João Paulo Ramos Agra Méllo

Relatório de Estágio

Local: Laboratório de Eletrônica Industrial e Acionamento de Máquinas - UFCG

Campina Grande, Brasil

7 de agosto de 2014

João Paulo Ramos Agra Méllo

Relatório de Estágio

Local: Laboratório de Eletrônica Industrial e Acionamento de Máquinas - UFCG

Relatório de Estágio submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande - como parte dos requisitos necessários para a obtenção do título de Graduado em Engenharia Elétrica.

Universidade Federal de Campina Grande - UFCG

Unidade Acadêmica de Engenharia Elétrica

Orientador: Alexandre Cunha Oliveira

Campina Grande, Brasil

7 de agosto de 2014

João Paulo Ramos Agra Méllo

Relatório de Estágio

Local: Laboratório de Eletrônica Industrial e Acionamento de Máquinas - UFCG

Relatório de Estágio submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande - como parte dos requisitos necessários para a obtenção do título de Graduado em Engenharia Elétrica.

Relatório aprovado. Campina Grande, Brasil, 7 de agosto de 2014:

Alexandre Cunha Oliveira
Orientador

Cursino Brandão Jacobina
Convidado

Campina Grande, Brasil
7 de agosto de 2014

Dedico este trabalho à minha família,
sem a qual não poderia realizar meu sonho,
ou sequer sonhar.

Agradecimentos

Agradeço a Deus, por ter me permitido chegar até aqui e pelas oportunidades que tive ao longo da vida.

Agradeço também à minha família por ter me dado todo o apoio de que precisei e que preciso na minha caminhada. À minha mãe, por todo amor e paciência transmitidos, pelas orientações. Ao meu pai, pelo eterno suporte às minhas decisões, pelos conselhos e senso de dignidade e justiça nos quais me espelho. Aos meus irmãos, por sempre estarem comigo e contribuírem na minha formação pessoal, me permitindo ser quem sou hoje.

Agradeço ao professor Alexandre pela orientação e ao professor Cursino pelo suporte.

Finalmente, agradeço aos amigos e colegas de curso e de laboratório, em especial aos amigos com quem convivo no dia-a-dia e que me ajudaram de algum modo na elaboração deste trabalho e na conclusão de outras atividade que me permitiram chegar até este ponto.

"Sapere aude!"
(Quintus Horatius Flaccus)

Lista de ilustrações

Lista de abreviaturas e siglas

ADC	Conversor Analógico-Digital (*Analog-to-Digital Converter*)
CCS	Code Composer Studio
CMOS	Semicondutor Metal-Óxido Complementar (*Complementary Metal-Oxide Semiconductor*)
CPU	Unidade Central de Processamento (*Central Processing Unit*)
DSC	Controlador de Sinais Digitais (*Digital Signal Controller*)
DSP	Processador de Sinais Digitais (*Digital Signal Processor*)
ECCE	*IEEE Energy Conversion Congress and Exposition*
EPWM	PWM Aprimorado (*Enhanced PWM*)
GPIO	Entrada/Saída de Uso Geral (*General Purpose Input/Output*)
LED	Diodo Emissor de Luz (*Light Emissor Diode*)
LEIAM	Laboratório de Eletrônica Industrial e Acionamento de Máquinas
PI	Proporcional-Integral
PIE	Expansão de Interrupção dos Periféricos (*Peripheral Interrupt Expansion*)
PLL	*Loop* Travado em Fase (*Phase Locked Loop*)
PWM	Modulação por Largura de Pulso (*Pulse Width Modulation*)
S/H	*Sample-and-Hold*
SOC	Começo da Conversão (*Start-of-Conversion*)
SOCA	Começo da Conversão A (*Start-of-Conversion A*)
SOCB	Começo da Conversão B (*Start-of-Conversion B*)
TBCLK	*Time Base Clock*
TI	Texas Instruments
UFCG	Universidade Federal de Campina Grande

Sumário

1 Introdução

O Laboratório de Eletrônica Industrial e Acionamento de Máquinas (LEIAM) é um dos laboratórios de pesquisa do Departamento de Engenharia Elétrica (DEE) da Universidade Federal de Campina Grande (UFCG). O projeto e desenvolvimento de novas topologias de conversores estáticos é um dos ramos de pesquisa do Laboratório. Estes conversores têm diversas aplicações, mas de modo geral cumprem a função de interligar de maneira eficiente um sistema de geração de energia elétrica com um sistema consumidor.

Existem diversos aspectos importantes que devem ser trabalhados no desenvolvimento de um sistema de conversão. Dentre eles está o estudo e aplicação da estratégia PWM para o chaveamento dos conversores, bem como o estudo e aplicação de um sistema de controle. No Laboratório, o dispositivo do sistema de conversão que é central para a aplicação dos elementos citados é o microcontrolador.

Assim, o estágio relatado neste documento teve como principal objetivo o estudo do DSP (*digital signal processor*), especificamente do dispositivo TMS320F28335, da Texas Instruments, que atualmente é reconhecido como um microcontrolador. Este é o dispositivo utilizado na maior parte das bancadas de experimentos do Laboratório, e sua compreensão é de fundamental importância para a realização dos projetos de pesquisa, sobretudo para testar e implementar estratégias PWM diversificadas.

Além do estudo do dispositivo referenciado, que daqui em diante será chamado de forma abreviada por DSP28335, foi também proposta do estágio a elaboração de um documento guia que servisse de base para a compreensão das funções que o DSP28335 desempenha nos sistemas de conversão implementados, de forma a facilitar a introdução de novos alunos no grupo de pesquisa e nas atividades práticas do Laboratório. O material deve servir também de auxílio na criação de projetos para o dispositivo, permitindo facilitar o processo de configuração do DSP28335 e a criação das rotinas que serão executadas por ele. Assim, este relatório foi escrito de modo a cumprir também a função deste guia.

A estrutura do relatório é dividida em seis capítulos, incluindo introdução e conclusão, ao longo dos quais são detalhados os procedimentos de criação e elaboração de um projeto base para o DSP28335 atendendo às necessidades das aplicações do Laboratório, utilizando o *software* CCS v3.3. Além disso, são citadas atividades adicionais realizadas durante o período de estágio.

No capítulo 2 é relatado e descrito o processo de criação de um projeto em branco para o DSP28335, i.e., é mostrado desde como foram obtidos os arquivos e instalados os programas e pacotes necessários, até como foi estruturado o projeto, deixando os arquivos prontos para serem editados.

Já no capítulo 3 são feitos o relato e a descrição da programação do arquivo principal do projeto e dos arquivos de configuração. Isto é realizado mostrando e descrevendo os códigos utilizados no arquivo principal e nos arquivos de configuração do DSP e dos periféricos que devem ser utilizados.

No capítulo 4 é descrito e relatado como o DSP foi programado para implementar a estratégia PWM e o sistema de controle, de acordo com um exemplo de sistema de conversão dado.

No capítulo 5 são citadas outras atividades que foram feitas no período de estágio, que consistiram basicamente no estudo, simulação, programação e montagem de sistemas de conversão a na publicação de artigos científicos, esta que é uma das principais atividades do Laboratório.

Por fim, na conclusão é feita uma análise geral do estágio e dos objetivos e resultados alcançados.

2 Criação de um Projeto para o DSP28335 no CCS v3.3

O DSP utilizado no Laboratório é o microcontrolador TMS320F28335, fabricado pela Texas Instruments, que daqui em diante será referenciada como TI. De fato, atualmente ele não é mais reconhecido como um DSP (*digital signal processor*), mas como um DSC (*digital signal controller*), um microcontrolador. Por motivos históricos, continuaremos a chamá-lo de DSP.

O TMS320F28335, daqui em diante referido por DSP28335, é um dispositivo de alta performance baseado em tecnologia CMOS, podendo operar a $1,8V$ ou a $1,9V$. Neste último caso pode chegar a uma frequência de $150MHz$, o que corresponde a um tempo de ciclo de $6,67ns$. Ele possui ainda uma CPU de 32 bits, barramentos com arquitetura Havard, um controlador DMA de seis canais, interface externa de 16 ou 32 bits, memória interna (Flash 256K x 16, SARAM 34K x 16 e OTP-ROM 1K x 16) e uma memória ROM 8K x 16 para *boot*. Além disso, possui diversos outros recursos como 3 temporizadores integrados à CPU, 88 pinos configuráveis de entrada e saída de uso geral (GPIO), 12 saídas EPWM, um módulo ADC de 16 canais de 12-bits cada, com taxa de conversão de $80ns$, dentre outros (consultar documentação disponível em *<http://www.ti.com/product/TMS320F28335/technicaldocuments>* para informação complementar.)

Dessa forma, para fazer uso das muitas funcionalidades do DSP é necessário antes configurá-lo e programá-lo adequadamente. Nas seções seguintes são relatados os procedimentos que foram realizados para adquirir e instalar os recursos necessários à criação de um projeto base para o DSP. Este projeto é o que permitirá configurar e programar o dispositivo de modo a possibilitar a utilização de recursos necessários às aplicações do Laboratório. Além disso, é relatado o procedimento de criação de um projeto base em branco.

2.1 Instalação do CCS v3.3 e do Pacote de Projetos Exemplo

O software utilizado para programar e construir projetos no Laboratório para o DSP28335 é o CCS (Code Composer Studio) v3.3, da TI. Esta não é a versão mais atual do programa, mas é a versão utilizada no local de estágio, o LEIAM. Deste modo, todos os procedimentos de instalação e criação de projetos mostrados neste relatório são feitos com base no CCS v3.3. O disco de instalação pode ser adquirido com o pessoal do Laboratório, i.e., professores, alunos de pós-graduação ou técnicos, para utilização dentro da UFCG e

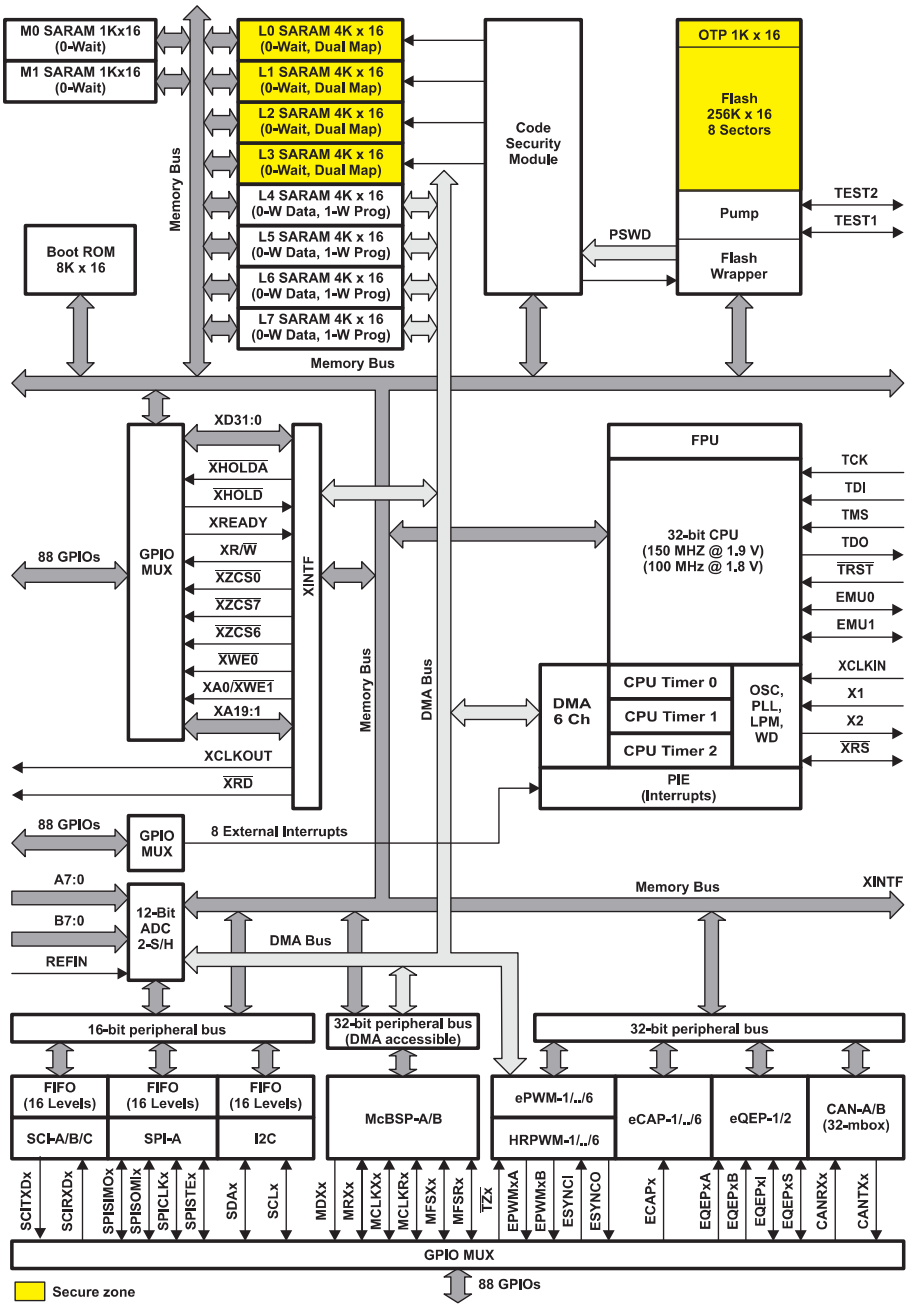

Figura 1 – Diagrama de blocos funcional do DSP28335.

com fins de ensino e pesquisa.

Nesta seção são mostrados os procedimentos que foram necessários para prepara um computador, de modo que com ele fosse possível criar um projeto para o DSP, bem como configurá-lo e programá-lo para executar as tarefas desejadas, considerando as aplicações feitas no LEIAM, i.e., comando de conversores estáticos e controle de sistemas que empreguem estes equipamentos.

Basicamente, foi necessário se realizar três instalações antes da criação de um projeto para, i.e.:

- Instalar o CCS v3.3;
- Instalar os pacotes de *drivers*/*target* para a plataforma eZdsp28335;
- Instalar o pacote de arquivos *header* e projetos exemplo do DSP.

As duas primeiras instalações foram feitas a partir do disco de instalação do CCS, já a última foi feita a partir de um arquivo que deve ser baixado do site da TI. Os procedimentos são detalhados nas subseções seguintes.

2.1.1 Instalando o CCS v3.3

O procedimento de instalação do CCS é muito simples. Uma vez de posse do disco de instalação, deve-se inseri-lo no drive de CD/DVD do computador e o *autorun* deve ser executado, levando à janela mostrada na Figura 2. A escolha da opção "*INSTALL PRODUCTS*" levará à janela mostrada na Figura 3. A opção "*CODE COMPOSER STUDIO v3.3*" deve ser escolhida e levará ao processo de instalação do CCS v3.3. A outra opção levará à instalação do pacote de *drivers* e *target* do DSP28335 e deve ser escolhida posteriormente.

Nas janelas seguintes serão dadas recomendações e instruções sobre a instalação, bem como será feita uma verificação do sistema com relação aos pré-requisitos de instalação, além de ser mostrado o acordo de licença do *software*. Deve-se observar tudo o que é mostrado nestas janelas e clicar em "*Next*", concordando com o acordo de licença, até chegar à janela mostrada na Figura 4. Aqui deve ser escolhida a opção de instalação "*Typical Install*", o que levará a janela seguinte em que deve-se escolher o diretório de instalação. Recomenda-se a escolha do diretório padrão, i.e., "*C:\CCStudio_v3.3*". Assim, deve-se clicar em "*Next*" e por fim, na janela seguinte, confirmar a instalação clicando em "*Install Now*".

A conclusão da instalação deve ser aguardada. É possível que, quando se chegue na etapa mostrada na Figura 5, a instalação demore a progredir. Mesmo assim, deve-se aguardar pacientemente a conclusão, que deve ser confirmada na janela final clicando em "*Finish*".

2.1.2 Instalando o pacote de *drivers* e *target*

A segunda instalação feita foi a do pacote de *drivers* e *target* para o eZdsp28335. O procedimento é muito parecido com o da instalação do CCS v3.3. Na janela da Figura 3 deve-se escolher a opção "*eZdsp28335 Drivers and Target Content*". Isto deve levar à janela mostrada na Figura 6. Ao clicar em "*Next*", a janela do acordo de licença será aberta. Deve-se concordar com o acordo para prosseguir com a instalação, o que levará à janela mostrada na Figura 7. Nesta janela deve-se escolher a opção "*Typical*" e clicar em

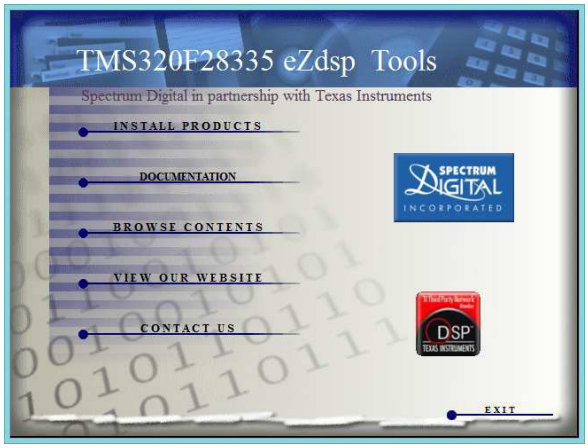

Figura 2 – Menu do disco de instalação do CCS v3.3.

Figura 3 – Menu de instalação dos produtos disponíveis no disco.

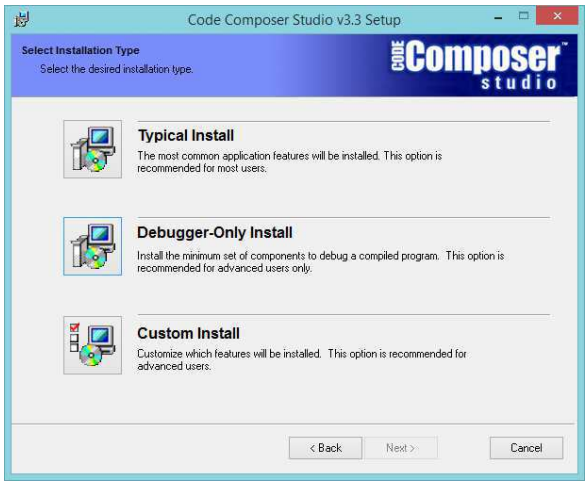

Figura 4 – Janela de seleção do tipo de instalação do CCS v3.3.

Figura 5 – Instalação do CCS v3.3.

"*Next*". A janela seguinte é referente à escolha do diretório de instalação, que deve ser o mesmo no qual o CCS foi instalado. Caso a recomendação de escolha do diretório padrão tenha sido obedecida, não deve-se alterar nada nesta janela, clicando em "*Next*" para dar sequência à instalação. Na janela seguinte deve-se confirmar as opções de instalação clicando em "*Next*" novamente e aguardar a conclusão da instalação, que deve ser rápida. Na janela final, após a instalação, deve-se simplesmente confirmar a conclusão clicando em "*Finish*".

2.1.3 Instalando o pacote de arquivos *header* e projetos exemplo do DSP

É fortemente recomendável que um projeto para o DSP28335 seja construído a partir de algum projeto base existente. Esta foi a metodologia adotada para a construção do projeto base, de forma que as descrições do relatório são pautadas na premissa de que a maior parte dos arquivos e códigos utilizados no projeto foram obtidos de projetos exemplo e mantidos inalterados. Deste modo, o foco das discussões e explicações feitas

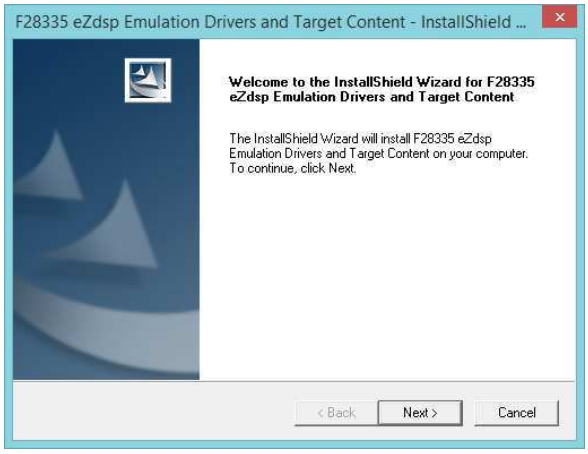

Figura 6 – Janela inicial da instalação do pacote de *drivers* e *target*.

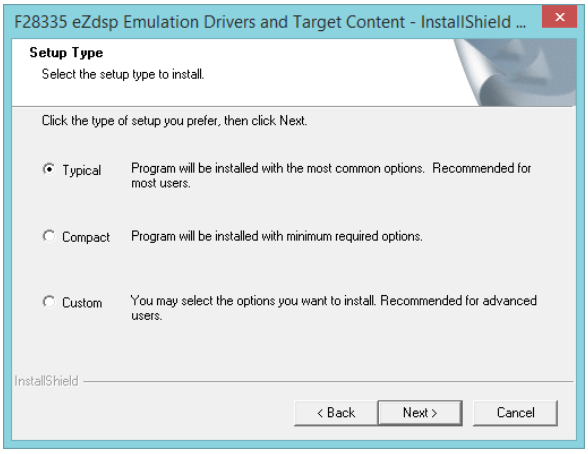

Figura 7 – Janela de seleção do tipo de instalação do pacote.

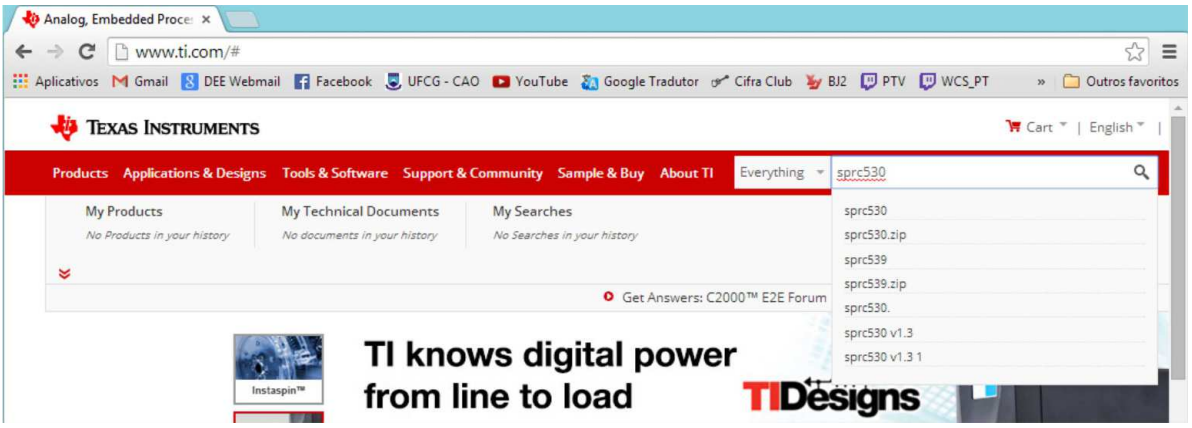

Figura 8 – Página inicial da Texas Instruments.

foi dado ao que deve ser acrescentado ou modificado a partir dos projetos exemplo, sem entrar em detalhes no que é mantido inalterado.

Assim, objetivando-se esta praticidade, foi adquirido no site da TI um pacote de projetos exemplo para DSPs, que pode ser baixado diretamente do endereço *<http://www.ti.com/lit/zip/sprc530>*. Alternativamente, pode-se fazer uma pesquisa por "sprc530" na página inicial do site da TI, no endereço *<http://www.ti.com/>*, como mostrado na Figura 8. Esta pesquisa irá levar à página mostrada na Figura 9. O *link* do lado direito desta página, no qual está escrito "*SPRC530*" em vermelho, irá levar à página de *download* do pacote "*sprc530.zip*", mostrada na Figura 10. Para fazer o *download* basta clicar no *link* mostrado na Figura 10, destacado em vermelho.

Este pacote com exemplos é baixado na forma de um arquivo "*.zip*", que deve ser extraído. Dentre os conteúdos está um arquivo executável com o nome "*setup_DSP2833x_v131.exe*". Este arquivo deve ser executado para que o pacote seja instalado. Esta ação causará a abertura da janela inicial de instalação do pacote, mostrada na Figura 11.

Clicar em "*Next*" nesta janela levará à janela com o acordo de licença, que deve ser aceito. Clicando em "*Next*" chega-se à janela seguinte, que deve ser checada. Clicando-se

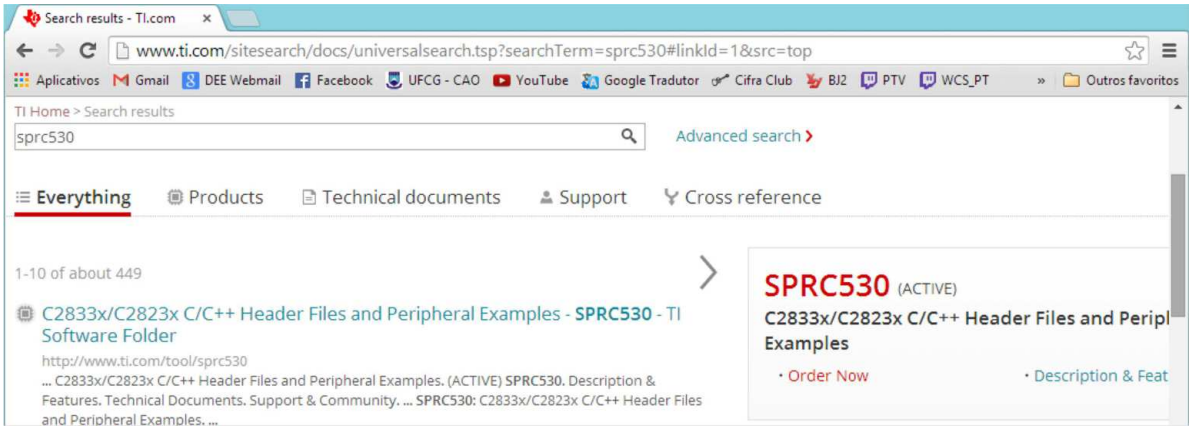

Figura 9 – Página destino da pesquisa pelo pacote de projetos exemplo para o DSP28335.

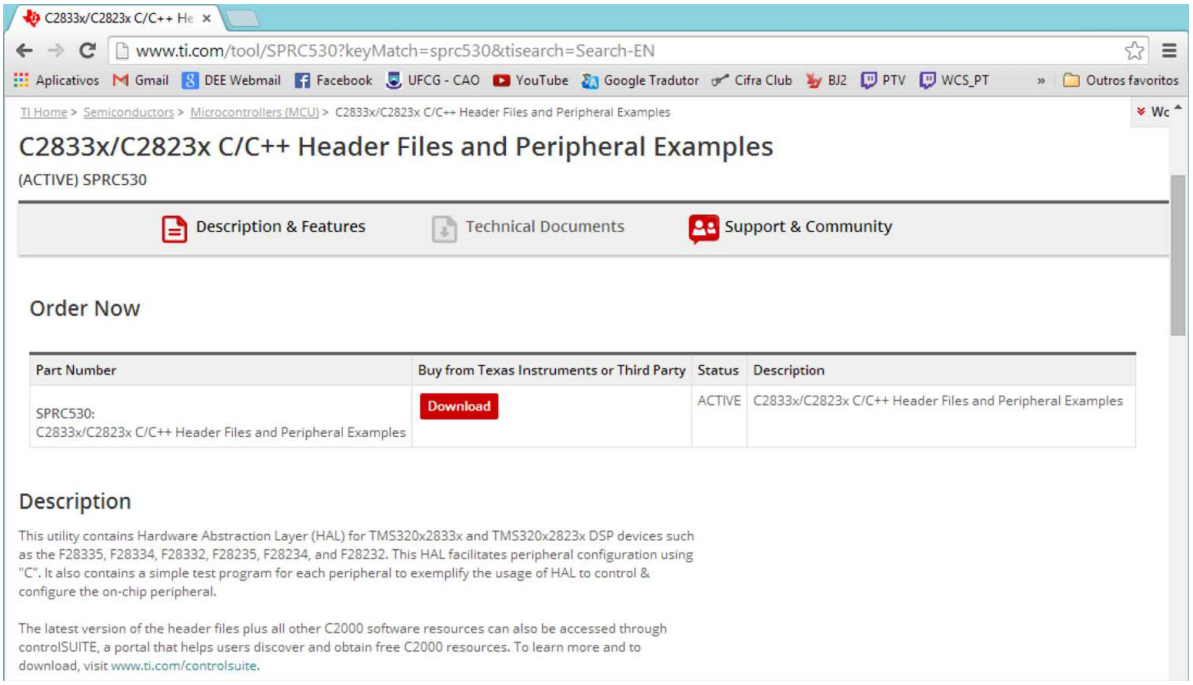

Figura 10 – Página de download do pacote de projetos exemplo para o DSP28335.

novamente em "*Next*" chega-se à janela de escolha do diretório de instalação, mostrada na Figura 12. Novamente, recomenda-se a escolha do diretório padrão, i.e., "*C:\tidcs\c28\DSP2833x\v131*". Então, segundo esta recomendação, nesta janela não é alterado nada e deve-se prosseguir com a instalação clicando em "*Next*". Finalmente, na janela seguinte, a instalação deve ser confirmada clicando-se em "*Install*", devendo-se em seguida aguardar o termino da extração dos arquivos, que não deve demorar. Na tela seguinte, deve-se clicar em "*Next*" e em seguida concluir a instalação clicando em "*Finish*" (Figura 13).

2.2 Criação do projeto base

Após a instalação do CCS conforme descrito na seção 2.1, foi realizada a criação de um projeto base em branco, conforme é descrito nesta seção.

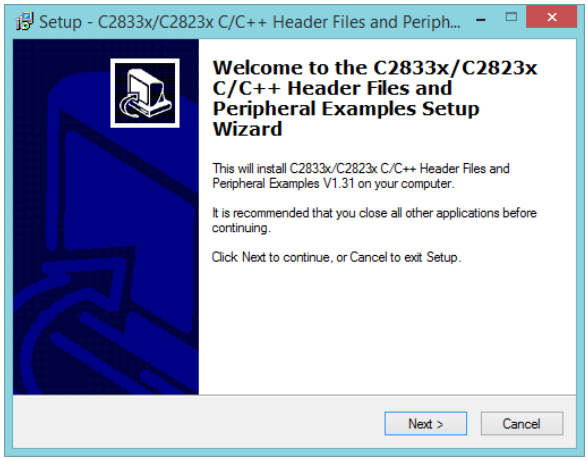

Figura 11 – Janela inicial de instalação do pacote de projetos exemplo para o DSP28335.

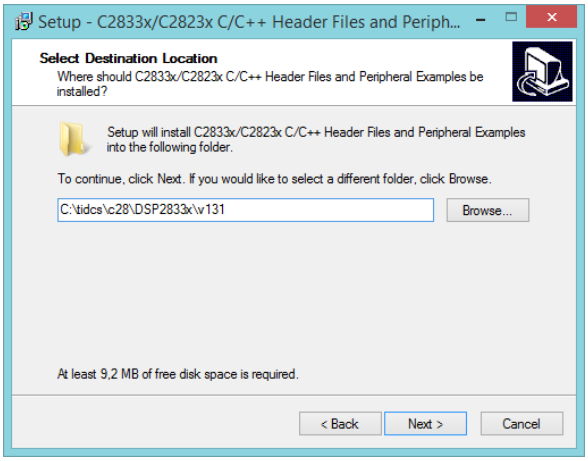

Figura 12 – Janela de escolha do diretório de instalação do pacote de projetos exemplo para o DSP28335.

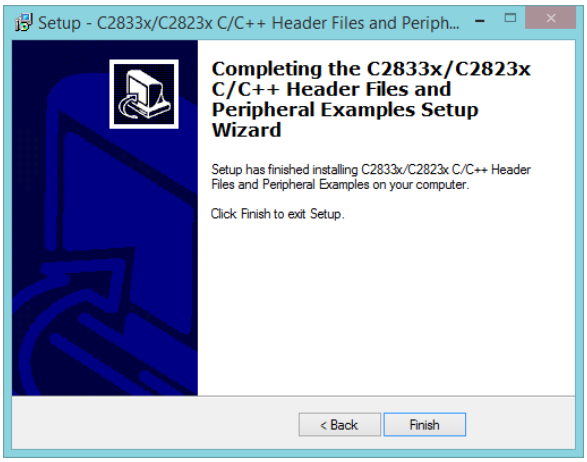

Figura 13 – Janela final da instalação do pacote de projetos exemplo para o DSP28335.

Com os ícones mostrados na Figura 14 disponíveis na área de trabalho do computador, deve-se dar um duplo clique no ícone com o nome "*F28335 eZdsp CCStudio v3.3*". Caso a placa do DSP não esteja conectada ao computador, a tela mostrada na Figura 15 deve aparecer. Para a criação do projeto não é necessário que a placa com o DSP esteja conectada, então nesta janela deve-se clicar na opção "*Ignore*", o que levará à janela inicial para criação e edição de projetos.

Caso o pacote de exemplos da TI tenha sido instalado no diretório padrão, os projetos exemplo compatíveis com o DSP28335 podem ser encontrados em "*C:\tidcs\c28\DSP2833x\v131\DSP2833x_examples*". Eles podem ser abertos escolhendo "*Project*" e em seguida "*Open...*" no menu da parte superior da janela inicial de criação e edição de projetos do CCS, mostrada na Figura 16. Ao se fazer isto, será aberta uma janela para a escolha de um arquivo com a extensão "*.pjt*" que será aberto. Estes arquivos estão nas pastas dos respectivos exemplos, e é recomendável que eles sejam consultados como forma complementar de estudo. Esta prática foi uma das adotadas durante o estágio e também para a elaboração deste relatório.

Figura 14 – Ícones disponíveis na área de trabalho após as instalações do CCS e dos pacotes.

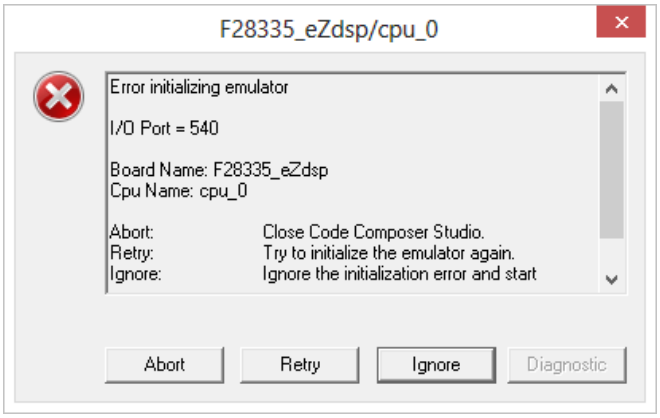

Figura 15 – Janela de erro ao se tentar iniciar o CCS sem a placa do DSP28335 estar conectada.

Um projeto construído para o DSP28335 no CCS v3.3 é bem estruturado e seus arquivos estão separados em seções pelos seus tipos. Isto pode ser percebido abrindo qualquer projeto exemplo do pacote, ou mesmo criando um conforme será descrito. Para um projeto padrão utilizado no LEIAM, são incluídos a partir dos projetos exemplo 26 arquivos de cabeçalho, 12 arquivos fonte e 2 arquivos *batch* do tipo "*.cmd*". Além disso, mais 4 arquivos fonte devem ser criados, cujos códigos e funções serão detalhas no capítulo 3. Os arquivos de cabeçalho ficam na seção "*Include*" do projeto, já os arquivos fonte ficam na seção "*Source*". Os arquivos "*.cmd*" ficam na seção principal.

Para a criação de um projeto, deve-se escolher "*Project*" e em seguida "*New...*" no menu da parte superior da tela inicial, conforme mostrado na Figura 16. A janela mostrada na Figura 17 se abrirá, e nela deve ser digitado o nome do projeto e escolhido o diretório em que ele ficará alocado. Estas escolhas são arbitrárias, de forma que as mostradas na figura foram tomadas simplesmente como exemplo para facilitar a compreensão do procedimento. Nada mais deve ser alterado nesta janela, de modo que foram escolhidos o nome "*ProjExemplo*" para o projeto e o diretório "*C:\ProjExemplo*" para alocá-lo. Ao se clicar em "*Finish*" o projeto é criado segundo a estrutura mostrada na Figura 18.

Em seguida, deve-se acrescentar arquivos ao projeto, que são os arquivos de cabeçalho, arquivos fonte e arquivos "*.cmd*", conforme descrito. Estes arquivos devem ser buscados no pacote de exemplos, cujo procedimento de aquisição e instalação foi mostrado na seção 2.1. Ainda, alguns arquivos fonte além dos disponíveis no pacote de exemplos devem ser criados. Estes procedimentos são relatados nas subseções que seguem.

2.2.1 Acrescentando os arquivos de cabeçalho

Os 26 arquivos de cabeçalho necessários ao projeto base são também usados em qualquer um dos projetos exemplo do pacote. Não é necessário fazer qualquer alteração neles após incluí-los. Assim, eles são listados a seguir de acordo com os diretórios em que

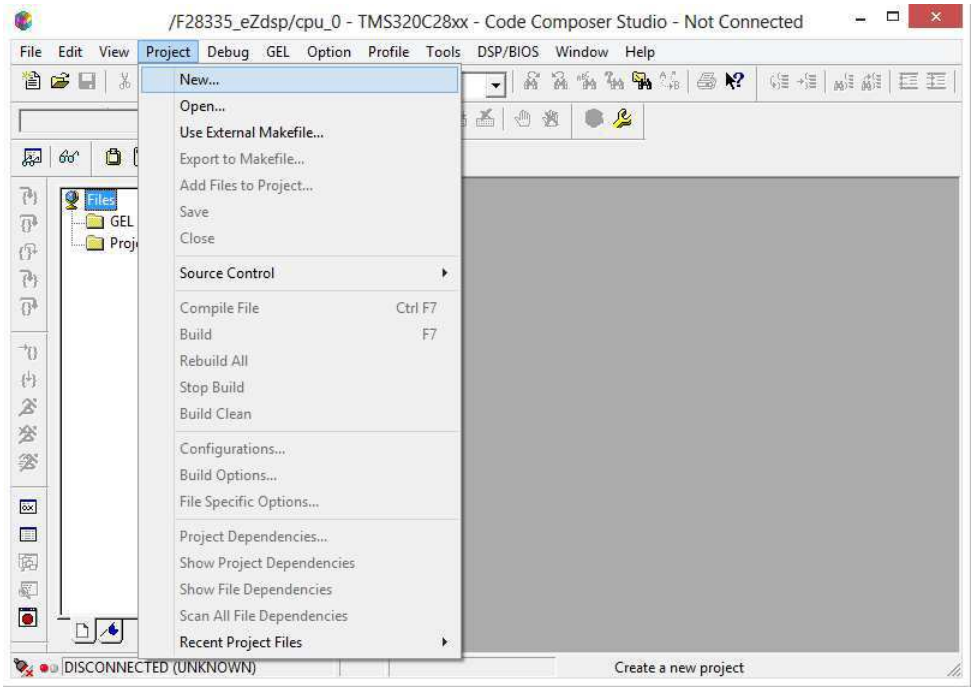

Figura 16 – Janela inicial de criação e edição de projetos do CCS para o DSP28335, mostrando a seção de projetos do menu na parte superior da janela.

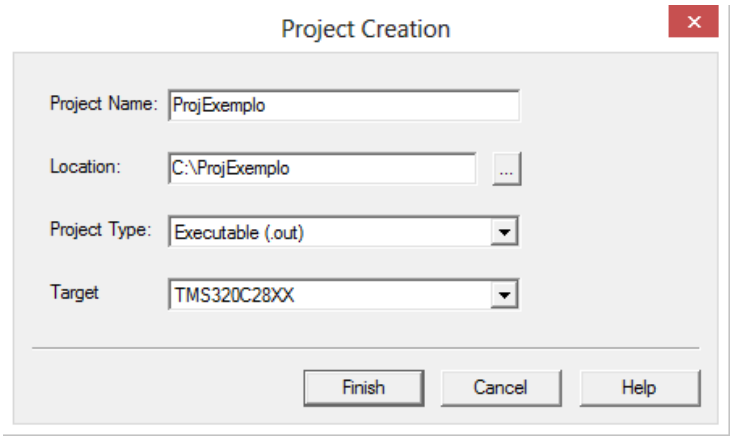

Figura 17 – Panela de escolha do nome e do diretório do projeto.

estão disponíveis, considerando a instalação padrão do pacote.

Os arquivos disponíveis no diretório "*C:\tidcs\c28\DSP2833x\v131\DSP2833x_headers\include*" são os seguintes:

- DSP2833x_Adc.h
- DSP2833x_CpuTimers.h
- DSP2833x_DevEmu.h
- DSP2833x_Device.h
- DSP2833x_DMA.h

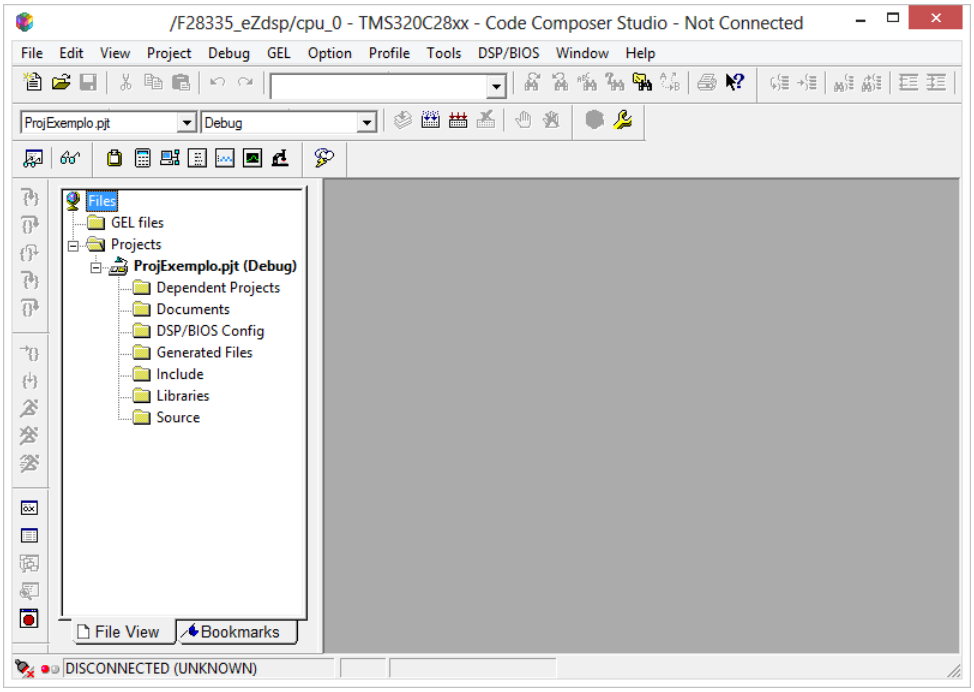

Figura 18 – Janela inicial após a criação do projeto.

- DSP2833x_ECan.h
- DSP2833x_ECap.h
- DSP2833x_EPwm.h
- DSP2833x_EQep.h
- DSP2833x_Gpio.h
- DSP2833x_I2c.h
- DSP2833x_Mcbsp.h
- DSP2833x_PieCtrl.h
- DSP2833x_PieVect.h
- DSP2833x_Sci.h
- DSP2833x_Spi.h
- DSP2833x_SysCtrl.h
- DSP2833x_Xintf.h
- DSP2833x_XIntrupt.h

O restante dos arquivos de cabeçalho está disponível no diretório "*C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\include*" e são os seguintes:

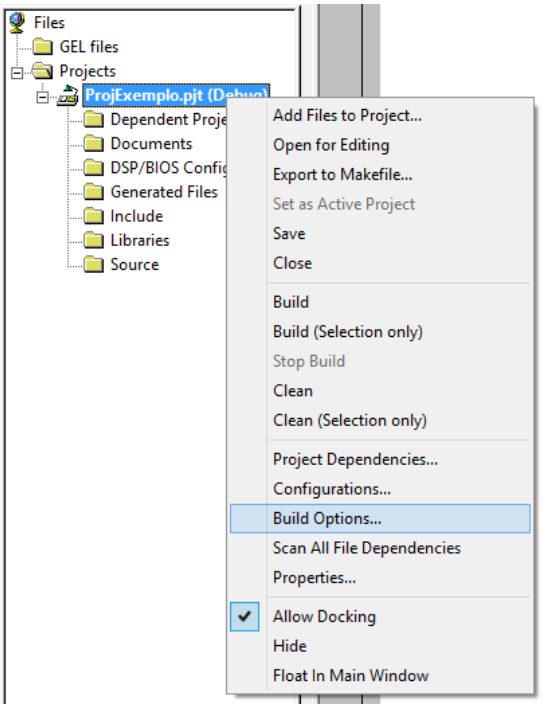

Figura 19 – Alterando o diretório de busca dos arquivos de cabeçalho, passo 1.

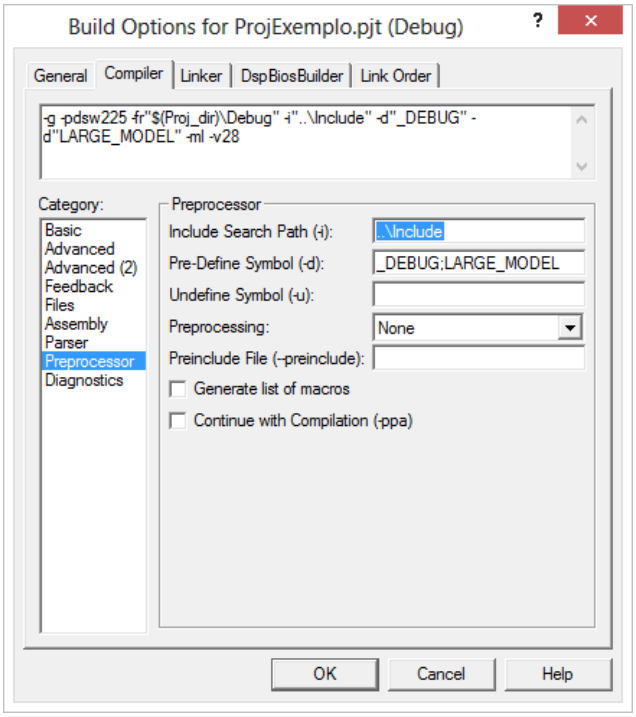

Figura 20 – Alterando o diretório de busca dos arquivos de cabeçalho, passo 2.

- DSP28x_Project.h
- DSP2833x_DefaultIsr.h
- DSP2833x_Dma_defines.h
- DSP2833x_EPwm_defines.h
- DSP2833x_Examples.h
- DSP2833x_GlobalPrototypes.h
- DSP2833x_I2c_defines.h

Para adicioná-los ao projeto foi criada uma pasta com o nome "*Include*" no diretório no qual o projeto base foi criado. No caso este diretório é "*C:\ProjExemplo*". Então, os arquivos listados foram copiados dos diretórios originais e colados para a pasta criada.

Desse modo, teve-se que modificar o diretório padrão no qual o pré-processador busca os arquivos de cabeçalho. Para isso, na janela do CCS deve-se clicar com o botão direito do mouse no nome do projeto criado, conforme mostrado na Figura 19, e escolher a opção "*Build Options...*". Fazendo isto, a janela mostrada na Figura 20 se abrirá. Na aba "*Compiler*", coluna "*Category:*", deve-se clicar em "*Processor*". No campo "*Include Search Path (-i):*" deve-se digitar "*..\Include*". Então, a janela deve ser fechada clicando-se em "*OK*", de modo que os arquivos de cabeçalho serão adicionados ao projeto quando ele for

compilado, o que só deve acontecer quando todos os arquivos necessários forem incluídos e criados.

2.2.2 Acrescentando os arquivos fonte e *batch*

Dentre os 16 arquivos fonte necessários ao projeto, 12 deles foram retirados do pacote de projetos exemplos. Eles são todos listados a seguir:

- DSP2833x_Adc.c
- DSP2833x_ADC_cal.asm
- DSP2833x_CodeStartBranch.asm
- DSP2833x_DefautIsr.c
- DSP2833x_EPwm.c
- DSP2833x_GlobalVariableDefs.c
- DSP2833x_PieCtrl.c
- DSP2833x_PieVect.c
- DSP2833x_Sci.c
- DSP2833x_Spi.c
- DSP2833x_SysCtrl.c
- DSP2833x_usDelay.asm

Todos os arquivos fonte supracitados estão disponíveis no diretório "*C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\source*", com exceção do arquivo "*DSP2833x_GlobalVariableDefs.c*", que está disponível no diretório "*C:\tidcs\c28\DSP2833x\v131\DSP2833x_headers\source*".

Para adicionar estes arquivos fonte ao projeto, foi criada uma pasta com o nome "*Source*" no diretório no qual o projeto base foi criado, similarmente a como foi feito para adicionar os arquivos de cabeçalho. Então, os arquivos listados foram copiados dos diretórios originais e colados para a pasta criada. Em seguida, com o projeto base aberto na janela do CCS, a inclusão destes arquivos foi concretizada clicando-se com o botão direito do mouse na seção "*Source*" e escolhendo a opção "*Add Files to Project...*", conforme mostrado na Figura 21.

Na janela que se abre deve-se navegar até a pasta para a qual os arquivos fonte foram copiados. No campo "*Files of type:*" desta janela deve-se escolher a opção "*All Files*

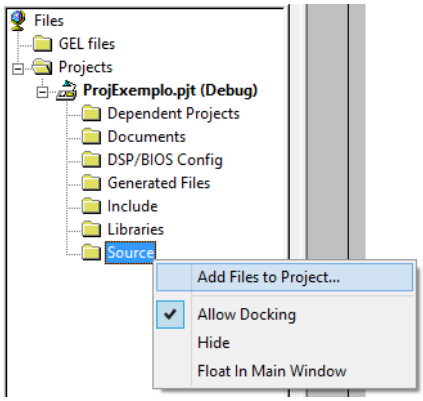

Figura 21 – Adicionando arquivos fonte.

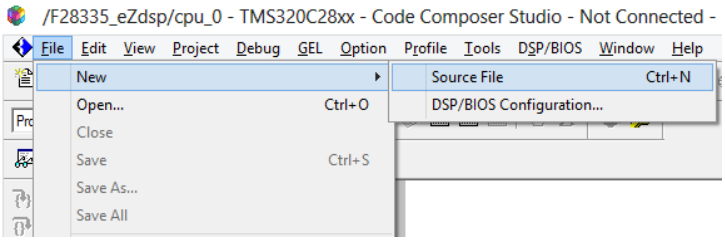

Figura 22 – Criando arquivos fonte.

(.*)*", caso contrário não será possível visualizar os arquivos do tipo "*.asm*", com códigos em *assembly*. Então, deve-se selecionar todos os arquivos da pasta e clicar em "*Open*". Isto adiciona os arquivos ao projeto imediatamente após a janela se fechar, de modo que se torna possível visualizar seus conteúdos e editá-los.

Dentre estes arquivos fonte, apenas um deve ter seu código modificado após ser adicionado ao projeto, o "*DSP2833x_EPwm.c*". Nele, todas as diretivas "*#if*" e "*#endif*" relacionadas as funções "*InitEPwm4Gpio()*", "*InitEPwm5Gpio()*" e "*InitEPwm6Gpio()*" foram retiradas, assim os seis módulos ePWM do DSP são sempre inicializados na chamada da função "*InitEPwmGpio()*". Mais detalhes sobre esta função serão dados no capítulo 3.

Já os 2 arquivos *batch* necessários ao projeto são listados a seguir, seguidos dos diretórios em que estão disponíveis:

- DSP2833x_Headers_nonBIOS.cmd
  Disponível em "*C:\tidcs\c28\DSP2833x\v131\DSP2833x_headers\cmd*"
- 28335_RAM_lnk.cmd
  Disponível em "*C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\cmd*"

Para adicioná-los ao projeto o procedimento é muito parecido com o de adicionar os arquivos fonte. Foi criada uma pasta com o nome "*cmdFiles*" no diretório do projeto e no lugar de se clicar com o botão direito na seção "*Source*" foi clicado no próprio nome do projeto e escolhida a opção "*Add Files to Project...*". Na janela que se abre deve-se navegar para a pasta na qual os arquivos foram copiados, selecionando-os e clicando em "*Open*" para finalizar o processo.

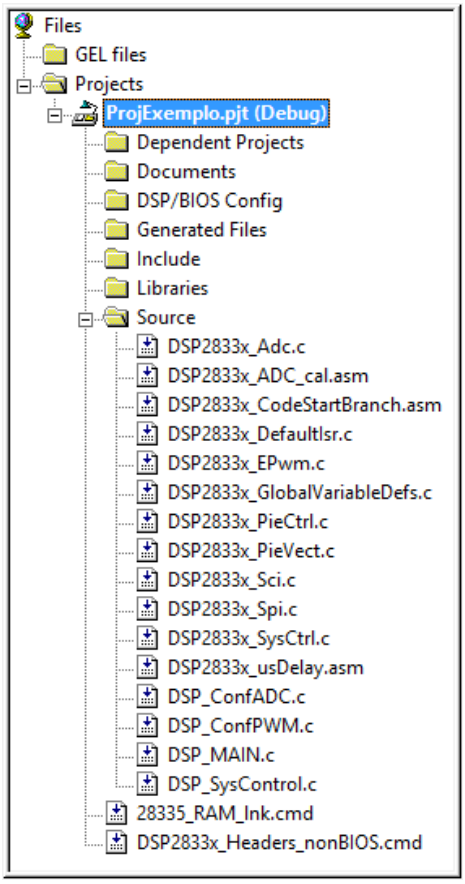

Figura 23 – Estrutura final do projeto base antes da primeira compilação.

2.2.3 Criando arquivos fonte adicionais

Restam ainda quatro arquivos fonte a serem adicionados, que não estão presentes no pacote adquirido no site da TI. Estes arquivos foram criados e seus conteúdos serão mostrados e explicados nos capítulos 3 e 4. Para o objetivo da criação de um projeto base em branco basta incluí-los sem conteúdo, por enquanto.

Para isso, no menu da parte superior da janela do CCS deve-se clicar em "*File*", "*New*" e escolher a opção "*Source File*", conforme mostrado na Figura 22. Uma forma mais rápida de fazer a mesma coisa é pressionar "*Ctrl+N*" no teclado. Um arquivo em branco se abrirá no campo de edição. Então, deve-se novamente clicar em "*File*" no menu e em seguida escolher "*Save as...*". Este arquivo deve ser salvo com o nome "*DSP_MAIN.c*" na pasta "*Source*", criada no diretório do projeto para os arquivos fonte. Mais três arquivos devem ser salvos seguindo este mesmo procedimento com os nomes "*DSP_ConfADC.c*", "*DSP_ConfPWM.c*" e "*DSP_SysControl.c*". Em seguida estes arquivos devem ser adicionados ao projeto seguindo o mesmo procedimento dos outros arquivos fonte já incluídos. Ao final, o projeto base deve ter a estrutura mostrada na Figura 23.

3 O Arquivo *DSP_MAIN.c* e a Configuração do DSP28335

O DSP28335 possui diversos periféricos, motivo pelo qual ele é considerado um microcontrolador. Dentre estes periféricos, dois são de maior importância para o projeto base. Eles são o módulo ADC e os módulos ePWM. O módulo ADC é importante principalmente para o sistema de controle, uma vez que as medições de tensão e corrente feitas no sistema de potência chegam no DSP por ele. Já os módulos ePWM são de fundamental importância para a aplicação da estratégia PWM, por meio da qual será feito o chaveamento dos braços dos conversores. Para que estes módulos sejam utilizados, eles precisam antes ser inicializados e configurados, o que foi feito por meio de funções descritas nos arquivos fonte "*DSP_ConfADC.c*" e "*DSP_ConfPWM.c*", de modo que o código utilizado nestes arquivos são mostrados e detalhados neste capítulo.

Mas antes de se entrar em detalhes nas especificidades destas configurações, é preciso compreender o projeto num escopo mais amplo. Por este motivo, o estudo dos códigos do projeto foi iniciado a partir da função principal "*main()*", cujo código foi inserido no arquivo fonte "*DSP_MAIN.c*". Então, a partir da "*main()*" foi feito o detalhamento das instruções e funções que são chamadas subsequentemente. É importante destacar que a descrição dos códigos apresentadas neste capítulo foi resultado do estudo exaustivo da documentação técnica do DSP28335, sobretudo dos documentos (1–5). Deste modo, é recomendado que esta documentação seja consultada. O que é exposto neste relatório de forma alguma substitui a informação que é apresentada nela.

Feitas estas considerações, neste capítulo são mostrados e descritos as códigos que foram inseridos nos arquivos fonte "*DSP_MAIN.c*", "*DSP_ConfADC.c*" e "*DSP_ConfPWM.c*".

3.1 O arquivo *DSP_MAIN.c* e as funções *main()* e *ADC_Int()*

O projeto base criado no CCS para o DSP deve descrever uma sequência de instruções e funções a serem executadas. Estas últimas, por sua vez, podem conter mais sequências de instruções e outras funções aninhadas. De modo geral, estas funções servem para configurar o DSP e seus periféricos, além de descrever os cálculos e ações que serão executadas uma vez que tudo tenha sido configurado. Em particular, as funções são executadas segundo uma estrutura em árvore, conforme mostrado no fluxograma da Figura 24.

A maior parte destas funções são executadas apenas uma vez e servem para inici-

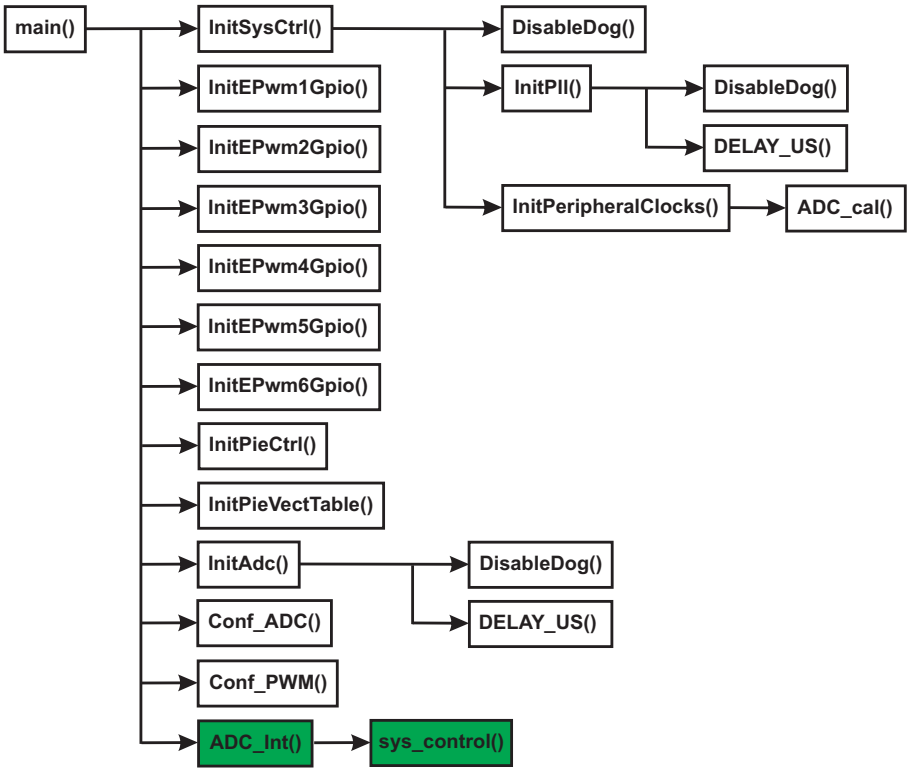

Figura 24 – Fluxograma de execução das funções no projeto compilado do DSP. A sequência de execução é de cima para baixo, e as setas indicam as funções que estão aninhadas em outras. Na origem da seta está a função raiz, na ponta está a função aninhada. As funções em verde são executadas dentro da interrupção do módulo ADC.

alizar e configurar periféricos, o que em essência é feito setando registradores de controle. Apenas as funções destacadas em verde na figura, no final da sequência, são executadas periodicamente, durante a interrupção do módulo ADC do DSP. Uma vez que o dispositivo tenha sido inicializado e configurado, ele vai operar segundo o que é descrito nestas funções, já que a interrupção ocorrerá periodicamente até que o funcionamento do sistema seja suspenso. A função "*sys_control()*" só será abordada no capítulo 4.

Como o próprio nome sugere, o arquivo fonte principal do projeto é o "*DSP_MAIN.c*", porque nele é descrita a função "*main()*", que é central para o projeto e dita a sequência de execução das funções e instruções responsáveis por configurar e definir as operações que serão executadas pelo dispositivo. Nele é definida também a função "*ADC_Int()*", que descreve quais ações serão executadas periodicamente a cada interrupção do módulo ADC do DSP, uma vez que todas as configurações tenham sido executadas.

A seguir é mostrado o código que foi inserido no arquivo "*DSP_MAIN.c*":

```
1 #include "DSP28x_Project.h"
2 #include "math.h"
3
4 #define AM 512        // Tamanho do Buffer Geral
5 #define pi 3.14159265 // Numero PI
```

```
6
7  // ***************************************************
8  //               DECLARACAO DE FUNCOES
9  // ***************************************************
10
11 extern void Conf_ADC(void);
12 extern void Conf_PWM(void);
13 extern void sys_control(long i, float var0, float var1,
14                         float var2, float var3, float var4,
15                         float var5, float var6, float var7,
16                         float var8, float var9, float var10,
17                         float var11, float var12, float var14);
18 interrupt void ADC_Int(void);
19
20 // ***************************************************
21 //         DECLARACAO DE VARIAVEIS GLOBAIS
22 // ***************************************************
23
24 int i = -1;
25
26 long AD0=0, AD1=0, AD2=0, AD3=0, AD4=0;
27 long AD5=0, AD6=0, AD7=0, AD8=0, AD9=0;
28 long AD10=0, AD11=0, AD12=0, AD13=0;
29 long AD14=0, AD15=0;
30
31 float var0=0., var1=0., var2=0., var3=0.;
32 float var4=0., var5=0., var6=0., var7=0.;
33 float var8=0., var9=0., var10=0., var11=0.;
34 float var12=0., var13=0., var14=0.;
35
36 // ***************************************************
37 //                 FUNCAO PRINCIPAL
38 // ***************************************************
39
40 void main(){
41     InitSysCtrl(); // Inicializa Sistema de Controle:
42                    // PLL, WatchDog, habilita perifericos;
43
44     // Configura os pinos de GPIO como saidas ePWMx
45     InitEPwm1Gpio(); // Inicializa ePWM1
46     InitEPwm2Gpio(); // Inicializa ePWM2
47     InitEPwm3Gpio(); // Inicializa ePWM3
```

```
48      InitEPwm4Gpio(); // Inicializa ePWM4
49      InitEPwm5Gpio(); // Inicializa ePWM5
50      InitEPwm6Gpio(); // Inicializa ePWM6
51
52      EALLOW;
53      GpioCtrlRegs.GPBMUX1.bit.GPIO32 = 0; // Configura GPIO32 como
54      GpioCtrlRegs.GPBDIR.bit.GPIO32 = 1; // saida de uso geral
55      EDIS;
56
57      GpioDataRegs.GPBDAT.bit.GPIO32=0; // Indica se a atualizacao do buffer
58                                        // ja foi concluida
59
60      InitPieCtrl(); // Inicializa os Registradores de Controle PIE
61                     // O estado padrao destes registradores eh zero
62                     // Os flags sao limpos
63
64      InitPieVectTable(); // Inicializa a tabela de vetores PIE
65
66      // Interrupts that are used in this example are re-mapped to
67      // ISR functions found within this file.
68      EALLOW;
69      PieVectTable.ADCINT = &ADC_Int;
70      EDIS;
71
72      EALLOW;
73      SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;
74      EDIS;
75
76      InitAdc(); // Inicializa o modulo ADC
77      Conf_ADC(); // Configura o modulo ADC
78      Conf_PWM(); // Configura os modulos ePWM;
79
80      EALLOW;
81      SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;
82      EDIS;
83
84      IER = 0xfff; // Habilita todas as interrupcoes da CPU controladas
85                   // pelo bloco PIE
86
87      // Habilita o ADCINT no PIE
88      PieCtrlRegs.PIECTRL.bit.ENPIE = 1; // Habilita o bloco PIE
89      PieCtrlRegs.PIEIER1.bit.INTx6 = 1;
```

```
90
91      // Habilita as interrupcoes globais e
92      // eventos debug em tempo real de maior prioridade:
93      EINT; // Habilita as interrupcoes no nivel da CPU
94      ERTM; // Habilita a interrupcao global em tempo real DBGM
95
96      // Espera a interrupcao do ADC
97      for(;;) {}
98  }
99
100 // ***************************************************
101 //          FUNCAO INTERRUPCAO DO ADC
102 // ***************************************************
103
104 interrupt void ADC_Int(void){
105     AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;       // Reinicia SEQ1
106     AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;     // Limpa o flag das
107                                             // interrupcoes SEQ1
108     PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Interrupcao PIEACK do PIE
109
110     AD0 = AdcRegs.ADCRESULT0 >> 4;
111     AD1 = AdcRegs.ADCRESULT1 >> 4;
112     AD2 = AdcRegs.ADCRESULT2 >> 4;
113     AD3 = AdcRegs.ADCRESULT3 >> 4;
114     AD4 = AdcRegs.ADCRESULT4 >> 4;
115     AD5 = AdcRegs.ADCRESULT5 >> 4;
116     AD6 = AdcRegs.ADCRESULT6 >> 4;
117     AD7 = AdcRegs.ADCRESULT7 >> 4;
118     AD8 = AdcRegs.ADCRESULT8 >> 4;
119     AD9 = AdcRegs.ADCRESULT9 >> 4;
120     AD10 = AdcRegs.ADCRESULT10 >> 4;
121     AD11 = AdcRegs.ADCRESULT11 >> 4;
122     AD12 = AdcRegs.ADCRESULT12 >> 4;
123     AD13 = AdcRegs.ADCRESULT13 >> 4;
124     AD14 = AdcRegs.ADCRESULT14 >> 4;
125
126     // DEFINICAO DAS CONSTANTES DE CALIBRACAO DOS SENSORES:
127     #define AD0_nDC 0.
128     #define AD0_Gain 1.
129
130     #define AD1_nDC 0.
131     #define AD1_Gain 1.
```

```
132
133     #define AD2_nDC 0.
134     #define AD2_Gain 1.
135
136     #define AD3_nDC 0.
137     #define AD3_Gain 1.
138
139     #define AD4_nDC 0.
140     #define AD4_Gain 1.
141
142     #define AD5_nDC 0.
143     #define AD5_Gain 1.
144
145     #define AD6_nDC 0.
146     #define AD6_Gain 1.
147
148     #define AD7_nDC 0.
149     #define AD7_Gain 1.
150
151     #define AD8_nDC 0.
152     #define AD8_Gain 1.
153
154     #define AD9_nDC 0.
155     #define AD9_Gain 1.
156
157     #define AD10_nDC 0.
158     #define AD10_Gain 1.
159
160     #define AD11_nDC 0.
161     #define AD11_Gain 1.
162
163     #define AD12_nDC 0.
164     #define AD12_Gain 1.
165
166     #define AD13_nDC 0.
167     #define AD13_Gain 1.
168
169     #define AD14_nDC 0.
170     #define AD14_Gain 1.
171
172     // Ad's --> var's:
173     var0 = (AD0-AD0_nDC)*AD0_Gain;
```

```
174     var1 = (AD1-AD1_nDC)*AD1_Gain;
175     var2 = (AD2-AD2_nDC)*AD2_Gain;
176     var3 = (AD3-AD3_nDC)*AD3_Gain;
177     var4 = (AD4-AD4_nDC)*AD4_Gain;
178     var5 = (AD5-AD5_nDC)*AD5_Gain;
179     var6 = (AD6-AD6_nDC)*AD6_Gain;
180     var7 = (AD7-AD7_nDC)*AD7_Gain;
181     var8 = (AD8-AD8_nDC)*AD8_Gain;
182     var9 = (AD9-AD9_nDC)*AD9_Gain;
183     var10 = (AD10-AD10_nDC)*AD10_Gain;
184     var11 = (AD11-AD11_nDC)*AD11_Gain;
185     var12 = (AD12-AD12_nDC)*AD12_Gain;
186     var13 = (AD13-AD13_nDC)*AD13_Gain;
187     var14 = (AD14-AD14_nDC)*AD14_Gain;
188
189     sys_control(i,var0,var1,var2, var3, var4,
190                 var5, var6, var7, var8, var9,
191                 var10, var11, var12, var14);
192
193     if(i<AM-1)
194         i++;
195     else
196         i=0;
197 }
```

Analisando o código do arquivo "*DSP_MAIN.c*", percebe-se que primeiramente os arquivos de cabeçalho "*DSP28x_Project.h*" e "*math.h*" são incluídos. Este último possibilita o uso de funções matemáticas, como a "*sin()*" e a "*cos()*". Já o primeiro incluí mais dois outros arquivos de cabeçalho, o "*DSP2833x_Examples.h*" e o "*DSP2833x_Device.h*".

Dentre estes, o primeiro é responsável por definir várias constante importantes para o funcionamento do DSP, e.g., a constante "*CPU_RATE*", que é usada para definir o *clock* da CPU. O arquivo "*DSP2833x_Examples.h*" também incluí outro arquivo de cabeçalho, o "*DSP2833x_GlobalPrototypes.h*", que declara os protótipos das funções globais compartilhadas entre os arquivos fonte do projeto.

Já a inclusão do arquivo "*DSP2833x_Device.h*" é importante porque nele são definidas mais constantes que são utilizadas nos arquivos do projeto, além de serem também definidas macros de instruções em *assembly*, como a "*EALLOW*" e a "*EDIS*", presentes em praticamente todos os arquivos fonte e que servem respectivamente para habilitar e desabilitar a escrita em registradores protegidos do DSP. Neste arquivo de cabeçalho também são definidos alguns tipos de variáveis usadas no projeto, como os tipos "*int16*", o

"*int32*" e o "*int64*", além de também serem incluídos todos os arquivos de cabeçalho importantes para todos os periféricos do DSP28335, independentemente deles serem usados ou não.

Depois da inclusão dos arquivos de cabeçalho, algumas constantes úteis à execução do programa são definidas, como a "*AM*", que é o número de valores que podem ser armazenados em vetores que servirão para visualizar variáveis do programa durante o funcionamento do sistema. Esta funcionalidade é muito útil para fazer a depuração do sistema como um todo, permitindo identificar falhas no programa, nos equipamentos de instrumentação e até na própria montagem.

Depois são declarados os protótipos de funções que não foram declarados nos arquivos de cabeçalho incluídos, e na sequência são declaradas as variáveis que serão utilizadas. Percebe-se que a declaração destas variáveis é feita de forma global, antes da função "*main()*".

Em seguida as funções "*main()*" e "*ADC_Int()*" são definidas. Suas descrições são feitas nas subseções seguintes.

3.1.1 A função *main()*

Analisando o código da função "*main()*", percebe-se que a primeira instrução executada é a chamada da função "*InitSysCtrl()*", que é definida no arquivo fonte "*DSP2833x_SysCtrl.c*". Ela tem como função inicializar o sistema de controle do DSP. A inicialização é feita setando os registradores relacionados para valores iniciais conhecidos. Basicamente, esta função desabilita o *watchdog* do sistema, inicializa o controle de PLL setando o registrador de controle do PLL (PLLCR) e inicializa os *clocks* dos periféricos do DSP, setando o pré-escalonador para *clocks* de alta e baixa frequência e os habilitando.

Depois, são chamadas as funções de "*InitEPwm1Gpio()*" a "*InitEPwm6Gpio()*", que estão definidas no arquivo fonte "*DSP2833x_EPwm.c*" e servem para inicializar os seis módulos ePWM disponíveis no DSP28335. Cada módulo possui duas saídas independentes, o que totaliza 12 PWMs. A inicialização dos módulos consiste basicamente em habilitar os resistores *pull-up* das saídas ePWM de cada módulo e configurar os respectivos pinos de GPIO como saídas ePWM.

As instruções seguintes servem para configurar o pino de GPIO 32. A instrução "**GpioCtrlRegs.GPBMUX1.bit.GPIO32 = 0;**" configura o pino como entrada ou saída de uso geral. Já a instrução "**GpioCtrlRegs.GPBDIR.bit.GPIO32 = 1;**" indica, no caso deste pino ser efetivamente usado como GPIO, se ele é entrada (0) ou saída (1). Neste caso ele é configurado como saída. Na placa do DSP do Laboratório este pino está associado a um LED. Após ele ser devidamente configurado, a linha de código "**GpioDataRegs.GPBDAT.bit.GPIO32 = 0;**" serve para enviar o nível lógico 0 (zero)

para este pino, o que deve apagar este LED, servindo como indicador de que o programa está em execução.

Em seguida é chamada a função "*InitPieCtrl()*", descrita no arquivo fonte "*DSP 2833x_PieCtrl.c*" e que serve para inicializar os registradores de controle do bloco PIE para valores iniciais conhecidos, i.e., zero. Este bloco é a parte do DSP responsável por administrar as interrupções geradas pelos periféricos. Ele serve como multiplexador de várias fontes de interrupção em um número menor de entradas, e pode suportar até 96 interrupções de periféricos, dentre as quais apenas 58 são usadas pelos periféricos do DSP28335, sendo que as restantes são reservadas versões futuras do dispositivo.

Depois a função "*InitPieVectTable()*" é chamada. Ela é definida no arquivo "*DSP 2833x_PieVect.c*" e serve para inicializar a tabela de vetores do bloco PIE. Nesta tabela todas as fontes de interrupção do DSP são relacionadas e divididas em grupos. A interrupção do módulo ADC, que é de interesse para o projeto base, está no grupo 1.

A linha de instrução "**PieVectTable.ADCINT = &ADC_Int;**" serve para especificar ao bloco PIE que as instruções presentes na função "*ADC_Int()*" devem ser executadas quando a interrupção do módulo ADC ocorrer.

As linhas de instrução seguintes estão relacionadas com a inicialização e configuração do módulo ADC e com a configuração dos módulos PWM. As execução das instruções "**SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;**" e "**SysCtrlRegs.PCLKCR0. bit.TBCLKSYNC = 1;**" que estão no começo e no fim do trecho referenciado, entre as macros "*EALLOW*" e "*EDIS*", é um procedimento padrão recomendável quando se deseja habilitar o *clock* dos módulos PWM perfeitamente sincronizados com os respectivos TBCLKs. Isto é, primeiramente o bit TBCLKSYNC do registrador PCLKCR0 é setado igual a zero para desabilitar os *clocks* dos módulos ePWM. Então, eles são cofigurados segundo a função "*Conf_PWM()*", no arquivo fonte "*DSP_ConfPWM.c*", e depois os *clocks* são reabilitados para que sejam sincronizados com os TBCLKs.

Na sequência, a função "*InitADC()*" é chamada. Ela é descrita no arquivo "*DSP 2833x_Adc.c*" e inicializa o módulo ADC para um estado inicial conhecido. Analisando o código da função, percebe-se que primeiramente o *clock* do ADC é habilitado através da instrução "**SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 1;**". O resgistrador PCLKCR0 é um dentre os três registradores de controle do *clock* dos periféricos (os outros dois são o PCLKCR1 e o PCLKCR2). Depois é chamada a função "*ADC_cal()*", que copia os valores de calibração para o ADC da memória ROM para os devidos registradores do módulo. Antes da chamada desta função, o *clock* do ADC deve obrigatoriamente ter sido habilitado. Na sequência, as três últimas linhas de código servem respectivamente para dar o *power-up* nos circuitos de *bandgap*, de referência e analógico do ADC, além de definir um *delay* de $5ms$ antes da primeira conversão, para que haja tempo do circuito analógico se estabilizar (linha de código "*DELAY_US(ADC_usDELAY);*").

As funções que são chamadas a seguir são a "*Conf_ADC()*" e a "*Conf_PWM()*". Elas estão definidas respectivamente nos arquivos fonte "*DSP_ConfADC.c*" e "*DSP_Conf PWM.c*" e terão seções específicas para a descrição de cada uma, mas em termos gerais elas servem respectivamente para fazer a configuração do módulo ADC e dos módulos PWM inicializados.

Depois que os *clocks* dos módulos ePWM são reabilitados, a instrução "*IER = 0xffff;*" é executada para habilitar todos os níveis de interrupções do bloco PIE. As instruções "*PieCtrlRegs.PIECTRL.bit.ENPIE = 1;*" e "*PieCtrlRegs.PIEIER1.bit.INTx6 = 1;*" são executadas em seguida para respectivamente habilitar o bloco PIE e habilitar a interrupção do ADC (o bit INTx6 do registrador PIEIER1 habilita a interrupção do ADC localmente). Por fim, as instruções "**EINT;**" e "**ERTM;**" são macros definidas no arquivo de cabeçalho "*DSP2833x_Device.h*" que executam instruções em *assembly* para limpar os bits INTM e DBGM, de modo que todas as interrupções bloqueáveis por *software* sejam globalmente habilitadas e os eventos de *debug* estejam habilitados em tempo real.

Por fim, um loop infinito é executado pelo laço "**for(;;){ }**", de modo que o DSP fique apenas aguardando pela ocorrência de uma interrupção, que no caso deste projeto base é a interrupção do módulo ADC, cuja configuração será descrita ainda neste capítulo.

3.1.2 A função *ADC_Int()*

Esta função é executada depois de ser gerado um evento de interrupção após cada conversão do módulo ADC, cuja a configuração é detalhada na seção 3.2. Na primeira linha, a instrução "**AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;**" serve para reiniciar o sequenciador SEQ1 do módulo ADC, de modo que a sequência de conversão do módulo seja reiniciada. Mais detalhes sobre isto serão dados na seção 3.2.

Na linha seguinte, a instrução "**AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;**" limpa o *flag* das interrupções do sequenciador SEQ1 do ADC. Isto deve ser feito manualmente após o início de cada chamada da função "*ADC_Int()*" para que outra interrupção só seja requisitada depois da ocorrência de uma nova conversão do módulo.

A instrução seguinte, "**PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;**", serve para limpar o bit de reconhecimento de interrupção do grupo 1 do bloco PIE, permitindo que ele envie um pulso para a entrada de interrupção da CPU se uma nova interrupção estiver pendente para este grupo. Na prática, isto possibilita que uma nova interrupção seja requisitada.

As linhas seguintes, que vão da instrução "**AD0 = AdcRegs.ADCRESULT0 » 4;**" até a "**AD14 = AdcRegs.ADCRESULT14 » 4;**", servem para fazer a leitura dos 15 registradores de resultados do ADC que são utilizados, armazenando seus valores. As linhas contendo as diretivas "*#define*" que seguem servem para definir cons-

tantes de calibração que serão aplicadas aos valores lidos dos registradores de resultado do ADC, fazendo as correções necessárias segundo um processo de calibração que consiste basicamente em observar valores lidos do ADC e comparar com valores medidos por outro instrumento de confiabilidade garantida, aplicando em seguida valores a essas constantes que permitam remover o *offset* da medição e corrigi-la segundo uma constante de proporcionalidade adequada. As instruções que vão da "**var0 = (AD0 - AD0_nDC)*AD0_Gain;**" até a "**var14 = (AD14 - AD14_nDC)*AD14_Gain;**" efetivam estas correções, atribuindo os valores corrigidos às variáveis da "*var0*" a "*var14*".

Por fim, a função "*sys_control()*" é chamada. Em linhas gerais, esta função executa a estratégia PWM e implementa o sistema de controle. Ela está definida no arquivo "*DSP_SysControl.c*" e será discutida em detalhes no capítulo 4. Depois de sua execução, o valor do índice "*i*" é atualizado. Este índice é utilizado na função "*sys_control()*" para salvar variáveis em vetores globalmente declarados e inicializados. O objetivo é que estas variáveis possam ser observadas graficamente no CCS durante a execução do programa no DSP, permitindo depurar erros no sistema de modo geral.

3.2 O arquivo *DSP_ConfADC.c* e a função *Conf_ADC()*

O arquivo "*DSP_ConfADC.c*" tem como principal objetivo fazer a configuração do módulo ADC. O código que foi inserido nele é mostrado a seguir:

```
1  #include "DSP28x_Project.h"
2
3  #define ADC_MODCLK 3
4  #define ADC_SHCLK 15
5
6  void Conf_ADC(void){
7      AdcRegs.ADCTRL1.bit.CPS = 1;
8      AdcRegs.ADCTRL1.bit.ACQ_PS = ADC_SHCLK;
9      AdcRegs.ADCTRL1.bit.CONT_RUN = 0;
10     AdcRegs.ADCTRL1.bit.SEQ_CASC = 1;
11
12     AdcRegs.ADCTRL2.bit.EPWM_SOCA_SEQ1 = 1; // Habilita o SOCA do ePWM
13                                             // para iniciar o SEQ1
14     AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1; // Habilita a interup. para o SEQ1
15
16     AdcRegs.ADCTRL3.bit.SMODE_SEL=0; // Desabilita conversao simultanea dos S/H
17     AdcRegs.ADCTRL3.bit.ADCCLKPS = ADC_MODCLK;
18
19     AdcRegs.ADCMAXCONV.all = 0x000E; // Numero maximo de conversoes
```

```
20
21      // Configura a ordem das conversoes
22      AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x0000; // Define a 1a conversao
23      AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 0x0001; // Define a 2a conversao
24      AdcRegs.ADCCHSELSEQ1.bit.CONV02 = 0x0002; // Define a 3a conversao
25      AdcRegs.ADCCHSELSEQ1.bit.CONV03 = 0x0003; // Define a 4a conversao
26      AdcRegs.ADCCHSELSEQ2.bit.CONV04 = 0x0004; // Define a 5a conversao
27      AdcRegs.ADCCHSELSEQ2.bit.CONV05 = 0x0005; // Define a 6a conversao
28      AdcRegs.ADCCHSELSEQ2.bit.CONV06 = 0x0006; // Define a 7a conversao
29      AdcRegs.ADCCHSELSEQ2.bit.CONV07 = 0x0007; // Define a 8a conversao
30      AdcRegs.ADCCHSELSEQ3.bit.CONV08 = 0x0008; // Define a 9a conversao
31      AdcRegs.ADCCHSELSEQ3.bit.CONV09 = 0x0009; // Define a 10a conversao
32      AdcRegs.ADCCHSELSEQ3.bit.CONV10 = 0x000A; // Define a 11a conversao
33      AdcRegs.ADCCHSELSEQ3.bit.CONV11 = 0x000B; // Define a 12a conversao
34      AdcRegs.ADCCHSELSEQ4.bit.CONV12 = 0x000C; // Define a 13a conversao
35      AdcRegs.ADCCHSELSEQ4.bit.CONV13 = 0x000D; // Define a 14a conversao
36      AdcRegs.ADCCHSELSEQ4.bit.CONV14 = 0x000E; // Define a 15a conversao
37
38      EPwm1Regs.ETSEL.bit.SOCAEN = 1; // Habilita o SOC no grupo A
39      EPwm1Regs.ETSEL.bit.SOCASEL = 2; // Seleciona o SOC do CPMA no upcount
40      EPwm1Regs.ETPS.bit.SOCAPRD = 1; // Gera o pulso no 1o eventos
41  }
```

A primeira linha de instrução da função "*Conf_ADC()*", "**AdcRegs.ADCTRL1.bit.CPS = 1;**", serve para configurar o bit CPS do registrador de controle ADCTRL1 do ADC. Este bit é um dos dois pré-escalonadores do *clock* do módulo ADC. Caso ele seja setado, o fator de divisão do *clock* do ADC será 2, que é o caso da configuração feita.

Depois, na linha de instrução "**AdcRegs.ADCTRL1.bit.ACQ_PS = ADC_SHCLK;**" é definido o tamanho da janela de aquisição do S/H em função do *clock* do ADC. O ACQ_PS é um campo do registrador ADCTRL1 que tem 4 bits, podendo variar numericamente de 0 a 15. O tamanho da janela de aquisição será dado por (ACQ_PS+1) vezes o período do *clock* do ADC. Neste caso ele é 16 vezes o período do *clock*, de acordo com o valor definido para a constante ADC_SHCLK.

A linha de instrução seguinte, "**AdcRegs.ADCTRL1.bit.CONT_RUN = 0;**", configura os sequenciadores para operar no modo *start-stop*. Eles podem operar em dois modos, o *continuous conversion* ou o *start-stop*. No primeiro modo os sequenciadores operam continuamente, i.e., sempre que o sinal de gatilho ocorrer, a sequência de conversão é realizada e os registradores de resultado são sobrescritos de acordo com uma sequência a ser definida. No segundo modo os sequenciadores devem ser reiniciados antes via *software*

para que uma nova conversão seja feita, caso contrário quando o gatilho ocorrer os registradores manterão o mesmo valor da última sequência de conversão feita. Como mostrado e explicado na seção 3.1, na parte destinada à função "*ADC_Int()*", esta reinicialização é feita sempre no início das rotinas de interrupção do ADC.

Depois, a instrução "**AdcRegs.ADCTRL1.bit.SEQ_CASC = 1;**" configura outro modo de operação dos sequenciadores do ADC, que neste caso são ajustados para operar em cascata. O módulo ADC tem dois sequenciadores de 8 bits, o SEQ1 e o SEQ2. Eles podem operar de forma independente, cada um com o seu gatilho, ou operar em cascata como um único sequenciador de 16 bits, com um gatilho em comum.

A linha de instrução seguinte, "**AdcRegs.ADCTRL2.bit.EPWM_SOCA_SEQ1 = 1;**", configura as conversões dos sequenciadores do ADC para serem iniciadas pelo gatilho SOCA, que vem dos módulos ePWM. Deste modo, os ePWMs podem ser programados para iniciar as conversões do ADC.

Já a instrução "**AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1;**" serve para habilitar o pedido de interrupção do sequenciador SEQ1 à CPU após a realização de uma conversão.

O ADC tem a capacidade de amostrar duas entradas analógicas simultaneamente, desde que uma entrada analógica esteja entre as entradas ADCINA0 e ADCINA7 e a outra esteja entre as entradas ADCINB0 e ADCINB7. A linha de instrução "**AdcRegs.ADCTRL3.bit.SMODE_SEL=0;**" desabilita esta função, de modo que apenas uma amostragem é feita por vez.

O outro pré-escalonador do *clock* do ADC é configurado pela linha de instrução seguinte, "**AdcRegs.ADCTRL3.bit.ADCCLKPS = ADC_MODCLK;**". O campo ADCCLKPS do registrador de controle ADCTRL3 do ADC tem um tamanho de 4 bits, podendo ajustar um fator de divisão que vai de 1 a 30, variando de 2 em 2 a partir de 2. Combinado com o bit CPS do registrador ADCTRL1, o fator de divisão para gerar o *clock* pré-escalonado do ADC será dador por "$2 \cdot ADCCLKPS \cdot (CPS+1)$", caso ADCCLKPS não seja 0 (zero), e "$CPS+1$" caso ADCCLKPS seja 0 (zero).

Depois é definido o número máximo de conversões que devem ser realizadas pelo ADC, por meio da linha de instrução "**AdcRegs.ADCMAXCONV.all = 0x000E;**". Neste caso, o número máximo é ajustado para 15, ou seja, segundo esta configuração poderão ser usados 15 dos 16 canais ADC disponíveis. Em seguida, nas próximas 15 linhas é definida a ordem de conversão dos sequenciadores, i.e., é definida em que sequência os sinais das entradas analógicas vão ser convertidos.

As três últimas linhas que seguem são necessárias para configurar o módulo ePWM1 para gerar o gatilho para os sequenciadores do ADC. A instrução "**EPwm1Regs.ETSEL.bit.SOCAEN = 1;**" habilita a geração do pulso de gatilho para o ADC. Já a instru-

ção "**EPwm1Regs.ETSEL.bit.SOCASEL = 2;**" seleciona uma dentre as seis opções disponíveis de eventos que servirão de referência para a geração dos pulsos. Neste caso a opção selecionada é quando o contador TBCTR for igual ao período definido no registrador TBPRD. Os detalhes sobre o que são o contador TBCTR e o registrador TBPRD serão dados na seção 3.3, mas na prática este evento ocorre a cada período da portadora triangular do PWM, quando ela atingir seu valor máximo. Por fim, a instrução seguinte, "**EPwm1Regs.ETPS.bit.SOCAPRD = 1;**", configura quantos eventos são necessários para que um pulso seja gerado. Podem ser escolhidos de 0 a 3 eventos. A escolha de 0 eventos significa desabilitar a geração do gatilho. Neste caso ele é configurado para ser gerado a cada evento.

3.3 O arquivo *DSP_ConfPWM.c* e a função *Conf_PWM()*

A função "*Conf_PWM()*" é descrita no arquivo "*DSP_ConfPWM.c*", e nela é feita a configuração dos seis módulos ePWM. Mas antes de se iniciar a explicação referente a estas configurações, deve-se destacar que dentre os módulos o ePWM1 deve ser configurado como mestre, de modo que o restante deles sejam escravos. Mesmo assim, as configurações são muito semelhantes, assim a descrição é feita para o ePWM1 e generalizada para os outros, fazendo-se apenas considerações sobre as alterações que devem ser feitas, quando necessário.

O código que foi inserido no arquivo "*DSP_ConfPWM.c*" é mostrado a seguir:

```
1  #include "DSP28x_Project.h"
2
3  #define EPWM1_TIMER_TBPRD 7500; // Definicao do periodo
4  #define EPWM2_TIMER_TBPRD 7500; // Definicao do periodo
5  #define EPWM3_TIMER_TBPRD 7500; // Definicao do periodo
6  #define EPWM4_TIMER_TBPRD 7500; // Definicao do periodo
7  #define EPWM5_TIMER_TBPRD 7500; // Definicao do periodo
8  #define EPWM6_TIMER_TBPRD 7500; // Definicao do periodo
9
10 void Conf_PWM(void){
11     // ************************************************
12     //              ePWM1 configuration
13     // ************************************************
14
15     // Set period and phase
16     EPwm1Regs.TBPRD = EPWM1_TIMER_TBPRD; // Set timer period
17     EPwm1Regs.TBPHS.half.TBPHS = 0;   // Set phase
18
```

```
19      // Setup counter mode and TBCLK prescalers
20      EPwm1Regs.TBCTR = 0;                         // Clear counter
21      EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up/down
22      EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;   // Clock ratio to SYSCLKOUT
23      EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV1;      // Clock ratio to SYSCLKOUT
24
25      // Setup phase loading
26      EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE; // Dis. phase loading (master)
27      EPwm1Regs.TBCTL.bit.PRDLD = TB_SHADOW;  // Enable shadow mode for TBPHS
28      EPwm1Regs.TBCTL.bit.SYNCOSEL = TB_CTR_ZERO; // Sync down-stream module
29
30      // Setup shadow register for CMPA and CMPB to load on ZERO
31      EPwm1Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW; // Enable shadow mode for CMPA
32      EPwm1Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW; // Enable shadow mode for CMPB
33      EPwm1Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // Load CMPA when TBCTR = 0
34      EPwm1Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // Load CMPB when TBCTR = 0
35
36      // Set actions when TBCTR = CMPA or TBCTR = CMPB
37      EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR; // Clear PWM1A on event A, UP count
38      EPwm1Regs.AQCTLA.bit.CAD = AQ_SET; // Set PWM1A on event A, DOWN count
39      EPwm1Regs.AQCTLB.bit.CBU = AQ_CLEAR; // Clear PWM1A on event A, UP count
40      EPwm1Regs.AQCTLB.bit.CBD = AQ_SET; // Set PWM1A on event B, DOWN count
41
42      // Deadtime configuration
43      EPwm1Regs.DBCTL.bit.OUT_MODE = DB_DISABLE; // Disable deadtime
44
45
46      // ************************************************
47      //              ePWM2 configuration
48      // ************************************************
49
50      // Set period and phase
51      EPwm2Regs.TBPRD = EPWM2_TIMER_TBPRD; // Set timer period
52      EPwm2Regs.TBPHS.half.TBPHS = 0;   // Set phase
53
54      // Setup counter mode and TBCLK prescalers
55      EPwm2Regs.TBCTR = 0;                         // Clear counter
56      EPwm2Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up/down
57      EPwm2Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;   // Clock ratio to SYSCLKOUT
58      EPwm2Regs.TBCTL.bit.CLKDIV = TB_DIV1;      // Clock ratio to SYSCLKOUT
59
60      // Setup phase loading
```

```
61      EPwm2Regs.TBCTL.bit.PHSEN = TB_ENABLE;  // Ena. phase loading (slave)
62      EPwm2Regs.TBCTL.bit.PHSDIR = TB_DOWN;   // Count DOWN on sync event
63      EPwm2Regs.TBCTL.bit.PRDLD = TB_SHADOW;  // Enable shadow mode for TBPHS
64      EPwm2Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN ; // Set sync flow-through
65
66      // Setup shadow register for CMPA and CMPB to load on ZERO
67      EPwm2Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW; // Enable shadow mode for CMPA
68      EPwm2Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW; // Enable shadow mode for CMPB
69      EPwm2Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // Load CMPA when TBCTR = 0
70      EPwm2Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // Load CMPB when TBCTR = 0
71
72      // Set actions when TBCTR = CMPA or TBCTR = CMPB
73      EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR; // Clear PWM2A on event A, UP count
74      EPwm2Regs.AQCTLA.bit.CAD = AQ_SET; // Set PWM2A on event A, DOWN count
75      EPwm2Regs.AQCTLB.bit.CBU = AQ_CLEAR; // Clear PWM2A on event A, UP count
76      EPwm2Regs.AQCTLB.bit.CBD = AQ_SET; // Set PWM2A on event B, DOWN count
77
78      // Deadtime configuration
79      EPwm2Regs.DBCTL.bit.OUT_MODE = DB_DISABLE; // Disable deadtime
80
81
82      // ***************************************************
83      //              ePWM3 configuration
84      // ***************************************************
85
86      // Set period and phase
87      EPwm3Regs.TBPRD = EPWM3_TIMER_TBPRD; // Set timer period
88      EPwm3Regs.TBPHS.half.TBPHS = 0;   // Set phase
89
90      // Setup counter mode and TBCLK prescalers
91      EPwm3Regs.TBCTR = 0;                        // Clear counter
92      EPwm3Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up/down
93      EPwm3Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;   // Clock ratio to SYSCLKOUT
94      EPwm3Regs.TBCTL.bit.CLKDIV = TB_DIV1;      // Clock ratio to SYSCLKOUT
95
96      // Setup phase loading
97      EPwm3Regs.TBCTL.bit.PHSEN = TB_ENABLE;  // Ena. phase loading (slave)
98      EPwm3Regs.TBCTL.bit.PHSDIR = TB_DOWN;   // Count DOWN on sync event
99      EPwm3Regs.TBCTL.bit.PRDLD = TB_SHADOW;  // Enable shadow mode for TBPHS
100     EPwm3Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN ; // Set sync flow-through
101
102     // Setup shadow register for CMPA and CMPB to load on ZERO
```

```
103     EPwm3Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW; // Enable shadow mode for CMPA
104     EPwm3Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW; // Enable shadow mode for CMPB
105     EPwm3Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // Load CMPA when TBCTR = 0
106     EPwm3Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // Load CMPB when TBCTR = 0
107
108     // Set actions when TBCTR = CMPA or TBCTR = CMPB
109     EPwm3Regs.AQCTLA.bit.CAU = AQ_CLEAR; // Clear PWM3A on event A, UP count
110     EPwm3Regs.AQCTLA.bit.CAD = AQ_SET; // Set PWM3A on event A, DOWN count
111     EPwm3Regs.AQCTLB.bit.CBU = AQ_CLEAR; // Clear PWM3A on event A, UP count
112     EPwm3Regs.AQCTLB.bit.CBD = AQ_SET; // Set PWM3A on event B, DOWN count
113
114     // Deadtime configuration
115     EPwm3Regs.DBCTL.bit.OUT_MODE = DB_DISABLE; // Disable deadtime
116
117
118     // ************************************************
119     //              ePWM4 configuration
120     // ************************************************
121
122     // Set period and phase
123     EPwm4Regs.TBPRD = EPWM4_TIMER_TBPRD; // Set timer period
124     EPwm4Regs.TBPHS.half.TBPHS = 0;   // Set phase
125
126     // Setup counter mode and TBCLK prescalers
127     EPwm4Regs.TBCTR = 0;                        // Clear counter
128     EPwm4Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up/down
129     EPwm4Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;   // Clock ratio to SYSCLKOUT
130     EPwm4Regs.TBCTL.bit.CLKDIV = TB_DIV1;      // Clock ratio to SYSCLKOUT
131
132     // Setup phase loading
133     EPwm4Regs.TBCTL.bit.PHSEN = TB_ENABLE;  // Ena. phase loading (slave)
134     EPwm4Regs.TBCTL.bit.PHSDIR = TB_DOWN;   // Count DOWN on sync event
135     EPwm4Regs.TBCTL.bit.PRDLD = TB_SHADOW;  // Enable shadow mode for TBPHS
136     EPwm4Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN ; // Set sync flow-through
137
138     // Setup shadow register for CMPA and CMPB to load on ZERO
139     EPwm4Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW; // Enable shadow mode for CMPA
140     EPwm4Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW; // Enable shadow mode for CMPB
141     EPwm4Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // Load CMPA when TBCTR = 0
142     EPwm4Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // Load CMPB when TBCTR = 0
143
144     // Set actions when TBCTR = CMPA or TBCTR = CMPB
```

```
145     EPwm4Regs.AQCTLA.bit.CAU = AQ_CLEAR; // Clear PWM4A on event A, UP count
146     EPwm4Regs.AQCTLA.bit.CAD = AQ_SET; // Set PWM4A on event A, DOWN count
147     EPwm4Regs.AQCTLB.bit.CBU = AQ_CLEAR; // Clear PWM4A on event A, UP count
148     EPwm4Regs.AQCTLB.bit.CBD = AQ_SET; // Set PWM4A on event B, DOWN count
149
150     // Deadtime configuration
151     EPwm6Regs.DBCTL.bit.OUT_MODE = DB_DISABLE; // Disable deadtime
152
153
154     // ************************************************
155     //              ePWM5 configuration
156     // ************************************************
157
158     // Set period and phase
159     EPwm5Regs.TBPRD = EPWM5_TIMER_TBPRD; // Set timer period
160     EPwm5Regs.TBPHS.half.TBPHS = 0;   // Set phase
161
162     // Setup counter mode and TBCLK prescalers
163     EPwm5Regs.TBCTR = 0;                          // Clear counter
164     EPwm5Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up/down
165     EPwm5Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;   // Clock ratio to SYSCLKOUT
166     EPwm5Regs.TBCTL.bit.CLKDIV = TB_DIV1;      // Clock ratio to SYSCLKOUT
167
168     // Setup phase loading
169     EPwm5Regs.TBCTL.bit.PHSEN = TB_ENABLE;  // Ena. phase loading (slave)
170     EPwm5Regs.TBCTL.bit.PHSDIR = TB_DOWN;   // Count DOWN on sync event
171     EPwm5Regs.TBCTL.bit.PRDLD = TB_SHADOW;  // Enable shadow mode for TBPHS
172     EPwm5Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN ; // Set sync flow-through
173
174     // Setup shadow register for CMPA and CMPB to load on ZERO
175     EPwm5Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW; // Enable shadow mode for CMPA
176     EPwm5Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW; // Enable shadow mode for CMPB
177     EPwm5Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // Load CMPA when TBCTR = 0
178     EPwm5Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // Load CMPB when TBCTR = 0
179
180     // Set actions when TBCTR = CMPA or TBCTR = CMPB
181     EPwm5Regs.AQCTLA.bit.CAU = AQ_CLEAR; // Clear PWM5A on event A, UP count
182     EPwm5Regs.AQCTLA.bit.CAD = AQ_SET; // Set PWM5A on event A, DOWN count
183     EPwm5Regs.AQCTLB.bit.CBU = AQ_CLEAR; // Clear PWM5A on event A, UP count
184     EPwm5Regs.AQCTLB.bit.CBD = AQ_SET; // Set PWM5A on event B, DOWN count
185
186     // Deadtime configuration
```

```
187     EPwm5Regs.DBCTL.bit.OUT_MODE = DB_DISABLE; // Disable deadtime
188
189
190     // ***************************************************
191     //              ePWM6 configuration
192     // ***************************************************
193
194     // Set period and phase
195     EPwm6Regs.TBPRD = EPWM6_TIMER_TBPRD; // Set timer period
196     EPwm6Regs.TBPHS.half.TBPHS = 0;   // Set phase
197
198     // Setup counter mode and TBCLK prescalers
199     EPwm6Regs.TBCTR = 0;                        // Clear counter
200     EPwm6Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up/down
201     EPwm6Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;   // Clock ratio to SYSCLKOUT
202     EPwm6Regs.TBCTL.bit.CLKDIV = TB_DIV1;      // Clock ratio to SYSCLKOUT
203
204     // Setup phase loading
205     EPwm6Regs.TBCTL.bit.PHSEN = TB_ENABLE;  // Ena. phase loading (slave)
206     EPwm6Regs.TBCTL.bit.PHSDIR = TB_DOWN;   // Count DOWN on sync event
207     EPwm6Regs.TBCTL.bit.PRDLD = TB_SHADOW;  // Enable shadow mode for TBPHS
208     EPwm6Regs.TBCTL.bit.SYNCOSEL = TB_SYNC_IN ; // Set sync flow-through
209
210     // Setup shadow register for CMPA and CMPB to load on ZERO
211     EPwm6Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW; // Enable shadow mode for CMPA
212     EPwm6Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW; // Enable shadow mode for CMPB
213     EPwm6Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO; // Load CMPA when TBCTR = 0
214     EPwm6Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO; // Load CMPB when TBCTR = 0
215
216     // Set actions when TBCTR = CMPA or TBCTR = CMPB
217     EPwm6Regs.AQCTLA.bit.CAU = AQ_CLEAR; // Clear PWM6A on event A, UP count
218     EPwm6Regs.AQCTLA.bit.CAD = AQ_SET; // Set PWM6A on event A, DOWN count
219     EPwm6Regs.AQCTLB.bit.CBU = AQ_CLEAR; // Clear PWM6A on event A, UP count
220     EPwm6Regs.AQCTLB.bit.CBD = AQ_SET; // Set PWM6A on event B, DOWN count
221
222     // Deadtime configuration
223     EPwm6Regs.DBCTL.bit.OUT_MODE = DB_DISABLE; // Disable deadtime
224 }
```

Inicialmente é feito o include do arquivo de cabeçalho "*DSP28x_Project.h*", procedimento padrão em todos os arquivos fonte adicionados ao projeto. Depois, são definidas

constantes que serão utilizadas posteriormente para configurar os períodos dos sinais de saída dos módulos ePWM.

Já na função "*Conf_PWM()*", a primeira linha de instrução para a configuração do módulo ePWM1, "**EPwm1Regs.TBPRD = EPWM1_TIMER_TBPRD;**", serve para ajustar o período TBPRD. Este valor serve para definir a frequência dos PWMs gerados pelo módulo. De modo geral, o TBPRD define um valor limite para o contador TBCTR. Este contador, por sua vez, representa a portadora triangular do PWM. Quando o sistema estiver em operação, o TBCTR será periodicamente incrementado a partir de 0 (zero) até o valor TBPRD, e/ou decrementado de TBPRD até 0 (zero). Este comportamento será definido por outra configuração, que será abordada em seguida. Para os módulos do ePWM2 ao ePWM6 esta configuração é feita da mesma forma.

A instrução seguinte, "**EPwm1Regs.TBPHS.half.TBPHS = 0;**", serve para atribuir um valor ao registrador TBPHS do módulo, que define a defasagem da portadora em relação à base de tempo que fornece o sinal de entrada para sincronização. Como o ePWM1 é o módulo mestre, o sinal de sincronização deve partir dele. Portanto esta configuração é irrelevante uma vez que o TBPHS deste módulo não é carregado. No entanto, para os módulos do ePWM2 ao ePWM6 esta configuração é importante e o valor atribuído deve seguir a fórmula "$TBPHS = (angulo/360) \cdot TBPRD$", onde "*angulo*" é o valor de fase desejado do módulo escravo em relação ao ePWM1, em graus.

A próxima linha de configuração, "**EPwm1Regs.TBCTR = 0;**", simplesmente inicializa o contador TBCTR em 0 (zero). Isto é feito para todos os módulos, garantindo um estado inicial nulo.

A linha seguinte, "**EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;**", define a forma da portadora dos PWMs gerados pelo módulo. Neste caso, a configuração é ajustada para que ela seja uma portadora triangular, i.e., o contador TBCTR é incrementado a cada ciclo de *clock* do módulo até atingir o valor TBPRD que foi definido. Depois, ele será decrementado até atingir o valor 0 (zero), reiniciando o ciclo. Esta configuração é a mesma para os módulos do ePWM2 ao ePWM6.

Como o período no qual o TBCTR é incrementado e decrementado é constante, a frequência da portadora triangular do PWM é dada em função TBPRD e do TBCLK, sendo este último o *clock* do módulo. Assim, a frequência do PWM é dada em Hertz pela fórmula "$frequencia = TBCLK/(2 \cdot TBPRD)$". Como será visto mais adiante, o TBCLK é configurado para ser igual ao *clock* do DSP. Como todos os TBPRDs dos módulos são ajustados para 7500, a frequência dos PWMs dos módulos deve ser de $10kHz$.

A linha "**EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;**" a seguir serve para configurar um dos pré-escalonadores do *clock* do módulo. Neste caso, a opção selecionada ajusta o valor dele para 1. A linha "**EPwm1Regs.TBCTL.bit.CLKDIV**

= TB_DIV1;" ajusta o outro pré-escalonador do *clock*, que neste caso também é 1. Desta forma, o *clock* TBCLK do ePWM1 será igual ao do sistema. Estas configurações são repetidas para os módulos do ePWM2 ao ePWM6.

A instrução a seguir, "**EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE;**", desabilita o carregamento do registrador TBPHS, que determina a defasagem do módulo durante os eventos de sincronização. Nos módulos do ePWM2 ao ePWM6 este carregamento deve ser habilitado, trocando-se a opção "TB_DISABLE" por "TB_ENABLE".

A seguir, deve ser acrescentada uma linha de configuração somente para os módulos do ePWM2 ao ePWM6. No caso do ePWM2, esta linha é "**EPwm2Regs.TBCTL.bit. PHSDIR = TB_DOWN;**". Ela serve para dizer se o contador TBCTR deve continuar a ser incrementado ou decrementado após ume vento de sincronização. Neste caso ele é configurado para ser decrementando (opção "TB_DOWN"). Isto quer dizer que, quando ocorrer ume vento de sincronização, o TBCTR do ePWM2 será igualado com o TBCTR do ePWM1, e a partir daquele ponto ele irá continuar seu ciclo sendo decrementado. Esta configuração é repetida para todos os outros módulos escravos, mas não existe na configuração do ePWM1.

A instrução "**EPwm1Regs.TBCTL.bit.PRDLD = TB_SHADOW;**" que segue serve para configurar o modo *shadow* do registrador TBPRD, i.e., como deve ser gerenciada a alteração do valor do TBPRD durante a operação do sistema. Segundo a configuração feita, caso seja dada alguma instrução para alterar o valor de TBPRD, o novo valor irá primeiro para um registrador *shadow*. Quando o TBCTR for 0 (zero), o valor do registrador *shadow* será copiado para TBPRD. A outra configuração possível permite a modificação instantânea do TBPRD. Esta configuração é a mesma para o ePWM1 e para todos os outros.

A linha de instrução seguinte, "**EPwm1Regs.TBCTL.bit.SYNCOSEL = TB_ CTR_ZERO;**", serve para configurar qual o sinal que será usado para a sincronização do módulo. Como o ePWM1 é o módulo mestre, a opção "TB_CTR_ZERO" é escolhida, tornando-o a referência de sincronização. Nos módulos do ePWM2 ao ePWM6 a opção de configuração deve ser "TB_SYNC_IN", de modo que todos terão o ePWM1 como referência.

Nas duas linhas que seguem, que contêm a instrução "**EPwm1Regs.CMPCTL. bit.SHDWAMODE = CC_SHADOW;**" e a instrução "**EPwm1Regs.CMPCTL. bit.SHDWBMODE = CC_SHADOW;**", é feita a configuração o modo *shadow* dos registradores CMPA e CMPB respectivamente, de modo similar a como é feito para o registrador TBPRD, conforme foi explicado. Neste caso o modo *shadow* é ativado. Estas configurações são também iguais para os módulos do ePWM2 ao ePWM6.

No caso do modo *shadow* dos registradores CMPA e CMPB, o evento de atuali-

zação deles é configurável, o que é feito nas duas instruções seguintes, a "**EPwm1Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;**" e a "**EPwm1Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;**". Neste caso, o valor dos registradores *shadow* será copiado quando o contador TBCTR for 0 (zero). Da mesma forma, estas configurações sao repetidas para os módulos do ePWM2 ao ePWM6.

Os registradores CMPA e CMPB de um determinado módulo ePWM são continuamente comparados com o contador TBCTR do mesmo módulo. Assim, quando $CMPA = TBCTR$ ou $CMPB = TBCTR$ são gerados eventos. As ações decorrentes destes eventos irão interferir nos dois sinais de saída de cada módulo ePWM, de modo a se obter os sinais modulados da forma como se deseja utilizá-los.

As quatro linhas de código que seguem servem justamente a este propósito, de configurar qual a ação que será tomada quando os eventos de comparação dos registradores CMPA e CMPB com o contador TBCTR ocorrerem.

A instrução "**EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR;**" configura a saída EPWM1A para ser limpa, i.e., "$EPWM1A = 0$'', quando o evento "$CMPA = TBCTR$" ocorrer e TBCTR estiver sendo incrementado. Já a instrução "**EPwm1Regs.AQCTLA.bit.CAD = AQ_SET**;" configura a saída EPWM1A para ser setada, i.e., "$EPWM1A = 1$", quando o evento "$CMPA = TBCTR$" ocorrer e TBCTR estiver sendo decrementado. As duas instruções seguintes configuram de forma análoga as ações que são tomadas para a saída EPWM1B quando o evento "$CMPB = TBCTR$" ocorre. Assim, as saídas correspondem a sinais PWM modulado na forma convencional. Estas configurações são repetidas para os módulos do ePWM2 ao ePWM6.

A configuração do ePWM1 é concluída com a instrução "**EPwm1Regs.DBCTL.bit.OUT_MODE = DB_DISABLE;**". Ela serve para configurar o tempo morto que será aplicado aos sinais EPWM1A e EPWM1B de saída. Ele pode ser aplicado na somente na descida dos sinais (opção "*DBA_ENABLE*"), somente na subida (opção "*DBB_ENABLE*"), na subida e na descida (opção DB_FULL_ENABLE) ou não ser aplicado (opção "*DB_DISENABLE*"). Esta última opção é a utilizada na configuração. Como as saídas de um mesmo módulo ePWM são usadas de forma independente na bancada de experimentos, i.e., uma não é necessariamente o complementar da outra, o ajuste do tempo morto é feito externamente, não sendo necessário fazê-lo no projeto. Esta configuração é repetida para os módulos do ePWM2 ao ePWM6, o que também conclui a configuração deles.

4 O Arquivo *DSP_SysCtrl.c* e a Programação da Estratégia PWM e do Sistema de Controle

No arquivo "*DSP_SysCtrl.c*" foi incluído o código que implementa a estratégia PWM e o sistema de controle de um sistema de conversão. Desta forma, antes de ser mostrado o código é necessário que seja mostrado um exemplo de sistema de conversão que sirva de referência, apresentando ainda a sua estratégia PWM e o seu sistema de controle. Isto é feito nas seção 4.1. Já na seção 4.2 o código do arquivo é mostrado e descrito.

4.1 O sistema de conversão

O sistema de conversão escolhido para a aplicação da estratégia PWM e do sistema de controle foi um conversor AC/DC (retificador) trifásico. Este é um sistema simples e permite demonstrar os procedimentos básicos para programar a função "*sys_control()*". O diagrama do sistema é mostrado na Figura 26. A seguir são detalhadas a estratégia PWM e o sistema de controle utilizados.

4.1.1 A estratégia PWM

A estratégia PWM utilizada no sistema de conversão foi a escalar, que pode ser resumida nas equações de 4.1 a 4.5. O desenvolvimento destas equações é omitido porque não é objetivo deste relatório detalhá-lo, mas apenas mostrar como utilizar o resultado para fazer a implementação da estratégia por meio de um código em C, que será mostrado posteriormente. Então, considerando que o sobrescrito "$*$" é utilizado para definir uma variável de referência relacionada a uma variável real do sistema, e.g., "v_x^*" é a referência de "v_x", e que $j = \{1, 2, 3\}$, tem-se:

$$v^*_{gtj0t} = v^*_{gj} + v^*_{0g0t}, \tag{4.1}$$

$$v^*_{0g0t} = \frac{1}{3}\sum_{j=1}^{3} v^*_{gtj0t}, \tag{4.2}$$

$$v^*_{0g0t} = \mu_{0g0t} \cdot v^*_{0g0tMAX} + (1 - \mu_{0g0t}) \cdot v^*_{0g0tMIN}, \tag{4.3}$$

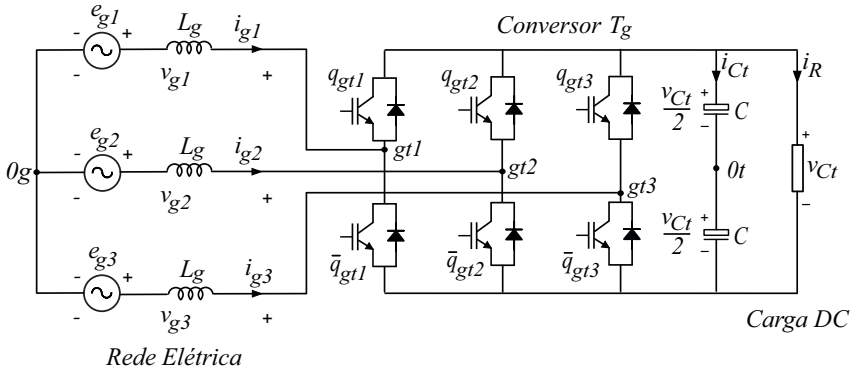

Figura 25 – Sistema de conversão de um retificador trifásico.

$$v^*_{0g0tMAX} = \frac{v^*_{Cm}}{2} - max\{v^*_{g1}, v^*_{g2}, v^*_{g3}\}, \tag{4.4}$$

$$v^*_{0g0tMIN} = -\frac{v^*_{Cm}}{2} - min\{v^*_{g1}, v^*_{g2}, v^*_{g3}\}, \tag{4.5}$$

onde v^*_{gtj0t} são as referências das tensões de polo do conversor, v^*_{0g0t} é a referência da tensão de neutro do ponto $0g$ ao ponto $0t$ do sistema, e $v^*_{0g0tMAX}$ e $v^*_{0g0tMIN}$ são respectivamente os valores máximo e mínimo que a referência v^*_{0g0t} pode assumir, e μ_{0g0t} é o fator de normalização da escolha de v^*_{0g0t}, de modo que $0 \leqslant \mu_{0g0t} \leqslant 1$. Mais detalhes sobre a estratégia PWM escalar podem ser obtidos consultando-se (6).

4.1.2 O sistema de controle

O objetivo principal do sistema de controle é manter a tensão do barramento DC constante, de modo a fornecer uma tensão estável à carga acoplada. Além disso, deseja-se manter o fator de potência da rede elétrica o mais próximo possível de 1, reduzindo a demanda de reativos.

Deste modo, para controlar a tensão no barramento DC é utilizado um esquema de controle em cascata. Na malha interna é feito o controle de corrente da rede, enquanto na malha externa é feito o controle de tensão. Além disso, para manter o fator de potência unitário é utilizado um controlador de ângulo por PLL que permite sincronizar o ângulo das referências de corrente com o das tensões da rede, a partir da medida de uma delas.

Assim, o diagrama de blocos do sistema de controle utilizado é o mostrado na Figura 26. O modelo dos controladores discretos utilizados será mostrado no código da função *sys_control()*, na seção 4.2.

4.2 O arquivo *DSP_SysCtrl.c* e a função *sys_control()*

O código que foi inserido no arquivo *DSP_SysCtrl.c* é mostrado a seguir:

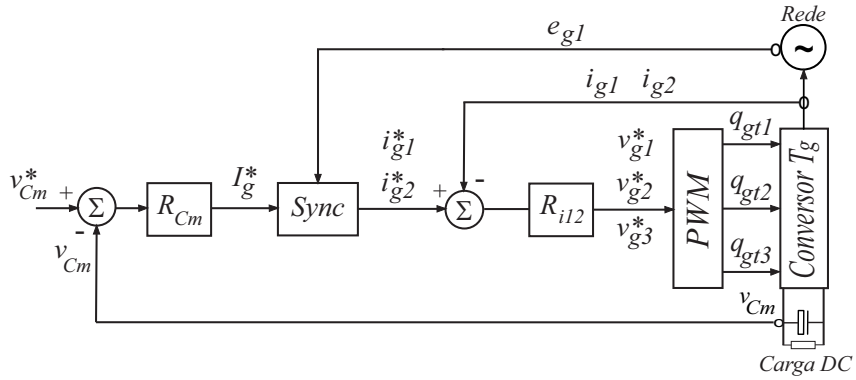

Figura 26 – Sistema de conversão de um retificador trifásico.

```
#include "DSP28x_Project.h"
#include "math.h"

#define AM 512 // Tamanho do Buffer Geral
#define pi 3.14159265

float buffer1[AM]={0.};
float buffer2[AM]={0.};

// Variaveis do controlador PLL:
float A1 = -8.796*5., A2 = -39.48*5.*5., A3 = 1.0, A4 = 0.0;
float B1 = 1., B2= 0.0, C1 = 0.0, C2 = 39.48*5.*5., D1 = 0.0;
float Pd = 0.0, teta_ig = 0.0;
float wff = 376.9911, X1t = 0.0, X2t = 0.0, dX1t = 0.0, dX2t = 0.0;
float X1tf = 0.0, X2tf = 0.0, Pdf = 0.0, e_pd = 0.0;
float X1a = 0., X2a = 0.0, dX1a = 0.0, dX2a = 0.0;
float X1af = 0.0, X2af = 0.0, kix = 0.0, kpx = 0.0, wf = 0.0;

float vaf_off=0., vbf_off=0., vcf_off=0., vdf_off=0.;
float vef_off=0., vff_off=0., vgf_off=0., vhf_off=0.;
float vif_off=0., vjf_off=0., vlf_off=0., vmf_off=0.;

float h = 0.0001; // Periodo discreto (p/ ePWM a 10kHz)

// Variaveis eletricas do sistema:
float wg = 0.; // Frequencia da rede
float eg1=0.; // Tensao da fase 1 da rede
float vCm=0.; // Tensao do barramento DC
float vCm_f=0., avCm_f=0., avCm=0.;
float ig1=0., ig2=0., ig3=0.; // Correntes da rede
```

```
31
32 // Variavei do cont. de tensao
33 float vCm_ref=0.; // Tensao de ref. do barramento DC
34 float erro_vCm=0., x_RCm=0.;
35 float kp_RCm=0., ki_RCm=0.; // Ganhos do cont. RCm de tensao
36
37 // Variaveis dos conts. de corrente:
38 float Ig=0.;
39 float ig1_ref=0., ig2_ref=0., ig3_ref=0.; // Refs. de corrente
40 float erro_ig1=0., erro_ig2=0.;
41 float cos_wgh=0., sin_wgh=0., cte_1_wg=0.;
42 float xa1_ant=0., xb1_ant=0., xa1=0., xb1=0.;
43 float xa2_ant=0., xb2_ant=0., xa2=0., xb2=0.;
44 float kp_Ri12=0., ki_Ri12=0.; // Ganho dos conts. Ri12 de corrente
45
46 // Variaveis de ref. do PWM:
47 float vg1_ref=0., vg2_ref=0., vg3_ref=0.;
48 float vgt10t_ref=0., vgt20t_ref=0., vgt30t_ref=0.;
49 float mi_0g0t=.5, v0g0t_ref=0.;
50 float v0g0t_ref_MAX=0., v0g0t_ref_MIN=0.;
51 float vg_ref_MAX=0., vg_ref_MIN=0.;
52
53 #pragma CODE_SECTION(sys_control, "IsrCode")
54
55 void sys_control(int32 i, float ad0, float ad1, float ad2,
56                  float ad3, float ad4, float ad5, float ad6,
57                  float ad7,float ad8, float ad9, float ad10,
58                  float ad11, float ad12, float ad14){
59     eg1 = ad0;
60     vCm = ad10;
61     ig1 = ad1;
62     ig2 = ad2;
63     ig3 = -(ig1+ig2);
64
65     // ----- Filtragem capacitores -----:
66     vCm_f = 0.9391*avCm_f + 0.03046*vCm + 0.03046*avCm;
67     avCm_f = vCm_f;
68     avCm = vCm;
69     // ----------
70
71     // ----- PLL da Potencia -----:
72     kix = 2.5; //4.5; //3.5; //2.5; //1.0; // Valores
```

```
73      kpx = 0.55; //0.55;                 // recomendados
74
75      // Detector de fase:
76      Pd = eg1*cos(teta_ig);
77
78          // Filtro PB:
79      X1a = X1af;
80      X2a = X2af;
81
82      dX1a = A1*X1a + A2*X2a + B1*Pd;
83      dX2a = A3*X1a + A4*X2a + B2*Pd;
84
85      X1af = X1a + dX1a*h;
86      X2af = X2a + dX2a*h;
87
88      Pdf = C1*X1af + C2*X2af + D1*Pd; // Saida do Filtro
89          // ---
90      // ---
91
92      e_pd = -(0.-Pdf); // Erro de fase
93
94      // Controlador PI + Integrador:
95      X1t = X1tf;
96      X1tf = X1t + kix*e_pd*h;
97      dX1t = X1tf + kpx*e_pd;
98      wf = wff + dX1t;
99      // ---
100
101     teta_ig = teta_ig + wf*h; // Saida do PLL
102     if(teta_ig >= 2.*pi)
103         teta_ig = teta_ig - 2.*pi;
104     if(teta_ig < 0.)
105         teta_ig = teta_ig + 2.*pi;
106     // ----- (FIM do PLL) -----
107
108     // ----- PI discreto de tensao -----:
109     vCm_ref = 100.;
110
111     kp_RCm = 0.08; // Ganho proporcional
112     ki_RCm = 0.1; // Ganho integral
113
114     x_RCm = x_RCm + h*erro_vCm;
```

```
115     erro_vCm = vCm_ref - vCm_f;
116     Ig = kp_RCm*erro_vCm + ki_RCm*x_RCm;
117
118     if(Ig > 10.)
119         Ig = 10.;
120     else if(Ig < 0.)
121         Ig = 0.;
122     // ----------
123
124     // ----- PIs discretos de corrente -----:
125     ig1_ref = Ig*sin(teta_ig);
126     ig2_ref = Ig*sin(teta_ig-2.094395102393195);
127     ig3_ref = Ig*sin(teta_ig+2.094395102393195);
128
129     kp_Ri12 = 15.; // Ganho proporcional
130     ki_Ri12 = 500.; // Ganho integral
131
132     wg = 376.9911184307752; // 2.*pi*60.;
133     cos_wgh = cos(wg*h);
134     sin_wgh = sin(wg*h);
135     cte_1_wg = 1./wg;
136
137     // ----- PI1 -----
138     xa1_ant = xa1;
139     xb1_ant = xb1;
140     xa1 = cos_wgh*xa1_ant
141           + cte_1_wg*sin_wgh*(xb1_ant + 2.*ki_Ri12*erro_ig1);
142     xb1 = -wg*sin_wgh*xa1_ant + cos_wgh*xb1_ant
143           + 2.*ki_Ri12*(cos_wgh - 1.)*erro_ig1;
144     erro_ig1 = -(ig1_ref - ig1);
145     vg1_ref = xa1 + kp_Ri12*erro_ig1;
146     // -----
147
148     // ----- PI2 -----
149     xa2_ant = xa2;
150     xb2_ant = xb2;
151     xa2 = cos_wgh*xa2_ant
152           + cte_1_wg*sin_wgh*(xb2_ant + 2.*ki_Ri12*erro_ig2);
153     xb2 = -wg*sin_wgh*xa2_ant + cos_wgh*xb2_ant
154           + 2.*ki_Ri12*(cos_wgh - 1.)*erro_ig2;
155     erro_ig2 = -(ig2_ref - ig2);
156     vg2_ref = xa2 + kp_Ri12*erro_ig2;
```

```
157     // -----
158
159     vg3_ref = -(vg1_ref+vg2_ref);
160     // -----
161
162     // Determinacao da tensao de referencia v0g0t_ref:
163     mi_0g0t = 0.5;
164
165     if(vg1_ref >= vg2_ref){
166         vg_ref_MAX = vg1_ref;
167         vg_ref_MIN = vg2_ref;
168     }
169     else{
170         vg_ref_MAX = vg2_ref;
171         vg_ref_MIN = vg1_ref;
172     }
173
174     if(vg3_ref > vg_ref_MAX)
175         vg_ref_MAX = vg3_ref;
176     else if(vg3_ref < vg_ref_MIN)
177         vg_ref_MIN = vg3_ref;
178
179     v0g0t_ref_MAX = 0.5*vCm_ref - vg_ref_MAX;
180     v0g0t_ref_MIN = -0.5*vCm_ref - vg_ref_MIN;
181     v0g0t_ref = mi_0g0t*v0g0t_ref_MAX + (1.-mi_0g0t)*v0g0t_ref_MIN;
182     // -----
183
184     // Tesoes de polo:
185     vgt10t_ref = vg1_ref + v0g0t_ref;
186     vgt20t_ref = vg2_ref + v0g0t_ref;
187     vgt30t_ref = vg3_ref + v0g0t_ref;
188     // -----
189
190     vaf_off = 7500.*(0.5 + vgt10t_ref/vCm_ref); // braco A1
191     vbf_off = 7500.*(0.5 + vgt20t_ref/vCm_ref); // braco A2
192     vcf_off = 7500.*(0.5 + vgt30t_ref/vCm_ref); // braco A3
193     if(vaf_off > 7500.)
194         vaf_off = 7490.;
195     else if(vaf_off < 10.)
196         vaf_off = 10.;
197     if(vbf_off > 7500.)
198         vbf_off = 7490.;
```

```
199     else if(vbf_off < 10.)
200         vbf_off = 10.;
201     if(vcf_off > 7500.)
202         vcf_off = 7490.;
203     else if(vcf_off < 10.)
204         vcf_off = 10.;
205
206     /*
207     vdf_off = 7500.*(0.5 + vsb20_ref/vCb_ref); // braco B1
208     vef_off = 7500.*(0.5 + vsb40_ref/vCb_ref); // braco B2
209     vff_off = 7500.*(0.5 + vsb60_ref/vCb_ref); // braco B3
210     if(vdf_off>7500.)
211         vdf_off=7490.;
212     else if(vdf_off<10.)
213         vdf_off=10.;
214     if(vef_off>7500.)
215         vef_off=7490.;
216     else if(vef_off<10.)
217         vef_off=10.;
218     if(vff_off>7500.)
219         vff_off=7490.;
220     else if(vff_off<10.)
221         vff_off=10.;
222
223     vgf_off = 7500.*(0.5 + vgc10c_ref/vCc_ref); // braco C1
224     vhf_off = 7500.*(0.5 + vgc20c_ref/vCc_ref); // braco C2
225     vif_off = 7500.*(0.5 + vgc30c_ref/vCc_ref); // braco C3
226     if(vgf_off>7500.)
227         vgf_off=7490.;
228     else if(vgf_off<10.)
229         vgf_off=10.;
230     if(vhf_off>7500.)
231         vhf_off=7490.;
232     else if(vhf_off<10.)
233         vhf_off=10.;
234     if(vif_off>7500.)
235         vif_off=7490.;
236     else if(vif_off<10.)
237         vif_off=10.;
238
239     vjf_off = 7500.*(0.5 + vgd10d_ref/vCd_ref); // braco D1
240     vlf_off = 7500.*(0.5 + vgd20d_ref/vCd_ref); // braco D2
```

```
241     vmf_off = 7500.*(0.5 + vgd30d_ref/vCd_ref); // braco D3
242     if(vjf_off>7500.)
243         vjf_off=7490.;
244     else if(vjf_off<10.)
245         vjf_off=10.;
246     if(vlf_off>7500.)
247         vlf_off=7490.;
248     else if(vlf_off<10.)
249         vlf_off=10.;
250     if(vmf_off>7500.)
251         vmf_off=7490.;
252     else if(vmf_off<10.)
253         vmf_off=10.;
254     */
255
256     // Referencias para o conversor 1:
257     EPwm3Regs.CMPA.half.CMPA = vaf_off; // A1
258     EPwm3Regs.CMPB = vbf_off;            // A2
259     EPwm4Regs.CMPA.half.CMPA = vcf_off;  // A3
260
261     /*
262     // Referencias para o conversor 2:
263     EPwm1Regs.CMPA.half.CMPA = vdf_off; // B1
264     EPwm2Regs.CMPA.half.CMPA = vef_off; // B2
265     EPwm2Regs.CMPB = vff_off;            // B3
266
267     // Referencias para o conversor 3:
268     EPwm6Regs.CMPA.half.CMPA = vgf_off; // C1
269     EPwm5Regs.CMPA.half.CMPA = vhf_off; // C2
270     EPwm5Regs.CMPB = vif_off;            // C3
271
272     // Referencias para o conversor 4:
273     EPwm4Regs.CMPB = vjf_off; // D1
274     EPwm1Regs.CMPB = vlf_off; // D2
275     EPwm6Regs.CMPB = vmf_off; // D3
276     */
277
278     buffer1[i] = vCm_f;
279     buffer2[i] = ig1;
280 }
```

Nas primeiras linhas de código do arquivo são feitas inclusões de arquivos de cabeçalho necessários e definidas algumas constantes, como a constante "AM", que é usada para definir o tamanho dos vetores nos quais podem ser salvas algumas variáveis para posterior visualização, e a constante "pi", que representa o número π.

Depois são feitas as declarações desses vetores e de todas as variáveis utilizadas na função "*sys_control()*". Estas declarações são feitas de forma global, de modo a forçar estas variáveis a reter seus valores ao final da execução da função "*sys_control()*", além de deixá-las disponíveis num escopo global.

A diretiva "**#pragma CODE_SECTION(sys_control, "IsrCode")**" deve ser chamada antes da definição da função *sys_control()*. Esta função, por usa vez, ao ser chamada deve receber o índice "i" para indexar os vetores declarados quando variáveis forem salvas neles, e também as 15 leituras do módulo ADC que foram salvas.

No caso do exemplo de sistema de conversão proposto, é necessário serem feitas medidas apenas da tensão da fase 1 da rede (e_{g1}), da tensão do barramento DC (v_{Cm}) e das correntes das fases 1 e 2 da rede (i_{g1} e i_{g2}). Dessa forma, nas primeiras 4 linhas de código da função estes valores medidos são obtidos dentre os resultados que são passados para a função, de acordo com o canal que foi utilizado.

Em seguida é descrito o código do controlador de ângulo por PLL. Ele recebe como entrada a tensão "*eg1*" medida e fornece como saída o ângulo "*teta_ig*", que é usado posteriormente para gerar as referências de corrente sincronizadas com as tensões da rede. As referências precisam necessariamente serem geradas com uma função seno, conforme pode ser visto mais adiante no código. Isto deve ocorrer porque logo no início do código do PLL existe um detector de fase que se baseia na identidade trigonométrica "$sin(a)\cdot cos(b) = \frac{1}{2}sin(a-b)+\frac{1}{2}sin(a+b) \approx \frac{1}{2}(a-b)+\frac{1}{2}sin(a+b)$", na qual a aproximação é válida uma vez que a diferença "$a-b$" seja pequena. Portanto, na linha de código "**Pd = eg1*cos(theta);**" supõe-se que "*eg1*" é gerado por uma função seno, de modo que "*ig1_ref*" também deve ser gerado por uma função seno para que o cálculo da fase esteja de acordo com esta convenção. Assim, depois dessa linha segue um filtro passa-baixas discreto de segunda ordem que filtra a variável "*Pd*", fornecendo assim na saída uma aproximação da diferença entre as fases das variáveis "*eg1*" e "*ig1_ref*". O restante do código é a versão discreta do modelo tradicional de um PLL.

Em seguida, é descrito o controlador PI de tensão. Este modelo discreto pode ser obtido a partir da discretização, pelo método de Euler, do modelo contínuo no espaço de estados do controlador PI convencional. O controlador recebe como sinal de entrada o erro de tensão "*erro_vCm*" e fornece como saída a amplitude das correntes de referência, "*Ig*".

Depois, esta amplitude é aplicada a um conjunto trifásico de sinais senoidais gerados pela função "*sin()*" segundo o ângulo "*teta_ig*" atualizado, fornecido pelo PLL, o que resulta nas três correntes de referência da rede, "*ig1_ref*", "*ig2_ref*" e "*ig3_ref*". As referências "*ig1_ref*" e "*ig2_ref*" são aplicadas a 2 controladores PI ressonantes discretos, cujo modelo pode ser consultado com detalhes em (7). Estes controladores recebem como entrada os sinais de erro de corrente e fornecem como saída as tensões de referência "*vg1_ref*", "*vg2_ref*" e "*vg3_ref*" são também um conjunto trifásico.

Estas tensões são então submetidas à estratégia PWM, cujas equações foram mostradas na seção 4.1 e são reproduzidas nas linhas de código que seguem. Ao final, são geradas as tensões de polo de referência, representadas pelas variáveis "*vgt10t_ref*", "*vgt20t_ref*" e "*vgt30t_ref*".

Na sequência, estas tensões de polo de referência são normalizadas segundo a tensão de referência do barramento DC "*vCm_ref*" e condicionadas para poderem ser comparadas com as portadoras triangulares geradas pelos módulos ePWM. Estas referências condicionadas são respectivamente as variáveis "*vaf_off*", "*vbf_off*" e "*vcf_off*". O trecho de código comentado em seguida faria a mesma coisa caso fossem utilizadas mais tensões de polo de referência de outros conversores.

Estas referências são atribuídas aos registradores CMPA e CMPB dos devidos módulos ePWM nas três linhas de código seguintes, de acordo com as saídas ePWM que estiverem associadas a cada braço dos conversores. Esta relação depende das montagens físicas realizadas entre a placa do DSP e o sistema de aquisição ligado à parte de potência da bancada. Assim, cada saída é utilizada para chavear um braço, i.e., um par de chaves. A chave de cima de um braço recebe o sinal conforme ele sai do ePWM, já a chave de baixo recebe o complementar desse sinal, gerado em um circuito externo ao DSP. As linhas de código que seguem comentadas teriam a mesma função e seriam úteis caso mais braços e/ou conversores fossem utilizados.

No final, as variáveis desejadas devem ser salvas nos vetores declarados, caso desejese visualizá-las depois durante a execução do código no DSP.

Deve-se destacar que todos os modelos discretos apresentados no código consideram que o passo de cálculo discreto, representado pela variável "h", é $0,0001$. Isto porque a rotina descrita na função *sys_control()* é chamada uma vez a cada período do PWM, que é a frequência em que ocorre a interrupção do módulo ADC, segundo as configurações feitas e descritas até então. Como a frequência dos módulos ePWM foi ajustada para $10kHz$, então o período discreto dos modelos discretizados é o inverso disto.

Esta seção concluí o processo de criação do código base para o DSP28335 utilizando o CCS v3.3. Para depurar o projeto basta compilá-lo sem que o DSP esteja necessariamente conectado. Os arquivos objeto devem ser gerados e o relatório da compilação não

deve apontar nenhum erro, o que deve acontecer caso os passos descritos no relatório sejam reproduzidos. Após a primeira compilação deve ser possível visualizar na janela do CCS, na parte que mostra a estrutura do projeto, todos os arquivos de cabeçalho dentro da seção "*Include*".

5 Atividades Adicionais do Estágio

As outras atividades realizadas durante o período de estágio foram em essência voltadas à publicação de um artigo científico intitulado "*AC/DC Converters with Open-End Grid for AC Machine Conversion Systems*", no ECCE 2014. Elas consistiram basicamente no estudo e simulação de 5 sistemas de conversão e na montagem e obtenção de resultados experimentais de 3 dentre eles. Além disso, a redação e submissão do artigo também ocorreu durante o período de estágio.

Os sistemas de conversão podem ser observados nas figuras de 27 a 31. Todos eles foram simulados no *software* PSIM para obtenção de resultados de simulação, mas apenas os circuitos das figuras 27, 29 e 31 puderam ser montados, devido a limitações temporárias do Laboratório. Sistemas similares aos apresentados podem ser encontrados em (8) e (9).

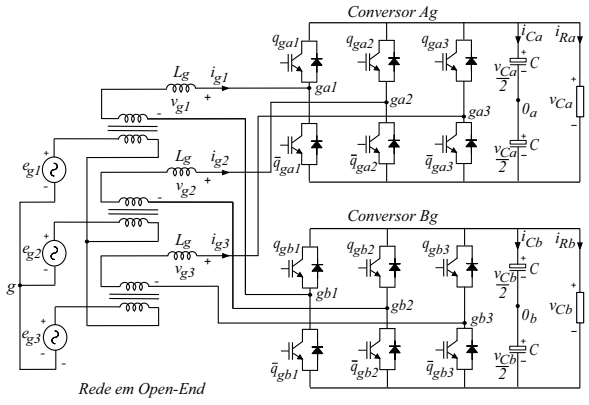

Figura 27 – Circuito A1, simulado e montado para obtenção de resultados experimentais para o artigo.

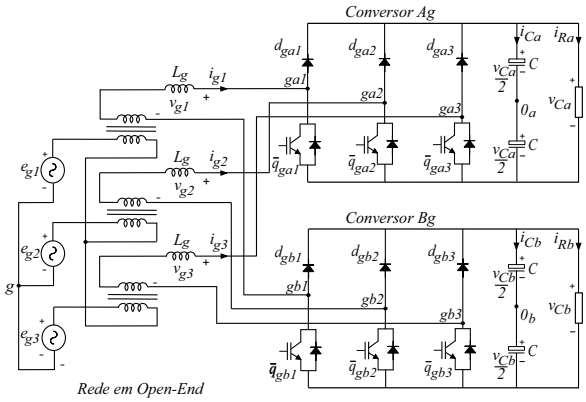

Figura 28 – Circuito A2, apenas simulado.

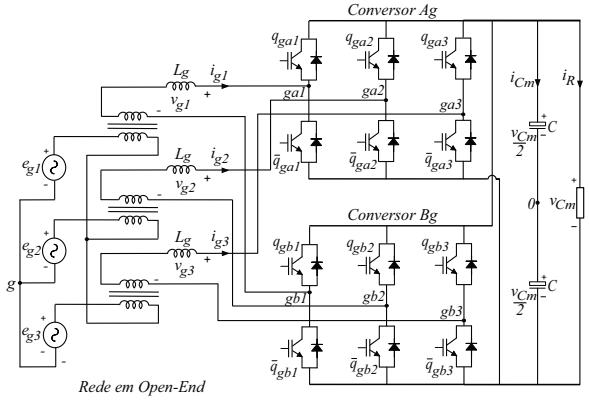

Figura 29 – Circuito B1, simulado e montado para obtenção de resultados experimentais para o artigo.

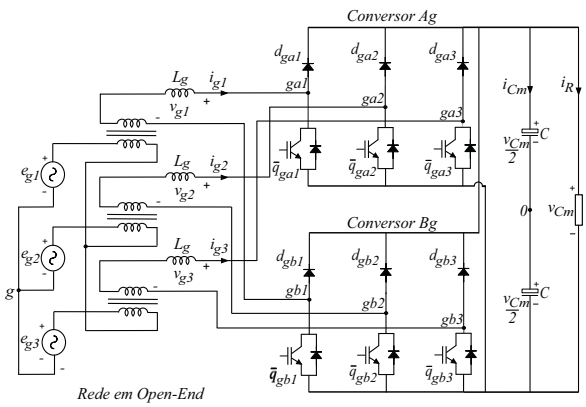

Figura 30 – Circuito B2, apenas simulado.

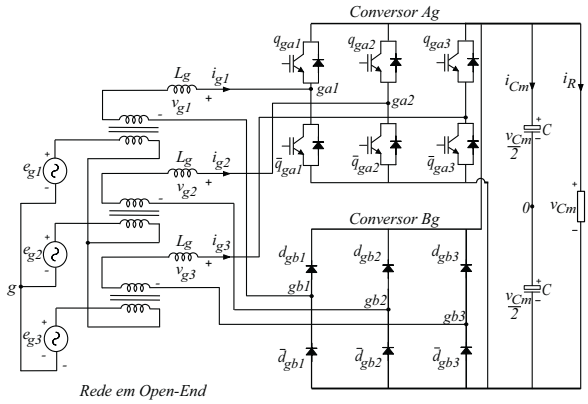

Figura 31 – Circuito B3, simulado e montado para obtenção de resultados experimentais para o artigo.

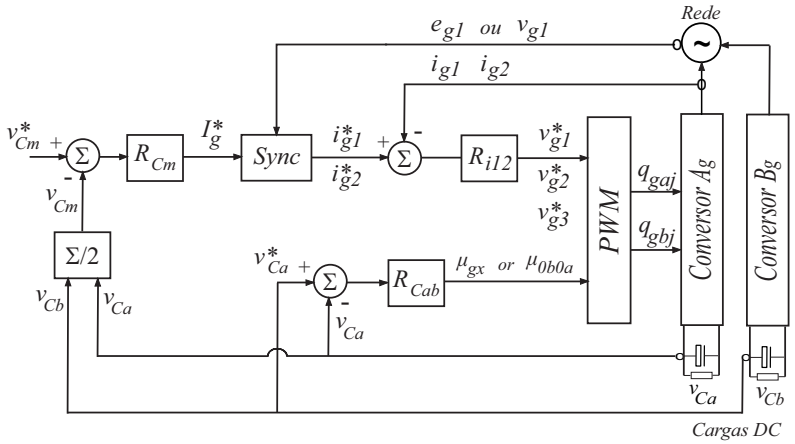

Figura 32 – Diagrama de blocos do sistema de controle dos circuitos A1 e A2, com $j = 1, 2, 3$.

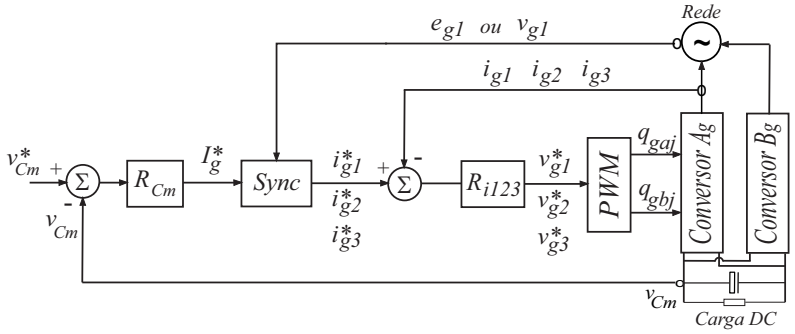

Figura 33 – Diagrama de blocos do sistema de controle dos circuitos B1, B2 e B3, com $j = 1, 2, 3$.

A estratégia PWM empregada em todos os casos foi a *level shifted PWM*, de modo a otimizar o aproveitamento dos níveis gerados pelos conversores. Os sistemas de controle empregados podem ser vistos nos diagramas de blocos mostrados na Figura 32 para os circuitos A1 e A2, e na Figura 33 para os circuitos B1, B2 e B3.

6 Considerações Finais

Com a realização destes estágio foi possível aprender muito sobre o DSP e adquirir mais intimidade com o dispositivo, que é muito importante para as atividades práticas do Laboratório. Além disso, foi possível iniciar e progredir consideravelmente no desenvolvimento do guia de criação de projetos para o DSP, considerando as aplicações do LEIAM. Este guia foi apresentado na forma deste relatório, e para sua conclusão definitiva restou apenas serem realizados testes experimentais, o que não foi possível durante o período de estágio pela constante demanda de utilização da bancada por outros integrantes do grupo de pesquisa.

Por este mesmo motivo não foi possível concretizar outras atividades inicialmente previstas para o estágio, como a realização de algumas melhorias na bancada de experimentos de uso geral. No entanto, a conclusão definitiva do material e das melhorias previstas serão efetivados futuramente, uma vez que o estagiário se tornará membro efetivo do grupo de pesquisa como mestrando.

Dentre as atividades do estágio estiveram também a simulação de 5 sistemas de conversão e a montagem de 3 dentre estes, seguida da redação e submissão de um artigo científico. Estas atividades foram satisfatoriamente concluídas, devendo-se apenas fazer a ressalva de que 2 dos sistemas de conversão não foram montados devido às limitações temporárias do Laboratório, que serão eliminadas uma vez que se realizem as melhorias originalmente propostas para o estágio.

Referências

1 TEXAS INSTRUMENTS. *TMS320F28335, TMS320F28334, TMS320F28332, TMS320F28235, TMS320F28234, TMS320F28232 Digital Signal Controllers (DSCs) Data Manual (SPRS439M)*. [S.l.], 2012. Citado na página 17.

2 TEXAS INSTRUMENTS. *TMS320x2833x Analog-to-Digital Converter (ADC) Module Reference Guide (SPRU812A)*. [S.l.], 2007. Citado na página 17.

3 TEXAS INSTRUMENTS. *TMS320x2833x, 2823x Enhanced Pulse Width Modulator (ePWM) Module Reference Guide (SPRUG04A)*. [S.l.], 2009. Citado na página 17.

4 TMS320X2833X, 2823x System Control and Interrupts Reference Guide (SPRUFB0D). [S.l.]. Citado na página 17.

5 TMS320C28X CPU and Instruction Set Reference Guide (SPRU430E). [S.l.]. Citado na página 17.

6 JACOBINA, C. B. et al. Digital scalar pulse-width modulation: a simple approach to introduce nonsinusoidal modulating waveforms. *Power Electronics*, IEEE Transactions, v. 16, n. 3, p. 351 – 359, May 2001. Citado na página 40.

7 JACOBINA, C. B. et al. Current control of unbalanced electrical systems. *Industrial Electronics*, IEEE Transactions, v. 48, n. 3, p. 517 – 524, June 2001. Citado na página 49.

8 ZHOU, Y.; NIAN, H. Investigation on open winding pmsg system with the integration of full controlled and uncontrolled converter. In: *Energy Conversion Congress and Exposition (ECCE), 2013 IEEE*. [S.l.: s.n.], 2013. p. 3912 – 3917. Citado na página 51.

9 WANG, Y. et al. Open-winding power conversion systems fed by half-controlled converters. In: *Power Electronics, IEEE Transactions on*. [S.l.: s.n.], 2013. v. 28, n. 5, p. 2427 – 2436. Citado na página 51.