Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Departamento de Engenharia Elétrica

Ravi Barreto Doria Figueiredo

ESTUDO DA PLACA BLACKFIN ADSP-BF537 PARA USO NO MONITOR MULTIPARAMÉTRICO LIFETOUCH.10

Campina Grande, Paraíba

Fevereiro, 2015

Ravi Barreto Doria Figueiredo

ESTUDO DA PLACA BLACKFIN ADSP-BF537 PARA USO NO MONITOR MULTIPARAMÉTRICO LIFETOUCH.10

Relatório de Estágio Supervisionado submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Processamento Digital de Sinais

Orientador: Edmar Candeia Gurjão

Campina Grande, Paraíba

Fevereiro, 2015

Ravi Barreto Doria Figueiredo

ESTUDO DA PLACA BLACKFIN ADSP-BF537 PARA USO NO MONITOR MULTIPARAMÉTRICO LIFETOUCH.10

Relatório de Estágio Supervisionado submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Edmar Candeia Gurjão
Orientador

Luciana Ribeiro Veloso
Avaliador

Campina Grande, Paraíba
Fevereiro, 2015

Aos meus colegas de projeto, a quem desejo toda a felicidade possivel.

Agradecimentos

Uma fase de minha vida se inicia com o termino deste trabalho e gostaria de agradecer a algumas pessoas que me ajudaram nessa jornada.

Aos meu companheiro de projeto Lucas Barbosa, Livia Carolline e a Denis Dantas.

Aos meus pais Renan Tavares e Marizi Barreto por terem me ensinado a caminhar e a viver!

A minha Amada Lucianna Marques por ser tudo aquilo que me complementa

Aos professores José Ewerton, Alexandre Jean Renes, Edmar Gurjão, Rômulo Maranhão e Professor Rubem Alves da Silva por toda a força que me deram.

Aos colegas de curso, Bruno Vinicius, Claudio Junior, Juliano Leal, Pablo Silvestre, Rodrigo Nicodemos, Andre Silva Fiuza, Carlos Ângelo por partilhar momentos de alegrias, de festas, de noites mal dormidas, de mal humor, de notas baixas e altas. Uma amizade que se propaga além da sala de aula e que eu levo pra minha vida toda!

"Ficar louco de vez em quando é necessidade básica para permanecer são."
- Osho

Resumo

Neste relatório será feito um estudo sobre a placa Balckfin ADSP-BF537 para o desenvolvimento do projeto do Monitor multiparamétrico Lifetouch.10 da Lifemed para uma futura fabricação deste monitor. Este relatório esta focado no estudo da arquitetura da placa. O sistema de Boot e Reset, o sistema de interrupção e o manuseio das JPIO.A analise de códigos no VisualDSP++ e os serviços de sistema oferecidos por esta plataforma e pesquisar a portabilidade desses códigos para o sistema embarcado μCLinux.

Palavras-chaves: VisualDSP++. μCLinux. GPIO. ADSP-BF537.

Abstract

In this report, it's a studied about the board Blackfin ADSP-BF537 for the development of the Multiparametric Monitor project used with the model Lifetouch.10 from Lifemed. This report is focus in the study of the architecture of the board, the boot system and reset, the interrupt system and de management of the GPIO. A code analysis on VisualDSP++ and the system services of the plataform and a research of the portablility of these codes to the μCLinux embedded system.

Keywords: VisualDSP++. μCLinux. GPIO. ADSP-BF537.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CEC	Controlador de Eventos do Processador
SIC	Controlador de Interrupções do Sistema
EVT	Tabela de Vetores de Eventos
GPIO	Portas de uso Geral
DMA	Acesso Direto à Memoria
API	Interface de Programação
PC	Controle de Potência
IC	Controle de Interrupção
IVG	Grupo de Vetores de Interrupção
UFCG	Universidade Federal de Campina Grande

Sumário

1 Introdução

A empresa Lifemed fechou um acordo com a universidade estadual UEPB para a realização do projeto do monitor multiparamétrico. Com este convenio a empresa Lifemed propos ao laboratorio NUTES o estudo da placa ADSP-BF537 para o uso no monitor. A finalidade do projeto é usar o sistema embarcado μCLinux para o controle da placa e este relatório esta no inicio deste projeto com o estudo da arquitetura da placa e a portabilidade dos códigos para o μCLinux.

1.1 Monitor

1.1.1 Descrição

O Lifetouch.10 é um monitor de sinais vitais, multiparâmetrico, pré-configurável em bloco único integrado com até 7 parâmetros simultâneos e visualização de 8 curvas, tela LCD colorida HD de 10.4", sistema operacional por comandos touch screen e simultaneamente por encoder ótico (chave rotativa) para tornar a operação simples, intuitiva, rápida e mais segura durante a utilização. É compacto, leve, formato slim line, portátil, alça incorporada para transporte, bateria interna recarregável do tipo Lítio-ion, armazena dados de tendência com programação gráfica e tabular até 96 horas, impressora térmica integrada opcional, permite atualizações (upgrade) e sistema de alimentação AC full range(90-260 Vac;47-67 Hz) .

Figura 1 – Monitor da LifeMed Lifetouch.10.

1.2 Objetivo

O objetivo deste relatorio é o estudo da arquitetura da placa Blackfin ADSP-BF537 usando a plataforma VisualDSP++ para acesso aos seus componentes. Analisar os códigos usados no VisualDSP++ e verificar a possibilidade de uso destes códigos na plataforma de sistema embarcado μCLinux.

2 Arquitetura da placa Blackfin ADSP-BF537

2.1 Introdução

As empresas Analog Devices e Intel criaram um microprocessador de sinal (Micro signal Architecture - MSA), que a Intel usa os microprocessadores em chipsets de telefones móveis e a Analog Devices na família de processadores Blackfin. A MSA tem a vantagem de ter um conjunto de instruções para microprocessadores de tipo RISK limpo e ortogonal. Combina uma unidade MAC dupla (multiplicativo e acumulativo), Possui um poderoso núcleo de processamento digital de sinais e capacidade de instrução-única para múltiplos dados (single-instruction multiple data - SIMD) em um conjunto de instruções de arquitetura.

A placa fornece tanto o uso como processador como processamento de sinal permitindo flexibilidade em particionar o uso da placa.O processador é de 600 MHz, e os recursos de processamento de sinal incluem uma porta de instruções e duas portas mapeadas separadamente de dados unificadas em um espaço de memória de 4GB, dois multiplicadores de rendimento de ciclo único de 16 bits, dois ALUs de 40 bit cada, quatro ALUs de vídeo de 8 bits cada. As características de micro-controlador são: Manipulação de bits arbitrária, um "mixed" para códigos de alta densidade de 16 ou 32 bits, proteção à memória, stack pointers, scratch SRAM, gerenciamento maleável do consumo de energia e interrupções priorizáveis para controle em tempo real. O diagrama de bloco da placa junto com os periféricos é mostrado na Figura 2.

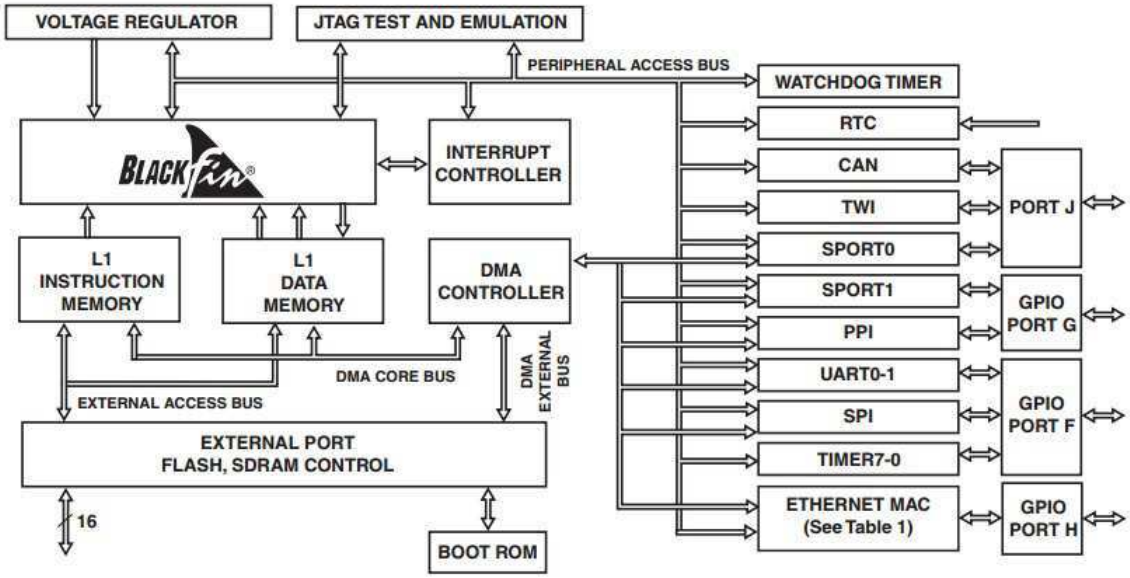

Figura 2 – Diagrama de blocos da placa ADSP-BF537.

2.2 Periféricos

A placa ADSP-BF537 contém uma rede de comunicação dedicada de alta velocidade tanto serial como paralela, um controle de interrupção para um gerenciamento flexível das interrupções para os periféricos da placa e fora dela, um gerenciador de potência com funções para customizar as características de potência do processador. A figura 3 demonstra os periféricos da placa em comparação com as placas similares do mesmo fabricante.

Features		ADSP-BF534	ADSP-BF536	ADSP-BF537
Ethernet MAC		—	1	1
CAN		1	1	1
TWI		1	1	1
SPORTs		2	2	2
UARTs		2	2	2
SPI		1	1	1
GP Timers		8	8	8
Watchdog Timers		1	1	1
RTC		1	1	1
Parallel Peripheral Interface		1	1	1
GPIOs		48	48	48
Memory Configuration	L1 Instruction SRAM/Cache	16K bytes	16K bytes	16K bytes
	L1 Instruction SRAM	48K bytes	48K bytes	48K bytes
	L1 Data SRAM/Cache	32K bytes	32K bytes	32K bytes
	L1 Data SRAM	32K bytes	—	32K bytes
	L1 Scratchpad	4K bytes	4K bytes	4K bytes
	L3 Boot ROM	2K bytes	2K bytes	2K bytes
Maximum Speed Grade		500 MHz	400 MHz	600 MHz
Package Options: CSP_BGA CSP_BGA		 208-Ball 182-Ball	 208-Ball 182-Ball	 208-Ball 182-Ball

Figura 3 – Tabela comparativa entre placas ADSP-BF534,BF536 e BF537.

2.3 Boot

Quando o sinal de reset é ativado, o processador começa uma procura de endereço das instruções e ás executas tanto para memória externa assíncrona quanto para o boot ROM interno. O boot ROM interno contém um pequeno boot do Kernell que carrega dados do aplicativo de uma memoria externa ou de um dispositivo *host*. Espera-se que esses dados estejam em um formato definido chamado *boot stream*, que consiste em múltiplos blocos de dados junto com comandos que instruem o boot do kernel em como inicializar a memoria SRAM L1 como também as memórias voláteis externas.

O boot do Kernell processa o boot stream bloco por bloco até encontrar uma função especial que termina o processo e aponta para o endereço de reset *0xFFA0 0000* na memoria L1. Este processo chama-se *booting*.

Existe três pinos dedicados a seleção do mode de boot chamado de *BMODE[2:0]* que instruem o processador em como se comportar após o reset. Se todos os pinos estiverem em nível baixo quando o botão de reset é solto, o processador pula o boot ROM e começa a execução de código no endereço *0x2000 0000* no banco de memoria externo 0. Um dispositivo com 16 bits de memoria tem que estar conectado a AMS0, caso contrario a execução do programa começa no endereço *0xEF00 0000* que contém o boot ROM. As

opções de boot possíveis usando os pinos *BMODE* estão apresentador na tabela 4.

Boot Source	BMODE[2:0]	Execution Start Address
Bypass boot ROM; execute from 16-bit external memory connected to ASYNC Bank 0	000	0x2000 0000
Use boot ROM to boot from 8-bit or 16-bit memory (PROM/flash)	001	0xEF00 0000
Reserved	010	0xEF00 0000
Boots from 8-, 16-, or 24-bit addressable SPI memory in SPI master mode with support for Atmel AT45DB041B, AT45DB081B, and AT45DB161B DataFlash® devices	011	0xEF00 0000
Boot from SPI host (slave mode)	100	0xEF00 0000
Boot from serial TWI memory (EEPROM/flash)	101	0xEF00 0000
Boot from TWI host (slave mode)	110	0xEF00 0000
Boot from UART host (slave mode)	111	0xEF00 0000

Figura 4 – Tabela com os modos de boot e seus endereços de inicialização.

2.4 Reset

O estado de reset inicializa o processo lógico, durante este estado, tanto aplicativos como o sistema operacional não são executados e o clock e parado nesse estado. O processador permanece neste estado enquanto durar o sinal do reset. Quando o sinal é desativado o processador completa a sequencia de reset e passa para o modo Supervisor, que executa o código encontrado no vetor de eventos do reset.Pode-se invocar o estado de reset no modo supervisor usando a instrução *RAISE*. Pode-se inicializar o reset das seguintes maneiras:

2.4.1 Hardware Reset

Este modo de reset é um evento assíncrono. O pino de entrada $RE\bar{S}ET$ tem que estar desativado para acionar o hardware reset. Este reset aplica um reset no sistema inteiro incluindo tanto o processador como os periféricos. Após o pino é desativado o processador garante que todos os periféricos sejam resetados. Após o reset o processador entra na sequencia de boot configurado no estado *BMODE*.

2.4.2 System Software Reset e Watchdog Timer

Existem três tipos de inicialização:

1. Configurando apropriadamente o watchdog timer.
2. escrevendo os bits *b#111* no registrador *SWRST* no endereço endereço *0xFFC0 0100*.
3. Usando a instrução *RAISE 1*.

O Watchdog Timer reseta o processador e os periféricos, o processo entra na sequencia de boot se o processador estiver no modo ativo. O boot é controlado pelo estado dos pinos *BMODE* e *NOBOOT* (no boot on software reset). Se o pino *NOBOOT* estiver desativado quem determina o boot é o pino *BMODE*.

O software reset pode ser inicializado ativando o *Reset Field* no registrado de software reset *(SWRST)* mostrado na figura 5.

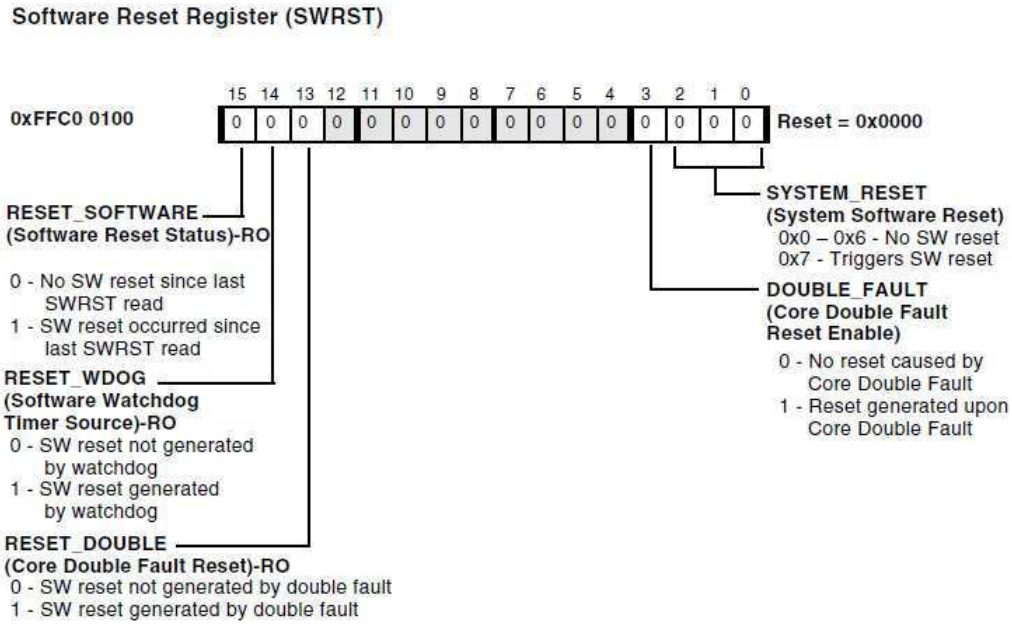

Figura 5 – Registrador Software Reset.

O software reset também pode ser inicializado usando a instrução *RAISE 1* ou ativando o bit *SYSRST* via o registrador de controle de Debug *(DBGCTL)* via emulação de software via JTAG. Em alguns processos o software reset afeta somente o processador, ja em outros o código de boot inicializa o sistema.

2.4.3 Core e System Reset

Para ativar o Core e System Reset é necessário inserir o código abaixo, este reset precisa de cinco ciclos de clock para se completar e é necessário que esteja na memoria L1.

$/ * Issuesystemsoftreset * /$
$P0.L = LO(SWRST);$

$P0.H = HI(SWRST);$
$R0.L = 0x0007;$
$W[P0] = R0;$
$SSYNC;$
$/* Waitforsystemresettocomplete(needstobe5SCLKs). */$
$/* AssumingaworstcaseCCLK : SCLKratio(15:1), use5*15 = 75 */$
$/* astheloopcount. */$
$P1 = 75;$
$LSETUP(start, end)LCO = P1;$
$start:$
$end:$
$NOP;$
$/* Clearsystemsoftreset */$
$R0.L = 0x0000;$
$W[P0] = R0;$
$SSYNC;$
$RAISE1;$

2.5 Sistema de interrupção

Existe dois mecanismos para o controle de interrupções e eventos

- Controlador de eventos do processador *(CEC)*: usa o clock especifico do processador *CCLK*, interage próximo a sequência de programação e administra a tabela de vetores de eventos *EVT*. O CEC esta relacionado a interrupções do tipo exceções, erro do processador, eventos de emulação e suporta interrupções de software.
- Controlador de interrupções do sistema *(SIC)*: Opera com o clock do sistema *SCLK*, usado para priorizar interrupções requerida dos periféricos e encaminha para o *CEC*.

A figura 6 demonstra como os periféricos são ligados ao SIC e como os registradores *SIC_IARx* gerenciam o controle das tarefas ligando-as as nove entrada disponíveis no *CEC*. Os registradores *ILAT, IMASK, IPEND* mapeados em memória são parte do controlador *CEC*.

2.5.1 Eventos e Sequencias

Existe dois níveis de controle de eventos no processador, o controlador SIC trabalha junto com o CEC para controlar e priorizar as interrupções do sistema. O SIC mapeia

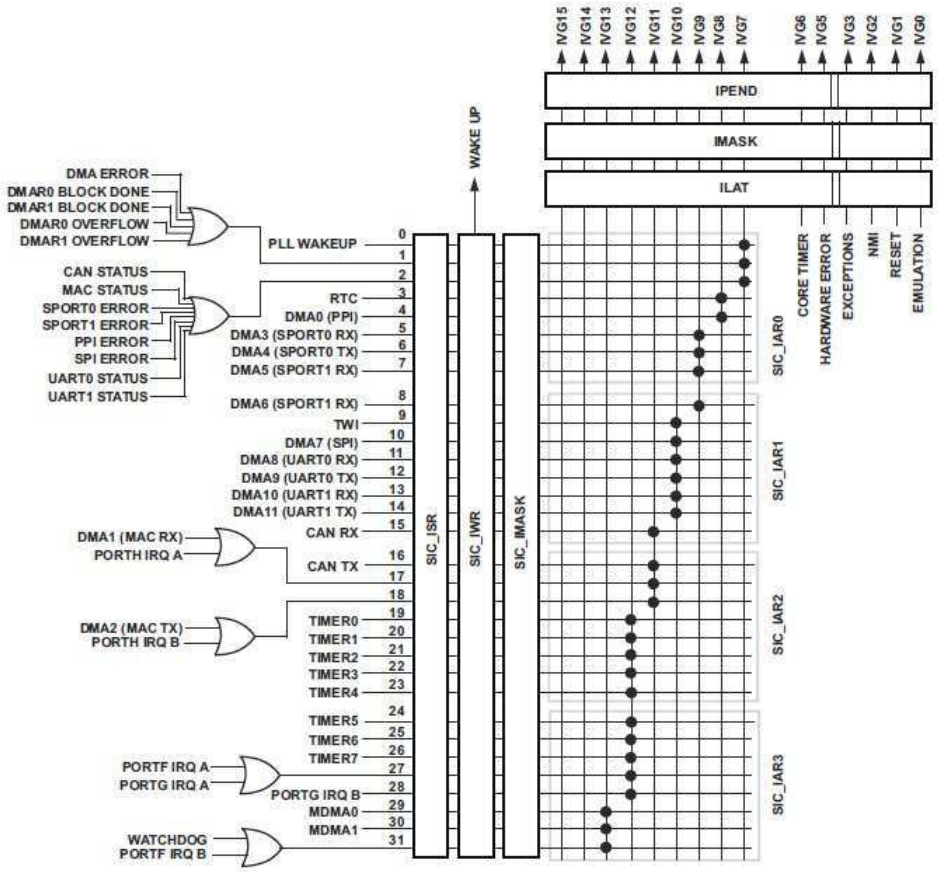

Figura 6 – Visão global da rotina de interrupção.

as interrupções vindas dos periféricos e prioriza as interrupções vindas do processador. Esse mapeamento é programável e interrupções podem ser mascaradas individualmente. O CEC administra cinco tipos de eventos:

1. Emulação.
2. Reset.
3. Interrupções não-mascaráveis (NMI).
4. Exceções.
5. Interrupções.

A interrupções é o evento que modifica o fluxo de instruções do processador e é um evento assíncrono. A exceção é um evento iniciado por software que é síncrono com o fluxo de instruções. Os eventos são aninhados e priorizados,na tabela 4.1 do manual TAL estão as interrupções do sistema junto com seu nível de prioridade.

2.5.2 Tabela de Vetores de Eventos

A tabela de vetores de eventos (*EVT*) é uma tabela do hardware com 16 entrada de 32 bits. A EVT contém uma entrada para cada evento possível no processador. As entradas são acessadas através do mapeamento de memoria (*memory map register - MMR*) e cada uma pode ser programada no reset, com o endereço apropriado, na rotina de serviço de interrupção. Quando um evento ocorre, as instruções procuram o endereço na tabela EVT para cada evento. Interrupções não são determinada por endereços fixos, com isso é possível usar um endereço qualquer para uma interrupção na tabela EVT. No texto (ANALOG DEVICES, INC, 2013) encontra-se a tabela em usa totalidade.

2.5.3 Sistema de interrupções dos periféricos

Para as interrupções dos periféricos, o SIC contém 32 entradas de interrupções e 9 saídas que se conectam com o CEC. A principais funções do SIC é mascarar, agrupar e priorizar solicitações de interrupção e direcioná-las para as 9 interrupções de propósito geral do CEC (IVG 7 - IVG 15). o SIC pode ativar individualmente qualquer Interrupção de periférico de um estado inativo. As noves interrupções de propósito geral (IVG 7 - IVG 15) do processador tem prioridade fixa. O *IVG0* tem a maior prioridade e o *IVG15* a menor, por tanto as designações ocorridas no registrador *SIC_IARx* alem de agrupar as interrupções também as prioriza. Porém, a prioridade das interrupções do proposito geral podem ser modificadas no processador. Se mais de uma fonte de interrupção é mapeada na mesma interrupção, a operação logica *OR* é utilizada sem priorização. Dependendo de como as interrupções são mapeados no processador, a rotina de serviços de interrupção (ISR) terá que interrogar vários bits de status para determinar a fonte de interrupção. Uma das primeiras instruções a ser lida é o registrador *SIC_ISR*, que é um registrado com bits individuais para cada fonte de interrupção vinda dos periféricos.

2.5.4 Modelo de Programação

A inicialização do sistema de interrupção envolve os seguintes passos.

- Inicialização da tabela EVT.
- Inicialização do registrador *IMASK*.
- Desmascarar a interrupção do periférico específico no registrado *SIC_MASK*.

A Figura 7 mostra os passos ocorridos quando uma interrupção é chamada, quando ocorrido, as seguintes ações são ocorridas.

1. O log do *SIC_ISR* é requisitado e acompanha as interrupções do sistema ativas mas não em serviço.

2. *SIC_IWR* verifica se é necessária alternar o estado do processador para um estado ativo para realizar a requisição.

3. *SIC_IMASK* ativa as interrupções dos periféricos a nível de processador, se a interrupção não esta mascarada, a requisição pula para o passo 4.

4. Os registradores *SIC_IARx* determinam a prioridade da interrupção.

5. O *ILAT* adiciona a interrupção requerida ao log de interrupções ativadas mas não em serviço.

6. O *IMASK* desmascara ou ativa evento em diferentes níveis de prioridade, se a interrupção não for mascarada pula-se esta etapa.

7. A tabela EVT é acessada para adicionar a interrupção no nível de interrupção adequado.

8. Quando a interrupção estiver ativa no processador, o bit apropriado do *IPEND* é ativado, que zera o respectivo bit *ILAT*, por isso que o bit *IPEND* rastreia todas as interrupções pendentes.

9. Quando a rotina ISR para a interrupção requerida for executada, a instrução *RTI* zera o bit apropriado do *IPEND*, porem o bit *SIC_ISR* não é zerado que o ISR retire os mecanismo que ativam a interrupção requerida.

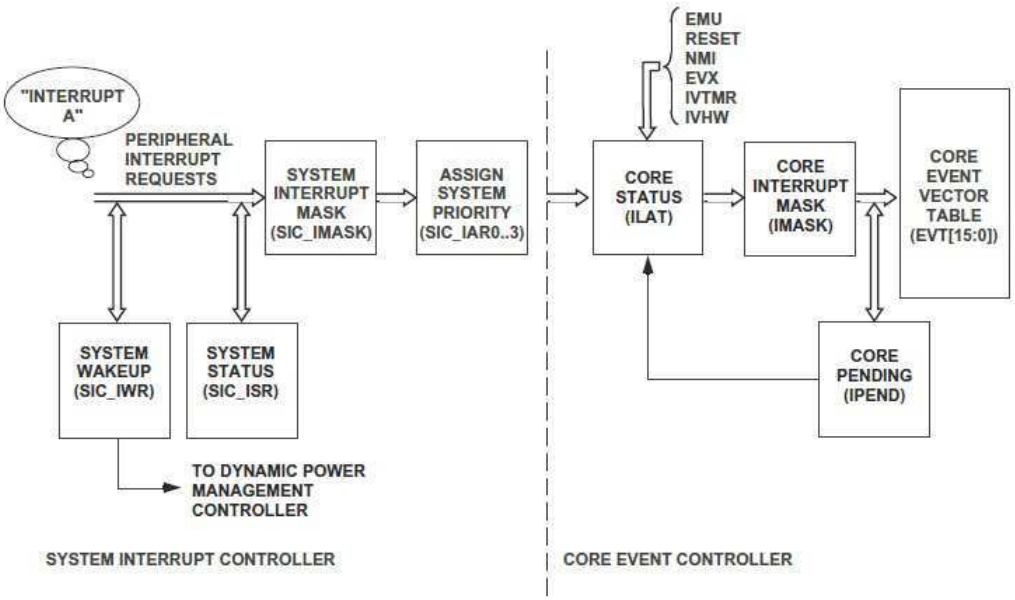

Figura 7 – Diagrama de blocos do processo de interrupção.

2.6 Timers

Existem noves timers programaveis no processador, destes, oito tem um pino que os configura tanto como um PWM ou como um timer de saída. Os timers podem ser sincronizados com um clock externo conectados pelo pino *PF1*, pelo pino *PPI_CLK* ou pelo clocl interno *SCLK*. O timer pode gerar interrupções no processador para inicializar eventos. O nono timer é usado é usado internamente como clock para interrupções internar do sistema.

2.7 Portas de uso geral (GPIO)

Existem 48 pinos bidirecionais para uso geral (*GPIO*) alocados em três modulos GPIOs: PORTFIO, PORTGIO e PORTHIO associados com as portas F, G e H. Cada pino pode ser controlado individualmente a partir da manipulação das portas, dos registrados de status e de interrupção que são os seguintes:

- O registrador GPIO Direction Control, que especifica a direção de cada GPIO individualmente como entrada ou saída.
- Registrador GPIO Control and Status, o processador emprega o "*write one to modify*" que permite qualquer combinação de pinos do GPIO para serem modificados em uma única instrução sem afetar qualquer outro pino GPIO.
- Registrador GPIO Interrupt Mask, os dois registradores existentes permitem que cada pino funcionem como uma interrupção para o processador.
- Registrador GPIO Interrupt Sensitivity, os dois registradores existentes especifica quando o pino for sensível a nível ou a mudança de nível, se for a mudança de nível, configura para subida ou descida do nivel.

3 Análise de códigos no VisualDSP++

A análise dos códigos no VisualDSP++ exige um estudo das bibliotecas mais usadas nessa IDE. É necessário um estudo sobre os serviços oferecidos das bibliotecas e os driver para cada periférico. Os serviços do sistema formam uma coleção de funções que normalmente encontrados em sistemas embarcados, cada serviço é focado para uma funcionalidade especifica como acesso direto á memoria (DMA), controle de potência (PM), controle de interrupção (IC) entre outros.

Cada serviço exporta uma interface de programação (API) que define a interface para cada serviço. Aplicações acessam as APIs para ter acesso as funcionalidades que querem ser controladas. Cada API pode ser acessada usando comandos em C ou em assembly. As vezes quando se quer acessar uma API, o serviço requer que outra API seja chamada, na maioria das vezes os serviços são independentes, porém redundancias são superadas quando permite-se que um serviço acesse o outro. A Figura 8 mostra a coleção de serviços do sistema e suas interações.

Os códigos usado como teste estão no anexo.

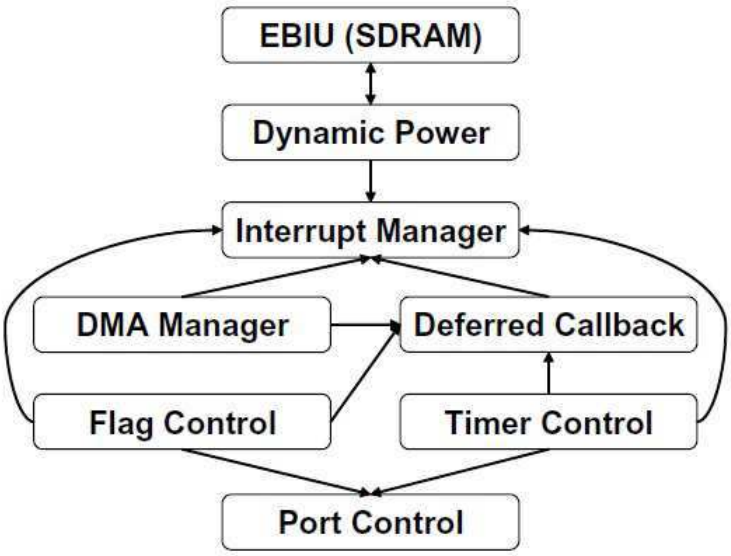

Figura 8 – Serviços do sistema e suas interações.

Neste estudo aprofundaremos nos serviço Flag control, Timer control, Deferred Callback e interrupt Manager que foram os serviços estudados.

3.1 Interrupt Manager

O controlador de interrupção prove com funções que permite controlar todos os aspectos do sistema de interrupção e eventos. O processador fornece 16 níveis de interrupção e eventos chamados de grupo de vetores de interrupção (IVG), que são enumerado de 0 á 15 e com ordem de prioridade decrescente. Alguns níveis são dedicados para certos eventos por exemplos para emulação, e outros são de uso geral (IVG7 - IVG15).

Todo o processamento do IVG é feito no CEC, quando um nível IVG é ativado, o CEC procura a entrada adequada na tabela de vetores de eventos (EVT) e aponta o vetor de execução para o endereço da tabela.

O controlador de interrupção permite que que a aplicação tenho total controle sobre como as interrupções são manuseadas, quer estejam mascaradas ou desmascaradas, mapeadas uma á uma ou todas juntas, se é necessário ativar o processador ou não e muitas outras. Permite também a criação de interrupções em cadeias, que a aplicação pode ligar (hook) qualquer numero de interrupções a qualquer nível IVG. Quando vários eventos são ligados ao mesmo nível IVG, isto permite que cada evento seja designado independente e permite que a aplicação processe as interrupções de uma maneira direta.

Todos os comandos do serviço de interrupção usa uma linguagem única, como para qualquer serviço. Valores de enumerações, declarações *typedef* e macros usam o prefixo $ADI\_INT\_$, enquanto as funções do serviço usam o prefixo $adi\_int\_$. Todas as funções retornam retornam o valor em ADI_INT_RESULT como em todos os serviços do sistema. Quando o código for bem executado o código $ADI\_INT\_RESULT\_SUCCESS$ terá como resultado 0.

3.1.1 Inicialização do controlador de interrupção

3.2 Deferred Callback Manager

Funções Callback são geralmente usadas quando a aplicação requer do gerenciador que notifique quando um evento for completado, por exemplo uma transferência DMA. A necessidade de executar uma função Callback se executa a rotina de serviços de interrupção (ISR) em uma prioridade alta, para tal prioridade o ISR tenta executa-la no mínimo tempo possível. Por outro lado Callback podem ser lentos e não determinístico. Muitas vezes o usuário prefere deferir a execução do Callback para uma prioridade mais baixa, com isso a requisição do ISR pode ser feita sem atrasos.

A biblioteca de serviços do callback deferido provê esse serviço através da gestão dos pedidos de Callback deferidos, essas funções são invocadas tipicamente como funções de baixo nível de prioridade em relação ao resto da aplicação. O numero de Callback deferidos em uma aplicação é determinado pelo usuário na inicialização. Embora só é

possível ter uma requisição por nível IVG, pode-se usar um software de priorização, não ha limite para o numero de requisições que podem ser usadas neste software. As requisições são executadas em ordem de prioridade dentro do nível IVG.

3.2.1 Usando o Gerenciador de Callback Deferido

O funcionamento do gerenciador DCB da-se da seguinte maneira

1. Configurando o gerenciador DCB.
   a) Inicializando o DCB.
   b) Abrindo uma fila.
2. Gerenciando a requisição.
   a) Nomeando um Callback para a fila.
   b) Remeter o callback para o nivel de prioridade.
3. fechando a fila.
4. Finalizando o gerenciador DCB.

3.3 Serviço de Flags programáveis

O serviço de Flags provê com uma fácil interface para programação dos flags, também conhecido como GPIO.

Usando os recursos dos outro serviços, o serviço de flag consegue controlar a direção dos flags, mudar os valores dos flags e notificar sobre a mudança do pino do flag via callback live ou deferred. Este serviço é independente dos serviços de interrupção e do DCB. Se os callback não forem "deferred", mas live, o serviço DCB não é necessário e se callback não são necessários o serviço de interrupção não é necessário.

3.3.1 Operação

Para usar o serviço Flags é necessário inicia-lo usando a função *adi_flag_Init*. Nesta função, um parâmetro é usado para proteger a região critica do código e uma parte da memoria que este serviço estiver usando. Para controlar e administrar os callback, o serviço flag necessita de um pequeno espaço de memoria para cada callback. O tamanho exato da memoria é definida no macro *ADI_FLAG_CALLBACK_MEMORY*. O usuário provê a quantidade mínima especificada pelo macro vezes a quantidade de callback a ser usado. Quando os serviços do callback não forem mais necessários o usuário finaliza o serviço com o comando *adi_flag_Terminate*.

No capitulo 7 do manual *Device Drivers and System Services Manual for Blackfin® Processors* esta disponível todos os comandos usado por este serviço

3.4 Timer Service

O serviço timer provê com uma fácil interface para o uso dos timers do processador, o Watchdog timer e os timers de uso geral.

Usando os recursos dos outros serviços o serviço timer permite ao usuário controlar todos os timer de uma forma consistente e instalar callback para que sejam notificados quando expirados. Este serviço é independente do serviço de interrupção e o DCB. Se os callback não forem deferred, mas live, o serviço DCB não é necessário e se callback não são necessários o serviço de interrupção não é necessário.

3.4.1 Operação

Para usar o serviço timer é necessário inicia-lo usando a função *adi_tmr_Init()*. Nesta função, um parâmetro é usado para proteger a região critica do código e uma parte da memoria que este serviço estiver usando. Quando os serviços do timer não forem mais necessários o usuário finaliza o serviço com o comando *adi_tmr_Terminate()*.

No capitulo 8 do manual *Device Drivers and System Services Manual for Blackfin® Processors* esta disponível todos os comandos usado por este serviço.

3.5 Codigos para VisualDSP++

Alguns códigos foram usados para testar os periféricos da placa.

3.5.1 Acionando Leds por registradores

Na Figura 9 usou-se uma FFT no filtro, que mostra a resposta do filtro no dominio da frequência. Código em anexo

3.5.2 ECHO via porta serial

Este programa demonstra o uso da porta serial(UART) através do programa chamado “ECHO”. Este programa detecta a taxa de Baud automaticamente com o uso de um carácter inicial definido pelo usuário, barra de espaço. O programa recebe o carácter do computador e manda de volta pela porta serial.

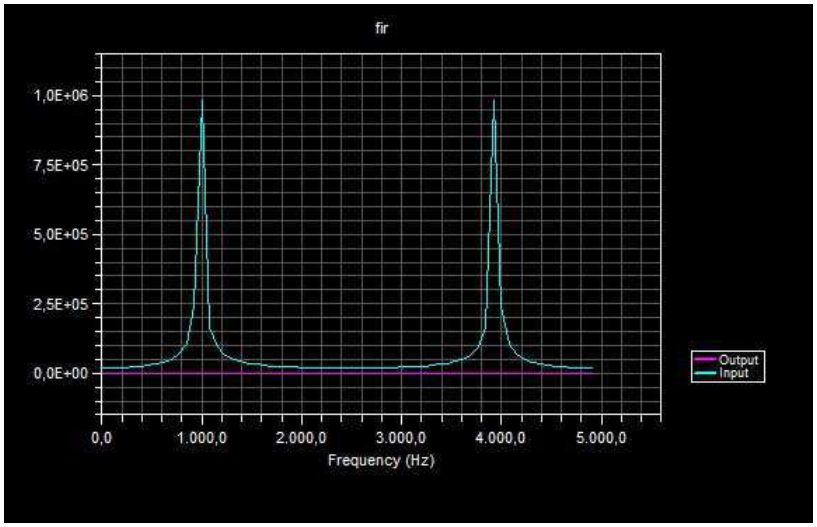

Figura 9 – Performance de FFT no filtro.

4 Portabilidade para o Linux

4.1 Portabilidade de códigos Assembly do VisualDSP++ para o GNU Toolchain

Antes de fazer essa transição é necessário saber que formato de executável esta sendo usado.

- FLAT: arquivos binarios Flat mais conhecido como *BFLT*, que são executáveis simples e leves baseados no formato de saida a.out.
- ELF: formato que executa e "linka", originalmente desenvolvido pelos laboratórios Unix System que tornou-se o formato padrão. O formato ELF é mais poderoso e flexível do que o FLAT porém são mais pesados necessitando mais espaço no disco.

4.1.1 Nome dos arquivos

O codigo Assembly do VisualDSP++(VDSP) usa o prefixo *.asm* enquanto o GNU assembler (GAS) usa o prefixo *.s* ou *.S*. Ambos arquivos contém comandos de pre-processamento, diretivas assembles comentários etc, que precisam ser pre-processados. Arquivos GNU não passam por um pre-processador C. Como GAS ou gcc não reconhece os arquivos .asm por padrão, é necessário dizer-lhe explicitamente que são arquivos assembler. isto é feito através do flag *-x assembler or -x assembler-with-cpp*. Usando este comando as sintaxes de comando do VDSP podem ser usado no GNU Toolchain.//

- A opção *-x assembler-with-cpp* comunica ao GCC que o arquivo de entrada é um assembly que precisa passar por um pre-processador C e que é um equivalente ao arquivo .s
- A opção -x assembler comunica ao GCC que o arquivo é um código assembly e é equivalente ao s.

4.1.2 Diretivas Assembly

Diretivas em um codigo assembly de controle contém os processos de assembly. Como as instruções em assembly, diretivas não produzem opcodes. Muitas das diretivas do VDSP não são reconhecidas pelo GCC e vice-versa. é necessario saber as diretivas equivalentes. No site (PORTING...,) exite uma lista comparativa.

4.2 Portabilizando codigos do VisualDSP++ para o Linux

O VisualDSP++ é um excelente software para desenvolver algoritmos de processamento de sinal, porém não é provido com um sistema de serviço robusto como o Linux. Pode ser muito útil recompilar os códigos do compilador VDSP com um gcc, g++ ou gas para que se possa ser usado no Linux. O sistema Linux é diferente do VDSP em muitas maneiras e a portabilidade tem ser avaliada individualmente, mas quando a portabilidade é feita pode-se usar como qualquer outro aplicativos Linux. a tabela tal mostra as ferramentas equivalentes do dois sistemas

Tool	$VDSP++$		GNU Toolchain	
	Executável	Extensão	Executável	Extensão
assembler	easmBLKFN	.asm	bfin-xxx-as	.S
Compilador C		.c	bfin-xxx-gcc	.c
Compilador C++	pp	.cpp	bfin-xxx-cpp	.cpp
archiver	elfar	.dlb	bfin-xxx-ar	.a
Formato do arquivo Bootrom	elfloader		bfin-xxx-ldr	
Linker	linker	.ldf	bfin-xxx-ld	.lds

4.2.1 Portabilizando códigos CC++

Em muitos casos portabilizar códigos CC++ requer simplesmente uma recompilação do código, mas é necessário tomar cuidado com as diferenças dos compiladores. Como cada compilador define seu próprio conjunto de extensões, é necessário saber as extensões equivalentes. Por exemplo:

- No VDSP o comando *#pragma* é escrito assim:
  - #pragma align 4
    char a;
- No Compilador GCC é escrito desta maneira
  - $chara\_\_attribute\_\_((aligned(4)))$;

No site **??** tem uma lista completa das extensões equivalente entre os dois compiladores.

5 Conclusão

Aprofundou-se o conhecimento da arquitetura da placa ADSP-BF537,averigou-se coomo a geração de códigos é criada no VisualDSP++ e como usar os serviços do sistema para uma criação de códigos de entendimento mais fácil e rápida. Estudou-se uma introdução a portabilidade de códigos para o μCLinux, a portabilidade de códigos tem que ser feita individualmente porque nem todas as bibliotecas estão disponíveis nos dois sistemas. O projeto irá continuar e este estudou complementou um pouco este projeto.

Referências

ANALOG DEVICES. *ADSP-BF537 EZ-KIT Lite Evaluation System Manual*. [S.l.], 2014.

ANALOG DEVICES, INC. *ADSP-BF537 Blackfin Processor Hardware Reference*. 3.4. ed. [S.l.], 2013. Citado na página 21.

ANALOG DEVICES,INC. *Device Drivers and System Services Manual for Blackfin Processors*. 3.2. ed. [S.l.], 2008.

ANALOG DEVICES,INC. *C/C++ Compiler and Library Manual for Blackfin Processors*. 5.4. ed. [S.l.], 2011.

ANALOG DEVICES,INC. *Blackfin Processor Programming Reference*. 2.2. ed. [S.l.], 2013.

ANALOG DEVICES,INC. *ADSP-BF534/ADSP-BF536/ADSP-BF537*. [S.l.], 2014.

PORT C/C++ source code from VDSP to GCC. Disponível em: <https://blackfin.uclinux.org/doku.php?id=visualdsp:port_c_code>.

PORTING Visual DSP++ assembly source code to the GNU Toolchain. Disponível em: <https://blackfin.uclinux.org/doku.php?id=visualdsp:port_assembly_code>. Citado na página 29.

6 Anexo - Codigos

6.0.2 Leds

Este código usou os comandos dos registradores sem o uso dos serviços de sistema. É um código mais complicado de se entender por manipular diretamente os registradores.

```
#include<cdefBF537.h>
// Codigo para acionar LEDs com os botões da placa.
int main( void )
{

*pPORTFIO_DIR = 0x0FC0;

*pPORTFIO = 0x0000;

*pPORTFIO_INEN = 0x003C;
while(1)
{
*pPORTFIO = (*pPORTFIO & (0x003C)) <<4;
}

return 0;
}
```

6.0.3 Filtro FIR

```
/*****************************************************************************
 Copyright (c) 2000 Analog Devices Inc. All rights reserved.
 *****************************************************************************
  File Name      : fir_test.c
  Description    : This module tests the fir function.

*****************************************************************************/

/* Includes */
#include "mds_def.h"
```

```
#include "filter.h"
#include "fir_coeff.h"
#include "fir_input.h"

/* Constants */
#define DELAY_SIZE       BASE_TAPLENGTH

fract16 delay[DELAY_SIZE];
fract16 OUT[BUFFER_SIZE];

void main()
{
int i,
nsamples,
tapLength;

    fir_state_fr16 s;

    nsamples = BUFFER_SIZE;
    tapLength = BASE_TAPLENGTH ;

fir_init(s, h, delay, tapLength);

_fir(IN, OUT, nsamples, &s);

}
```

6.0.4 Codigo para porta serial

```
/*****************************************************************

Copyright(c) 2004 Analog Devices, Inc. All Rights Reserved.

This software is proprietary and confidential.  By using this
software you agree to the terms of the associated Analog Devices
License Agreement.

$RCSfile: UART_Autobaud.c,v $
$Revision: 4 $
```

```
$Date: 2009-07-28 11:25:38 -0400 (Tue, 28 Jul 2009) $

*****************************************************************/

/*****************************************************************

Include files

*****************************************************************/

#include <services/services.h>// system services
#include <drivers/adi_dev.h>// device manager includes
#include <drivers/uart/adi_uart.h>// uart driver includes

#include "ezkitutilities.h" // EZ-Kit utilities

/*****************************************************************

User configurations:

Callbacks can be either "live" meaning they happen at hardware interrupt
time, or "deferred" meaning that the Deferred Callback Service is used
to make callbacks at a lower priority interrupt level.  Deferred
callbacks usually allow the system to process data more efficiently with
lower interrupt latencies.

The macro below can be used to toggle between "live" and "deferred"
callbacks.  All drivers and system services that make callbacks into an
application are passed a handle to a callback service.  If that handle
is NULL, then live callbacks are used.  If that handle is non-NULL,
meaning the handle is a real handle into a deferred callback service,
then callbacks are deferred using the given callback service.

Also included is a macro that can be used to set the baud rate for
the UART.
```

```
*****************************************************************/

#define USE_DEFERRED_CALLBACKS // enables deferred callbacks

/*****************************************************************

Autobaud initiation Macros

The Autobaud detection is initiated by a specific character sent from
the Terminal program. User can set their own character used in
Autobaud operation. The character can be defined using two macros.

DIVISOR_BITS macro sets the number of UART bits to be captured. This
value is the number of bits counted up to the first pulse width of the
character, including start bit.
Example: for space character, the hex equivalent is 0x20. So the
DIVISOR_BITS are calculated as 1 startbit + 5 bits (bit 0 - 4 are 0)
The command 'ADI_UART_CMD_SET_DIVISOR_BITS' sets appropriate value
in the PDD

AUTOBAUD_CHAR macro directly sets the decimal value of the Autobaud
initiation character & the driver calculates the divisor bits from it
The command 'ADI_UART_CMD_SET_AUTOBAUD_CHAR' sets appropriate character
in the PDD

Use one of these macros and enable the appropriate command
*****************************************************************/

// Set the number of captured UART Bits

//#define DIVISOR_BITS (6)
// Autobaud character is `(Grave Accent) or Space

// Set Autobaud initiation character (Decimal Value)

#define AUTOBAUD_CHAR (32)
// set Space as initiation character

/*****************************************************************
```

```
Static data

*****************************************************************/

// Create two buffer chains. One chain will be used for one of
// the frames,the other chain for the other frame. Note that
// in this example there is only 1 buffer in each chain.
ADI_DEV_1D_BUFFER InboundBuffer;
ADI_DEV_1D_BUFFER OutboundBuffer;

// data for the inbound and outbound buffer
u8 InboundData;
u8 OutboundData;


// Deferred Callback Manager data
// (memory for 1 service plus 4 posted callbacks)
#if defined(USE_DEFERRED_CALLBACKS)
static u8 DCBMgrData[ADI_DCB_QUEUE_SIZE + (ADI_DCB_ENTRY_SIZE)*4];
#endif

// Device Manager data (base memory + memory for 1 device)
static u8 DevMgrData[ADI_DEV_BASE_MEMORY + (ADI_DEV_DEVICE_MEMORY * 1)];

// Handle to the UART driver
static ADI_DEV_DEVICE_HANDLE  DriverHandle;

// String to be displayed in the Terminal screen
static unsigned char s1[] = "EZ-Kit Blackfin UART";
static unsigned char s2[] = " communicating at baudrate = 0x";
static unsigned char s3[] = "Type any character in the screen and
                             Blackfin will echo it\n";
static unsigned char s4[] = "-----------------------------------------
                             -----------\n\n";

/*****************************************************************

Static Functions
```

```
*****************************************************************/

static void SendString ( // Transmits a String via UART
unsigned char s[]
);

static void SendChar ( // Transmits a character via UART
unsigned char c
);

static void HextoASCII ( // Converts Hex Value to ASCII equivalent
u8 HexValue
);

/*****************************************************************

Function: ExceptionHandler
HWErrorHandler

Description: We should never get an exception or hardware error,
but just in case we'll catch them and simply turn
on all the LEDS should one ever occur.

*****************************************************************/

static ADI_INT_HANDLER(ExceptionHandler) // exception handler
{
ezErrorCheck(1);
return(ADI_INT_RESULT_PROCESSED);
}


static ADI_INT_HANDLER(HWErrorHandler) // hardware error handler
{
ezErrorCheck(1);
return(ADI_INT_RESULT_PROCESSED);
}
```

```
/*********************************************************************

Function: Callback

Description: Each type of callback event has it's own unique ID
so we can use a single callback function for all
callback events.  The switch statement tells us
which event has occurred.

In the example, we've configured the buffers going
to the driver for inbound data to generate a callback
but the buffers going to the driver for outbound
data will not generate a callback.

When we get a callback for an inbound buffer, we
simply copy the data to the outbound buffer, then
send the outbound buffer to the device for
transmission and re-queue the inbound buffer so that
it can receive the next piece of data.

*********************************************************************/

static void Callback(   // call back function specific for UART0
void *AppHandle,
u32  Event,
void *pArg)
{

ADI_DEV_BUFFER *pBuffer; // pointer to the buffer that was processed
u32 THR;

// CASEOF (event type)
switch (Event) {

// CASE (buffer processed)
case ADI_DEV_EVENT_BUFFER_PROCESSED:

// identify which buffer is generating the callback
// we're setup to only get callbacks on inbound buffers so we know this
```

```
// is an inbound buffer
pBuffer = (ADI_DEV_BUFFER *)pArg;

// copy the data to the outbound buffer
OutboundData = InboundData;

// send the outbound buffer
ezErrorCheck(adi_dev_Write(DriverHandle, ADI_DEV_1D,
             (ADI_DEV_BUFFER *)&OutboundBuffer));

// requeue the inbound buffer
ezErrorCheck(adi_dev_Read(DriverHandle, ADI_DEV_1D,
             (ADI_DEV_BUFFER *)&InboundBuffer));

// cycle the LEDs so it looks like we're doing something important
ezCycleLEDs();
break;

// CASE (Autobaud detection Complete)
case ADI_UART_EVENT_AUTOBAUD_COMPLETE:

SendString("\n"); // Skip a line
SendChar(0x0D); // Carriage Return

SendString(s1); // Display string s1
SendChar('0'); // Display Character 0
SendString(s2); // Display String s2

// Display the Divisor value calculated
HextoASCII((*(u32 *)pArg) >> 12); // Display nibble3
HextoASCII((*(u32 *)pArg) >> 8); // Display nibble2
HextoASCII((*(u32 *)pArg) >> 4); // Display nibble1
HextoASCII(*(u32 *)pArg); // Display nibble0

SendString("\n"); // Skip a line
SendChar(0x0D); // Carriage Return
SendString(s3); // Display String s3
SendChar(0x0D); // Carriage Return
SendString(s4); // Display String s4
```

```
SendChar(0x0D); // Carriage Return

break;

// CASE (an error)
case ADI_UART_EVENT_BREAK_INTERRUPT:
case ADI_UART_EVENT_FRAMING_ERROR:
case ADI_UART_EVENT_PARITY_ERROR:
case ADI_UART_EVENT_OVERRUN_ERROR:

// turn on all LEDs and wait for help
ezTurnOnAllLEDs();
while (1) ;

// ENDCASE
}

// return
}


/*****************************************************************
*
* Function: SendString
*
* Description: Transmit a string via the UART
*
*****************************************************************/
static void SendString (
unsigned char s[]
){
u32 i; // count value
for(i=0;s[i]!=0x00;i++)
SendChar(s[i]);

// return
}

/*****************************************************************
```

```
*
* Function: SendChar
*
* Description: Transmit a character via the UART
*
*********************************************************************/
static void SendChar (
unsigned char c
){

    u32     THRstatus;

   THRstatus = FALSE;
   while (1){
    if (THRstatus){
    // Is the transmitter ready to accept another data?
        OutboundData = c; // yes, send the next data
        // send the outbound buffer
        ezErrorCheck(adi_dev_Write(DriverHandle, ADI_DEV_1D,
             (ADI_DEV_BUFFER *)&OutboundBuffer));
        break; }
    else
     // check the transmitter buffer status
       ezErrorCheck(adi_dev_Control(DriverHandle,
               ADI_UART_CMD_GET_TEMT,(void *)&THRstatus));
}

// return
}

/*********************************************************************
*
* Function: HextoASCII
*
* Description: Converts Hex value to ASCII equivalent
*  & sends via UART
*********************************************************************/
static void HextoASCII ( // Converts Hex Value to ASCII equivalent
u8 HexValue
```

```
){
HexValue &= 0x0F;
if(HexValue >= 0x0A)
SendChar(HexValue+0x37);
else
SendChar(HexValue+0x30);
}

/*************************************************************************
*
* Function: main
*
*************************************************************************/

void main(void) {

ADI_DCB_HANDLE DCBManagerHandle;
// handle to the callback service
ADI_DEV_MANAGER_HANDLE  DeviceManagerHandle;
// handle to the Device Manager
u32  ResponseCount;
// response counter
u32 i; //loop variable

ADI_DEV_CMD_VALUE_PAIR ConfigurationTable [] = {
// configuration table for the UART driver
{ ADI_DEV_CMD_SET_DATAFLOW_METHOD,  (void *)ADI_DEV_MODE_CHAINED },
{ ADI_UART_CMD_SET_DATA_BITS,  (void *)8 },
{ ADI_UART_CMD_ENABLE_PARITY,  (void *)FALSE },
{ ADI_UART_CMD_SET_STOP_BITS,  (void *)1 },
// { ADI_UART_CMD_SET_DIVISOR_BITS,(void *)DIVISOR_BITS },
 // Only if DIVISOR_BITS Macro enabled
{ ADI_UART_CMD_SET_AUTOBAUD_CHAR,(void *)AUTOBAUD_CHAR },
// Only if AUTOBAUD_CHAR Macro enabled
{ ADI_DEV_CMD_END,NULL },
};

// initialize EZ-Kit
ezInit(1);
```

```
// initialize the Interrupt Manager and hook the exception
// and hardware error interrupts all interrupts are on
// unique IVGs so we don't need any secondary handler memory
ezErrorCheck(adi_int_Init(NULL, 0, &ResponseCount, NULL));
ezErrorCheck(adi_int_CECHook(3, ExceptionHandler, NULL, FALSE));
ezErrorCheck(adi_int_CECHook(5, HWErrorHandler, NULL, FALSE));

// initialize the Deferred Callback Manager and setup a queue
#if defined(USE_DEFERRED_CALLBACKS)
ezErrorCheck(adi_dcb_Init(&DCBMgrData[0], ADI_DCB_QUEUE_SIZE,
             &ResponseCount, NULL));
ezErrorCheck(adi_dcb_Open(14, &DCBMgrData[ADI_DCB_QUEUE_SIZE],
(ADI_DCB_ENTRY_SIZE)*4, &ResponseCount, &DCBManagerHandle));
#else
DCBManagerHandle = NULL;
#endif

//Initialize the flag service, memory is not passed because callbacks
//  are not being used
ezErrorCheck(adi_flag_Init(NULL, 0, &ResponseCount, NULL));

//Initialize all LEDS
for (i = EZ_FIRST_LED; i < EZ_NUM_LEDS; i++){
        ezInitLED(i);
}

// turn off all LEDs
ezTurnOffAllLEDs();

// initialize the Device Manager
ezErrorCheck(adi_dev_Init(DevMgrData, sizeof(DevMgrData),
             &ResponseCount, &DeviceManagerHandle, NULL));

// create two buffers that will be initially be placed on
//the inbound queue only the inbound buffer will have a callback
InboundBuffer.Data = &InboundData;
```

```
InboundBuffer.ElementCount = 1;
InboundBuffer.ElementWidth = 1;
InboundBuffer.CallbackParameter = &InboundBuffer;
InboundBuffer.ProcessedFlag = FALSE;
InboundBuffer.pNext = NULL;

OutboundBuffer.Data = &OutboundData;
OutboundBuffer.ElementCount = 1;
OutboundBuffer.ElementWidth = 1;
OutboundBuffer.CallbackParameter = NULL;
OutboundBuffer.ProcessedFlag = FALSE;
OutboundBuffer.pNext = NULL;

// open the UART driver for bidirectional data flow
ezErrorCheck(adi_dev_Open(DeviceManagerHandle, &ADIUARTEntryPoint,
             0, NULL, &DriverHandle,
             ADI_DEV_DIRECTION_BIDIRECTIONAL, NULL,
             DCBManagerHandle, Callback));

// configure the UART driver with the values from the configuration table
    ezErrorCheck(adi_dev_Control(DriverHandle, ADI_DEV_CMD_TABLE,
             ConfigurationTable));

    // Enable Autobaud detection
ezErrorCheck(adi_dev_Control(DriverHandle, ADI_UART_CMD_AUTOBAUD, NULL));

// give the device the inbound buffer
ezErrorCheck(adi_dev_Read(DriverHandle, ADI_DEV_1D,
             (ADI_DEV_BUFFER *)&InboundBuffer));

// enable data flow
ezErrorCheck(adi_dev_Control(DriverHandle, ADI_DEV_CMD_SET_DATAFLOW,
             (void *)TRUE));



// wait until the last push button is pressed
while (ezIsButtonPushed(EZ_LAST_BUTTON) == FALSE) ;

// close down the device driver
```

```
ezErrorCheck(adi_dev_Close(DriverHandle));

// close the Device Manager
ezErrorCheck(adi_dev_Terminate(DeviceManagerHandle));

// close down the Deferred Callback Manager
#if defined(USE_DEFERRED_CALLBACKS)
ezErrorCheck(adi_dcb_Terminate());
#endif

// turn off all LEDs
ezTurnOffAllLEDs();

// return
}
```

6.0.5 Código testado para μCLinux

Código de simples multiplicação de dois vetores para testar a capacidade da placa para processamento de sinais. Este cdigo usa tanto as memorias internas e externas para comparação da velocidade de processamento.

```
#include <stdio.h>

void test1();
void test2();
void test3();
void test4();

#define N 100 /* tamanho do vetor                          */
#define M 10  /* numero de vezes que é repetido o testes*/

int before;   /* contador de ciclos antes do loop */
int after;    /* contador de ciclos depois do loop  */

int main() {
  printf("Theoretical best case is N/2 = %d cycles\n", N/2);
  test1();
```

```
  test2();
  test3();
  test4();
  return 0;
}

/* Multiplicação de dois vetores, versão C */

int dot(short *x, short *y, int len)
{
   int i,dot;

   dot = 0;
   for(i=0; i<len; i++)
     dot += x[i] * y[i];

   return dot;
}

/* Multiplicação de dois vetores */

int dot1(short *x, short *y, int len)
{
   int dot;

      __asm__
   (
   "I0 = %3;\n\t"
   "I1 = %4;\n\t"
   "A1 = A0 = 0;\n\t"
   "R0 = [I0++] || R1 = [I1++];\n\t"
   "LOOP dot%= LC0 = %5 >> 1;\n\t"
   "LOOP_BEGIN dot%=;\n\t"
   "A1 += R0.H * R1.H,A0+=R0.L*R1.L||R0=[I0++]||R1=[I1++];\n\t"
   "LOOP_END dot%=;\n\t"
   "R0 = (A0 += A1);\n\t"
   "%0 = R0 >> 1;\n\t"
/* Correção com shift á esquerda durante multiplicação */
   : "=&d" (dot), "=&d" (before), "=&d" (after)
```

```
   : "a" (x), "a" (y), "a" (len)
   : "I0", "I1", "A1", "A0", "R0", "R1"
   );

   return dot;
}

/* Multiplicação de dois pontos medido em ciclos */

int dot2(short *x, short *y, int len)
{
   int dot;

   __asm__
   (
   "I0 = %3;\n\t"
   "I1 = %4;\n\t"
   "A1 = A0 = 0;\n\t"
   "R0 = [I0++] || R1 = [I1++];\n\t"
   "R2 = CYCLES;\n\t"
   "%1 = R2;\n\t"
   "LOOP dot%= LC0 = %5 >> 1;\n\t"
   "LOOP_BEGIN dot%=;\n\t"
      "A1+=R0.H*R1.H,A0+=R0.L*R1.L||R0=[I0++]||R1=[I1++];\n\t"
   "LOOP_END dot%=;\n\t"
   "R2 = CYCLES;\n\t"
   "%2 = R2;\n\t"
   "R0 = (A0 += A1);\n\t"
   "%0 = R0 >> 1;\n\t"
 /* Correção com shift á esquerda durante multiplicação */
   : "=&d" (dot), "=&d" (before), "=&d" (after)
   : "a" (x), "a" (y), "a" (len)
   : "I0", "I1", "A1", "A0", "R0", "R1", "R2"
   );

   return dot;
}

/* Função C que retorna o valor do registrador CYCLES */
```

```
int cycles() {
  int ret;

   __asm__ __volatile__
   (
   "%0 = CYCLES;\n\t"
   : "=&d" (ret)
   :
   : "R1"
   );

   return ret;
}

void test1() {
  short        x[N];
  short        y[N];
  int          i,k, ret;
  unsigned int before, after;
  unsigned int time[M];

  for(i=0; i<N; i++) {
    x[i] = 1;
    y[i] = 1;
  }

  for(k=0; k<M; k++) {
    before = cycles();
    ret = dot(x, y, N);
    after = cycles();
    time[k] = after - before;
  }

  printf("Test 1: Vanilla C\n");
  printf("  ret = %d: run time: ",ret);
  for(k=0; k<M; k++)
    printf("%u ", time[k]);
  printf("\n");
```

```
}

void test2() {
  short        x[N];
  short        y[N];
  int          i,k, ret;
  unsigned int before, after;
  unsigned int time[M];

  for(i=0; i<N; i++) {
    x[i] = 1;
    y[i] = 1;
  }

  for(k=0; k<M; k++) {
    before = cycles();
    ret = dot1(x, y, N);
    after = cycles();
    time[k] = after - before;
  }

  printf("Test 2: data in external memory, outboard cycles function\n");
  printf("  ret = %d: run time: ",ret);
  for(k=0; k<M; k++)
    printf("%u ", time[k]);
  printf("\n");
}

void test3() {
  short        x[N];
  short        y[N];
  int          i,k, ret;
  unsigned int time[M];

  for(i=0; i<N; i++) {
    x[i] = 1;
    y[i] = 1;
  }
```

```
  for(k=0; k<M; k++) {
    ret = dot2(x, y, N);
    time[k] = after - before;
  }

  printf("Test 3: data in external memory, inboard cycles\n");
  printf("  ret = %d: run time: ",ret);
  for(k=0; k<M; k++)
    printf("%u ", time[k]);
  printf("\n");
}

void test4() {
    short       *x=(short*)0xff904000 - N*sizeof(short);
                            /* Top of Data B SRAM */
  short       *y=(short*)0xff804000 - N*sizeof(short);
                              /* Top of Data A SRAM */

  int          i,k, ret;
  unsigned int time[M];

  for(i=0; i<N; i++) {
    x[i] = 1;
    y[i] = 1;
  }

  for(k=0; k<M; k++) {
    ret = dot2(x, y, N);
    time[k] = after - before;
  }

  printf("Test 4: data in internal memory, inboard cycles\n");
  printf("  ret = %d: run time: ",ret);
  for(k=0; k<M; k++)
    printf("%u ", time[k]);
  printf("\n");

}
```