campo, mas os nomes dos campos são diferentes dos usado neste trabalho.

○ Exemplo 9 – Usando uma mini stdio

A maioria dos programas feitos em C usa um numero relativamente pequeno de rotinas. As principais são:

printf

puts

fgets

getchar

putchar

O pacote de ferramentas para ARM obtido em launchpad.net/gcc-arm-embedded já vem com uma biblioteca C incorporada baseada na newlib[11]. Assim já existem as rotinas acima e muitas outras, mas como usá-las será visto adiante. Para evitar problemas, o Makefile deve ser modificado para não usar a libc do compilador durante a ligação.

O arquivo ministdio.c mostra a implementação das rotinas printf, puts e fputs usando a rotina putchar para saída. A rotina fgets foi implementada usando apenas a rotina getchar e permite um grau limitado de edição da linha. Assim definindo-se apenas as funções getchar e putchar, tem-se as funcionalidades das outras funções[12]. No programa principal em main.c estas estão definidas para usar as rotinas da UART.

```
int getchar(void) { return UART_ReceiveChar(); }
int putchar(int c) { UART_SendChar(c); return c; }
```

A rotina printf é mais complexa e usa o cabeçalho varargs.h para acessar um número variável de parâmetros. Não há suporte para números de ponto flutuante e opções de formatação como largura e preenchimento.

Como as rotinas estão em um arquivo que faz parte de um projeto, todas elas são incorporadas ao arquivo executável. Quando se usam bibliotecas, somente os arquivos que tem símbolos mencionados no projeto são incorporados ao arquivo executável.

11 Ver https://sourceware.org/newlib/

12 A blblioteca newlib usa uma abordagem semelhante, mas o número de rotinas implementadas é muito maior.

Exemplo 10 – Usando Newlib

Para sistemas embarcados, o uso de uma biblioteca padrão C como as usadas em Linux (ou outro sistema operacional) é inviável pelo seu grande tamanho e pelo número de funcionalidades existentes, que não são necessárias.

Existem diversos esforços para enxugar a biblioteca padrão C, existem iniciativas como dietlib, musl,uclibc, entre outros. No entanto, todas elas são para um sistema POSIX como o Linux.

Lib C (fread, fwrite)
POSIX API (read,write)
Kernel
Hardware

Para sistemas embarcados, existe a biblioteca newlib. Ela apresenta um conjunto (pequeno) das funcionalidades de uma biblioteca padrão C, mas suficiente para sistemas embarcados. Por exemplo, estão incorporadas a newlib rotinas para manipulação de cadeia de caracteres (strcpy, strcat, strlen, etc.), aritméticas (sin, cos, sqrt, etc) entre outras. Não fazem parte rotinas como

Lib C (fread, fwrite)
syscall
Hardware

A implementação faz uso de um conjunto de funções, que no exemplo, está no arquivo syscall.c. As principais são _read e _write. Observe que em bibliotecas C para Linux, estas rotinas têm os nomes read e write, como especificado no POSIX, poluindo o espaço de nomes. O newlib corretamente usa nomes reservados para implementação.

```
int _read(int file, char *ptr, int len) {
int nc = 0;
char c;
int nb;

    while( nc < len ) {
        if( SerialStatus()==0 )
            break;
        c = SerialRead();
        *ptr++ = c;
        nc++;
    }
    return nc;
}

int _write(int file, char *ptr, int len) {
int todo;

    for (todo = 0; todo < len; todo++) {
        SerialWrite(*ptr++);
    }
```

```
    return len;
}
```
As demais rotinas, na maior parte dos sistemas embarcados, somente definem os símbolos ou retornam um código de erro.

```
int _close(int file) {
    return -1;
}
```

Um comentário final é que a interface serial tem que ser inicializada. A melhor posição para isto é na rotina _main, que é chamada antes da main.

```
int _main(void) {
      SerialInit();
}
```

Exemplo 11 – Esquema muito simples de leitura do botão

Neste projeto o botão SW1 será usado para inverter o ciclo de piscamento dos LEDs. A variável sentido é usada para determinar qual o próximo LED a acender. O piscamento é controlado por uma máquina de estado. Assim, a sequencia é determinada pelo estado que deverá seguir ao atual como mostrado abaixo.

```
case 0:
    GPIO_WritePins(LED_ALL, LED_RED);
    Delay(1000);
    if( sentido )
        state = 1;
    else
        state = 2;
    break;
```

No final de cada piscamento, é lido o estado atual dos pinos de entrada com a rotina GPIO_ReadPins(). Se o bit correspondente a chave SW1 for 1, o sentido é invertido.

```
bits = GPIO_ReadPins();
if( (bits&SW1) != 0 ) { // No debounce yet
    sentido = !sentido;
}
```

Há vários problemas neste código. A leitura só é feita ao final do tempo de acendimento. Outro é que não há *debounce*.

Exemplo 12 – Uso do SysTick para temporização e leitura

Este exemplo é bem similar ao anterior. Uma diferença é que o tempo é controlado pelo SysTick. Outra é que o botão é testado a cada 1 ms.

```
void SysTick_Handler(void) {
uint32_t bits;

    tick++;
    bits = GPIO_ReadPins();
    if( (bits&SW1) != 0 ) { // No debounce yet
        sentido = !sentido;
    }

}
```

Deve ser observado que a variável sentido agora é uma variável global. Um protótipo da rotina GPIO_ReadPins teve que ser inserido no código antes da rotina acima, pois a implementação é mostrada depois.

Apresenta também os mesmos problemas de falta de debounce do exemplo anterior.

Exemplo 13 – Uso de callback

Ao invés de testar periodicamente se o botão está pressionado ou não, pode-se configurar o periférico GPIO para gerar uma interrupção toda vez que o nível de tensão correspondente ao botão passar de baixo para alto.

A abstração usada para o GPIO foi estendido com a possibilidade de se especificar uma rotina que será chamada toda vez que o evento subida do sinal for detectado. Esta técnica é determinada callback. Enquanto não for especificado qual a rotina, o apontador gpiocallback é nulo e não será acionada nenhuma rotina.

```
uint32_t
GPIO_EnableInterrupt(uint32_t pins, void (*callback)(uint32_t) ) {
GPIOA_Type *gpio;

#ifdef USE_AHB
    gpio = GPIOF_AHB;
#else
    gpio = GPIOF;
#endif
    gpio->IM = pins;
    intpins  = pins;
    gpiocallback = callback;
}
```

A rotina de callback é simples.

```
void switchmonitor(uint32_t w) {
    sentido = !sentido;
}
```

Como no chamado da rotina GPIO_EnableInterrupt foi especificado o SW_ALL, qualquer dos botões inverterá o sentido do piscamento.

A rotina de interrupção se chama GPIOF_IRQHandler e redefine uma rotina definida como fraca (weak) no arquivo startup_tm4c123.c.

Exemplo 14 – Interface GPIO reusável

Neste projeto, todo a implementação da interface para a GPIO foi transferida para o arquivo gpio.c. Para que as rotinas possam ser usadas, os protótipos das funções foram colocados no arquivo gpio.h

```
#ifndef GPIO_H
#define GPIO_H
/**
 * @file     gpio.h
 * @brief    Header for gpio.c
 * @version  V1.0
 * @date     23/01/2016
 *
 * @note     CMSIS library used
 * @note     Uses bit-banding
 *
 **/

void GPIO_Init(uint32_t outputs, uint32_t inputs);
void GPIO_WritePins(uint32_t zeroes, uint32_t ones);
uint32_t GPIO_ReadPins(void);
uint32_t GPIO_EnableInterrupt(uint32_t pins, void (*callback)(uint32_t) );

#endif
```

No começo da rotina, foi colocado um teste que evita que o arquivo seja incluido mais do que uma vez. Caso isto acontecesse, haveria mensagens de erro devido a redefinição das rotinas. Este tipo de teste e definição de simbolo do preprocessador é muito comum, mas está sendo substuido pelo comando

```
#pragma once
```

Neste caso, o preprocessador mantem uma lista dos arquivos já incluidos e que tem este comando, evitando não só a inclusão, mas também a leitura do arquivo.

Exemplo 15 – Lista de rotinas chamadas na interrupção

É possível e também muito comum, se colocar todo o processamento nas rotinas de interrupção. Neste caso, há uma lista das rotinas a serem chamadas quando da interrupção com as informações sobre o intervalo entre chamadas e quanto falta para a próxima chamada.

```
#define NTASKS 10
typedef struct {
    void        (*func)(void);
    uint32_t    period;
    uint32_t    timecounter;
} Task_t;
static Task_t TaskTable[NTASKS];

static int TaskCount = 0;
```

A rotina TaskAdd insere uma rotina na lista. Quando da interrupção, é verificado o valor do timecounter. Se este for zero, a rotina correspondente é chamada e o timecounter recarregado.

Um problema sério desta implementação é que o tempo de processamento da interrupção pode ser muito grande e haver problemas de temporização.

Exemplo 16 – Usando acionamento por tempo

Exemplo 17 – Usando protothreads

Adam Dunkels apresentou uma maneira de se emular um sistema multitarefas cooperativos, sem que as tarefas aparentassem ser RTC (Run To Completion), mas na verdade, o são.

Geralmente o código para tarefas em sistemas multitarefas como FreeRTOS e uc/os tem o modelo abaixo.

```
Void Task(void) {
      // Inicializacao
      while(1) {
            // processo
            // espera
      }
}
```

Em protothreads, é usada uma técnica baseada no mecanismo de Duff, que foi usado originalmente para acelerar a copia de memória. A codificação convencional para isto em C é mostrada abaixo.

```
send(char *to, char *from, int count) {
int n = count;
      while( n-- > 0 ) {
            *to++ = *from++;
      }
}
```

Na implementação acima, a cada cópia é feito o controle, comparando e decrementando o contador n. Mas usando o fato de que o padrão C não impõe restrições às instruções que podem ser colocadas dentro de um switch e que, se não houver um break, as instruções seguintes são executadas, pode-se ter um controle a cada 8 instruções, acelerando o processo de cópia.

```
send(char *to, char *from, int count) {
int n=(count+7)/8;
      switch(count%8){
       case 0: do{     *to = *from++;
       case 7:         *to = *from++;
       case 6:         *to = *from++;
       case 5:         *to = *from++;
       case 4:         *to = *from++;
       case 3:         *to = *from++;
       case 2:         *to = *from++;
       case 1:         *to = *from++;
                 }while(--n>0);
       }
}
```

Primeiro é calculado quantas vezes o ciclo deve ser repetido, considerando que a cada ciclo, 8 bytes são copiados. Na primeira vez que o switch é executado, o resto da divisão por 8 é o número de bytes copiados. Quando se chega ao while, passa-se a copiar 8 bytes a cada ciclo.

Em protothreads, é feito uso intensivo de MACROS para que uma tarefa tenha a aparência da tarefa Task mostrada acima.

```
PT_THREAD(Blink_Green(struct pt *pt)) {
static uint32_t tstart;

    PT_BEGIN(pt);
    while(1) {
        // processo
        if( blinking )
            LED_Toggle(LED_GREEN);
        else
            LED_Write(0,LED_GREEN);
        // espera
        tstart = msTick;
        PT_WAIT_UNTIL(pt,((msTick-tstart)>=semiperiod_green));
    }
    PT_END(pt);
}
```

Este código após passar pelo preprocessador tem a forma abaixo.

```
char link_Green(struct pt *pt)) {
static uint32_t tstart;

    switch(pt->lc) {
    case 0:
      while(1) {
         // processo
         if( blinking )
            LED_Toggle(LED_GREEN);
         else
            LED_Write(0,LED_GREEN);
         // espera
          tstart = msTick;
         pt->lc = 1;
      case 1:
          if( ! (msTick-tstart)>=semiperiod_green) ) return PT_WAITING;
           pt->lc = 0;
       }
    }
    return PT_ENDED;
}
```

No código acima, pode-se ver que a rotina retorna com o código PT_WAITING ou PT_ENDED.

Também pode-se ver algumas restrições que devem ser obedecidas:

1. Todas as variáveis locais devem ser estáticas, para manterem os valores entre chamados;
2. O uso de *switch* é complicado pelo fato de que pode se emaranhar com o implantado pelo protothread.
3. O otimizador pode reordenar as instruções e atrapalhar o funcionamento do mecanismo.

O ciclo principal é simples.

```
void main(void) {

    LED_Init(LED_ALL);

    Button_Init();
```

```
    PT_INIT(&pt_BlinkGreen);
    PT_INIT(&pt_BlinkRed);
    PT_INIT(&pt_ButtonProc);

    SysTick_Config(SystemCoreClock/1000);   /* 1 ms */

    for (;;) {
        PT_SCHEDULE(Blink_Green(&pt_BlinkGreen));
        PT_SCHEDULE(Blink_Red(&pt_BlinkRed));
        PT_SCHEDULE(ButtonProc(&pt_ButtonProc));
    }
}
```

A detecção da botão pressionado é feito na rotina ButtonProc.

```
static
PT_THREAD(ButtonProc(struct pt *pt)) {
static uint32_t tstart;
uint32_t b;

    PT_BEGIN(pt);
    while(1) {
        PT_WAIT_UNTIL(pt,Button_Read()&SW1);
        blinking = ! blinking;
        tstart = msTick;
        PT_WAIT_UNTIL(pt,((msTick-tstart)>=DEBOUNCE_TIME));
        PT_WAIT_UNTIL(pt,!(Button_Read()&SW1));
    }
    PT_END(pt);
}
```

O acendimento dos LEDs é controlado na rotina BlinkGreen e BlinkRed que tem a mesma estrutura.

```
static
PT_THREAD(Blink_Green(struct pt *pt)) {
static uint32_t tstart;

    PT_BEGIN(pt);
    while(1) {
        if( blinking )
            // static int state = 0
            // if (state == 0) {
            //    LED_Write(LED_GREEN,0);
            // } else {
            //    LED_Write(0,LED_GREEN);
            // }
            //
            LED_Toggle(LED_GREEN);
        else
            LED_Write(0,LED_GREEN);
        tstart = msTick;
        PT_WAIT_UNTIL(pt,((msTick-tstart)>=semiperiod_green));
    }
    PT_END(pt);
}
```

A tećnica Protothreads foi usada para implementar no kernel Contiki bastante usado em Internet of Things. Também foi usada para implementar software para desktops.

Exemplo 18 - Usando o Super Simple Tasker (SST)[13]

Robert Ward em 2003 [11] e Miro Samek e Robert Ward em 2008 [12][14] apresentaram um esquema que permite uma forma restrita de preempção entre tarefas. O conceito de níveis de interrupção foi expandido para tarefas. Assim, uma tarefa de prioridade alta pode interromper uma tarefa de prioridade baixa. Para isto, há uma interrupção periódica (tick), quando se verifica se uma tarefa de prioridade mais alta deve se executada.

Todas as tarefas devem ser do tipo Run To Completion, isto é, retornar, para dar vez a tarefas de prioridade mais baixa. Só é necessária uma pilha, pois tarefas só podem ser executadas quando todas as tarefas de prioridade mais alta foram executadas.

O Super Simple Tasker é a base da implementação de um sistema de implementação de máquinas hierarquicas de estado, QP apresentado no livro de Miro Samek, Practical UML Statecharts in C/C++ [13].

O programa principal é simples como mostrado abaixo.

```
void main(void) {

    LED_Init(LED_ALL);

    Button_Init(&joystick_callback);

    SST_init();

    //SST_task(myTask,myTask_ID, myTask_EQ, myTask_EVQL, SST_SIGNAL_TASKINIT,
0);

    SST_task(Task_Blink_Green, BLINK_GREEN_PRIO, BlinkGreenQueue, QUEUE_SIZE,
             SST_SIGNAL_TASKINIT, 0);
    SST_task(Task_Blink_Red, BLINK_RED_PRIO,     BlinkRedQueue,   QUEUE_SIZE,
             SST_SIGNAL_TASKINIT, 0);
    SST_task(Task_Button, BUTTON_PRIO,           0, 0, SST_SIGNAL_TASKINIT, 0);

    SysTick_Config(SystemCoreClock/1000);   /* 1 ms */


    SST_run();
}
```

A rotina de piscar um LED tem uma estrutura simples. A mensagem SST_SIGNAL_TASKINIT é passada na primeira vez que a rotina é chamada para se fazer inicialização. A rotina deve sempre retornar rapidamente.

```
void Task_Blink_Green(SSTEvent event){
static uint32_t lasttick = 0;

    if(event.sig!=SST_SIGNAL_TASKINIT) {
        if( blinking ) {
            if( msTick > (lasttick+semiperiod_green) ) {
                LED_Toggle(LED_GREEN);
                lasttick = msTick;
            };
        } else {
            LED_Write(0,LED_GREEN);
```

13 A versão para ARM Cortex M pode ser achada em https://github.com/upiitacode/SST_ARM.

14 O artigo e o código (para DOS) podem ser achados em http://www.state-machine.com/doc/articles.html

```
        }
    }

}
```

Mas muita coisa acontece nas rotinas de interrupção. Na rotina de interrupção de tempo SysTick_Handler deve ser verificado se não há tarefas de prioridade mais alta a serem executadas. Para evitar problemas de corrida, um protocolo deve se seguido. No inicio da rotina, deve ser chamada a rotina (na verdade, uma macro) SST_ISR_ENTRY. E no termino do processamento da interrupção, deve ser chamada a rotina (macro) SST_ISR_EXIT.

```
void SysTick_Handler(void) {
int pin;

    SST_ISR_ENTRY(pin,ISR_TICK_PRIO);
//  SST_post(myTask_ID,1,0);
    msTick++;
    SST_post(BLINK_GREEN_PRIO, ISR_TICK_SIG, 0);
    SST_post(BLINK_RED_PRIO, ISR_TICK_SIG, 0);
//    SST_post(BUTTON_PRIO, ISR_TICK_SIG, 0);
    SST_ISR_EXIT(pin,(SCB->ICSR = SCB_ICSR_PENDSVSET_Msk));

}
```

As rotinas de interrupção do joystick devem seguir o mesmo protocolo. Assim elas tem a forma abaixo.

```
void EXTI0_IRQHandler(void) { /* CENTER BUTTON */
int pin;
    SST_ISR_ENTRY(pin,ISR_EXTI0_ID);

    if( (EXTI->IMR1&EXTI_IMR1_IM0) && (EXTI->PR1&EXTI_PR1_PIF0) ) {
        if( callback.CenterButtonPressed )
            callback.CenterButtonPressed();
        EXTI->PR1 |= EXTI_PR1_PIF0;

    };

    SST_ISR_EXIT(pin,(SCB->ICSR = SCB_ICSR_PENDSVSET_Msk));
}
```

A abordagem baseada em SST apresenta algumas vantagens:

1 – Necessita uma única pilha.

2 – Apresenta possibilidade de preempção.

3 – É relativamente simples e pequena.

Exemplo 19 – Usando o FreeRTOS[15]

O FreeRTOS [14] é um núcleo de tempo real preemptivo altamente portável. Como é um sistema preemptivo sem restrições, cada tarefa deve ter uma pilha separada. Basicamente o chaveamento de tarefas corresponde a uma troca de pilha, pois cada pilha tem o endereço de retorno.

O FreeRTOS tem uma API (Application Programming Interface) grande e bem completa. Tem suporte para vários mecanismos de comunicação entre tarefas.

O programa principal tem a forma abaixo. São criadas várias tarefas e dado inicio ao escalonamento com o chamada da rotina vTaskStartScheduler, que em condições normais, nunca retorna.

```
int main(void) {

    LED_Init(LED_ALL);

    ButtonInit();

    xTaskCreate(Task_Blink_Red,"Red", 1000,0,1,0);
    xTaskCreate(Task_Blink_Green,"Green", 1000,0,2,0);

    vTaskStartScheduler();

    while(1) {}   // Just in case
}
```

As tarefas são independentes e apresentam a estrutura clássica para sistemas preemptivos, nunca retornando. Por exemplo, as rotinas Task_Blink_Green (mostrada abaixo) e Task_Blink_Red têm a mesma estrutura.

```
void Task_Blink_Green(void *pvParameters){
const portTickType xFrequency = 500;
portTickType xLastWakeTime=xTaskGetTickCount();

    while(1) {
        if( blinking ) {
            LED_Toggle(LED_GREEN);
            vTaskDelayUntil(&xLastWakeTime,xFrequency);
        } else {
            LED_Write(0,LED_GREEN);
            vTaskDelay(semiperiod_green);
        }
    }
}
```

A rotina Task_Button implementa um esquema de debounce usando os mecanismos de temporização fornecidos pelo FreeRTOS.

```
void Task_Button(void *pvParameters) {

    while(1) {
        if( (Button_Read()&SW1) != 0 ) {
            blinking = ! blinking;
```

15 www.freertos.org

```
            vTaskDelay(DEBOUNCE_TIME);
            while( (Button_Read()&SW1) == 0 ) {
                vTaskDelay(DEBOUNCE_TIME);
            }
        }
    }

}
```

Esta técnica representa uma das formas mais poderosas de construir um sistema de tempo real. Um dos problemas é o custo em termos de uso de memória RAM, pois cada tarefa deve ter sua proproa pilha e ela deve ser dimensionada para o máximo.

Para se verificar a temporização, pode-se usar o Rate Monotonic Analysis (RMA) baseado em Liu e Wayland [15]. Considerando-se que [16] as *n* tarefas são independentes e que o final o período é o limite para a resposta, e sabendo-se de cada tarefa, o período T_i e a carga (tempo de execução do processo) C_i, pode-se calcular a taxa de utilização de CPU de cada processo U_i.

$$U_i = \frac{C_i}{T_i}$$

e a carga total U

$$U = \sum_{i=1}^{n} U_i$$

Se U for maior que 1, é impossível a alocação. Se U for menor que $n\left(2^{\frac{1}{n}} - 1\right)$ a alocação é possível. Entre este valor e 1, deve ser feita uma analise minuciosa (através de simulação) para verificar a viabilidade.

n	$n\left(2^{\frac{1}{n}} - 1\right)$
1	1
2	0,83
3	0,78
4	0.76
5	0,74
10	0,71
100	0,69

Em geral, esta abordagem leva a soluções garantidas, mas muito custosas, pois C_i é considerado para o pior caso e o sistema considera a possibilidade (baixa) de todos as tarefas demandarem o máximo ao mesmo tempo.

Além disso, quando as tarefas são interdependentes, deve ser considerado o problema da inversão de prioridade, quando uma tarefa de baixa prioridade controla um recurso (por exemplo, um semáforo), que passa a ser demandado por uma tarefa de prioridade alta. Como a tarefa tem baixa prioridade, ela não consegue ser escalonada e não consegue liberar o recurso, impedindo a tarefa de prioridade alta de ser escalonada.

Também, existe o problema de ocorrer um *deadlock*, quando se alocam recursos. Duas tarefas necessitam dos recursos A e B. A tarefa 1 solicita na sequencia A e B e a tarefa 2, na sequencia B e A. Se a tarefa 1 for interrompida logo após conseguir o recurso A, pode acontecer da tarefa 2 conseguir o recurso B e solicitar o A, quando então ela entra em espera. Quando a tarefa 1 é escalonada novamente, ela solicita o recurso B, e entra em espera. Nesta configuração, nenhuma das tarefas consegue ser executada.

Referencias

[1] Tiva TM4C123GH6PM Microcontroller Data Sheet. SPM376e

[2] Tiva TM C Series TM4C123G LaunchPad Evaluation Board User's Guide. SMPU296.

[3] Miro Samek, Practical UML Statecharts in C/C++.