Universidade Tecnológica Federal do Paraná
Campus Pato Branco

Desenvolvimento em microcontroladores baseados em processadores ARM Cortex-M4

Leandro Fabian Junior
Callebe Soares Barbosa

orientado por
Gustavo Weber Denardin

2016

Universidade Tecnológica Federal do Paraná
Campus Pato Branco

Leandro Fabian Junior
Callebe Soares Barbosa

orientado por
Gustavo Weber Denardin

Desenvolvimento em microcontroladores baseados em processadores ARM Cortex-M4

Recurso Educacional Aberto produzido com o fomento do Programa de Bolsas para o Desenvolvimento de Recursos Educacionais Abertos (PIBEA) por meio do Programa de Bolsas de Fomento às Ações de Graduação da UTFPR.

2016

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O material aqui contido busca iniciar o leitor no desenvolvimento em plataformas baseadas em processadores com arquitetura ARM.

O Hardware utilizado para os exemplos aqui apresentados é o Kit de avaliação da Texas Instruments Tiva^{TM} C Series TM4C1294.

Capítulo 2

Conhecendo o Processador ARM

2.1 Características ARM

O termo ARM é o acrônimo para *advanced RISC machine* e é o nome dado a uma família de processadores que implementa um conjunto reduzido de instruções, conhecido como RISC (do inglês *reduced instruction set computing*). As instruções da arquitetura RISC levam, aproximadamente, o mesmo período de tempo para serem executadas. Alguns ciclos de clock. Isso permite que as respostas possam ser mais rápidas que em outras arquiteturas com instruções mais complexas (CISC, do inglês *complex instruction set computing*) porém, que demandam de mais tempo de processamento.

Segundo Yiu [**?**] a arquitetura dos processadores Cortex-M3 e Cortex-M4, são ambas implementações da arquitetura ARMv7-M. Existem diferentes tipos de arquitetura ARM para diferentes tipos de processadores, que ainda podem variar conforme são atualizadas ao longo dos anos. Os detalhes da arquitetura ARMv7-M estão documentados no Manual de Referência, disponível no site da ARM Limited.

O Cortex-M3 e Cortex-M4 são essencialmente idênticos em seus aspectos construtivos, de modo que o diagrama de blocos da figura **??** apresenta uma visão geral interna adequada tanto do processador Cortex –M4 quanto -M3.

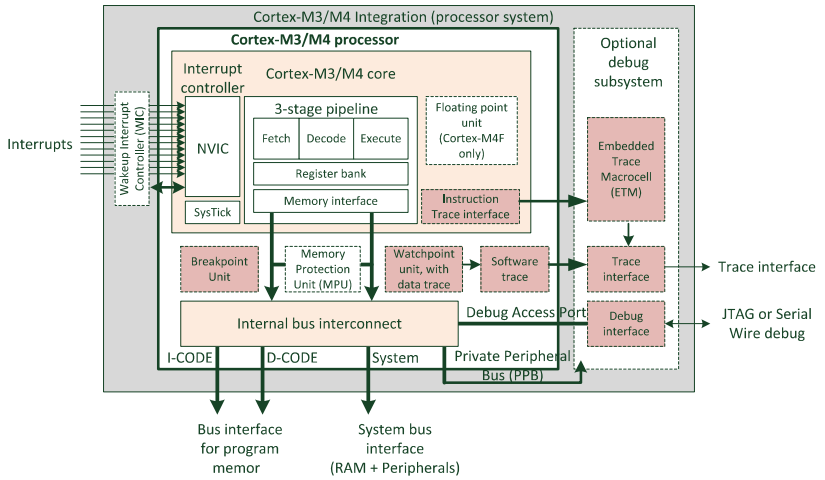

Figura 2.1: Diagrama de Blocos - Processador Cortex-M3 e Cortex-M4 [**?**]

Na figura **??** notamos a presença de elementos no processador como: o controlador de vetores de interrupção, NVIC (*Nested Vectored Interrupt Controller*); o controlador de acionamento de interrupção, WIC (*Wakeup Interrupt Controller*); o temporizador SysTick;

a unidade de proteção de memória MPU (*Memory Protection Unit*); e uma unidade de ponto flutuante presente apenas no Cortex –M4. Existe ainda um sistema de debug dentro do processador para realizar depuração de software e um sistema interno de barramentos para transferência de dados entre o núcleo do processador, o sistema de debug e o MPU.

Os processadores da família Cortex M são de 32 bits, podendo também trabalhar com dados de 8 bits e 16 bits de forma bastante eficiente. Já os processadores Cortex-M3 e Cortex-M4, mesmo sendo da família Cortex M, podem realizar uma série de operações envolvendo dados de 64 bits. Estas operações podem ser realizadas através de um *piperline* de três estágios com uma arquitetura de barramento do tipo *Harvard* permitindo instruções simultâneas de busca e acesso de dados.

Uma das grandes vantagens dos processadores Cortex M é seu baixo consumo de energia. Em especial os processadores Cortex M3 e Cotex M4 podem executar instruções com taxa de $200mA/MHz$ com alimentação de $1,8V$. Estes processadores possuem modos de suspensão que tornam possíveis desativar dispositivos de *Clock* para economizar energia, e um *hardware* adicional para despertar o processador dos modos de suspensão.

Devemos salientar aqui que estamos sempre nos referindo a apenas aos processadores, e que este é uma parte constituinte do microcontrolador. De modo que os demais componentes da placa são desenvolvidos pelos diferentes fabricantes. Assim existem vários tipos de microcontroladores com diferentes características de periféricos e recursos, porém a arquitetura empregada nos processadores é a mesma.

2.2 Modos de operação ARM Cortex-M4

O processador Cortex-M4 possui dois estados de operação, como mostrado na figura **??**, *debug state* e *Thumb state*. O *debug state* ocorre quando o processador é interrompido, por exemplo ao atingir um *breakpoint*, então a execução de instrução é interrompida. Já o *Thumb state* ocorre quando o código do programa está sendo executado. Diferente de outros processadores ARM, o Cortex-M não suporta instruções ARM.

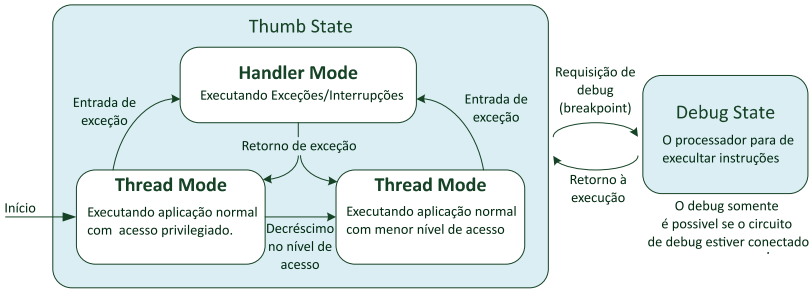

Figura 2.2: Modos de Operação [**?**]

No *Thumb state* ainda existem dois modos de operação, que dizem respeito ao nível de privilégio no acesso ao processador. Ao executar uma rotina de tratamento de interrupção o processador entra em um nível de acesso privilegiado, caracterizando o *handler mode*. Durante a execução de uma aplicação normal o processador pode estar tanto em nível de acesso privilegiado quanto em nível menor, sendo chamado de *thread mode*. Isso é controlado por um registrador específico.

A aplicação pode alterar seu nível de acesso durante o *thread mode*, para um nível menos privilegiado. Porém, para aumentar seu nível de acesso deve haver um mecanismo de exceção/interrupção por parte do processador. Tais mecanismos de controle de nível de acesso garantem uma maior robustez para o sistema, controlando o acesso à regiões críticas de memória.

2.3 Registradores internos

Para um controle melhor e um processamento de dados maior o Cortex-M4 possui registradores internos ao processador agrupados em um conjunto chamado de *banco de registradores*. Cada instrução enviada ao processador especifica a operação a ser executada, os registradores fonte e se necessário os registradores de destino. A arquitetura ARM é baseada no modelo conhecido como *load/store*, ou seja, para processar um conteúdo que esteja na memória é preciso carregá-lo para um registrador interno e então processá-lo. Se necessário, é preciso armazená-lo de volta na memória.

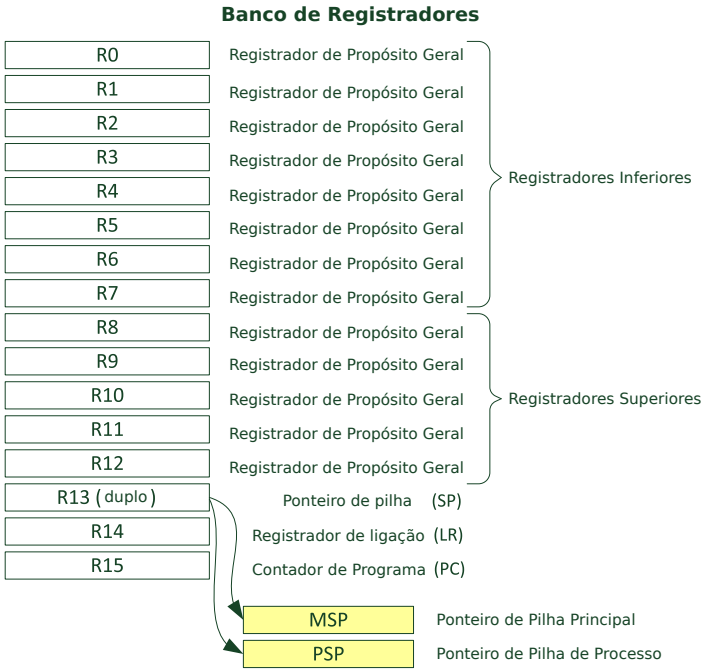

Figura 2.3: Banco de registradores internos [**?**]

O banco de registradores do Cortex-M4 possui 16 registradores de 32 bits, como mostrado na Figura **??**. Cada registrador possui seu propósito, como detalhado a seguir:

R0 - R12: Registradores de Propósito Geral
Devido ao número limitado no conjunto de instruções, muitas das de 16 bits somente acessam os registradores de R0 à R7, chamados de *registradores inferiores*. De R8 à R9, os *registradores altos*, podem ser usados com as instruções de 32 de bits e alguns com instruções 16 de bits. Os valores iniciais desses registradores são indefinidos.

R13: Ponteiro de Pilha (*Stack Pointer*, SP)
Usado para acessar a pilha de memória. Fisicamente há dois ponteiros de pilha, o principal (*Main Stack Pointer*, MSP) e o de processo (*Process Stack Pointer*, PSP)). O MSP é o ponteiro padrão, é selecionado após um *reset* do sistema ou quando o processador está em modo de exceção (Handler Mode). Seu valor inicial é o primeiro da memória na sequência de *reset*. Já o PSP é usado durante o Thread Mode, quando as tarefas da aplicação estão rodando, seu valor inicial é desconhecido.

Somente um dos ponteiros de pilha é visível durante a aplicação e os dois bits menos significativos de ambos são sempre nulos. Em aplicações que não fazem uso de um sistema operacional somente o MSP é usado.

R14: Registrador de Ligação (*Link Register*, LR)

Esse registrador armazena automaticamente o ponto em que uma rotina chama uma sub-rotina. Assim, ao fim da execução dessa sub-rotina, esse valor é carregado para o Contador de Programa e a execução continua de onde tinha anteriormente parado.

Se uma sub-rotina chamar outra sub-rotina, o valor nesse registrador será substituído e o ponto de retorno antigo se perderá, portanto é preciso que esse último valor seja salvo na pilha de memória.

Durante uma rotina de tratamento de exceção, o valor de LR é também sobrescrito mas por um valor de retorno de exceção, usado para disparar o retorno da exceção ao fim da rotina de tratamento.

R15: Contador de Programa (*Program Counter, PC*)

Marca o próximo endereço que deve ser executado na aplicação. Quando este registrador é lido, automaticamente seu valor decrementa de 4 (32 bits), apontando para o próximo endereço da execução. Já quando é feito uma operação de escrita, o programa pula para a posição apontada e passa a executar a aplicação a partir deste novo ponto.

O bit menos significativo do PC indica o tipo de instrução que está sendo executada, '0' para ARM e '1' para Thumb. Portanto no Cortex-M4, tal bit deve ser sempre '1' pois não são suportadas instruções ARM. Este fato deve ser lembrado quando é feita uma operação de escrita sobre o registrador.

Capítulo 3

Conhecendo a plataforma de trabalho

O hardware utilizado aqui será o TIVA ™ TM4C1294NCPDT, um kit de desenvolvimento da empresa Texas Instruments que possui um microcontrolador baseado no processador ARM Cortex-M4. A tabela **??** traz suas principais características.

Características	Descrição
Núcleo	ARM Cortex-M4F
Performance	Operação até 120-MHz; 150 DMIPS (Dhrystone MIPS) de performance
Memória Flash	1024 KB
SRAM	256 KB single-cycle System SRAM
EEPROM	6KB
ROM	ROM interna carregada com biblioteca TivaWare™ C Series
Interface de Periféricos Externos (EPI)	Interface dedicada de 8-/16-/32- bits dedicados a periféricos e memoria
Verificação de Redundância Cíclica (CRC)	Função Hash de 16-/32- bits, que suporta quatro formas de CRC
Universal Asynchronous Receivers/Transmitter (UART)	8 módulos UARTs
Quad Synchronous Serial Interface (QSSI)	Quatro módulos de SSI com Bi- , Quad- e suporte avançado de SSI
Inter-Integrated Circuit (I^2C)	10 módulos I^2C com 4 velocidades de transmissão
Controller Area Network (CAN)	2 controladores CAN 2.0 A/B
Ethernet MAC	10/100 Ethernet MAC
Ethernet PHY	PHY com IEEE 1588 PTP
Universal Serial Bus (USB)	USB 2.0 OTG/Host/Device com ULPI interface e suporte a Link Power Management (LPM)
Micro Acesso Direto à Memória (μDMA)	Controlador ARM© PrimeCell© 32-channel configurável μDMA
General-Purpose Timer (GPTM)	8 blocos 16/32-bit GPTM
Watchdog Timer (WDT)	2 Watchdog Timers
Hibernation Module (HIB)	Low-power battery-backed Hibernation module
General-Purpose Input/Output (GPIO)	15 physical GPIO blocks

Pulse Width Modulator (PWM)	1 modulo PWM , com 4 geradores PWM e um registador de controle, com um total de 8 saídas PWM.
Quadrature Encoder Interface (QEI)	Um modulo QEI
Analog-to-Digital Converter (ADC)	2 modulos ADC de 12-bit taxa de 2 milhões de amostras/segundo
Controlador Comparador Analógico	Três comparadores analógicos independentes
Comparador Digital	16 comparadores digitais
JTAG e Serial Wire Debug (SWD)	1 modulo JTAG com ARM SWD integrado
Encapsulamento	128-pin TQFP
Temperatura de Operação	$-40°C$ até $105°C$

Tabela 3.1: Características Básicas - TM4C1294NCPDT [**?**]

O TM4C1294NCPDT possui dois barramentos. Um desses é responsável pela conexão padrão entre o núcleo de processamento e os periféricos, o *Advanced Peripheral Bus* (APB). Já o outro, o *Advanced High-Performance Bus* (AHB), é um barramento especial que pode ser requisitado da maioria dos periféricos e, possui uma resposta muito mais rápida por ser exclusivo. O esquema dos barramentos é mostrado no diagrama de blocos da figura **??**.

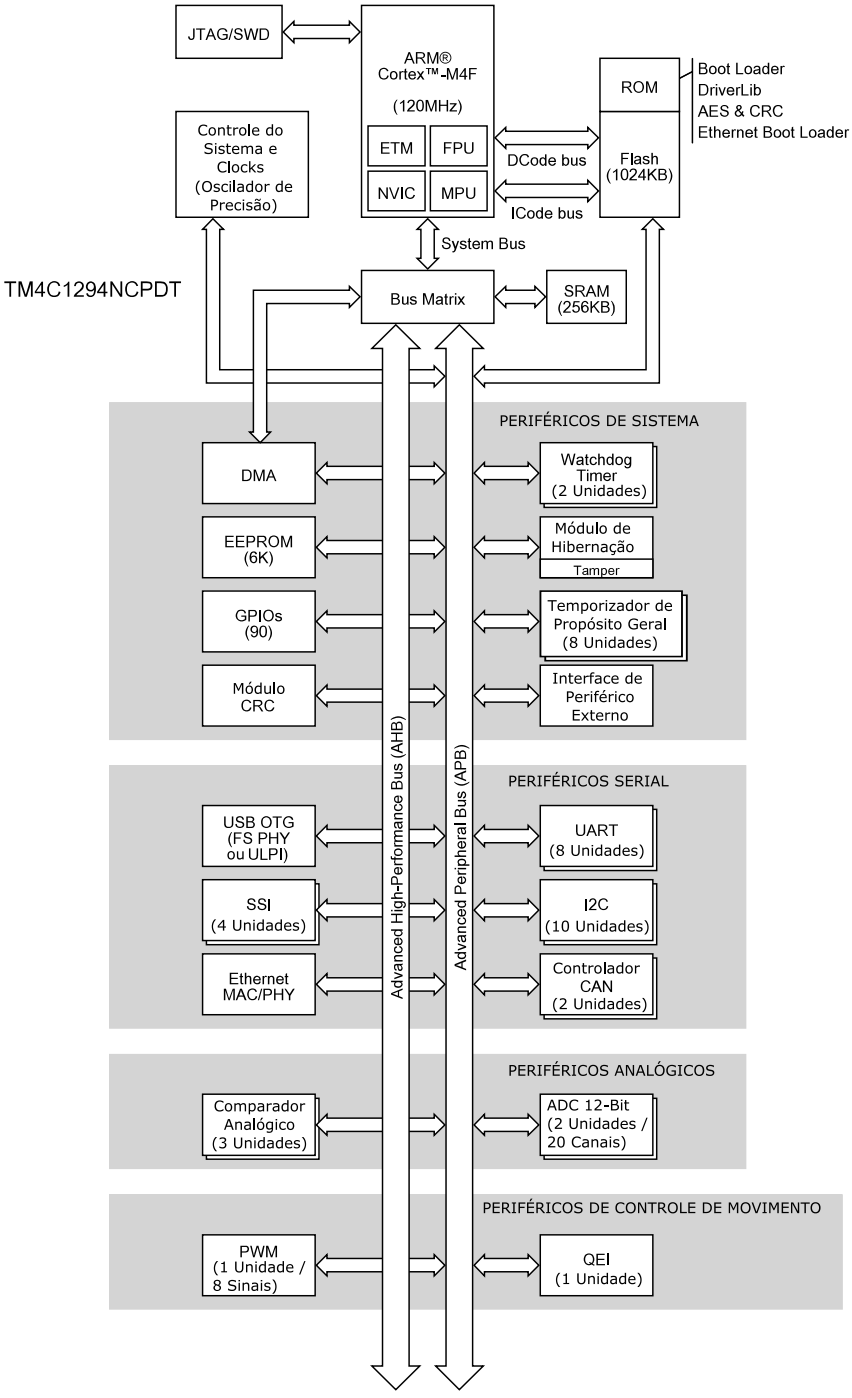

Figura 3.1: Diagrama de Blocos - TM4C1294NCPDT [?]

Capítulo 4

Iniciando um projeto no Code Composer

Os projetos abordados adiante farão uso da IDE Code Composer que é baseada em Eclipse em sua versão 6.1.2 que é a mais recente no momento em que este texto é escrito. oferecida gratuitamente mediante a um cadastro realizado no site da Texas Instruments.

Na hora de instalar a IDE é preciso que sejam marcadas as opções de compatibilidade com a placa em uso, a Tiva C Series TM4C1294 Connected LaunchPad, e ainda seu compilador GCC, caso contrário o projeto não poderá ser criado.

Após iniciar o Code Composer, inicie um novo projeto em **File** > **New** > **CCS Project** como mostrado na Figura **??**.

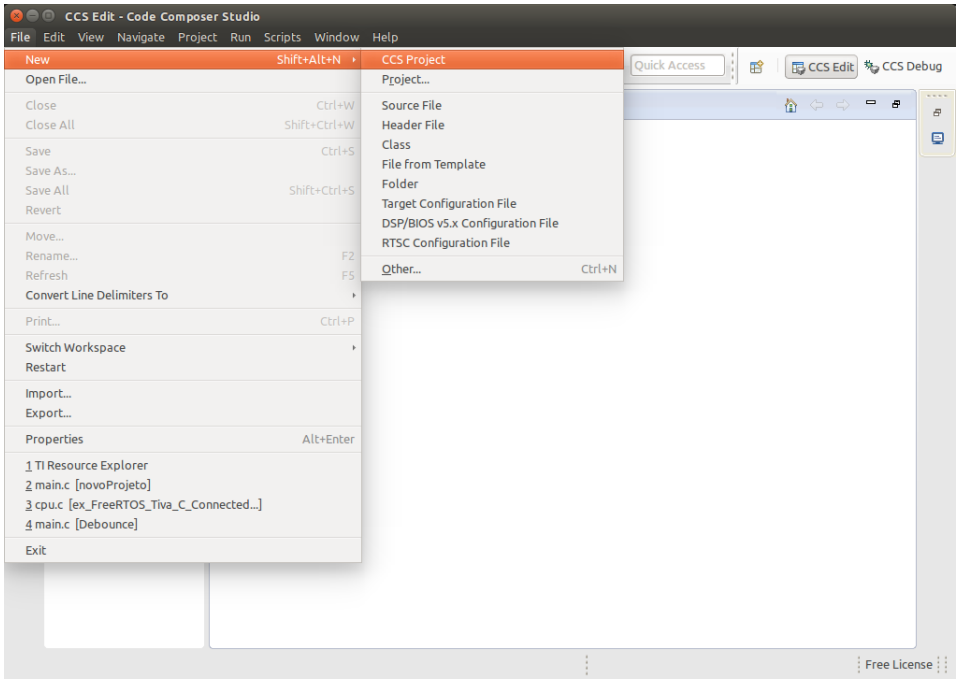

Figura 4.1: Criando um novo projeto

Uma janela de configurações será exibida para que o ambiente seja preparado para o hardware em uso, como na Figura **??**.

Para o hardware aqui utilizado, em *Target* escolhe-se a opção **Tiva C Series** e no

segundo campo **Tiva TM4C1294NCPDT**.

Em *Connection* será utilizada a **Stellaris In-Circuit Debug Interface** para a programação e debug do microcontrolador.

Após isso, escolhe-se um nome para o projeto e o diretório que será armazenado, que é normalmente o local do workspace padrão marcando a opção **Use default location**.

A Texas Instruments disponibiliza um compilador próprio porém será usado aqui o GCC, compilador de código aberto sob a licença GNU. Portanto, em *Compile version* escolhe-se a opção **GNU** com a versão mais recente. As outras opções não precisam ser alteradas.

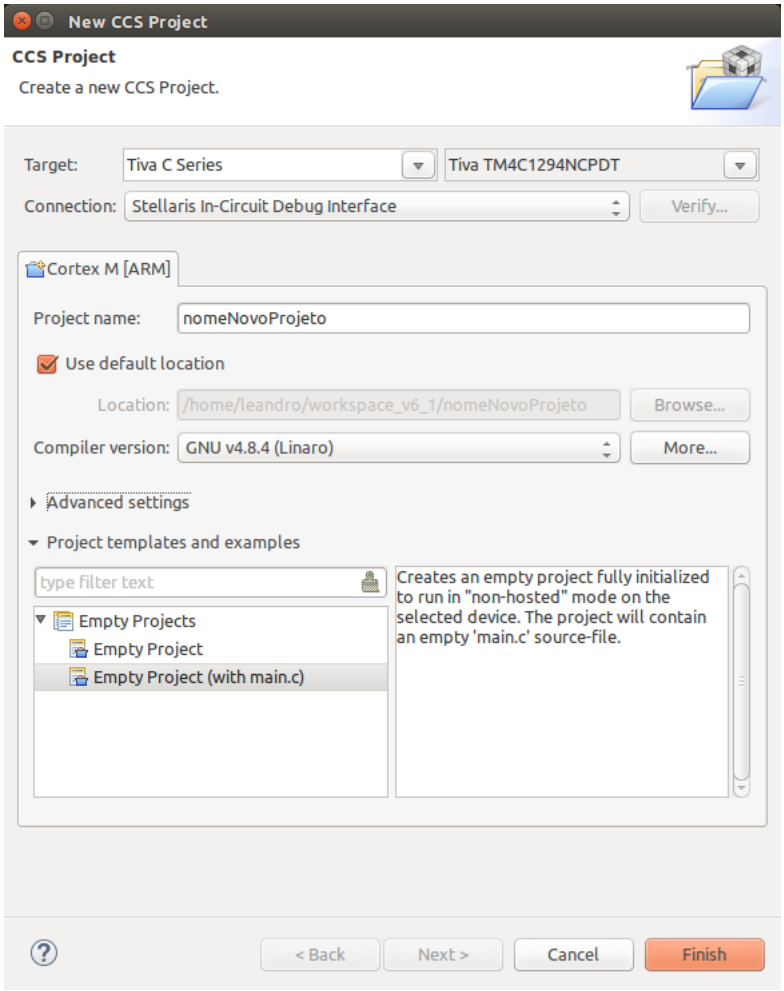

Figura 4.2: Configurando o projeto

Clicando em *Finish* o projeto será criado. Para o correto funcionamento do compilador GCC devem-se ainda ser feitos mais alguns ajustes.

Selecionando o projeto criado na barra lateral *Project Explorer*, vá em **Project** > **Properties** como na Figura **??**.

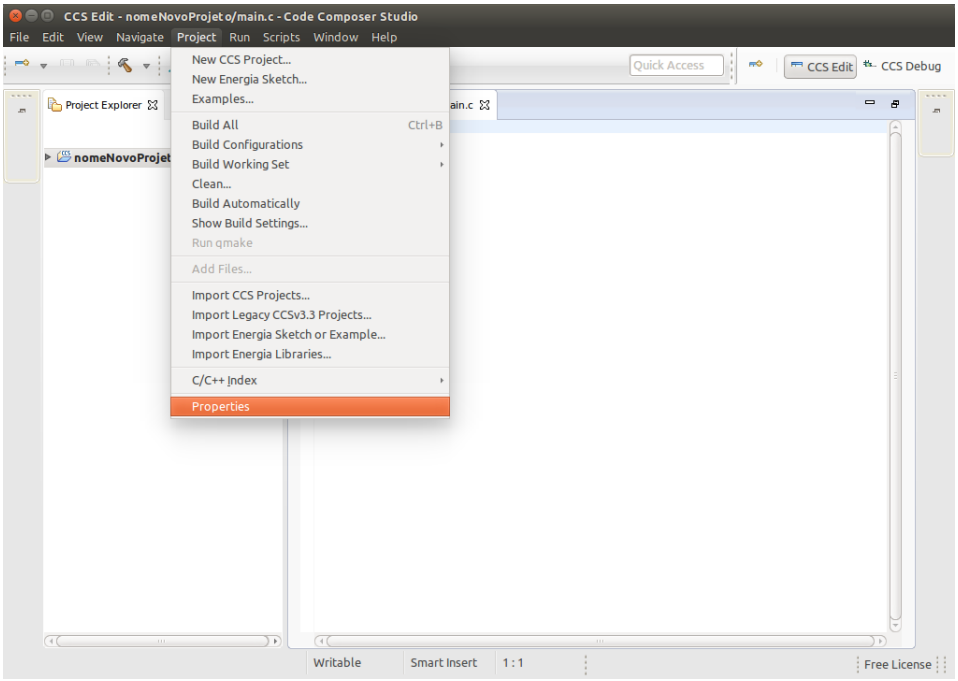

Figura 4.3: Abrindo propriedades do projeto

Na janela de propriedades selecione **Build** > **GNU Compiler** > **Symbols**. Adicione um novo símbolo clicando no botão *Add* como na Figura **??**. Na janela que se abre digite **TARGET_IS_TM4C129_RA1** e clique em *OK*. Adicione ainda o símbolo **gcc**. Esses símbolos não podem conter erros de escrita, caso contrário causarão erros na hora da compilação. Ao se ter os três símbolos mostrados na Figura **??**, selecione **Build** > **GNU Linker** > **Basic**. Na opção *Set start address* digite **_start** como na Figura **??** e clique em *OK*.

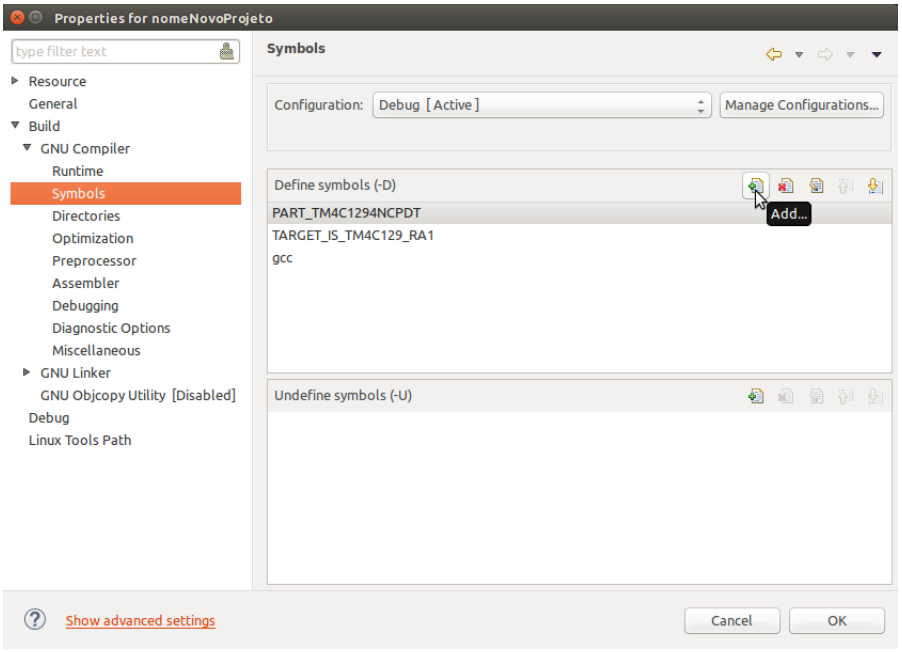

Figura 4.4: Adicionando símbolo para a compilação no GCC

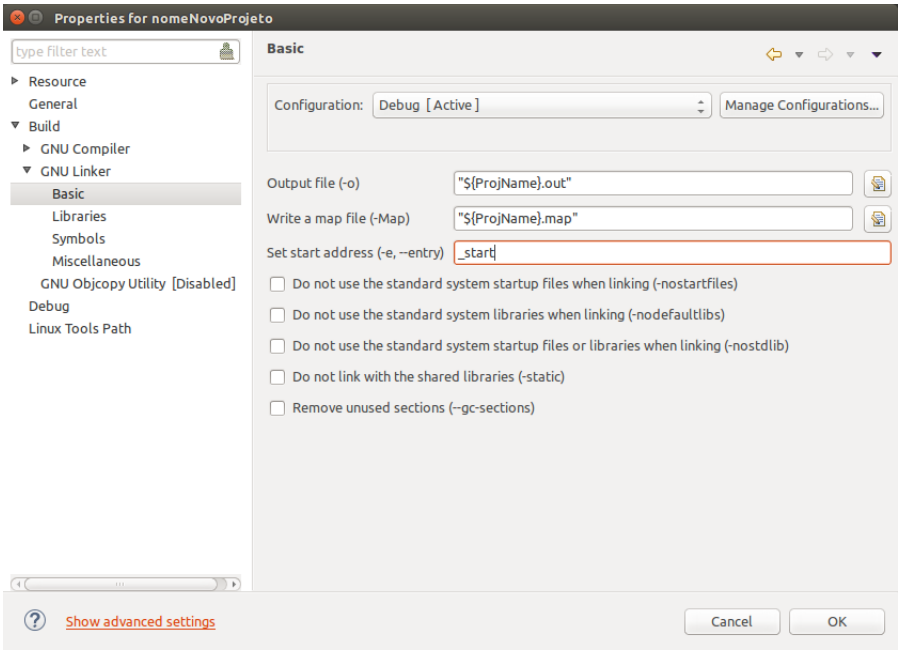

Figura 4.5: Configurando endereço de início do Linker do GCC

Ao fim desses passos o projeto estará criado e poderá ser compilado no Code Composer utilizando o GCC.

Capítulo 5

Biblioteca TivaWare

Para facilitar a programação do microcontrolador será feito o uso da biblioteca TivaWare fornecida pela Texas Instruments. Tal ferramenta facilita o controle do processador e acesso aos periféricos disponíveis. A TivaWare pode ser obtida no site da empresa gratuitamente.

O site disponibiliza somente a versão para o sistema Windows, que vem em formato executável, sendo preciso apenas dar um clique duplo sobre o arquivo e seguir os passos da instalação. Já para sistemas não derivados do *MS-DOS*, basta abrir este mesmo executável baixado com um aplicativo de descompactação de arquivos e copiar o conteúdo para um diretório qualquer desejado.

A estrutura do TivaWare é composta basicamente de dois diretórios [**?**]:

driverlib/ Contém o código fonte para os drivers do dispositivo

inc/ Contém os arquivos de cabeçalho que são usados pelos drivers para acessar os registradores do microcontrolador

Os outros arquivos contidos no pacote do TivaWare são extras que facilitam alguns usos do microcontrolador. Como o diretório *'examples/'* que contém códigos prontos para utilização em alguns dos microcontroladores e periféricos suportados, o *'utils/'* com algumas implementações frequentes e a biblioteca *'usblib/'* que implementa uma comunicação usb com portes para vários tipos de arquivos.

5.1 Incluindo a TivaWare ao projeto

Para a utilização da TivaWare nos projetos que serão apresentados é preciso que as aplicações desenvolvidas tenham acesso à tais bibliotecas. Tal comunicação pode ser feita de dois tipos: *linkando* ou copiando a biblioteca para o diretório do código fonte ou adicionando o diretório da biblioteca nos comandos de compilação.

5.1.1 Bibliotecas junto ao código fonte

Este método pode ser feito de dois modos, copiando as bibliotecas para o diretório do código fonte da aplicação, ou *linkando*-as a este diretório.

É importante notar que se os arquivos de código fonte forem portados para outra máquina, somente serão compilados se as bibliotecas estiverem disponíveis nesta. Portanto, sempre que houver memória disponível, é aconselhável que se copie as bibliotecas usadas na aplicação para junto de seu diretório.

Para copiar as bibliotecas é possível apenas copiar as pastas para o diretório do projeto que este será atualizado automaticamente ou ainda arrastar e soltar o diretório ou arquivo da biblioteca sobre o projeto na barra lateral *Project Explorer* no Code Composer que será aberta uma janela intermediária como na figura **??**.

Selecionando **'Copy files and folders'** os arquivos serão copiados para o diretório do projeto escolhido. Já as duas outras opções criarão somente um *link* do arquivo no diretório especificado na caixa de seleção **'Create link location relative to'**, deste modo o compilador verá os arquivos como se eles estivessem neste diretório, porém existe apenas o caminho para alcançá-los. Se acaso eles forem movidos haverá erros de compilação.

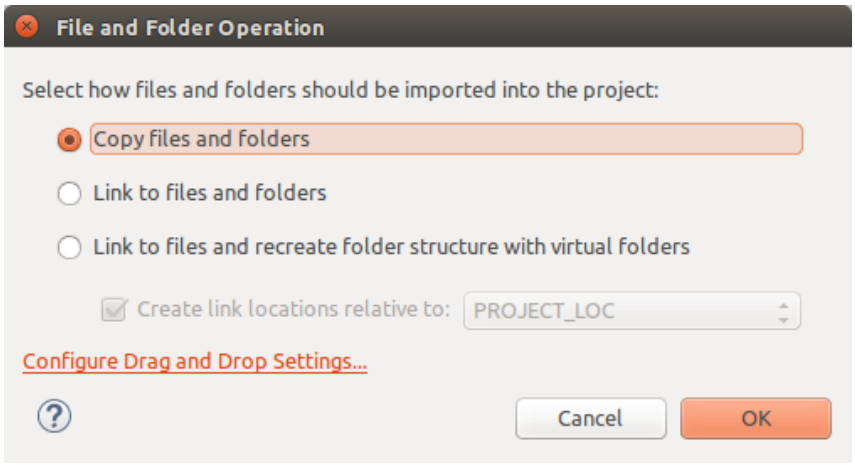

Figura 5.1: Janela de importação de arquivos

5.1.2 Inclusão de caminho na compilação

Um outro modo de juntar as bibliotecas ao código fonte é adicionando seu caminho à compilação. Com o projeto selecionado na janela lateral *Project Explorer*, vá em **Project** > **Properties** > **Build** > **GNU Compiler** > **Directories** e clique em *Add*, como na figura **??**.

Na janela aberta é possível digitar um caminho para o diretório ou arquivo, mas para prevenir erros existem os botões inferiores que abrirão uma navegação nos diretórios do sistema. Em **Workspace** é possível escolher o caminho para o diretório de um projeto ou de seus subdiretórios. Em **Variables** pode-se escolher o caminho armazenado em uma das variáveis de ambiente do projeto. E finalmente, em **Browse** é possível buscar um diretório navegando pelos arquivos do sistema.

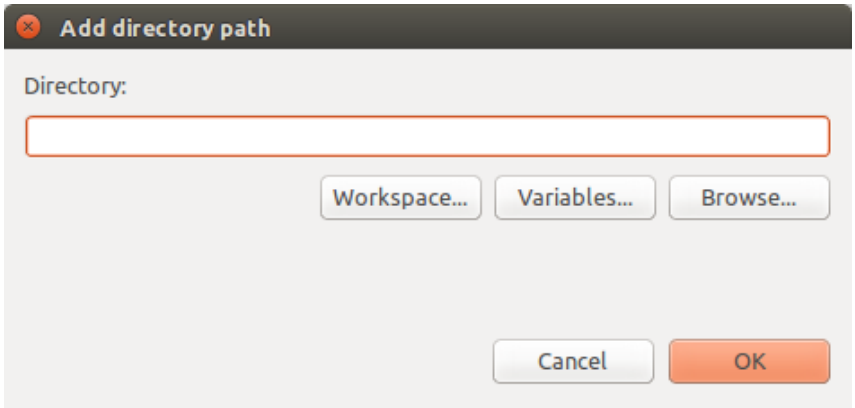

Figura 5.3: Escolhendo diretórios para incluir na compilação

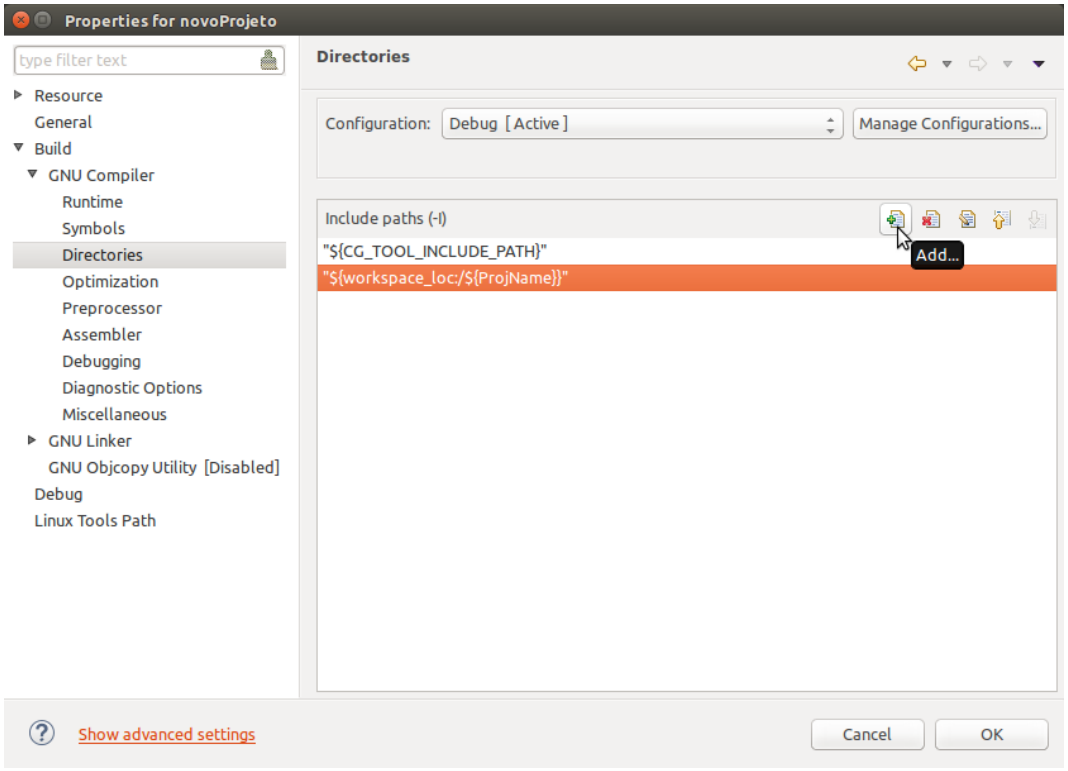

Figura 5.2: Incluindo diretórios para compilação

5.2 TivaWare na ROM

O TM4C1294NCPDT possui carregado na memória ROM uma parte da biblioteca de drivers do TivaWare. Isso possibilita a geração de um arquivo menor na hora da compilação, economizando memória de programa.

Para o uso das funções gravadas na ROM é necessário importar o arquivo de cabeçalho *'driverlib/rom.h'* e ainda usar o prefixo *'ROM_'* junto a função desejada. Por exemplo, para usar a função de configuração de clock do sistema

$$SysCtlClockFreqSet()$$

carregada na ROM, esta deve ser chamada como

$$ROM\_SysCtlClockFreqSet().$$

Porém, ao chamar tal função da ROM é possível que ela não seja encontrada na hora da compilação. Isso se deve ao fato de que nem todos os hardwares compatíveis com a TivaWare possuem uma memória ROM carregada com sua biblioteca ou mesmo não possua toda ela. Tal problema é resolvido adicionando-se o arquivo de cabeçalho *'driverlib/rom_map.h'* e usando o prefixo *'MAP_'* junto às funções ao invés de *'ROM_'*. Para o exemplo da função de configuração de clock, a chamada seria feita da forma

$$MAP\_SysCtlClockFreqSet().$$

Esse arquivo de cabeçalho implementa uma estrutura que confere se a função usada existe na ROM do dispositivo para o qual o código será compilado e só assim a substitui. O prefixo de mapeamento pode ser usado em todas as chamadas de funções implementadas pela TivaWare.

Capítulo 6

Sistema de Clock

Fontes de clock são elementos básicos para o funcionamento de qualquer DSP, e a sua configuração é uma das primeiras tarefas a se realizar em qualquer aplicação. No Tiva - TM4C1294NCPDT há múltiplas fontes de clock, os quais devem ser configuradas logo após um *Power-On Reset* (POR), ou seja, quando o dispositivo é iniciado ou recuperado de um *reset*.

6.1 Fontes de clock

As fontes disponíveis para o controle do TM4C1294NCPDT são:

Oscilador Interno de Precisão (do inglês, PIOSC)
Fonte de clock interna ao microcontrolador que é usada durante e logo após um POR. É o clock usado para iniciar a execução de uma aplicação. Não necessita de nenhum componente externo e fornece um clock de 16 MHz que apesar de ser preciso, varia com temperaturas mais extremas. O PIOSC é útil também para aplicações que não exijam uma fonte de clock tão precisa. Mesmo sendo, ou não, o clock do sistema, o PIOSC pode ser configurado como fonte de clock de um periférico.

Oscilador Principal (do inglês, MOSC)
O oscilador principal fornece um clock de precisão por meio de um desses métodos: uma fonte *single-end* de clock é conectada ao pino de entrada OSC0 do microcontrolador, ou um cristal externo é conectado entre o pino de entrada OSC0 e o pino de saída OSC1. Com o PLL em uso, o cristal deve ser de uma frequência entre 5 MHz e 25 MHz. Se não, pode variar de 4 MHz até 25 MHz.

Oscilador Interno de Baixa-frequência (do inglês, LFIOSC)
Clock com frequência nominal de 33 KHz com uma porcentagem de variação. É usado durante os modos de economia de energia *Deep-Sleep*. Estes modos proveem um número reduzido de periféricos em funcionamento e podem desligar o MOSC e/ou PIOSC enquanto o microcontrolador está neste estado.

Oscilador RTC do Módulo de Hibernação (do inglês, RTCOSC)
Fornece uma saída de clock para o sistema selecionável entre duas: um clock externo de 32.768 Hz ou um Clock de Baixa Frequência de Hibernação (HIB LFIOSC). O Módulo de Hibernação pode receber um sinal de clock de 32.768 Hz conectado ao pino XOSC0. O oscilador de 32.768 Hz pode ser usado para o clock do sistema, assim eliminando a necessidade de um cristal ou oscilador adicional. Alternativamente, o Módulo de Hibernação contem um oscilador de baixa frequência (HIB LFIOSC) que provê um RTC para o sistema e pode também prover um clock de precisão para os modos de economia de energia Deep-Sleep ou Hibernação. Note que o HIB LFIOSC é

uma fonte de clock diferente de LFIOSC, os dois possuem a mesma frequência nominal mas, enquanto o primeiro pode variar de 10 KHz à 90 KHz, o segundo varia de 10 KHz à 70 KHz.

O clock interno do sistema (SysClk), pode ser derivado de qualquer uma das fontes anteriormente listadas. Um PLL interno pode ser usado pelo PIOSC ou pelo MOSC, e somente estes, para gerar o SysClk e os clocks dos periféricos.

6.2 Circuito de verificação do MOSC

O microcontrolador possui circuitos de controle de clock que verificam se o Oscilador Principal está funcionando adequadamente e na frequência apropriada. O circuito sinaliza quando esta frequência se encontra fora dos valores permitidos para o cristal em uso.

Este circuito deve ser habilitado em tempo de execução. Quando ligado, se um erro for constatado, o MOSC é desligado, é ligado o PIOSC e o sistema é resetado levando o processador a uma interrupção não-mascarável (NMI).

6.3 Na TivaWare

As principais funções de configuração do clock no TivaWare estão listadas abaixo.

```
void SysCtlMOSCConfigSet(uint32_t ui32Config)
```

Configura o circuito monitor do oscilador principal.

`ui32Config`
Configura o controle do oscilador principal a partir da lógica OR de definições no formato **SYSCTL_MOSC_*k***, onde ***k*** pode assumir o valor de:

- **VALIDATE** para verificar uma falha do MOSC
- **INTERRUPT** quando se deseja gerar uma interrupção ao invés do reset do processador
- **NO_XTAL** se não há um oscilador esxterno nos pinos OSC0/OSC1, reduzindo o consumo de energia
- **PWR_DIS** se deseja-se que o MOSC seja desligado. Se este parâmetro não for especificado o oscilador permanece ligado
- **LOWFREQ** se a frequência do MOSC está abaixo de 10 MHZ
- **HIGHFREQ** se a frequência do MOSC está acima de 10 MHZ
- **SESRC** quando o MOSC é um oscilador *single-end* conectado ao pino OSC0. Se não especificado, assume-se que um cristal está em uso.

```
uint32_t SysCtlClockFreqSet(uint32_t ui32Config,
                            uint32_t ui32SysClock)
```

Configura o clock do sistema, frequência de entrada, seu oscilador fonte e o uso ou não do PLL. Retorna o valor da frequência definida em Hz.

`ui32Config`
Configuração do clock. Lógica OR das seguintes máscaras:

SYSCTL_XTAL_*k*MHZ indica o uso de um cristal externo de ***k*** MHz, podendo assumir os valores: **5**, **6**, **8**, **10**, **12**, **16**, **18**, **20**, **24** ou **25**.

SYSCTL_OSC_*k* corresponde ao oscilador usado. Sendo ***k***:

- **MAIN** para o oscilador principal
- **INT** para o oscilador interno de precisão de 16 MHz
- **INT30** para o oscilador interno de baixa frequência
- **EXT32** para o oscilador de 32.768 Hz do módulo de hibernação (apenas quando o módulo estiver disponível).

SYSCTL_USE_*k* fonte do clock do sistema. Podendo ***k*** assumir os valores:

- **PLL** quando a saída do PLL fornece o clock do sistema.
- **OSC** para o oscilador fonte alimentar o clock do sistema.

SYSCTL_CFG_VCO_*k* indica a frequência do VCO do PLL quando este está em uso. Os valores de ***k*** podem somente assumir os valores de **480** para 480 MHz e **320** para 320 MHz. O TivaWare escolhe o valor do divisor do PLL para gerar o clock mais próximo do valor desejado a partir destes valores de VCO.

`ui32SysClock`
Valor inteiro requerido para o clock do sistema. Se não for possível alcançá-lo com as configurações usadas é assumido o valor mais próximo de clock abaixo deste valor.

```
void SysCtlPeripheralEnable(uint32_t ui32Peripheral)
```

Habilita o clock em um dos periféricos. Há um período de 5 ciclos de clock da chamada da função até a real ligação do periférico. Cuidados devem ser tomados para que não haja o acesso ao periférico durante este curto espaço de tempo.

`ui32Peripheral`
Periférico a ser ligado o clock.

SYSCTL_PERIPH_*k* indica o periférico à ser habilitado, onde ***k*** deve ser:

- **ADC*k*** para representar o AD de número *k*.
- **CAN*k*** para representar o barramento CAN de número *k*.
- **CCM*k*** para representar o módulo CRC do barramento CAN de número *k*.
- **COMP*k*** para representar o comparador de número *k* do AD.
- **EEPROM*k*** para representar a EEPROM de número *k*.
- **EMAC** para representar o módulo Ethernet MAC.
- **EPHY** para representar o módulo Ethernet PHY.
- **EPI** para representar a Interface de Periféricos Externos.
- **GPIO*k*** para representar a GPIO de letra *k*.
- **HIBERNATE** para representar o módulo de hibernação.
- **I2C*k*** para representar o módulo I2C de número *k*.
- **PWM*k*** para representar o módulo de PWM de número *k*.
- **QEI*k***, **QEI1** para representar de número *k*.
- **SSI*k*** para representar o módulo SSI de número *k*.
- **TIMER*k*** para representar o contador de número *k*.
- **UART*k*** para representar o módulo UART de número *k*.
- **UDMA** para representar o módulo de Acesso Direto à Memória.
- **USB*k*** para representar o módulo USB de número *k*.

- **WDOG*k*** para representar o módulo de estouro de tempo de número *k*.
- **WTIMER*k*** para representar o contador do módulo de estouro de tempo de número *k*.

```
void SysCtlPeripheralDisable(uint32_t ui32Peripheral)
```

Desabilita o clock em um dos periféricos. Uma vez desabilitado, não responderá a nenhum comando.

`ui32Peripheral`
Periférico a ser desabilitado.

SYSCTL_PERIPH_*k* indica o periférico à ser habilitado, onde ***k*** deve ser:

- **ADC*k*** para representar o AD de número *k*.
- **CAN*k*** para representar o barramento CAN de número *k*.
- **CCM*k*** para representar o módulo CRC do barramento CAN de número *k*.
- **COMP*k*** para representar o comparador de número *k* do AD.
- **EEPROM*k*** para representar a EEPROM de número *k*.
- **EMAC** para representar o módulo Ethernet MAC.
- **EPHY** para representar o módulo Ethernet PHY.
- **EPI** para representar a Interface de Periféricos Externos.
- **GPIO*k*** para representar a GPIO de letra *k*.
- **HIBERNATE** para representar o módulo de hibernação.
- **I2C*k*** para representar o módulo I2C de número *k*.
- **PWM*k*** para representar o módulo de PWM de número *k*.
- **QEI*k***, **QEI1** para representar de número *k*.
- **SSI*k*** para representar o módulo SSI de número *k*.
- **TIMER*k*** para representar o contador de número *k*.
- **UART*k*** para representar o módulo UART de número *k*.
- **UDMA** para representar o módulo de Acesso Direto à Memória.
- **USB*k*** para representar o módulo USB de número *k*.
- **WDOG*k*** para representar o módulo de estouro de tempo de número *k*.
- **WTIMER*k*** para representar o contador do módulo de estouro de tempo de número *k*.

```
void SysCtlPeripheralSleepEnable(uint32_t ui32Peripheral)
```

Permite que o periférico continue operando mesmo quando o processador estiver em modo de economia de energia.

`ui32Peripheral`
Periférico a ser habilitado em modo de economia de energia.

SYSCTL_PERIPH_*k* indica o periférico à ser habilitado, onde ***k*** deve ser:

- **ADC*k*** para representar o AD de número *k*.
- **CAN*k*** para representar o barramento CAN de número *k*.
- **CCM*k*** para representar o módulo CRC do barramento CAN de número *k*.

- **COMP*k*** para representar o comparador de número *k* do AD.
- **EEPROM*k*** para representar a EEPROM de número *k*.
- **EMAC** para representar o módulo Ethernet MAC.
- **EPHY** para representar o módulo Ethernet PHY.
- **EPI** para representar a Interface de Periféricos Externos.
- **GPIO*k*** para representar a GPIO de letra *k*.
- **HIBERNATE** para representar o módulo de hibernação.
- **I2C*k*** para representar o módulo I2C de número *k*.
- **PWM*k*** para representar o módulo de PWM de número *k*.
- **QEI*k***, **QEI1** para representar de número *k*.
- **SSI*k*** para representar o módulo SSI de número *k*.
- **TIMER*k*** para representar o contador de número *k*.
- **UART*k*** para representar o módulo UART de número *k*.
- **UDMA** para representar o módulo de Acesso Direto à Memória.
- **USB*k*** para representar o módulo USB de número *k*.
- **WDOG*k*** para representar o módulo de estouro de tempo de número *k*.
- **WTIMER*k*** para representar o contador do módulo de estouro de tempo de número *k*.

```
void SysCtlPeripheralSleepDisable(uint32_t ui32Peripheral)
```

Desabilita o periférico quando o processador estiver em modo de economia de energia. Isso ajuda a diminuir a corrente usada no dispositivo.

`ui32Peripheral`

Periférico a ser desabilitado em modo de economia de energia.

SYSCTL_PERIPH_*k* indica o periférico à ser habilitado, onde ***k*** deve ser:

- **ADC*k*** para representar o AD de número *k*.
- **CAN*k*** para representar o barramento CAN de número *k*.
- **CCM*k*** para representar o módulo CRC do barramento CAN de número *k*.
- **COMP*k*** para representar o comparador de número *k* do AD.
- **EEPROM*k*** para representar a EEPROM de número *k*.
- **EMAC** para representar o módulo Ethernet MAC.
- **EPHY** para representar o módulo Ethernet PHY.
- **EPI** para representar a Interface de Periféricos Externos.
- **GPIO*k*** para representar a GPIO de letra *k*.
- **HIBERNATE** para representar o módulo de hibernação.
- **I2C*k*** para representar o módulo I2C de número *k*.
- **PWM*k*** para representar o módulo de PWM de número *k*.
- **QEI*k***, **QEI1** para representar de número *k*.
- **SSI*k*** para representar o módulo SSI de número *k*.
- **TIMER*k*** para representar o contador de número *k*.
- **UART*k*** para representar o módulo UART de número *k*.
- **UDMA** para representar o módulo de Acesso Direto à Memória.
- **USB*k*** para representar o módulo USB de número *k*.
- **WDOG*k*** para representar o módulo de estouro de tempo de número *k*.

- **WTIMER*k*** para representar o contador do módulo de estouro de tempo de número k.

```
void SysCtlPeripheralClockGating(bool bEnable)
```

Habilita e desabilita o clock dos periféricos em modo de economia de energia.

bEnable
: Valor booleano que deve ser **true** se os periféricos podem ser usados durante o modo de economia de energia, e **false** se eles não podem ser usados neste período.

6.4 Exemplo

Um exemplo de configuração do clock do microcontrolador é dado a seguir:

```
1  // Configurando circuito de verificacao
2  // do MOSC para frequencias acima de 10 MHZ
3  SysCtlMOSCConfigSet(SYSCTL_MOSC_HIGHFREQ);
4
5  // Fonte de clock externa de 25 MHz,
6  // provinda do oscilador principal,
7  // usando a saida do PLL com fvco = 480 MHz,
8  // gerando um clock de 120 MHz para o microcontrolador
9  int systemClockFreq = SysCtlClockFreqSet ( \
10             (SYSCTL_XTAL_25MHZ | \
11              SYSCTL_OSC_MAIN | \
12              SYSCTL_USE_PLL | \
13              SYSCTL_CFG_VCO_480), 120000000);
14
15 // Habilita o funcionamento da GPIO A e da GPIO B
16 SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
17 SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
18
19 // Habilita o uso somente da GPIO B em
20 // modo de economia de energia
21 // A GPIO A nao podera ser utilizado durante
22 // este periodo
23 SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOB);
24
25 // Habilita o clock nos perifericos em modo de economia de
      energia
26 SysCtlPeripheralClockGating(true);
```

Capítulo 7

Portas de Entrada e Saída de Propósito Geral (GPIOs)

O TM4C1294NCPDT possui 15 portas GPIOs de 8 pinos cada. Elas são nomeadas com as letras de *'A'* à *'Q'* menos as letras *'I'* e *'O'*. Algumas das especificações das GPIOs são:

- Possui mais de 90 GPIOS, dependendo da configuração usada.
- Pinos específicos possuem ligação com os periféricos do microcontrolador e suas funções devem ser configuradas.
- Tensão em configuração de entrada de 3,3 V.
- Todas as portas são conectadas ao Barramento de Alta Performance (AHB).
- Mudança rápida de nível de saída da porta a cada ciclo de clock em portas ligadas ao AHB.
- Interrupções por pinos nas portas P e Q por bordas de subida, descida ou ambas.
- Podem ser usadas para iniciar uma sequência de amostragem do A/D ou uma transferência μDMA.
- Estado dos pinos podem ser mantidos durante o modo de hibernação; variações de nível nos pinos da porta P podem ser usadas para acordar o sistema da hibernação.
- Pinos configurados como entradas digital utilizam circuitos Schmitt-trigger.
- Pinos possuem resistores de pull-up e pull-down e limites de corrente para 2, 4, 6, 8, 10 e 12 mA.
- Configuração dreno-aberto habilitada

7.1 Na TivaWare

As principais funções presentes na TivaWare responsáveis pela configuração e manipulação das GPIOs são listadas a seguir.

```
void GPIODirModeSet(uint32_t ui32Port,
                    uint8_t ui8Pins,
                    uint32_t ui32PinIO)
```

Configura a direção dos pinos da porta especificada.

ui32Port
: Base da porta a ser configurada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

ui32Pins
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

ui32PinIO
: Direção ou modo dos pinos especificados.

  GPIO_DIR_MODE_IN pinos configurados como de entrada

  GPIO_DIR_MODE_OUT pinos configurados como de saída

  GPIO_DIR_MODE_HW direção dos pinos controlada pelo pelo hardware de algum periférico

```
void GPIOPinConfigure(uint32_t ui32PinConfig)
```

Configura pino utilizado por algum periférico especial.

ui32PinConfig
: Valor de configuração do pino, especificado pelas definições no formato **GPIO_P*Ak*_*pin***. Sendo ***A*** uma letra representando a porta que contém o pino, ***k*** o número do pino referente à porta e ***pin*** a função atribuída ao pino.

```
uint32_t GPIODirModeGet(uint32_t ui32Port,
                        uint8_t ui8Pin)
```

Retorna a direção de pinos de uma determinada porta.

ui32Port
: Base da porta a ser consultada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

ui32Pins
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

```
uint32_t GPIOPadConfigSet(uint32_t ui32Port,
                          uint8_t ui8Pins,
                          uint32_t ui32Strength,
                          uint32_t ui32PinType)
```

Configura especificações de potência e tipo dos pinos das portas GPIOs.

ui32Port
: Base da porta a ser configurada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

ui32Pins
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

ui32Strength
: Especifica a corrente máxima que os pinos podem fornecer. Configurada através da definição **GPIO_STRENGTH_*k*MA** onde o máximo de corrente nos pinos é de ***k*** mA. Sendo que ***k*** pode assumir os valores: 2, 4, 6, 8, 10 e 12.

ui32PinType
: Valor que configura o tipo do pino. Definido por **GPIO_PIN_TYPE_*k***, onde k pode assumir os seguintes valores:

  STD pino do tipo *push-pull* (tipo padrão)

  STD_WPU pino do tipo *pull-up* fraco, ou seja, com um resistor de *pull-up* pouca corrente circulando

  STD_WPD pino do tipo *pull-down* fraco, ou seja, com um resistor de *pull-down* pouca corrente circulando

  OD pino do tipo coletor aberto

  ANALOG pino de entrada analógica

```
uint32_t GPIOPadConfigGet(uint32_t ui32Port,
                          uint8_t ui8Pin,
                          uint32_t *pui32Strength,
                          uint32_t *pui32PinType)
```

Retorna as informações de configuração utilizadas no pino de uma determinada porta.

ui32Port
: Base da porta a ser consultada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

ui32Pins
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

pui32Strength
: Ponteiro de um endereço de memória já alocado, onde estarão contidas as especificações de corrente.

pui32PinType
: Ponteiro de um endereço de memória já alocado, onde estarão contidas as especificações do tipo do pino.

```
int32_t GPIOPinRead(uint32_t ui32Port,
                    uint8_t ui8Pins)
```

Retorna os valores contidos nos pinos de entrada especificados.

`ui32Port`
: Base da porta a ser consultada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

`ui32Pins`
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

```
void GPIOPinWrite(uint32_t ui32Port,
                  uint8_t ui8Pins,
                  uint8_t ui8Val)
```

Configura os valores nos pinos de saída especificados.

`ui32Port`
: Base da porta a ser configurada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

`ui32Pins`
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

`ui32Val`
: Pacote de valores de cada pino, contendo 8 bits. Onde o bit 0 representa sinal alto no pino 0, o bit 1 representa sinal alto no pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

```
void GPIOIntTypeSet(uint32_t ui32Port,
                    uint8_t ui8Pins,
                    uint32_t ui32IntType)
```

Configura os tipos das interrupções nos pinos de entrada especificados.

`ui32Port`
: Base da porta a ser configurada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

`ui32Pins`
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

`ui32IntType`
: Definição do evento que dispara a interrupção.

  GPIO_FALLING_EDGE disparo de interrupção na borda de descida

  GPIO_RISING_EDGE disparo de interrupção na borda de subida

  GPIO_BOTH_EDGES disparo de interrupção em ambas as bordas

GPIO_LOW_LEVEL disparo de interrupção quando em nível baixo

GPIO_HIGH_LEVEL disparo de interrupção quando em nível alto

```
void GPIOIntEnable(uint32_t ui32Port,
                   uint32_t ui32IntFlags)
```

Habilita as interrupções nos pinos especificados.

ui32Port
: Base da porta a ser configurada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

ui32IntFlags
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_INT_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

```
void GPIOIntDisable(uint32_t ui32Port,
                    uint32_t ui32IntFlags)
```

Desabilita as interrupções nos pinos especificados.

ui32Port
: Base da porta a ser configurada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

ui32IntFlags
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_INT_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

```
void GPIOIntClear(uint32_t ui32Port,
                  uint32_t ui32IntFlags)
```

Limpa o *buffer* que armazena se já houve uma interrupção nos pinos especificados.

ui32Port
: Base da porta a ser configurada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

ui32IntFlags
: Pacote em OU binário de pinos com 8 bits. Onde o bit 0 representa o pino 0, o bit 1 representa o pino 1 e assim por diante. É possível utilizar das definições no formato **GPIO_INT_PIN_*k***, onde ***k*** é o número do pino de 0 a 7.

```
void GPIOIntRegister(uint32_t ui32Port,
                     void (*pfnIntHandler)(void))
```

Configura a função de tratamento de uma interrupção da GPIO. Aquela que é chamada quando ocorre a interrupção.

`ui32Port`
: Base da porta a ser configurada. Normalmente **GPIO_PORT*k*_BASE**, onde ***k*** é a letra identificadora da porta.

`pfnIntHandler`
: Ponteiro da função de tratamento. Esta não deve receber nada como parâmetro e nem retornar nada.

7.2 Exemplo

Um exemplo básico de configuração das portas GPIOs do microcontrolador é dado a seguir:

```
1
2  // Configura os pinos 0, 2 e 7 da porta A como sendo de entrada
3  GPIODirModeSet(
4      GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_2 |
5      GPIO_PIN_7, GPIO_DIR_MODE_IN);
6
7  // Configura os pinos 4, 5 e 6 da porta A como sendo de saida
8  GPIODirModeSet(GPIO_PORTA_BASE, GPIO_PIN_4 |
9      GPIO_PIN_5 | GPIO_PIN_6, GPIO_DIR_MODE_OUT);
10
11 // Configura os pinos 4 e 6 da porta A para poder
12 // fornecer ate 4 mA para o circuito
13 GPIOPadConfigSet(GPIO_PORTA_BASE,
14     GPIO_PIN_4 | GPIO_PIN_6, GPIO_STRENGTH_4MA);
15
16 // Armazena valor de entrada do pino 2 da porta A
17 int valor = GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_2);
18
19 // Configura pino de saida 4 em sinal alto e o 6 em baixo
20 GPIOPinWrite(
21     GPIO_PORTA_BASE, GPIO_PIN_4 | GPIO_PIN_6, GPIO_PIN_4);
```

Capítulo 8

Receptor/Transmissor Assíncrono Universal (UART)

O Transmissor/Receptor Assíncrono Universal (*Universal Asynchronous Receiver/Transmitter*, UART), é um periférico de transmissão e recepção de dados usado na comunicação entre dispositivos, sendo esta comunicação realizada de forma serial e assíncrona, ou seja sem a necessidade de transmissão do sinal de clock de referência. Este modo de transmissão faz necessário o uso de apenas duas vias de comunicação uma para a transmissão e outra para a recepção de dados.

8.1 Padrão da Comunicação

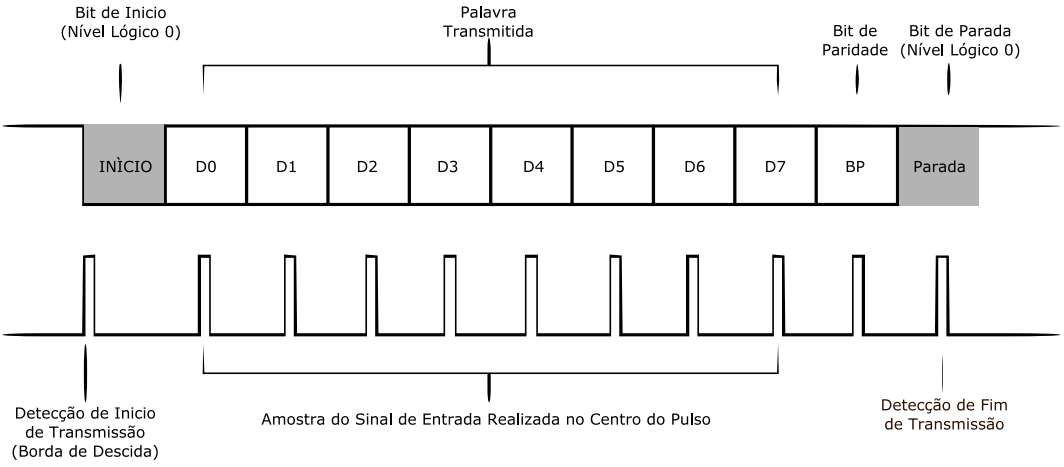

Figura 8.1: Protocolo de envio na comunicação UART

Para que a comunicação UART seja realizada é necessário que o sinal de transmissão obedeça a um protocolo. Quando uma palavra é transmitida, primeiro é enviado um bit de início de transmissão para o receptor. Este bit deve ser de nível logico 0 para que a ocorrência da borda de descida sinalize ao receptor que sincronize a amostragem do sinal a ser lido de modo que ela ocorra no meio de cada período de transmissão. Após transmitir os dados é necessário enviar um bit informando a existência de paridade ou não, e por último é enviado um bit de nível lógico alto para informar o fim da transmissão. Esta sintaxe pode ser observada na figura **??**.

8.2 UART do TM4C1294NCPDT

O Tiva TM4C1294NCPDT possui 8 módulos de comunicação UART. Cada um destes possuem um gerador de *baud-rate*, ou taxa de transmissão, que possibilitam transmissões de até 7,5 Mbps em modo de normal transmissão e 15 Mbps em modo *High Speed*.

Para que seja possível regular o *baud-rate* de forma mais precisa os módulos UART possuem um divisor de 22 bits, sendo 16 bits inteiros e 6 bits fracionários, pelo qual o módulo determina o período de transmissão de bit.

Já o buffer de leitura e transmissão do UART no Tiva tem um tamanho de 8 bits, porém para cada módulo existe uma FIFO de 16x8 bits tanto para transmissão quanto para recepção, sendo que o *trigger* de interrupção de estouro da FIFO é selecionável entre 1/8, 1/4, 1/2, 3/4, 7/8 ou 8/8.

O sinal de transmissão criado pelo UART do tiva pode transmitir dados seriais de 5,6,7 ou 8 bits de dados precedidos do bit de *Start* e acompanhados de um bit de paridade, se estiver habilitado, e 1 ou 2 bits de parada. A figura **??** apresenta o sinal característico da transmissão UART do Tiva.

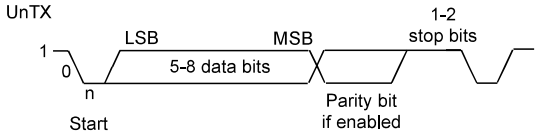

Figura 8.2: Sinal de Transmissão UART no Tiva TM4C1294NCPDT

A figura **??** apresenta os pinos de entrada e saída do UART que podem ser usados no Tiva TM4C1294NCPDT.

Tabela 8.1: Canais do UART - Tiva TM4C1294NCPDT [**?**]

Pino	Mux/Função	Tipo	Buffer	Descrição
UORx	PA0 (1)	I	TTL	UART Módulo 0, recepção do sinal
U0Tx	PA1 (1)	O	TTL	UART Módulo 0, transmissão do sinal
U1Rx	PB0 (1) PQ4 (1)	I	TTL	UART Módulo 1, recepção do sinal
U1Tx	PB1 (1)	O	TTL	UART Módulo 1, transmissão do sinal
U2Rx	PA6 (1) PD4 (1)	I	TTL	UART Módulo 2, recepção do sinal
U2Tx	PA7 (1) PD5 (1)	O	TTL	UART Módulo 2, transmissão do sinal
U3Rx	PA4 (1) (1) PJ0 (1)	I	TTL	UART Módulo 3, recepção do sinal
U3Tx	PA5 (1) PJ1 (1)	O	TTL	UART Módulo 3, transmissão do sinal
U4Rx	PK0 (1) PA2 (1)	I	TTL	UART Módulo 4, recepção do sinal
U4Tx	PK1 (1) PA3 (1)	O	TTL	UART Módulo 4, transmissão do sinal
U5Rx	PC6 (1)	I	TTL	UART Módulo 5, recepção do sinal
U5Tx	PC7 (1)	O	TTL	UART Módulo 5, transmissão do sinal
U6Rx	PP0 (1)	I	TTL	UART Módulo 6, recepção do sinal
U6Tx	PP1 (1)	O	TTL	UART Módulo 6, transmissão do sinal
U7Rx	PC4 (1)	I	TTL	UART Módulo 7, recepção do sinal
U7Tx	PC5 (1)	O	TTL	UART Módulo 7, transmissão do sinal

8.3 Na TivaWare

As principais funções para o uso da comunicação UART na TivaWare são apresentadas nesta seção.

```
void UARTClockSourceSet(uint32_t ui32Base,
                        uint32_t ui32Source);
```

Configura a fonte de clock utilizada no barramento UART.

ui32Base
: Base da UART a ser configurada. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

ui32Source
: Fonte de clock do UART.

  UART_CLOCK_SYSTEM clock fornecido pelo clock do sistema

  UART_CLOCK_PIOSC clock fornecido pelo oscilador interno de precisão

```
void UARTConfigSetExpClk(uint32_t ui32Base,
                         uint32_t ui32UARTClk,
                         uint32_t ui32Baud,
                         uint32_t ui32Config)
```

Configura os parâmetros da comunicação UART.

ui32Base
: Base da UART a ser configurada. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

ui32UARTClk
: Valor do clock a ser configurado no periférico da UART.

ui32Baud
: Baud rate da comunicação UART.

ui32Config
: Valor formado pela operação OU lógica de 3 parâmetros: número de bits de dados, número de bits de parada e a paridade.

  UART_CONFIG_WLEN_*k* protocolo com ***k*** bits de dados. Onde ***k*** pode assumir os valores: 5, 6, 7 e 8

  UART_CONFIG_STOP_*k* define 1 ou 2 bits de parada. Sendo que ***k*** assume os valores **ONE** ou **TWO**, respectivamente.

  UART_CONFIG_PAR_*k* onde ***k*** pode assumir os valores:

  - **NONE** sem bit de paridade
  - **EVEN** com bit de paridade par
  - **ODD** com bit de paridade ímpar
  - **ONE** bit de paridade é sempre 1
  - **ZERO** bit de paridade é sempre 0

```
void UARTEnable(uint32_t ui32Base)
```

Habilita o funcionamento da comunicação UART.

ui32Base
: Base da UART a ser habilitada. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

```
void UARTDisable(uint32_t ui32Base)
```

Desabilita o funcionamento da comunicação UART.

ui32Base
: Base da UART a ser desabilitada. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

```
int32_t UARTCharGet(uint32_t ui32Base)
```

Pega próximo caractere recebido pela UART. Se não houver nada para ser lido, o programa trava e espera até haver um caractere.

ui32Base
: Base da UART a ser lida. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

```
int32_t UARTCharGetNonBlocking(uint32_t ui32Base)
```

Pega próximo caractere recebido pela UART. Se não houver nada para ser lido, a leitura é ignorada e o programa continua normalmente. Nesse caso, a função retorna -1.

ui32Base
: Base da UART a ser lida. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

```
void UARTCharPut(uint32_t ui32Base,
                 unsigned char ucData))
```

Envia um caractere para ser transmitido pela UART. Se não houver espaço na fila de transmissão, o programa é travado e aguarda até o caractere ser colocado na fila.

ui32Base
: Base da UART a ser lida. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

ucData
: Caractere a ser transmitido pela UART.

```
bool UARTCharPutNonBlocking(uint32_t ui32Base,
                            unsigned char ucData))
```

Envia um caractere para ser transmitido pela UART. Se não houver espaço na fila de transmissão, a operação é ignorada e o programa continua normalmente. Nesse caso, a função retorna o valor *false* e a operação deve ser repetida posteriormente.

ui32Base
: Base da UART a ser lida. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

ucData
: Caractere a ser transmitido pela UART.

```
bool UARTCharsAvail(uint32_t ui32Base)
```

Informa se há algum caractere para ser lido na fila de recepção.

ui32Base
: Base da UART a ser lida. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

```
bool UARTSpaceAvail(uint32_t ui32Base)
```

Informa se há espaço para ser enviado um novo byte para a fila de transmissão.

ui32Base
: Base da UART a ser lida. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

```
void UARTIntRegister(uint32_t ui32Port,
                     void (*pfnIntHandler)(void))
```

Configura a função de tratamento de uma interrupção da UART. Aquela que é chamada quando ocorre a interrupção.

ui32Base
: Base da UART a ser configurada. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

pfnIntHandler
: Ponteiro da função de tratamento. Esta não deve receber nada como parâmetro e nem retornar nada.

```
void UARTIntEnable(uint32_t ui32Base,
                   uint32_t ui32IntFlags)
```

Habilita as interrupções especificadas da UART especificada.

`ui32Base`
: Base da UART a ser configurada. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

`ui32IntFlags`
: Pacote em formato de OU lógico com os parâmetros que especificam os eventos que podem causar a interrupção da UART. Cada parâmetro recebe uma definição no formato **UART_INT_*k***, onde ***k*** é um valor entre os seguintes:

  - **9BIT** interrupção por endereço de 9 bits
  - **OE** interrupção por transbordo de fila
  - **BE** interrupção por erro de parada
  - **PE** interrupção por erro de paridade
  - **FE** interrupção por erro de enquadramento
  - **RT** interrupção por estouro de tempo de recepção
  - **TX** interrupção por transmissão
  - **RX** interrupção por recepção
  - **DSR** interrupção por DSR
  - **DCD** interrupção por DCD
  - **CTS** interrupção por CTS
  - **RI** interrupção por RI

```
void UARTIntDisable(uint32_t ui32Base,
                    uint32_t ui32IntFlags)
```

Desabilita as interrupções especificadas da UART especificada.

`ui32Base`
: Base da UART a ser configurada. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

`ui32IntFlags`
: Pacote em formato de OU lógico com os parâmetros que especificam os eventos que podem causar a interrupção da UART. Cada parâmetro recebe uma definição no formato **UART_INT_*k***, onde ***k*** é um valor entre os seguintes:

  - **9BIT** interrupção por endereço de 9 bits
  - **OE** interrupção por transbordo de fila
  - **BE** interrupção por erro de parada
  - **PE** interrupção por erro de paridade
  - **FE** interrupção por erro de enquadramento
  - **RT** interrupção por estouro de tempo de recepção
  - **TX** interrupção por transmissão
  - **RX** interrupção por recepção
  - **DSR** interrupção por DSR
  - **DCD** interrupção por DCD

- **CTS** interrupção por CTS
- **RI** interrupção por RI

```
void UARTIntClear(uint32_t ui32Base,
                  uint32_t ui32IntFlags)
```

Limpa a flag que armazena a ocorrência das interrupções especificadas da UART especificada.

`ui32Base`
Base da UART a ser configurada. Normalmente **UART*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo configurada.

`ui32IntFlags`
Pacote em formato de OU lógico com os parâmetros que especificam as interrupção da UART a serem limpas. Cada parâmetro recebe uma definição no formato **UART_INT_*k***, onde ***k*** é um valor entre os seguintes:

- **9BIT** interrupção por endereço de 9 bits
- **OE** interrupção por transbordo de fila
- **BE** interrupção por erro de parada
- **PE** interrupção por erro de paridade
- **FE** interrupção por erro de enquadramento
- **RT** interrupção por estouro de tempo de recepção
- **TX** interrupção por transmissão
- **RX** interrupção por recepção
- **DSR** interrupção por DSR
- **DCD** interrupção por DCD
- **CTS** interrupção por CTS
- **RI** interrupção por RI

8.4 Exemplo

A seguir, é apresentado um exemplo de configuração da UART. Um exemplo mais abrangente pode ser visto na Seção **??**

```
1     // Habilita GPIO A usado na comunicacao da UART 0
2  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
3  // Aguarda 3 SysCtlDelay
4  // Aproximadamente 10 ciclos de clock
5  MAP_SysCtlDelay(3);
6  // Configura PA0 no modo Rx da UART 0
7  MAP_GPIOPinConfigure(GPIO_PA0_U0RX);
8  // Configura PA1 no modo Tx da UART 0
9  MAP_GPIOPinConfigure(GPIO_PA1_U0TX);
10
11 // Habilita UART 0
12 MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
13 // Configura PA0 e PA1 como pinos de comunicacao da UART
```

```
14 MAP_GPIOPinTypeUART(
15     GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
16 // Configura UART 0
17 // fonte de clock 120MHz para 115.200 baud 8N1
18 UARTConfigSetExpClk(UART0_BASE, 120000000, 115200,
19     (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
20     UART_CONFIG_PAR_NONE));
21
22 // Configura rotina de tratamento de interrupcao da UART
23 UARTIntRegister(UART0_BASE, UARTIntHandler);
24 // Habilita interrupcao da UART 0
25 MAP_IntEnable(INT_UART0);
26 // Configura pinos de interrupcao da UART 0
27 MAP_UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT);
```

Capítulo 9

Barramento SPI

A Interface de Periféricos Serial, ou SPI (*Serial Peripheral Interface*) é um dispositivo usado para a transmissão e recepção serial de dados. Sendo uma comunicação síncrona, a SPI necessita de uma fonte de clock de referência para se estabelecer, além de um sinal de *chip select*, ou **CS**, para ativar a recepção de dados no dispositivo receptor. Deste modo esta comunicação requer no minimo três vias de transmissão, sendo que a comunicação em cada barramento é unidirecional.

9.1 Padrão da Comunicação

A comunicação SPI possui a maior taxa de transmissão, ou *baud-rate*, dentre os demais protocolos de comunicação usados em microcontroladores, podendo chegar a até a 66Mpbs em periféricos com o AT45BD0100D da Adesto. O que possibilita um *baud-rate* tão elevado é o fato de que nesta comunicação a recepção e a transmissão de dados é feita separadamente e de forma direta, sem a necessidade de se transmitir bits de inicio ou termino de transmissão, e ainda de modo que o controle da transmissão é realizado pelo sinal CS (Chip Select) e pelo sinal CLK (Clock). A figura **??** apresenta o padrão de uma comunicação SPI.

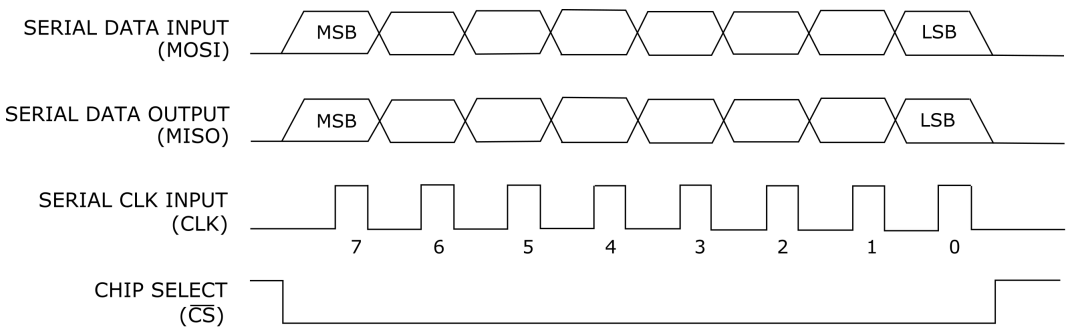

Figura 9.1: Padrão de Comunicação SPI

Para transmitir um dado de um dispositivo **Mestre** para um **Escravo** é necessário que o **Mestre** ative o sinal de CS do **Escravo** e forneça a ele o sinal de clock de referência. Em seguida bit a bit deve ser transmitido pela porta MOSI *Master Output - Slave Input* de ambos os dispositivos.

Quando for necessário transmitir um dado de um **Escravo** para um **Mestre**, novamente o **Mestre** deve ativar o sinal de CS do **Escravo** e fornecer a ele o sinal de clock de referência, porém o dado será transmitido bit a bit pela porta MISO *Master Input - Slave Output* de ambos os dispositivos.

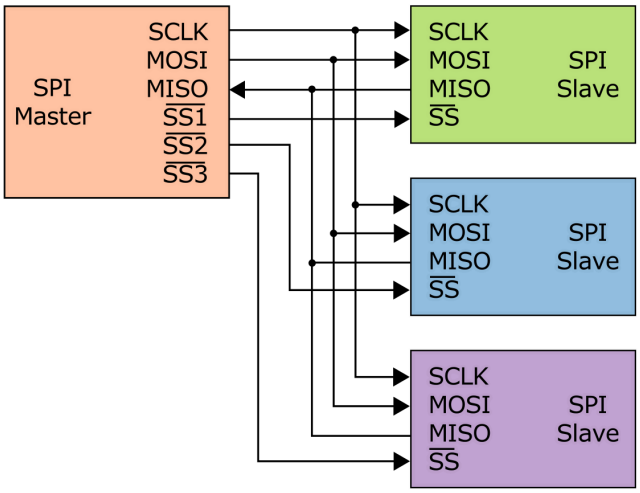

Figura 9.2: Diagrama de Comunicação SPI - Vários Escravos

A figura **??** apresenta um diagrama básico de uma comunicação entre **Mestre** e vários **Escravos** através dos barramentos de dados e de clock em comum.

9.2 SPI do TM4C1294NCPDT

No Tiva - TM4C1294NCPDT a comunicação SPI se dá através do Interface Serial Quad Sincrona (*Quad Synchronous Serial Interface*), ou QSSI. Há 4 módulos QSSI no Tiva, todos capazes de transmitir dados no modo Advanced, Bi-SSI e Quad-SSI.

No modo de transmissão Bi-SSI, dois pinos de dados são ativados para receber ou transmitir; SSInXDAT0 e o SSInXDAT1. Já no modo de transmissão Quad-SSI, quatro pinos são ativados para receber ou transmitir dados; SSInXDAT0, SSInXDAT1, SSInXDAT2 e SSInXDAT4.

No modo de transmissão Advanced, a transmissão se realiza de modo que ao transmitir um dado por um pino, o outro pino fica impossibilitado de receber, e vice-versa.

A forma de transmissão dos módulos QSSI podem ser alterados entre os formatos *Texas Instruments synchronous serial* e *Freescale SPI*. Logo para transmitir em modo SPI basta somente selecionar o modo *Freescale SPI*, podendo utilizar tanto o modo de transmissão Bi- ou Quad-SSI.

No modo de comunicação SPI, tem-se um *baud rate* máximo de 2MHz e uma FIFO para transmissão e outra para recepção ambas com capacidade de 16x8 bits. É possível alternar a fonte de clock de referência da transmissão entre o clock padrão do sistema (SYSCLK) e o clock alternativo (ALTCLK), contando ainda com um divisor de clock de 8 bits, que possibilita dividir o clock de 1 até 254 vezes.

Como é comum na comunicação SPI o Tiva possui portas para transmissão, recepção e clock exclusiva para cada módulo de comunicação. A tabela **??** apresenta as referidas portas para comunicação SPI.

Tabela 9.1: Canais do SPI - Tiva TM4C1294NCPDT [?]

Pino	Mux/Função	Tipo	Buffer	Descrição
SSI0CLK	PA2 (15)	I/O	TTL	SPI Módulo 0, sinal de clock
SSI0XDAT0	PA4 (15)	I/O	TTL	SPI Módulo 0, MISO
SSI0XDAT1	PA5 (15)	I/O	TTL	SPI Módulo 0, MOSI
SSI1CLK	PB5 (15)	I/O	TTL	SPI Módulo 1, sinal de clock
SSI1XDAT0	PE4 (15)	I/O	TTL	SPI Módulo 1, MISO
SSI1XDAT1	PE5 (15)	I/O	TTL	SPI Módulo 1, MOSI
SSI2CLK	PB5 (15)	I/O	TTL	SPI Módulo 2, sinal de clock
SSI2XDAT0	PD1 (15)	I/O	TTL	SPI Módulo 2, MISO
SSI2XDAT1	PD0 (15)	I/O	TTL	SPI Módulo 2, MOSI
SSI3CLK	PQ0 (14) PF3 (14)	I/O	TTL	SPI Módulo 3, sinal de clock
SSI3XDAT0	PQ2 (14) PF1 (14)	I/O	TTL	SPI Módulo 3, MISO
SSI3XDAT1	PQ3 (14) PF0 (14)	I/O	TTL	SPI Módulo 3, MOSI

9.3 Na TivaWare

As principais funções de configuração e utilização da SPI pela TivaWare são listadas a seguir. São utilizadas as funções da SSI, porém são destacadas somente as funções que se referem à SPI.

```
void SSIConfigSetExpClk(uint32_t ui32Base,
                        uint32_t ui32SSIClk,
                        uint32_t ui32Protocol,
                        uint32_t ui32Mode,
                        uint32_t ui32BitRate,
                        uint32_t ui32DataWidth)
```

Configura a interface serial síncrona.

`ui32Base`
: Base da interface serial síncrona a ser configurada. Normalmente **SSI*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

`ui32SSIClk`
: Frequência do clock da comunicação.

`ui32Protocol`
: Protocolo utilizado. No caso da SPI, é utilizado o valor **SSI_FRF_MOTO_MODE_*k***, onde ***k*** assume valores de 0 a 3.

`ui32Mode`
: Valor do modo de operação. Definido no formato **SSI_MODE_*k***, onde ***k*** assume os valores:

  - **MASTER** opera no modo mestre.
  - **SLAVE** opera no modo escravo.
  - **SLAVE_OD** opera no modo escravo com saída desabilitada.

```
void SSIEnable(uint32_t ui32Base)
```

Habilita a interface serial síncrona.

ui32Base
: Base da interface serial síncrona a ser configurada. Normalmente **SSI*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

```
void SSIDisable(uint32_t ui32Base)
```

Desabilita a interface serial síncrona.

ui32Base
: Base da interface serial síncrona a ser configurada. Normalmente **SSI*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

```
void SSIDataGet(uint32_t ui32Base,
                uint32_t *pui32Data)
```

Pega próximo dado recebido pela SSI. Se não houver nada para ser lido, o programa trava e espera até haver um dado.

ui32Base
: Base da SSI a ser lida. Normalmente **SSI*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo lida.

pui32Data
: Ponteiro para endereço de memória já alocada, onde será gravado o dado recebido.

```
int32_t SSIDataGetNonBlocking(uint32_t ui32Base,
                              uint32_t *pui32Data)
```

Pega próximo dado recebido pela SSI. Se não houver nada para ser lido, a leitura é ignorada e o programa continua normalmente. Nesse caso, a função retorna 0 (zero).

ui32Base
: Base da SSI a ser lida. Normalmente **SSI*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo lida.

pui32Data
: Ponteiro para endereço de memória já alocada, onde será gravado o dado recebido.

```
void SSIDataPut(uint32_t ui32Base,
                uint32_t ui32Data)
```

Envia um dado para ser transmitido pela SSI. Se não houver espaço na fila de transmissão, o programa é travado e aguarda até o dado ser colocado na fila.

ui32Base
: Base da SSI a ser lida. Normalmente **SSI*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo lida.

ui32Data
: Dado a ser transmitido pela SSI.

```
int32_t SSIDataPutNonBlocking(uint32_t ui32Base,
                              uint32_t ui32Data)
```

Envia um dado para ser transmitido pela SSI. Se não houver espaço na fila de transmissão, a operação é ignorada e o programa continua normalmente. Nesse caso, a função retorna o valor 0 (zero) e a operação deve ser repetida posteriormente.

ui32Base
: Base da SSI a ser lida. Normalmente **SSI*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo lida.

ucData
: Dado a ser transmitido pela SSI.

```
void SSIIntRegister(uint32_t ui32Base,
                    void (*pfnHandler)(void))
```

Configura a rotina de tratamento de interrupção da SSI especificada.

ui32Base
: Base da SSI a ser configurada. Normalmente **SSI*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo utilizada.

pfnIntHandler
: Ponteiro da função de tratamento. Esta não deve receber nada como parâmetro e nem retornar nada.

```
void SSIIntClear(uint32_t ui32Base,
                 uint32_t ui32IntFlags)
```

Apaga as *flags* de interrupção da SSI especificadas.

ui32Base
: Base da SSI a ser configurada. Normalmente **SSI*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo utilizada.

ui32IntFlags
: Pacote de parâmetros em formato de OU binário que especifica as interrupções da SSI. Cada parâmetro é representado no formato **SSI_*k***, onde ***k*** pode assumir os valores:

  - **TXFF** interrupção acionada pela fila de transmissão.
  - **RXFF** interrupção acionada pela fila de recepção.
  - **RXTO** interrupção acionada pelo estouro de tempo da recepção.

- **RXOR** interrupção acionada por sobrescrita na recepção.

```
void SSIIntEnable(uint32_t ui32Base,
                  uint32_t ui32IntFlags)
```

Habilita as interrupções da SSI especificadas.

ui32Base
: Base da SSI a ser configurada. Normalmente **SSI*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo utilizada.

ui32IntFlags
: Pacote de parâmetros em formato de OU binário que especifica as interrupções da SSI. Cada parâmetro é representado no formato **SSI_*k***, onde ***k*** pode assumir os valores:

  - **TXFF** interrupção acionada pela fila de transmissão.
  - **RXFF** interrupção acionada pela fila de recepção.
  - **RXTO** interrupção acionada pelo estouro de tempo da recepção.
  - **RXOR** interrupção acionada por sobrescrita na recepção.

```
void SSIIntDisable(uint32_t ui32Base,
                   uint32_t ui32IntFlags)
```

Desabilita as interrupções da SSI especificadas.

ui32Base
: Base da SSI a ser configurada. Normalmente **SSI*k*_BASE**, onde ***k*** é o número que identifica a base que está sendo utilizada.

ui32IntFlags
: Pacote de parâmetros em formato de OU binário que especifica as interrupções da SSI. Cada parâmetro é representado no formato **SSI_*k***, onde ***k*** pode assumir os valores:

  - **TXFF** interrupção acionada pela fila de transmissão.
  - **RXFF** interrupção acionada pela fila de recepção.
  - **RXTO** interrupção acionada pelo estouro de tempo da recepção.
  - **RXOR** interrupção acionada por sobrescrita na recepção.

9.4 Exemplo

Uma implementação de configuração da SPI é dada a seguir. Para um exemplo mais abrangente, consulte a Seção **??**.

```
1 // Habilita SSI
2 SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
3
4 // Habilita a GPIO A
5 SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
6
7 // Configura os pinos da comunicacao SPI
```

```
8  //      PA4 - SSI0Rx
9  //      PA3 - SSI0Fss
10 //      PA2 - SSI0CLK
11 GPIOPinConfigure(GPIO_PA2_SSI0CLK);
12 GPIOPinConfigure(GPIO_PA3_SSI0FSS);
13 GPIOPinConfigure(GPIO_PA4_SSI0RX);
14 GPIOPinConfigure(GPIO_PA5_SSI0TX);
15
16 // Configura pinos utilisados na SSI
17 GPIOPinTypeSSI(GPIO_PORTA_BASE,
18     GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 | GPIO_PIN_2);
19
20 // Configuracao da comunicacao
21 SSIConfigSetExpClk(SSI0_BASE, ui32SysClock, SSI_FRF_MOTO_MODE_0
       ,
22                    SSI_MODE_MASTER, 1000000, 8);
23
24 // Habilita SSI
25 SSIEnable(SSI0_BASE);
```

Capítulo 10

Conversor Analógico/Digital ADC

O ADC (*Analog-To-Digital Converter*) é um periférico responsável por realizar a conversão de uma grandeza analógica de tensão para um valor correspondente digital. Para realizar esta conversão pode-se implementar vários tipos circuito, como o conversor flash, ou o conversor de aproximações sucessivas, ou ainda conversor integrador simples ou de rampa dupla. Porém o circuito de conversão mais usado em circuito integrados atualmente é o conversor de aproximações sucessivas, o qual também usado como AD no Tiva TM4C1294NCPDT.

10.1 ADC de Aproximações Sucessivas

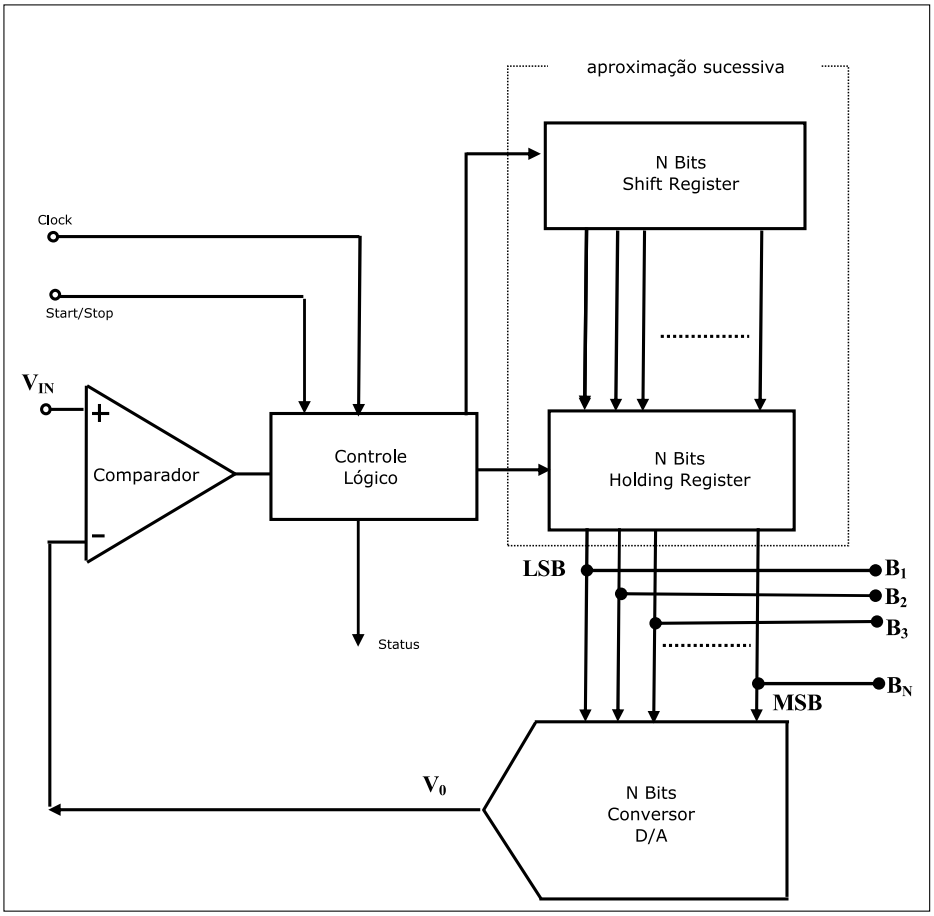

Figura 10.1: Conversor A/D tipo Aproximação Sucessiva

A figura **??** apresenta o diagrama básico de funcionamento de um conversor de aproximações sucessivas. Nota-se que tal conversor utiliza a técnica de realimentação para relacionar uma voltagem analógica de entrada com um código digital, através de um conversor DA (*Digital-To-Analog Converter*) e um comparador. O processo de conversão inicia quando o *Shift Register* e o *Holding Register* são zerados, e então o MSB (*Most Significant Bit*) do *Holding Register* vai para nível alto. Em seguida o comparador relaciona a saída do conversor DA com a tensão V_{IN}. Se $V_O < V_{IN}$ a conversão chega ao fim, porém se isso não for verdade a etapa se repete e MSB vai para nível baixo e o segundo SB vai para nível alto. E assim se dá a conversão.

10.2 ADC do TM4C1294NCPDT

O Tiva TM4C1294NCPDT possui 2 módulos de conversão AD de 12-bit, que podem ser usados em qualquer um das 20 entradas de sinal analógico. É possível realizar amostragens sequenciais entre os canais ou no mesmo canal repetidamente com intervalos de tempo programáveis. Cada módulo AD possui 8 comparadores que possibilitam realizar comparações entre os sinais de entrada e valores pré-definidos, para que assim possa ser realizado as mais diversas operações. Ainda é possível usar um *trigger* diferente para cada um dos módulos, ou usar um *trigger* para acionar ambos os módulos.

A tabela **??** apresenta os pinos de entrada para os módulos ADC0 e ADC1, com a descrição do nome do pino de entrada, o numero referente a este pino, sua função e o tipo de *buffer* usado. Nesta mesma tabela temos o pino chamado de $VREFA+$ que é o pino da tensão de referência usado pelo AD. O $VREFA+$ corresponde o valor máximo que o conversor DA, usado pelo AD para realizar a comparação com a tensão de entrada, pode atingir.

A tensão $VREFA+$ é extremamente importante para se realizar a conversão AD, pois se este valor não for selecionado de forma adequada pode-se acarretar problemas no valor digital. A tensão $VREFA+$ pode ser alternada entre uma fonte de referência interna ou externa.

Pino	n^o	Mux/Função	Tipo	Buffer	Descrição
AIN0	12	PE3	I	Analógico	ADC - Entrada 0
AIN1	13	PE2	I	Analógico	ADC - Entrada 1
AIN2	14	PE1	I	Analógico	ADC - Entrada 2
AIN3	15	PE0	I	Analógico	ADC - Entrada 3
AIN4	128	PD7	I	Analógico	ADC - Entrada 4
AIN5	127	PD6	I	Analógico	ADC - Entrada 5
AIN6	126	PD5	I	Analógico	ADC - Entrada 6
AIN7	125	PD4	I	Analógico	ADC - Entrada 7
AIN8	124	PE5	I	Analógico	ADC - Entrada 8
AIN9	123	PE4	I	Analógico	ADC - Entrada 9
AIN10	121	PB4	I	Analógico	ADC - Entrada 10
AIN11	120	PB5	I	Analógico	ADC - Entrada 11
AIN12	4	PD3	I	Analógico	ADC - Entrada 12
AIN13	3	PD2	I	Analógico	ADC - Entrada 13
AIN14	2	PD1	I	Analógico	ADC - Entrada 14
AIN15	1	PD0	I	Analógico	ADC - Entrada 15
AIN16	18	PK0	I	Analógico	ADC - Entrada 16
AIN17	19	PK1	I	Analógico	ADC - Entrada 17
AIN18	20	PK2	I	Analógico	ADC - Entrada 18
AIN19	21	PK3	I	Analógico	ADC - Entrada 19
VREFA+	9	fixo	-	Analógico	A tensão de referência é usada pelo AD para fixar o valor máximo de conversão.

Tabela 10.1: Canais de Entrada ADC - Tiva TM4C1294NCPDT [?]

10.3 Na TivaWare

As principais funções, da biblioteca TivaWare, responsáveis pela configuração e utilização do ADC, são listadas a seguir.

```
void ADCSequenceConfigure(uint32_t ui32Base,
                          uint32_t ui32SequenceNum,
                          uint32_t ui32Trigger,
                          uint32_t ui32Priority)
```

Configurações básicas do ADC especificado.

ui32Base
: Base do ADC a ser configurado. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

ui32Trigger
: Evento que aciona o amostrador. Definido no formato **ADC_TRIGGER_*k***, onde ***k*** pode assumir um dos valores:

  - **PROCESSOR** amostragem iniciada por comando de software.
  - **COMP0** amostragem iniciada pelo comparador 0 do ADC.
  - **COMP1** amostragem iniciada pelo comparador 1 do ADC.
  - **COMP2** amostragem iniciada pelo comparador 2 do ADC.
  - **EXTERNAL** amostragem iniciada por uma GPIO de entrada configurada.
  - **TIMER** amostragem iniciada pelo temporizador.
  - **PWM0** amostragem iniciada pelo PWM 0.
  - **PWM1** amostragem iniciada pelo PWM 1.
  - **PWM2** amostragem iniciada pelo PWM 2.
  - **PWM3** amostragem iniciada pelo PWM 3.
  - **ALWAYS** amostragem iniciada repetida e continuamente

```
void ADCSequenceStepConfigure(uint32_t ui32Base,
                              uint32_t ui32SequenceNum,
                              uint32_t ui32Step,
                              uint32_t ui32Config)
```

Configura o intervalo de amostragem do ADC especificado.

ui32Base
: Base do ADC a ser configurado. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

ui32Step
: Intervalo de amostragem do ADC.

ui32Config
: Configuração da amostragem. Pacote de OU lógico de valores no formato **ADC_CTL_*k***, onde ***k*** pode assumir os valores:

  - **TS** amostragem do sensor de temperatura interno.
  - **IE** amostragem gera interrupção.
  - **END** amostragem por sequencia e seleção.
  - **D** amostragem por seleção diferencial.

- **CH*k*** seleciona canal de entrada como canal ***k***, onde ***k*** assume valores de 0 a 23.
- **CMP*k*** seleciona comparador ***k*** para ser utilizado, onde ***k*** assume valores de 0 a 7.

```
void ADCSequenceEnable(uint32_t ui32Base,
                       uint32_t ui32SequenceNum)
```

Habilita a sequência de amostragem.

ui32Base
: Base do ADC a ser configurado. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

```
void ADCSequenceDisable(uint32_t ui32Base,
                        uint32_t ui32SequenceNum)
```

Desabilita a sequência de amostragem.

ui32Base
: Base do ADC a ser configurado. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

```
int32_t ADCSequenceDataGet(uint32_t ui32Base,
                           uint32_t ui32SequenceNum,
                           uint32_t *pui32Buffer)
```

Pega o valor gerado na amostragem.

ui32Base
: Base do ADC a ser lido. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

pui32Buffer
: Ponteiro para uma região de memória alocada. Onde será armazenado o valor lido.

```
int32_t ADCSequenceDataGet(uint32_t ui32Base,
                           uint32_t ui32SequenceNum,
                           uint32_t *pui32Buffer)
```

Pega o valor gerado na amostragem.

ui32Base
: Base do ADC a ser lido. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

pui32Buffer
: Ponteiro para uma região de memória alocada. Onde será armazenado o valor lido.

```
int32_t ADCSequenceOverflow(uint32_t ui32Base,
                            uint32_t ui32SequenceNum)
```

Informa se houve uma perda de leitura, antes de ter lido o valor antigo ocorreu uma nova amostragem.

ui32Base
: Base do ADC a ser lido. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

```
void ADCProcessorTrigger(uint32_t ui32Base,
                         uint32_t ui32SequenceNum)
```

Causa uma leitura do ADC invocada pelo processador. Gatilho por software.

ui32Base
: Base do ADC a ser chamado. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

```
void ADCIntRegister(uint32_t ui32Base,
                    uint32_t ui32SequenceNum,
                    void (*pfnHandler)(void))
```

Configura rotina de tratamento de interrupção do ADC.

ui32Base
: Base do ADC a ser configurada. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

pfnIntHandler
: Ponteiro da função de tratamento. Esta não deve receber nada como parâmetro e nem retornar nada.

```
void ADCIntEnable(uint32_t ui32Base,
                  uint32_t ui32SequenceNum)
```

Habilita as interrupções do ADC.

ui32Base
: Base do ADC a ser configurada. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

```
void ADCIntDisable(uint32_t ui32Base,
                   uint32_t ui32SequenceNum)
```

Habilita as interrupções do ADC.

ui32Base
: Base do ADC a ser configurada. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

```
void ADCIntClear(uint32_t ui32Base,
                 uint32_t ui32SequenceNum)
```

Limpa a *flag* de interrupções do ADC.

ui32Base
: Base do ADC a ser configurada. Normalmente **ADC*k*_BASE**, onde ***k*** é a letra identificadora do periférico.

ui32SequenceNum
: Número da sequência de amostragem.

10.4 Exemplo

A seguir, é apresentado um código de configuração do ADC. Um exemplo mais elaborado é apresentado na Seção **??**.

```
1 // Habilita ADC0
2 MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
3 // Aguarda 3 SysCtlDelay. Aproximadamente 10 ciclos de clock
4 MAP_SysCtlDelay(3);
5
6 // Desabilitar Interrupcao do ADC para configura-la
7 MAP_IntDisable(INT_ADC0SS0);
8 MAP_ADCIntDisable(ADC0_BASE, 0);
9 MAP_ADCSequenceDisable(ADC0_BASE, 0);
```

```
10
11 //Configurando ADC
12 MAP_ADCHardwareOversampleConfigure(ADC0_BASE, 4);
13 MAP_ADCSequenceConfigure(ADC0_BASE, 0,
14     ADC_TRIGGER_PROCESSOR, 0);
15 MAP_ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
16     ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
17 MAP_ADCSequenceEnable(ADC0_BASE, 0);
18
19 // Habilitando Interrupcao do ADC
20 MAP_ADCIntClear(ADC0_BASE, 0);
21 MAP_ADCIntEnable(ADC0_BASE, 0);
22 MAP_IntEnable(INT_ADC0SS0);
```

Capítulo 11

Modulação por largura de Pulso (PWM)

PWM (Pulse Width Modulation) é um modulação baseada na conversão linear de um valor em escala de tensão para outro em escala de *Duty Cicle* aplicado a uma onda quadrada de amplitude qualquer. Este tipo de modulação é utilizada em diversas aplicações eletrônicas.

11.1 Modo de Funcionamento

Para criar uma modulação PWM é necessário criar uma onda periódica linear e variante no tempo, que se consiga comparar com o sinal que se deseja converter em PWM. Logo o tipo de onda necessária neste modulação é a onda triangular ou a onda dente de serra. Para melhor compreensão a onda triangular ou dente de serra será denominada aqui de onda portadora, e o sinal ao qual se deseja converter em PWM será denominado sinal modulante.

Para a implementação digital deste método pode ser feito através uma comparação direta entre os módulos da onda portadora e o sinal modulante. Quando o modulo do sinal modulante for maior do que o módulo da portadora o sinal modulado vai para nível alto. Porém quando o módulo do sinal modulante for menor do que o módulo da portadora o sinal modulado vai para zero. A grande diferença da implementação digital é que o sinal da portadora é gerado internamente através de um contador, que pode trabalhar no modo de contagem *Down*, *Up* ou *UpDown*.

Quando o sinal da portadora é gerado através de um contador *Down* ou *Up* a frequência do sinal modulado será igual a frequência do contador dividido pelo numero máximo de contagens. Já quando o sinal da portadora é gerador por um contador *UpDown* a frequência do sinal modulado será igual a frequência do contador dividido por duas vezes o numero máximo de contagens. Sendo que o numero máximo de contagens deve ser maior ou igual ao modulo do sinal modulante. A Figura **??** e a Figura **??** apresentam melhor o modo como a geração de PWM é realizada digitalmente.

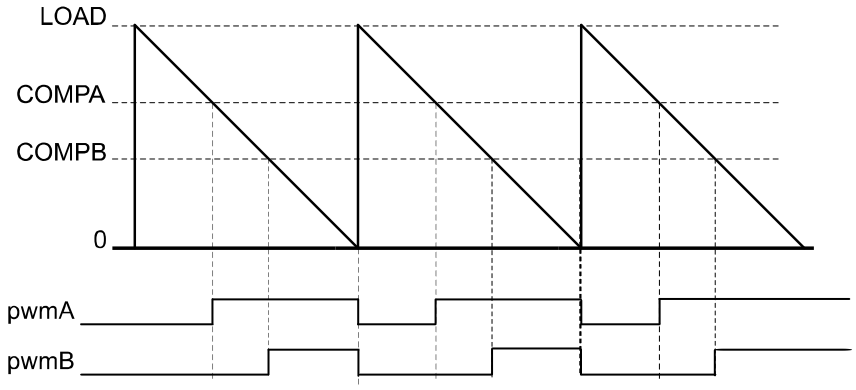

Figura 11.1: PWM modo Down [**?**]

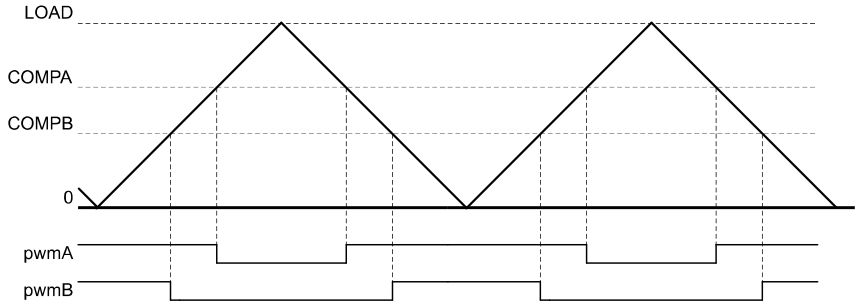

Figura 11.2: PWM modo Up-Down [**?**]

Tanto na Figura **??** quanto na Figura **??** os valores de *LOAD*, *COMPA*, *COMPB*, *pwmA* e *pwmB* são referentes aos registradores presente do Tiva - TM4C1294NCPD, responsáveis pela modulação PWM. Tais registradores serão melhor abordados na próxima seção.

11.2 PWM do TM4C1294NCPDT

O Tiva - TM4C1294NCPD possui um módulo PWM com quatro blocos geradores e seus respectivos blocos de controle, disponibilizando oito saídas PWM. Através dos blocos de controle é possível escolher qual a polaridade de cada sinal PWM, e o seu respectivo pino. Cada bloco gerador produz 2 sinais PWM com a mesma frequência, porém ambos podem ter *Duty Cicle* independentes ou *Duty Cicle* complementares, com uma intervalo de *Dead Band*.

Como a maioria das aplicações com PWM é destinada ao chaveamento, o Tiva - TM4C1294NCPD possui não só uma configuração de geração PWM complementar com *Dead Band*, recurso essencial para acionamento de pontes H, como também possui 4 pinos de entrada para um sistema de controle de falha, um para cada gerador PWM.

Para gerar a onda portadora o Tiva possui um contador de 16 bits capaz de realizar contagens no modo *Down* e *UpDown*, sendo possível atualizar o valor da contagem máxima (*LOAD*). Cada um dos geradores PWM possuem ainda dois comparadores distintos (*COMPA* e *COMPB*), responsáveis por gerar os sinais PWM e que podem ser usados para gerar interrupções.

Quando um comparador está configurado para provocar interrupções esta ocorre toda vez que o valor do comparador selecionado for maior do que o valor de *LOAD*. A Figura **??** demonstra o modo como as os sinais de interrupção são provocados pelos comparadores no modo *Down*, e a Figura **??** demonstra o mesmo no modo *UpDown*.

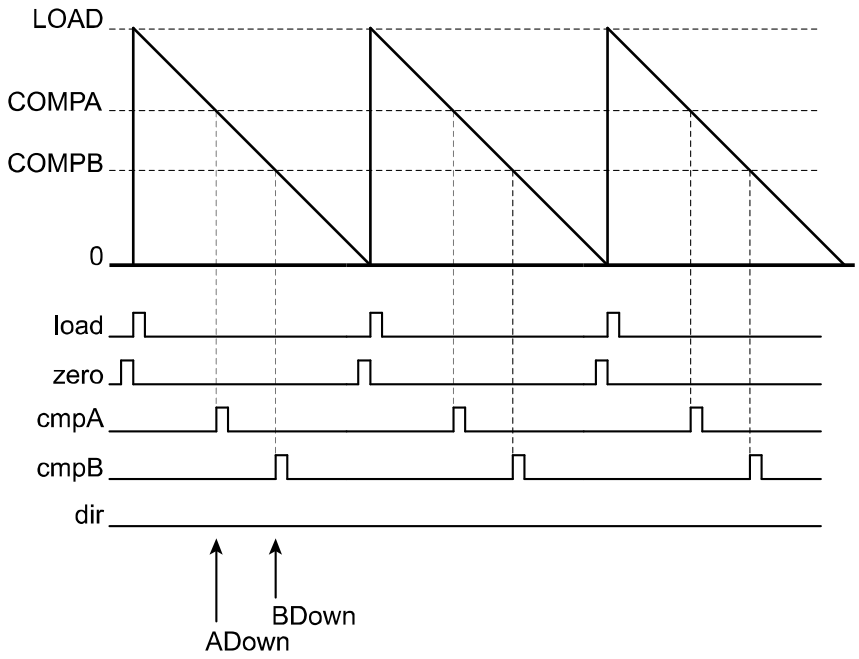

Figura 11.3: Interrupções do PWM modo Down [**?**]

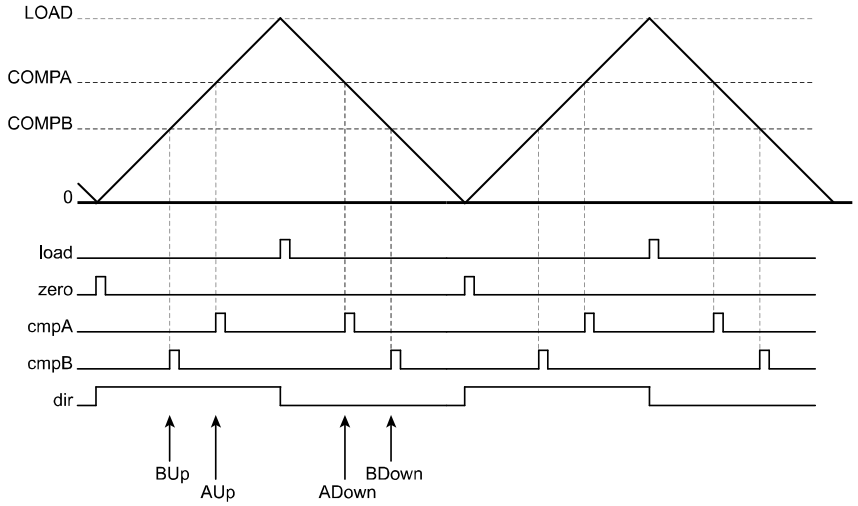

Figura 11.4: Interrupções do PWM modo Up-Down [**?**]

A tabela **??** apresenta os 8 pinos do módulo PWM, sendo estes pinos de saída dos sinais de PWM.

Pino	n^o	Mux/Função	Tipo	Descrição
M0PWM0	42	PF0 (6)	O	Saída PWM 0
M0PWM1	43	PF1 (6)	O	Saída PWM 1
M0PWM2	44	PF2 (6)	O	Saída PWM 2
M0PWM3	45	PF3 (6)	O	Saída PWM 3
M0PWM4	49	PG0 (6)	O	Saída PWM 4
M0PWM5	50	PG1 (6)	O	Saída PWM 5

M0PWM6	63	PK4 (6)	O	Saída PWM 6
M0PWM7	62	PK5 (6)	O	Saída PWM 7

Tabela 11.1: Canais PWM - Tiva TM4C1294NCPDT [?]

11.3 Na TivaWare

As principais funções de configuração e utilização dos geradores de PWM e suas interrupções são listadas a seguir.

```
void PWMClockSet(uint32_t ui32Base,
                 uint32_t ui32Config)
```

Configura a frequência do oscilador que alimenta a base de PWM especificada.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32Config
: Divisão do clock do sistema, para a alimentação do gerador do PWM. É definida no formato **PWM_SYSCLK_DIV_*k***, onde ***k*** pode ser um dos valores: 1, 2, 4, 8, 16, 32 ou 64.

```
void PWMGenConfigure(uint32_t ui32Base,
                     uint32_t ui32Gen,
                     uint32_t ui32Config)
```

Configura o gerador de PWM especificado.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é o valor identificador do gerador.

ui32Gen
: Gerador a ser configurado. Definido por **PWM_GEN_*k***, onde ***k*** pode ser um dos valores: 0, 1, 2 ou 3.

ui32Config
: Pacote de parâmetros em formato de OU lógico, onde cada parâmetro é representado no formato **PWM_GEN_MODE_*k*** e, k pode assumir os valores:

  - **DOWN** ou **UP_DOWN** para especificar o modo do contador.
  - **SYNC** ou **NO_SYNC** para especificar o modo de carregamento do contador e do comparador.
  - **DBG_RUN** ou **DBG_STOP** para especificar o comportamento em tempo de depuração.
  - **GEN_NO_SYNC**, **GEN_SYNC_LOCAL** ou **GEN_SYNC_GLOBAL** para especificar o modo de sincronização do contador do gerador.

- **DB_NO_SYNC**, **DB_SYNC_LOCAL** ou **DB_SYNC_GLOBAL** para especificar o modo de sincronização do parâmetro de *deadband*.
- **FAULT_LATCHED** ou **FAULT_UNLATCHED** para especificar se falhas serão travadas ou não.
- **MINPER** ou **NO_MINPER** para especificar se há ou não um período mínimo de falha.
- **FAULT_EXT** ou **FAULT_LEGACY** para especificar ou não o uso do suporte de seleção de fonte de falha estendida.

```
void PWMGenEnable(uint32_t ui32Base,
                  uint32_t ui32Gen)
```

Habilita o contador/gerador da base de PWM especificado.

ui32Base
: Base do gerador a ser habilitado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32Gen
: Gerador a ser habilitado. Definido por **PWM_GEN_*k***, onde ***k*** pode ser um dos valores: 0, 1, 2 ou 3.

```
void PWMGenDisable(uint32_t ui32Base,
                   uint32_t ui32Gen)
```

Desabilita o contador/gerador da base de PWM especificado.

ui32Base
: Base do gerador a ser desabilitado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32Gen
: Gerador a ser desabilitado. Definido por **PWM_GEN_*k***, onde ***k*** pode ser um dos valores: 0, 1, 2 ou 3.

```
void PWMGenIntTrigEnable(uint32_t ui32Base,
                         uint32_t ui32Gen,
                         uint32_t ui32IntTrig)
```

Configura os eventos que causam as interrupções no gerador da base de PWM especificada.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32Gen
: Gerador a ser configurado. Definido por **PWM_GEN_*k***, onde ***k*** pode ser um dos valores: 0, 1, 2 ou 3.

ui32Ints
: Pacote de parâmetros em formato de OU binário que especificam as interrupções do PWM. Cada parâmetro é representado no formato **PWM_INT_CNT_*k***, onde ***k*** pode assumir os valores:

  - **ZERO** interrupção acionada ao zerar o contador.
  - **LOAD** interrupção acionada ao chegar ao valor máximo do contador
  - **AU** interrupção acionada quando o contador é incrementado e alcança o valor especificado no comparador A.
  - **AD** interrupção acionada quando o contador é decrementado e alcança o valor especificado no comparador A.
  - **BU** interrupção acionada quando o contador é incrementado e alcança o valor especificado no comparador B.
  - **BD** interrupção acionada quando o contador é decrementado e alcança o valor especificado no comparador B.

```
void PWMGenIntClear(uint32_t ui32Base,
                    uint32_t ui32Gen,
                    uint32_t ui32Ints)
```

Limpa as *flags* que marcam a ocorrência das interrupções especificadas no gerador da base de PWM especificado.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32Gen
: Gerador a ser configurado. Definido por **PWM_GEN_*k***, onde ***k*** pode ser um dos valores: 0, 1, 2 ou 3.

ui32Ints
: Pacote de parâmetros em formato de OU binário que especificam as interrupções do PWM. Cada parâmetro é representado no formato **PWM_INT_CNT_*k***, onde ***k*** pode assumir os valores:

  - **ZERO** interrupção acionada ao zerar o contador.
  - **LOAD** interrupção acionada ao chegar ao valor máximo do contador
  - **AU** interrupção acionada quando o contador é incrementado e alcança o valor especificado no comparador A.
  - **AD** interrupção acionada quando o contador é decrementado e alcança o valor especificado no comparador A.
  - **BU** interrupção acionada quando o contador é incrementado e alcança o valor especificado no comparador B.
  - **BD** interrupção acionada quando o contador é decrementado e alcança o valor especificado no comparador B.

```
void PWMGenIntRegister(uint32_t ui32Base,
                       uint32_t ui32Gen,
                       void (*pfnIntHandler)(void))
```

Configura a rotina de tratamento de interrupção do gerador da base de PWM especificada.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32Gen
: Gerador a ser configurado. Definido por **PWM_GEN_*k***, onde ***k*** pode ser um dos valores: 0, 1, 2 ou 3.

pfnIntHandler
: Ponteiro da função de tratamento. Esta não deve receber nada como parâmetro e nem retornar nada.

```
void PWMGenPeriodSet(uint32_t ui32Base,
                     uint32_t ui32Gen,
                     uint32_t ui32Period)
```

Configura o período do sinal no gerador da base de PWM especificada.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32Gen
: Gerador a ser configurado. Definido por **PWM_GEN_*k***, onde ***k*** pode ser um dos valores: 0, 1, 2 ou 3.

ui32Period
: Período em formato de valor do contador entre 0 e 2^n- 1, onde n é o número de bits do contador. Dado pela razão entre a frequência do clock do gerador e a frequência desejada para o PWM.

```
void PWMIntEnable(uint32_t ui32Base,
                  uint32_t ui32GenFault)
```

Habilita as interrupções para o gerador da base de PWM especificada.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32GenFaults
: Pacote de parâmetros em formato de OU binário que especificam os geradores que causam interrupções do PWM. Cada parâmetro é representado no formato **PWM_INT_GEN_*k***, onde ***k*** pode assumir os valores: 0, 1, 2, 3.

```
void PWMIntDisable(uint32_t ui32Base,
                   uint32_t ui32GenFault)
```

Desabilita as interrupções para o gerador da base de PWM especificada.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32GenFaults
: Pacote de parâmetros em formato de OU binário que especificam os geradores que causam interrupções do PWM. Cada parâmetro é representado no formato **PWM_INT_GEN_*k***, onde ***k*** pode assumir os valores: 0, 1, 2, 3.

```
void PWMPulseWidthSet(uint32_t ui32Base,
                      uint32_t ui32PWMOut,
                      uint32_t ui32Width)
```

Desabilita as interrupções para o gerador da base de PWM especificada.

ui32Base
: Base do gerador a ser configurado. Normalmente **PWM*k*_BASE**, onde ***k*** é a letra identificadora do gerador.

ui32PWMOut
: Valor que representa a saída de PWM a ser configurada. Representado no formato **PWM_OUT_*k***, onde ***k*** pode assumir valores de 0 à 7.

ui32Width
: Valor do contador em que o sinal ficará em nível alto.

11.4 Exemplo

Um exemplo de configuração de PWM, utilizando a saída 0 do gerador 0 é apresentada a seguir. É gerada uma onda quadrada de 10 KHz. Outro exemplo, utilizando a interrupção do PWM, pode ser encontrado na Seção **??**.

```
1   // Configura clock do sistema de 120 MHz
2   SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
3       SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);
4   // Habilita o PWM 0
5   SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
6   // Habilita a porta F utilizada no pino de saida de PWM
7   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
8
9   // Habilita pino de saida1 do PWM 0
10  GPIOPinConfigure(GPIO_PF1_M0PWM1);
11  // Configura pino F1 como saida do PWM
12  GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1);
13
14  // PWM no gerador 0 em modo regressivo, nao sincronizado
15  PWMGenConfigure(
16      PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN |
17      PWM_GEN_MODE_NO_SYNC);
18  // CLock do gerador e o do sistema dividido por 4
19  PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_4);
```

```
20      // Periodo de 10 KHz -> 120000000 / 4 / 10000
21      PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, 3000);
22      // Largura de pulso de 50% -> 3000 * 0.5
23      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, 1500);
24      // Habilita gerador 0
25      PWMGenEnable(PWM0_BASE, PWM_GEN_0);
26      // Saidas de PWM a serem modificadas
27      PWMOutputState(PWM0_BASE,
28          (PWM_OUT_0_BIT | PWM_OUT_1_BIT), true);
29
30      // Habilita interrupcao do gerador 0
31      PWMIntEnable(PWM0_BASE, PWM_INT_GEN_0);
32      // Habilita interrupcao do PWM 0
33      IntEnable(INT_PWM0_0);
34      // Interrupcao disparada no estouro do contador
35      PWMGenIntTrigEnable(PWM0_BASE, PWM_GEN_0,
36          PWM_INT_CNT_LOAD);
```

Capítulo 12

Temporizador de Propósito Geral

Temporizadores em microcontroladores são usados como contadores de intervalo de tempo para eventos internos e externos ao DSP. Tais temporizadores possibilitam provocar interrupções a tempos programáveis, sendo esse recurso essencial em muitas aplicações.

12.1 GPTM no TM4C1294NCPDT

O temporizador de propósito geral, ou *General-Purpose Timers* (GPTM), do Tiva - TM4C129NCPDT possui blocos de contadores de 16/32 bits. Em cada bloco há dois contadores de 16 bits, sendo um do *Timer A* e o outro do *Timer B*, que podem ser concatenados em apenas um contador de 32 bits.

Qualquer temporizador do Tiva é capaz de ser usado como um gatilho para conversões do ADC. Porém não se pode usar mais de um temporizador como gatilho para o ADC.

O GPTM é o recurso temporizador padrão no Tiva, porém ele não é o único temporizador presente. Existem os temporizadores do módulo PWM, do módulo *WatchDog Timer*, e do módulo *Systick*, que também podem ser usados de modo parecido ao GPTM.

Dentro do GPTM há 8 blocos de contadores de 16/32 bits. A fonte de clock para os contadores pode ser selecionada entre o clock do sistema ou o ALTCLK (*Alternate CLock*). Tendo em vista que o ALTCLK pode ser proveniente do PIOSC, clock do módulo de hibernação ou o oscilador de baixa frequência (LFIOSC). As funcionalidades básicas dos contadores de cada bloco são:

- Provocar uma única interrupção após intervalo de tempo programável, com contador de 16/32 bits.
- Provocar interrupções periódicas a intervalo de tempo programável, com contador de 16/32 bits.
- Clock de tempo real utilizando clock externo de 32,768 kHz, com contador de 32 bits
- Captura de intervalo de tempo através de detecção de borda de sinal externo com divisor de clock de 8 bits, com contador de 16 bits.
- Modo PWM, com divisor de clock de 8 bits, com contador de 16 bits.
- Contador do modo contagem crescente ou decrescente.
- Modo de captura e comparação, com contador de 16/32 bits.
- Encadeamento de temporizadores para múltiplos disparos de interrupção.
- Gatilho para disparo do ADC.

A Tabela **??** apresenta os pinos de entrada e saída do GPTM para os modos de funcionamento captura, comparação e geração de PWM.

Pino	n^o	Mux/Função	Tipo	Descrição
T0CCP0	1 33 85	PD0 (3) PA0 (3) PL4 (3)	I/O	Timer 0 Captura/Comparação/PWM 0
T0CCP1	2 34 86	PA1 (6) PL5 (3) PL5 (3)	I/O	Timer 0 Captura/Comparação/PWM 1
T1CCP0	3 35 94	PD2 (3) PA2 (3) PL6 (3)	I/O	Timer 1 Captura/Comparação/PWM 0
T1CCP1	4 36 93	PD3 (3) PA3 (3) PL7 (3)	I/O	Timer 1 Captura/Comparação/PWM 1
T2CCP0	37 78	PA4 (3) PM0 (3)	I/O	Timer 2 Captura/Comparação/PWM 0
T2CCP1	38 77	PA5 (3) PM1 (3)	I/O	Timer 2 Captura/Comparação/PWM 1
T3CCP0	40 76 125	PA6 (3) PM2 (3) PD4 (3)	I/O	Timer 3 Captura/Comparação/PWM 0
T3CCP1	41 75 126	PA7 (3) PM3 (3) PD5 (3)	I/O	Timer 3 Captura/Comparação/PWM 1
T4CCP0	74 95 127	PM4 (3) PB0 (3) PD6 (3)	I/O	Timer 4 Captura/Comparação/PWM 0
T4CCP1	73 96 128	PM5 (3) PB1 (3) PD7 (3)	I/O	Timer 4 Captura/Comparação/PWM 1
T5CCP0	72 91	PM6 (3) PB2 (3)	I/O	Timer 5 Captura/Comparação/PWM 0
T5CCP1	71 92	PM7 (3) PB3 (3)	I/O	Timer 5 Captura/Comparação/PWM 1

Tabela 12.1: Canais Temporizador de Propósito Geral - Tiva TM4C1294NCPDT [**?**]

12.2 Na TivaWare

As principais funções responsáveis pela configuração e utilização do temporizador de propósito geral, presentes na TivaWare, estão listadas a seguir.

```
void TimerConfigure(uint32_t ui32Base,
                    uint32_t ui32Config)
```

Configura o temporizador especificado.

ui32Base
: Base do temporizador a ser configurado. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Config
: Pacote de parâmetros em formato de OU binário que configura o modo de operação do temporizador especificado. Cada parâmetro é representado no formato **TIMER_CFG_*k***, para o caso de ser utilizado a base como um único temporizador, ou, **TIMER_CFG_A_*k*** ou **TIMER_CFG_B_*k*** para o caso de ser utilizado como dois temporizadores separados. Onde ***k*** pode assumir os valores:

  - **ONE_SHOT** temporizador de um disparo.
  - **ONE_SHOT_UP** temporizador de um disparo, com contador incremental.
  - **PERIODIC** temporizador periódico.
  - **PERIODIC_UP** temporizador periódico, com contador incremental.
  - **RTC** temporizador com *clock* de tempo real.
  - **SPLIT_PAIR** dois temporizador de meia largura.
  - **CAP_COUNT** temporizador com contador por captura de borda.
  - **CAP_COUNT_UP** temporizador com contador por captura de borda, com contador incremental.
  - **CAP_TIME** temporizador com tempo por captura de borda.
  - **CAP_TIME_UP** temporizador com tempo por captura de borda, com contador incremental.
  - **CAP_PWM** temporizador com saída de PWM.

```
void TimerEnable(uint32_t ui32Base,
                 uint32_t ui32Timer)
```

Habilita a contagem no temporizador especificado.

ui32Base
: Base do temporizador a ser habilitado. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica quais temporizadores serão habilitados. Podendo assumir um de 3 valores:

  - **TIMER_A** para habilitar somente o temporizador A.
  - **TIMER_B** para habilitar somente o temporizador B.
  - **TIMER_BOTH** para habilitar ambos os temporizadores.

```
void TimerDisable(uint32_t ui32Base,
                  uint32_t ui32Timer)
```

Desabilita a contagem no temporizador especificado.

ui32Base
: Base do temporizador a ser desabilitado. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica quais temporizadores serão desabilitados. Podendo assumir um de 3 valores:

  - **TIMER_A** para habilitar somente o temporizador A.
  - **TIMER_B** para habilitar somente o temporizador B.
  - **TIMER_BOTH** para habilitar ambos os temporizadores.

```
void TimerControlLevel(uint32_t ui32Base,
                       uint32_t ui32Timer,
                       bool bInvert)
```

Controla o nível de saída do temporizador.

ui32Base
: Base do temporizador a ser configurado. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica quais temporizadores serão configurados. Podendo assumir um de 3 valores:

  - **TIMER_A** para habilitar somente o temporizador A.
  - **TIMER_B** para habilitar somente o temporizador B.
  - **TIMER_BOTH** para habilitar ambos os temporizadores.

bInvert
: Se *true*, o sinal de saída é dado por nível baixo, caso contrário, por nível baixo.

```
void TimerControlTrigger(uint32_t ui32Base,
                         uint32_t ui32Timer,
                         bool bEnable)
```

Habilita ou desabilita o gatilho de saída do temporizador.

ui32Base
: Base do temporizador a ser configurado. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica quais temporizadores serão configurados. Podendo assumir um de 3 valores:

  - **TIMER_A** para habilitar somente o temporizador A.
  - **TIMER_B** para habilitar somente o temporizador B.
  - **TIMER_BOTH** para habilitar ambos os temporizadores.

bEnable
: Se *true*, o sinal de gatilho do sinal de saída é habilitado, caso contrário, não.

```
void TimerControlEvent(uint32_t ui32Base,
                       uint32_t ui32Timer,
                       uint32_t ui32Event)
```

Controla o tipo do evento de saída do temporizador.

ui32Base
: Base do temporizador a ser configurado. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica quais temporizadores serão configurados. Podendo assumir um de 3 valores:

  - **TIMER_A** para habilitar somente o temporizador A.
  - **TIMER_B** para habilitar somente o temporizador B.
  - **TIMER_BOTH** para habilitar ambos os temporizadores.

ui32Event
: Valor que especifica o tipo do evento. É definido no formato **TIMER_EVENT_*k***, onde ***k*** pode assumir os valores:

  - **POS_EDGE** para evento disparado na borda positiva.
  - **NEG_EDGE** para evento disparado na borda negativa.
  - **BOTH_EDGES** para evento disparado em ambas as bordas.

```
void TimerLoadSet(uint32_t ui32Base,
                  uint32_t ui32Timer,
                  uint32_t ui32Value)
```

Configura o valor máximo do temporizador.

ui32Base
: Base do temporizador a ser configurado. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica quais temporizadores serão configurados. Podendo assumir um de 3 valores:

  - **TIMER_A** para habilitar somente o temporizador A.
  - **TIMER_B** para habilitar somente o temporizador B.
  - **TIMER_BOTH** para habilitar ambos os temporizadores.

ui32Value
: Valor a ser carregado no temporizador.

```
uint32_t TimerLoadGet(uint32_t ui32Base,
                      uint32_t ui32Timer)
```

Lê o valor máximo do temporizador.

ui32Base
: Base do temporizador a ser lido. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica qual temporizador será lido. Podendo assumir um de 3 valores:

- **TIMER_A** para habilitar somente o temporizador A.
- **TIMER_B** para habilitar somente o temporizador B.
- **TIMER_BOTH** para habilitar ambos os temporizadores.

```
void TimerMatchSet(uint32_t ui32Base,
                   uint32_t ui32Timer,
                   uint32_t ui32Value)
```

Configura o valor de comparação carregado no temporizador.

ui32Base
: Base do temporizador a ser lido. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica qual temporizador será lido. Podendo assumir um de 3 valores:

- **TIMER_A** para habilitar somente o temporizador A.
- **TIMER_B** para habilitar somente o temporizador B.
- **TIMER_BOTH** para habilitar ambos os temporizadores.

ui32Valor
: Valor no qual o contador irá parar ou, atualizar sua saída.

```
uint32_t TimerValueGet(uint32_t ui32Base,
                       uint32_t ui32Timer)
```

Lê o valor atual do contador do temporizador.

ui32Base
: Base do temporizador a ser lido. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica qual temporizador será lido. Podendo assumir um de 3 valores:

- **TIMER_A** para habilitar somente o temporizador A.
- **TIMER_B** para habilitar somente o temporizador B.

- **TIMER_BOTH** para habilitar ambos os temporizadores.

```
void TimerIntRegister(uint32_t ui32Base,
                      uint32_t ui32Timer,
                      void (*pfnHandler)(void))
```

Configura a rotina de tratamento de interrupção do temporizador.

ui32Base
: Base do temporizador a ser lido. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32Timer
: Valor que especifica qual temporizador será lido. Podendo assumir um de 3 valores:

  - **TIMER_A** para habilitar somente o temporizador A.
  - **TIMER_B** para habilitar somente o temporizador B.
  - **TIMER_BOTH** para habilitar ambos os temporizadores.

pfnIntHandler
: Ponteiro da função de tratamento. Esta não deve receber nada como parâmetro e nem retornar nada.

```
void TimerIntEnable(uint32_t ui32Base,
                    uint32_t ui32IntFlags)
```

Habilita as interrupções do temporizador.

ui32Base
: Base do temporizador a ser lido. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32IntFlags
: Pacote de parâmetros em formato de OU binário que especifica os eventos que podem causar interrupção. Cada parâmetro é definido no formato **TIMER_*k***,onde ***k*** pode assumir os valores:

  - **TIMB_DMA** quando o uDMA do temporizador B estiver completo.
  - **TIMA_DMA** quando o uDMA do temporizador A estiver completo.
  - **CAPA_EVENT** interrupção por evento de captura do temporizador A.
  - **CAPA_MATCH** interrupção por comparação de captura do temporizador A.
  - **TIMA_TIMEOUT** interrupção por estouro de tempo do temporizador A.
  - **CAPB_EVENT** interrupção por evento de captura do temporizador B.
  - **CAPB_MATCH** interrupção por comparação de captura do temporizador B.
  - **TIMB_TIMEOUT** interrupção por estouro de tempo do temporizador B.
  - **RTC_MATCH** interrupção pela máscara do RTC.

```
void TimerIntDisable(uint32_t ui32Base,
                     uint32_t ui32IntFlags)
```

Habilita as interrupções do temporizador.

ui32Base
: Base do temporizador a ser lido. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32IntFlags
: Pacote de parâmetros em formato de OU binário que especifica os eventos que podem causar interrupção. Cada parâmetro é definido no formato **TIMER_*k***,onde ***k*** pode assumir os valores:

  - **TIMB_DMA** quando o uDMA do temporizador B estiver completo.
  - **TIMA_DMA** quando o uDMA do temporizador A estiver completo.
  - **CAPA_EVENT** interrupção por evento de captura do temporizador A.
  - **CAPA_MATCH** interrupção por comparação de captura do temporizador A.
  - **TIMA_TIMEOUT** interrupção por estouro de tempo do temporizador A.
  - **CAPB_EVENT** interrupção por evento de captura do temporizador B.
  - **CAPB_MATCH** interrupção por comparação de captura do temporizador B.
  - **TIMB_TIMEOUT** interrupção por estouro de tempo do temporizador B.
  - **RTC_MATCH** interrupção pela máscara do RTC.

```
void TimerIntClear(uint32_t ui32Base,
                   uint32_t ui32IntFlags)
```

Limpa as *flags* de interrupção do temporizador.

ui32Base
: Base do temporizador a ser lido. Normalmente **TIMER*k*_BASE**, onde ***k*** é o valor identificador do temporizador.

ui32IntFlags
: Pacote de parâmetros em formato de OU binário que especifica os eventos que podem causar interrupção. Cada parâmetro é definido no formato **TIMER_*k***,onde ***k*** pode assumir os valores:

  - **TIMB_DMA** quando o uDMA do temporizador B estiver completo.
  - **TIMA_DMA** quando o uDMA do temporizador A estiver completo.
  - **CAPA_EVENT** interrupção por evento de captura do temporizador A.
  - **CAPA_MATCH** interrupção por comparação de captura do temporizador A.
  - **TIMA_TIMEOUT** interrupção por estouro de tempo do temporizador A.
  - **CAPB_EVENT** interrupção por evento de captura do temporizador B.
  - **CAPB_MATCH** interrupção por comparação de captura do temporizador B.
  - **TIMB_TIMEOUT** interrupção por estouro de tempo do temporizador B.
  - **RTC_MATCH** interrupção pela máscara do RTC.

12.3 Exemplo

Um exemplo simples de configuração do temporizador é mostrado a seguir. Na Seção **??** é apresentado um exemplo mais aprofundado sobre a configuração e utilização do temporizador.

```
1  // Configura o Temporizador A como um temporizador de disparo
       unico,
2  // e o temporizador B como um contador por captura de borda
3  TimerConfigure(TIMER0_BASE, (TIMER_CFG_SPLIT_PAIR |
       TIMER_CFG_A_ONE_SHOT |
4       TIMER_CFG_B_CAP_COUNT));
5
6  // Configura o contador do temporizador A como de diparo unico
7  TimerLoadSet(TIMER0_BASE, TIMER_A, 3000);
8
9  // Configura o contador do temporizador B para contar em ambas
       as bordas
10 TimerControlEvent(TIMER0_BASE, TIMER_B, TIMER_EVENT_BOTH_EDGES)
       ;
11
12 // Habilita os temporizadores
13 TimerEnable(TIMER0_BASE, TIMER_BOTH);
```

Capítulo 13

Exemplos de aplicação

Esta seção apresenta alguns exemplos práticos de implementação no TivaWare. É importante ressaltar que esses exemplos foram desenvolvidos para o TM4C1294NCPDT. Sendo assim, podem haver incompatibilidades presentes no carregamento destes códigos para algum outro hardware diferente, mesmo suportado pelo TivaWare.

13.1 LEDs das GPIOs controlados por Temporizadores

Este software de exemplo, ilustra um modo de implementação dos temporizadores e das GPIOs. Possui dois temporizadores, sendo que um inicia com um valor de contagem igual à metade do valor inicial do outro. No estouro de tempo desses temporizadores é disparada uma interrupção e as rotinas de interrupção acendem e apagam os LEDs presentes na placa conectados às GPIOs. Fazendo ascendam e, logo após, pisquem em tempos opostos.

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include "inc/hw_ints.h"
4  #include "inc/hw_memmap.h"
5  #include "inc/hw_types.h"
6  #include "driverlib/debug.h"
7  #include "driverlib/fpu.h"
8  #include "driverlib/gpio.h"
9  #include "driverlib/interrupt.h"
10 #include "driverlib/pin_map.h"
11 #include "driverlib/rom.h"
12 #include "driverlib/rom_map.h"
13 #include "driverlib/sysctl.h"
14 #include "driverlib/timer.h"
15 #include "driverlib/uart.h"
16 #include "utils/uartstdio.h"
17
18 // Armazena o clock do sistema
19 uint32_t g_ui32SysClock;
20
21 // Flag que controla o LED
22 uint32_t g_ui32Flags;
23
24 void
25 Timer0IntHandler(void)
26 {
27
28     // Limpa a interrupcao do temporizador
```

```
29      ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
30
31      // Troca valor da flag entre 0 e 1
32      HWREGBITW(&g_ui32Flags, 0) ^= 1;
33
34      // Acende o LED 0 conforme a flag
35      GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0, g_ui32Flags);
36
37      // Atualiza a interrupcao
38      ROM_IntMasterDisable();
39      ROM_IntMasterEnable();
40  }
41
42  void
43  Timer1IntHandler(void)
44  {
45
46    // Limpa a interrupcao do temporizador
47      ROM_TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
48
49      // Troca valor da flag entre 0 e 1
50      HWREGBITW(&g_ui32Flags, 1) ^= 1;
51
52      // Acende o LED 1 conforme a flag
53      GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, g_ui32Flags);
54
55      // Atualiza a interrupcao
56      ROM_IntMasterDisable();
57      ROM_IntMasterEnable();
58  }
59
60  int
61  main(void)
62  {
63      // Congfigura clock do sistema em 120 MHz
64      g_ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
65                                               SYSCTL_OSC_MAIN |
66                                               SYSCTL_USE_PLL |
67                                               SYSCTL_CFG_VCO_480),
      120000000);
68
69      // Habilita as GPIOS que contem os LEDs
70      ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);
71
72      // Habilita os pinos que contem os LEDs
73      ROM_GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0 | GPIO_PIN_1
      );
74
75
76      // Habilita os temporizadores 0 e 1
77      ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
78      ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);
79
80      // Habilita as interrupcoes no processador
81      ROM_IntMasterEnable();
82
83      // Configura os 2 temporizadores periodicos de 32 bits
84      ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
```

```
85      ROM_TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
86      ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, g_ui32SysClock);
87      ROM_TimerLoadSet(TIMER1_BASE, TIMER_A, g_ui32SysClock / 2);
88
89      // Configura as rotinas de tratamento de interrupcao do
        temporizadores
90      TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0IntHandler);
91      TimerIntRegister(TIMER1_BASE, TIMER_A, Timer1IntHandler);
92
93      // Configura as interrupcoes dos estouros de tempo dos
        temporizadores
94      ROM_IntEnable(INT_TIMER0A);
95      ROM_IntEnable(INT_TIMER1A);
96      ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
97      ROM_TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
98
99      // Habilita os temporizadores
100     ROM_TimerEnable(TIMER0_BASE, TIMER_A);
101     ROM_TimerEnable(TIMER1_BASE, TIMER_A);
102
103     // Superlaco de programa
104     while(1);
105
106     // Retorna sem erro
107     return 0;
108 }
```

13.2 Listagem de Periféricos na UART

O seguinte software, implementa uma comunicação UART na base de UART 0 do microcontrolador. Ao receber bytes, e detectado algum caractere 'p', é imprimido uma lista com os periféricos disponíveis no microcontrolador. Se for algum outro caractere, somente é exibida uma mensagem informativa.

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include <string.h>
4  #include "inc/hw_ints.h"
5  #include "inc/hw_memmap.h"
6  #include "driverlib/rom.h"
7  #include "driverlib/rom_map.h"
8  #include "driverlib/sysctl.h"
9  #include "driverlib/uart.h"
10 #include "driverlib/pin_map.h"
11 #include "driverlib/gpio.h"
12
13 // Numero maximo de perifericos existentes
14 #define peripheralQuantity  77
15
16 // Definicoes dos perifericos que podem estar disponiveis no sistema
17 const uint32_t peripheralAvailableId[peripheralQuantity] = {
18   SYSCTL_PERIPH_ADC0, SYSCTL_PERIPH_ADC1, SYSCTL_PERIPH_CAN0,
19   SYSCTL_PERIPH_CAN1, SYSCTL_PERIPH_COMP0, SYSCTL_PERIPH_EMAC0,
20   SYSCTL_PERIPH_EPHY0, SYSCTL_PERIPH_EPI0, SYSCTL_PERIPH_GPIOA,
21   SYSCTL_PERIPH_GPIOB, SYSCTL_PERIPH_GPIOC, SYSCTL_PERIPH_GPIOD,
22   SYSCTL_PERIPH_GPIOE, SYSCTL_PERIPH_GPIOF, SYSCTL_PERIPH_GPIOG,
23   SYSCTL_PERIPH_GPIOH, SYSCTL_PERIPH_GPIOJ, SYSCTL_PERIPH_HIBERNATE,
```

```
24    SYSCTL_PERIPH_CCM0, SYSCTL_PERIPH_EEPROM0, SYSCTL_PERIPH_FAN0,
25    SYSCTL_PERIPH_FAN1, SYSCTL_PERIPH_GPIOK, SYSCTL_PERIPH_GPIOL,
26    SYSCTL_PERIPH_GPIOM, SYSCTL_PERIPH_GPION, SYSCTL_PERIPH_GPIOP,
27    SYSCTL_PERIPH_GPIOQ, SYSCTL_PERIPH_GPIOR, SYSCTL_PERIPH_GPIOS,
28    SYSCTL_PERIPH_GPIOT, SYSCTL_PERIPH_I2C0, SYSCTL_PERIPH_I2C1,
29    SYSCTL_PERIPH_I2C2, SYSCTL_PERIPH_I2C3, SYSCTL_PERIPH_I2C4,
30    SYSCTL_PERIPH_I2C5, SYSCTL_PERIPH_I2C6, SYSCTL_PERIPH_I2C7,
31    SYSCTL_PERIPH_I2C8, SYSCTL_PERIPH_I2C9, SYSCTL_PERIPH_LCD0,
32    SYSCTL_PERIPH_ONEWIRE0, SYSCTL_PERIPH_PWM0, SYSCTL_PERIPH_PWM1,
33    SYSCTL_PERIPH_QEI0, SYSCTL_PERIPH_QEI1, SYSCTL_PERIPH_SSI0,
34    SYSCTL_PERIPH_SSI1, SYSCTL_PERIPH_SSI2, SYSCTL_PERIPH_SSI3,
35    SYSCTL_PERIPH_TIMER0, SYSCTL_PERIPH_TIMER1, SYSCTL_PERIPH_TIMER2,
36    SYSCTL_PERIPH_TIMER3, SYSCTL_PERIPH_TIMER4, SYSCTL_PERIPH_TIMER5,
37    SYSCTL_PERIPH_TIMER6, SYSCTL_PERIPH_TIMER7, SYSCTL_PERIPH_UART0,
38    SYSCTL_PERIPH_UART1, SYSCTL_PERIPH_UART2, SYSCTL_PERIPH_UART3,
39    SYSCTL_PERIPH_UART4, SYSCTL_PERIPH_UART5, SYSCTL_PERIPH_UART6,
40    SYSCTL_PERIPH_UART7, SYSCTL_PERIPH_UDMA, SYSCTL_PERIPH_USB0,
41    SYSCTL_PERIPH_WDOG0, SYSCTL_PERIPH_WDOG1, SYSCTL_PERIPH_WTIMER0,
42    SYSCTL_PERIPH_WTIMER1, SYSCTL_PERIPH_WTIMER2, SYSCTL_PERIPH_WTIMER3,
43    SYSCTL_PERIPH_WTIMER4, SYSCTL_PERIPH_WTIMER5
44  };
45
46  // Nomes dos perifericos que podem estar disponiveis no sistema
47  const uint8_t peripheralDescription[peripheralQuantity][23] = {
48    "Conversor A/D 0", "Conversor A/D 1", "Barramento CAN 0 ",
49    "Barramento CAN 1 ", "Comparador analogico 0", "Ethernet MAC 0",
50    "Ethernet PHY 0", "EPI0", "GPIO A",
51    "GPIO B", "GPIO C", "GPIO D",
52    "GPIO E", "GPIO F", "GPIO G",
53    "GPIO H", "GPIO J", "Modulo de hibernacao",
54    "CCM 0", "EEPROM 0", "FAN 0",
55    "FAN 1", "GPIO K", "GPIO L",
56    "GPIO M", "GPIO N", "GPIO P",
57    "GPIO Q", "GPIO R", "GPIO S",
58    "GPIO T", "I2C 0", "I2C 1",
59    "I2C 2", "I2C 3", "I2C 4",
60    "I2C 5", "I2C 6", "I2C 7",
61    "I2C 8", "I2C 9", "LCD 0",
62    "One Wire 0", "PWM 0", "PWM 1",
63    "QEI 0", "QEI 1", "SSI 0",
64    "SSI 1", "SSI 2", "SSI 3",
65    "Timer 0", "Timer 1", "Timer 2",
66    "Timer 3", "Timer 4", "Timer 5",
67    "Timer 6", "Timer 7", "UART 0",
68    "UART 1", "UART 2", "UART 3",
69    "UART 4", "UART 5", "UART 6",
70    "UART 7", "uDMA", "USB 0",
71    "Watchdog 0", "Watchdog 1", "Wide Timer 0",
72    "Wide Timer 1", "Wide Timer 2", "Wide Timer 3",
73    "Wide Timer 4", "Wide Timer 5"
74  };
75
76  // Imprime uma string na UART
77  void UARTprint(uint8_t *buffer) {
78    int i;
79    for(i = 0; i < strlen((char*)buffer); i++) {
80      UARTCharPut(UART0_BASE, buffer[i]);
81    }
```

```
82 }
83
84 // Imprime na UART os perifericos disponiveis no sistema
85 void UARTPrintPeripheralsAvailable() {
86   int i;
87   UARTprint((uint8_t*)"Perifericos disponiveis:\n\r");
88   for (i = 0; i < peripheralQuantity; i++) {
89     if (SysCtlPeripheralPresent(
90         (uint32_t)peripheralAvailableId[i])) {
91       UARTprint((uint8_t*)"- ");
92       UARTprint((uint8_t*)peripheralDescription[i]);
93       UARTprint((uint8_t*)"\n\r");
94     }
95   }
96 }
97
98 // Rotina de tratamento de interrupcao da UART
99 void UARTIntHandler(void) {
100   uint32_t statusInterrupt;
101
102   // Salva o status de interrupcao da UART 0
103   statusInterrupt = MAP_UARTIntStatus(UART0_BASE, true);
104
105   // Limpa interrupcoes encontradas na UART 0
106   MAP_UARTIntClear(UART0_BASE, statusInterrupt);
107
108   // Enquanto houver caracteres na FIFO de transmissao
109   // para serem enviados
110   while (MAP_UARTCharsAvail(UART0_BASE)) {
111
112     // Se o caractere informado por 'p' entao sao
113     // listados os perifericos disponiveis no sistema
114     if (UARTCharGet(UART0_BASE) == 'p') {
115       UARTPrintPeripheralsAvailable();
116     } else {
117       UARTprint((uint8_t*)"Para listar os "
118           "perifericos disponiveis envie a letra 'p'\n\r");
119     }
120   }
121 }
122
123 void UARTConfigure() {
124
125   // Habilita GPIO A usado na comunicacao da UART 0
126   MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
127   // Aguarda 3 SysCtlDelay. Aproximadamente 10 ciclos de clock
128   MAP_SysCtlDelay(3);
129   // Configura PA0 no modo Rx da UART 0
130   MAP_GPIOPinConfigure(GPIO_PA0_U0RX);
131   // Configura PA1 no modo Tx da UART 0
132   MAP_GPIOPinConfigure(GPIO_PA1_U0TX);
133
134
135   // Habilita UART 0
136   MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
137   // Configura PA0 e PA1 como pinos de comunicacao da UART
138   MAP_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
139   // Configura UART 0 com fonte de clock 120MHz para 115.200 baud 8N1
```

```
140   UARTConfigSetExpClk(UART0_BASE, 120000000, 115200,
141       (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
142       UART_CONFIG_PAR_NONE));
143
144
145   // Configura rotina de tratamento de interrupcao da UART
146   UARTIntRegister(UART0_BASE, UARTIntHandler);
147   // Habilita interrupcao da UART 0
148   MAP_IntEnable(INT_UART0);
149   // Configura pinos de interrupcao da UART 0
150   MAP_UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT);
151 }1
152
153 // Funcao principal do programa
154 int main(void) {
155
156   // Configura oscilador principal acima de 10 MHz
157   SysCtlMOSCConfigSet(SYSCTL_MOSC_HIGHFREQ);
158
159   // Configura clock para 120 MHz
160   MAP_SysCtlClockFreqSet(
161       (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
162       SYSCTL_CFG_VCO_480),
163       120000000);
164
165   // Funcao de inicializacao da UART
166   UARTConfigure();
167
168   // Super laco de programa
169   while (1);
170
171   return 0;
172 }
```

13.3 Largura de pulso do PWM controlada pelo ADC

Este exemplo, implementa um método de controle direto da largura de pulso de um PWM de 10 KHz pela tensão lida no ADC. São utilizadas as interrupções de ambos os periféricos.

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include "inc/hw_ints.h"
4  #include "inc/hw_memmap.h"
5  #include "inc/hw_types.h"
6  #include "inc/hw_gpio.h"
7  #include "driverlib/rom.h"
8  #include "driverlib/rom_map.h"
9  #include "driverlib/sysctl.h"
10 #include "driverlib/pin_map.h"
11 #include "driverlib/gpio.h"
12 #include "driverlib/adc.h"
13 #include "driverlib/pwm.h"
14
15 uint32_t ADCValue;
16 float PWMValue = 0;
17
18 void ADCConfigure() {
```

```
19
20    // Habilita ADC0
21    MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
22    // Aguarda 3 SysCtlDelay. Aproximadamente 10 ciclos de clock
23    MAP_SysCtlDelay(3);
24
25    // Desabilitar Interrupcao do ADC para configura-la
26    MAP_IntDisable(INT_ADC0SS0);
27    MAP_ADCIntDisable(ADC0_BASE, 0);
28    MAP_ADCSequenceDisable(ADC0_BASE, 0);
29
30    // Configurando ADC
31    MAP_ADCHardwareOversampleConfigure(ADC0_BASE, 4);
32    MAP_ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
33    MAP_ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
34      ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
35    MAP_ADCSequenceEnable(ADC0_BASE, 0);
36
37    // Habilitando Interrupcao do ADC
38    MAP_ADCIntClear(ADC0_BASE, 0);
39    MAP_ADCIntEnable(ADC0_BASE, 0);
40    MAP_IntEnable(INT_ADC0SS0);
41  }
42
43  void ADC_handler() {
44
45    // Limpando Interrupcao do ADC
46    ADCIntClear(ADC0_BASE, 0);
47    // Passando valor convertido pelo ADC para a variavel ADCValue
48    ADCSequenceDataGet(ADC0_BASE, 0, &ADCValue);
49  }
50
51  void PWMConfigure(void) {
52
53    // Configurando GPIO para PWM
54    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
55    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
56
57    GPIOPinConfigure(GPIO_PF1_M0PWM1);
58    GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1);
59
60    // Configurando PWM
61    PWMGenConfigure(PWM0_BASE, PWM_GEN_0,
62      PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
63    // Configurando Fonte Clock do PWM como
64    // fonte de 120 MHz dividido por 4 sendo um clock de 30 MHz
65    PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_4);
66    // Configurando contagem de 3000, para gerar um PWM com 10 KHz
67    PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, 3000);
68    // Configurando contagem de 1550
69    // para gerar um Duty Cicle inicial de 50%
70    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, 1500);
71
72    // Habilitando geracao PWM
73    PWMGenEnable(PWM0_BASE, PWM_GEN_0);
74    PWMOutputState(PWM0_BASE, (PWM_OUT_0_BIT | PWM_OUT_1_BIT), true);
75
76    // Habilitando interrupcao do PWM
```

```
77    PWMIntEnable(PWM0_BASE, PWM_INT_GEN_0);
78    IntEnable(INT_PWM0_0);
79    PWMGenIntTrigEnable(PWM0_BASE, PWM_GEN_0, PWM_INT_CNT_LOAD);
80  }
81
82  void PWM_handler() {
83
84    // Convertendo valor obtido pelo AD, na base 2^12, para base de 3000
85    PWMValue = ADCValue / 1.3653334; // ADCValue/(2^12)*3000
86
87    // Atualizando Duty Cicle
88    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, (int)(PWMValue));
89
90    // Limpando Interrupcao do ADC
91    ADCIntClear(ADC0_BASE, 0);
92
93    // Habilitando Interrupao do ADC
94    ADCProcessorTrigger(ADC0_BASE, 0);
95  }
96
97  int main(void) {
98
99    // Configuracao Basica do Sistema de clock
100   // selecionando a frequencia de 120 MHz
101   SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
102     SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);
103
104   // Habilitando Unidade de Ponto Flutuante
105   FPUEnable();
106   FPULazyStackingEnable();
107
108   // Configuracao ADC
109   ADCConfigure();
110
111   // Configuracao PWM
112   PWMConfigure();
113
114   while (1) {
115
116   }
117
118   return 0;
119 }
```

13.4 Comunicação SPI com Terminal UART

O seguinte exemplo, implementa uma comunicação SPI que envia e recebe 3 bytes. É inicializada uma comunicação UART secundária para a utilização de um terminal, onde são mostradas informações sobre o processo do sistema.

```
1  #include <stdbool.h>
2  #include <stdint.h>
3  #include "inc/hw_memmap.h"
4  #include "driverlib/gpio.h"
5  #include "driverlib/pin_map.h"
6  #include "driverlib/ssi.h"
7  #include "driverlib/sysctl.h"
```

```
8  #include "driverlib/uart.h"
9  #include "utils/uartstdio.h"
10
11 // Numero de bytes enviados e recebidos
12 #define NUM_SSI_DATA            3
13
14 void InitConsole(void) {
15     // Habilita GPIO A
16     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
17     // Configura os pinos de recepcao e transmissao da UART 0
18     GPIOPinConfigure(GPIO_PA0_U0RX);
19     GPIOPinConfigure(GPIO_PA1_U0TX);
20
21   // Habilita a UART 0
22     SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
23     // Clock da UART como o clock interno de 16 MHz
24     UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);
25     // Habilita pinos da comunicaco UART
26     GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
27     // Inicializa UART com baud rate de 115200
28     UARTStdioConfig(0, 115200, 16000000);
29 }
30
31 int main(void) {
32     uint32_t ui32SysClock;
33
34     uint32_t pui32DataTx[NUM_SSI_DATA];
35     uint32_t pui32DataRx[NUM_SSI_DATA];
36     uint32_t ui32Index;
37
38   // Configura clock do sistema como 25 MHz
39     ui32SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
40                                        SYSCTL_OSC_MAIN |
41                                        SYSCTL_USE_OSC), 25000000);
42
43     // Inicializa o console
44     InitConsole();
45
46     // Mostra a situcao do sistema na=o terminal UART
47     UARTprintf("SSI ->\n");
48     UARTprintf("  Mode: SPI\n");
49     UARTprintf("  Data: 8-bit\n\n");
50
51     // Habilita SSI
52     SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
53
54     // Habilita a GPIO A
55     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
56
57     // Configura os pinos da comunicacao SPI
58     //       PA4 - SSI0Rx
59     //       PA3 - SSI0Fss
60     //       PA2 - SSI0CLK
61     GPIOPinConfigure(GPIO_PA2_SSI0CLK);
62     GPIOPinConfigure(GPIO_PA3_SSI0FSS);
63     GPIOPinConfigure(GPIO_PA4_SSI0RX);
64     GPIOPinConfigure(GPIO_PA5_SSI0TX);
65
```

```
66      // Configura pinos utilisados na SSI
67      GPIOPinTypeSSI(GPIO_PORTA_BASE,
68      GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 | GPIO_PIN_2);
69
70      // Configuracao da comunicacao
71      SSIConfigSetExpClk(SSI0_BASE, ui32SysClock, SSI_FRF_MOTO_MODE_0,
72                         SSI_MODE_MASTER, 1000000, 8);
73
74      // Habilita SSI
75      SSIEnable(SSI0_BASE);
76
77      // Aguarda leitura de todos os dados recebidos
78      while(SSIDataGetNonBlocking(SSI0_BASE, &pui32DataRx[0]))
79      {
80      }
81
82      // Texto a ser enviado
83      pui32DataTx[0] = 's';
84      pui32DataTx[1] = 'p';
85      pui32DataTx[2] = 'i';
86
87      //  Indicacao de que comecara o envio
88      UARTprintf("Sent:\n  ");
89
90      // Envia 3 bytes de dado
91      for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
92      {
93          // Mostra na UART, o dado que esta sendo transferido
94          UARTprintf("'%c' ", pui32DataTx[ui32Index]);
95
96          // Envia byte dela SSI
97          SSIDataPut(SSI0_BASE, pui32DataTx[ui32Index]);
98      }
99
100     // Aguarda SSI estar pronta
101     while(SSIBusy(SSI0_BASE))
102     {
103     }
104
105     // Indicacao de que SSI esta recebendo dados
106     UARTprintf("\nReceived:\n  ");
107
108     // Recebe 3 bytes de dado
109     for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
110     {
111         // Recebe 3 bytes de dado pela SSI
112         SSIDataGet(SSI0_BASE, &pui32DataRx[ui32Index]);
113
114         // Conversao para 8 bits
115         pui32DataRx[ui32Index] &= 0x00FF;
116
117         // Mostra byte de dado recebido
118         UARTprintf("'%c' ", pui32DataRx[ui32Index]);
119     }
120
121     // Retorna sem erro
122     return(0);
123 }
```

Referências

[1] T. I. Incorporated. *TivaWare™ Peripheral Driver Library*. Texas Instruments Incorporated, Disponível em: http://www.ti.com/lit/ug/spmu298a/spmu298a.pdf . Acesso em: 11 Jan. 2016.

[2] T. I. Incorporated. *Tiva TM4C1294NCPDT Microcontroller - DATA SHEET*. Texas Instruments Incorporated, Disponível em: http://www.ti.com.cn/cn/lit/ds/symlink/tm4c1294ncpdt.pdf . Acesso em: 20 Jan. 2016, 6 2014.

[3] J. Yiu. *The Definitive Guide To ARM CORTEX™ -M3 AND ARM CORTEX™ -M4*. Newnes, Cambridge, UK, 3 ed edition, 2014.