UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ENGENHARIA DE SÃO CARLOS

FERNANDO YAMAGUTI TAKAHASHI

REPRODUTOR DE MÚSICA BASEADO EM MICROCONTROLADOR ARM7 PARA REPRODUÇÃO DE ARQUIVOS WAV ARMAZENADOS EM CARTÃO DE MEMÓRIA

São Carlos

2012

FERNANDO YAMAGUTI TAKAHASHI

REPRODUTOR DE MÚSICA BASEADO EM MICROCONTROLADOR ARM7 PARA REPRODUÇÃO DE ARQUIVOS WAV ARMAZENADOS EM CARTÃO DE MEMÓRIA

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos, da Universidade de São Paulo

Curso de Engenharia Elétrica com ênfase em Eletrônica

ORIENTADOR: Evandro Luís Linhari Rodrigues

São Carlos

2012

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

T136r

Takahashi, Fernando Yamaguti
Reprodutor de música baseado em microcontrolador ARM7 para reprodução de arquivos WAV armazenados em cartão de memória / Fernando Yamaguti Takahashi ; orientador Evandro Luís Linhari Rodrigues. -- São Carlos, 2012.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2012.

1. ARM. 2. ARM SAM7EX-256. 3. WAV format. 4. SD. 5. PWM. I. Titulo.

FOLHA DE APROVAÇÃO

Nome: Fernando Yamaguti Takahashi

Título: "Reprodutor de Música Baseado em Microcontrolador para Reprodução de Arquivos WAV Armazenados em Cartão de Memória"

Trabalho de Conclusão de Curso defendido e aprovado em 09 / 02 / 2012,

com NOTA 5,0 (cinco, zero), pela comissão julgadora:

Prof. Dr. Maximiliam Luppe - EESC/USP

Prof. Dr. Marcelo Andrade da Costa Vieira - EESC/USP

Prof. Associado Homero Schiabel
Coordenador da CoC-Engenharia Elétrica
EESC/USP

Dedicatória

Dedico este trabalho aos meus pais, Heigi e Glória, por sempre dedicarem uma parte de suas vidas para a minha formação, por apoiar em todas as decisões que tive até aqui e por motivar em todos os momentos difíceis. Obrigado por tudo até hoje, pelos ensinamentos, pelos momentos alegres e pelo companheirismo.

À minha namorada, Mariana, que esteve ao meu lado em todo o período de faculdade, e sempre permitindo que eu focasse em primeiro lugar os estudos. Obrigado por estar sempre presente, tantos nos momentos felizes como nos infelizes.

Agradecimentos

Agradeço a Deus por todas as realizações até hoje.

À toda a minha família, em especial aos meus irmãos, Marcelo e Petiula, e aos meus tios, Célia e Walter, por todo o apoio e incentivo antes e durante a graduação.

À família da minha namorada, por me acolher durante estes cinco anos, tonando-se uma segunda família.

Ao meu professor orientador, Evandro L.L. Rodrigues, por ser mais que um professor, ser um amigo. Por sempre orientar e dar dicas pra o meu futuro; e sempre estar aberto para conversas.

Aos amigos conquistados durante a faculdade. Em especial; Shamir, José e Márcio, pelo companheirismo nas noites na república e nas partidas de futebol; a Gabriel Silva e João, pelo companheirismo nas horas de estudos e laboratórios; a Bruno, Gabriel Stein, Igor e João Pedro por momentos engraçados e alegres fora da faculdade. Espero que todas estas amizades sejam duradouras independentemente da distância. E que todos sejam felizes no caminho que decidirem.

Resumo

Neste Trabalho de Conclusão de Curso foi desenvolvido um reprodutor de música em formato WAV. O projeto foi obtido utilizando-se como base o kit de ensino SAM7-EX256, produzido pela empresa Olimex, que utiliza-se um microcontrolador ARM7TDMI com 256*KBytes* de memória de programa e 64*Kbytes* de SRAM. Como resultado, obteve-se um reprodutor de música capaz de reproduzir qualquer arquivo de formato WAV, através da leitura do arquivo em um cartão de memória do tipo SD ou MMC, além de poder controlar o volume da reprodução, atrasar ou avançar para a próxima faixa, pausar e reproduzir a faixa.

Palavras chaves: ARM, ARM SAM7EX-256, WAV *format,* SD, PWM.

Abstract

In this Course Conclusion Paper it was developed a music player in WAV format. The project was conceived using the SAM7EX-256 development board, produced by Oimex, that uses a microcontrolller ARM7TDMI with 256kbytes of memory program and 64kbytes of SRAM. The final result was a music player capable of reproducing any WAV file format, by reading the file on a memory card like, SD or MMC, and controlling the music playback.

Key words: ARM, ARM SAM7EX-256, WAV format, SD, PWM.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Abreviaturas

ARM : *Advanced RISC Machine*

MP3 : *MPEG Audio Layer 3*

MPEG : *Moving Picture Experts Group*

WAV ou WAVE : *Waveform Audio file*

PCX : *Personal Computer eXchange*

DIB : *Device-Independent Bitmap*

SPI : *Serial Peripheral Interface*

PWM : *Pulse-width modulation*

SD : *Secure Digital* Card

MMC : *Multi Media Card*

SDHC : *Secure Digital High Capacity*

SDXC : *Secure Digital Extended Capacity*

ADC : *analog-to-digital converter*

USB : *Universal Serial Bus*

PCM : *Pulse-code modulation*

MBR : *Master Boot Record*

FAT : *File Allocation Table*

SS : *Slave Select*

CS : *Chip Select*

MOSI : *Master Out Slave In*

MISO : *Master In Slave Out*

SPCK : *Serial Clock*

NSS : *Slave Select*

PMC : *Power Management Controller*

AIC : *Advanced Interrupt Controller*

IDE *: Integrated Development Environment*

JVM : *Java Virtual Machine*

CPU : central processing unit

RS232 : *Recommended Standard 232*

DC : *Direct Current*

CAPÍTULO 1

1.1 Introdução

A música é uma forma de arte que se constitui basicamente em combinar sons e silêncio seguindo uma pré-organização ao longo do tempo.

É considerada por diversos autores como uma prática cultural e humana. Atualmente não se conhece nenhuma civilização ou agrupamento que não possua manifestações musicais próprias. Embora nem sempre seja feita com esse objetivo, a música pode ser considerada como uma forma de arte, considerada por muitos como sua principal função (Wikipedia, 2011).

O mundo da música é um dos muitos em que a eletrônica vem se desenvolvendo e ganhando cada dia mais importância. È difícil imaginar qualquer produção musica sendo concebida sem a utilização de algum instrumento ou alguma forma de captação e reprodução que não utilize sequer algum componente elétrico ou eletrônico.

Um tocador de música portátil utiliza-se a eletrônica para armazenar dados de sinal sonoro em forma digital, e para realizar a conversão e reprodução destes dados digitalizados para a forma analógica.

Atualmente existe centenas de tipos de tocadores de música portátil, como por exemplo Ipods (da empresa Apple) e Walkman (da empresa Sony), capazes de reproduzir sons armazenados de diferentes formatos (WAV, Mp3, MP4, WMV). E é neste tipo de produto que está baseado o protótipo produzido neste trabalho de conclusão de curso. O funcionamento do projeto é baseado nos comandos básicos encontrado em qualquer tipo de tocador de música portátil.

A utilização de microcontroladores para esta aplicação é uma alternativa muito eficiente, já que, a partir dele, é possível processar o sinal de áudio, armazenar dados e controlar os periféricos do sistema.

1.2 Objetivos

O objetivo geral deste Trabalho de Conclusão de Curso foi de desenvolver um projeto baseado em microcontrolador para obter um reprodutor de música portátil. O projeto é baseado nos reprodutores de músicas atuais, como por exemplo, *MP3 Players* e *Ipods*, e visa produdizr um produto semelhante a estes, nos quais o usuário consegue controlar a reprodução de músicas armazenadas em um cartão de memória, porém utilizando outro formato de arquivo de aúdio (WAV).

Para o desenvolvimento deste projeto, foram abordados os seguintes tópicos:

1. Estudo do kit de desenvolvimento SAM7-EX256.
2. Estudo do protocolo SPI para a comunicação com o cartão de memória.
3. Geração de sinal de áudio a partir do PWM do microcontrolador.
4. Estudo para decodificar e manipular arquivos de formato WAV.
5. Configurar interrupção por *timer* presente no microcontrolador.
6. Desenvolvimento do *software* para decodificar e reproduzir os arquivos de formato WAV.

CAPÍTULO 2

2.1 Microcontrolador ARM

Os microncontroladores ARM (*Advanced RISC Machine*) são uma família de microcontroladores que possuem um núcleo baseados na arquiterura RISC (*Reduced Instruction Set Computer*) de 32 *bits.* Esta Arquitetura tem como característica uma estrutura de instruções mais simplificada, que tem como objetivo otimizar a velocidade de execução do microcontrolador. Além das instruções de 32*bits* da arquitetura ARM, a partir da família ARM7TDMI foi criado um set de instruções de 16 *bits* denominado *Thumb.* Estas instruções, evidementemente, são mais simplificadas e possuem algumas limitações em relação às instruções de 32*bits,* no entanto reduzem consideravelmente o espaço de memória consumido pelo código do programa. Posteriormente, a partir da família ARM1156 foi introduzido o *Thumb-2,* que complementa algumas limitações do seu predecessor através da introdução de algumas instruções de 32bits. Dessa forma, o *Thumb-2* procura o equilíbrio entre o desempenho do conjunto de instruções ARM de 32*bits* e a densidade de código do Thumb. Outra característica importante dos microcontroladores ARM é o baixo consumo de energia, fazendo com que sejam amplamente utilizados em dispositivos móveis e em circuitos embarcados (PEREIRA, 2007).

A elaboração da arquitetura ARM foi feito pela empresa Acorn (inicialmente ARM era a sigla para *Acorn RISC Machine*). Posteriormente foi identificado o potencial dessa tecnologia e outras empresas tornaram-se parceiras no seu desenvolvimento, entre elas a Apple e a VLSI e foi criada a ARM *Holding,* detentora da licença desta arquitetura. Atualmente diversos fabricantes de semicondutores oferecem microcontroladores baseados nesta tecnologia, entre eles Atmel, Texas Instrument, Freescale, STMicroeletronics e NXP Semiconductors (Wikipedia, 2011).

Dentre os microcontroladores ARM existem diversas famílias que possuem algumas características específicas, dependendo da geração. Dentre todas as famílias ARM, as mais populares são ARM7, ARM9, ARM11 e Cortex, sendo a última, amplamente utilizado em aparelhos portáteis, como *tablets* e *smartphones* (ARM ltd., 2011).

Uma das características que diferenciam estas gerações é a quantidade de estágios de *pipeline*. O *Pipeline* representa a quantidade de instruções que o CPU pode deixar na "fila" para as execuções. Isso significa que não é necessário aguardar todos os ciclos de execução de uma instrução para que a próxima seja iniciada. Na prática, esse processo otimiza a velocidade de execução do programa, diminuindo a quantidade de ciclos de clock entre uma instrução e outra. No ARM7 existem três estágios de *pipeline* (Atmel *Corporation, 2009*), enquanto que nas famílias ARM9, ARM11 e Cortex existem, respectivamente, cinco, oito e onze estágios.

A versatilidade e a eficiência dos microcontroladores da família ARM fez com esta seja uma das arquiteturas mais bem sucedidas no mercado atualmente. Ela é utilizada em inúmeras aplicações que exigem desde um baixo consumo de potência até um alto desempenho de processamento.

2.2 Kit Olimex SAM7EX-256

O kit de desenvolvimento utilizado neste projeto foi o SAM7EX-256 da empresa Olimex, utiliza como processador central a CPU ATMEL AT91SAM7EX256, um processador ARM7 de arquitetura RISC de 32 bits, com 256 *Kbytes* de memória de programa e 64*Kbytes* de memória SRAM.

O kit possui como periféricos (Olimex Ltd, 2011):

- Conector JTAG padrão com o ARM (do tipo 2x10) para programação ou *debug*;
- Display LCD NOKIA 6610 128x128 TFT de 12-bits colorido com *backlight*;
- Ethernet 10/100 PHY com KS8721BL;
- Conector de USB;
- canais de interface e *drivers* de RS232;
- Conector para cartão SD e MMC;
- Joystick de 4 direções e "*push action*";
- botões;
- Plugue de entrada e saída de áudio para microfone e fone de ouvido;
- Alto-falante *on board*, com um potenciômetro para controle do volume;
- Potenciômetro conectado ao ADC;
- Termistor conectado ao ADC;

- Regulador de tensão de 3.3V com corrente de até 800mA;
- Fonte de alimentação simples de 6V AC ou DC, pode ser alimentado pela porta USB;
- Led da fonte de alimentação;
- Filtro capacitivo da fonte de alimentação;
- Botão e circuito de *reset*;

Dentre os periféricos listados, os que tiveram maior importância para implementação deste projeto foram Conector para Cartão SD e *Joystick*.

Abaixo encontra-se uma figura contendo os periféricos principais para o projeto:

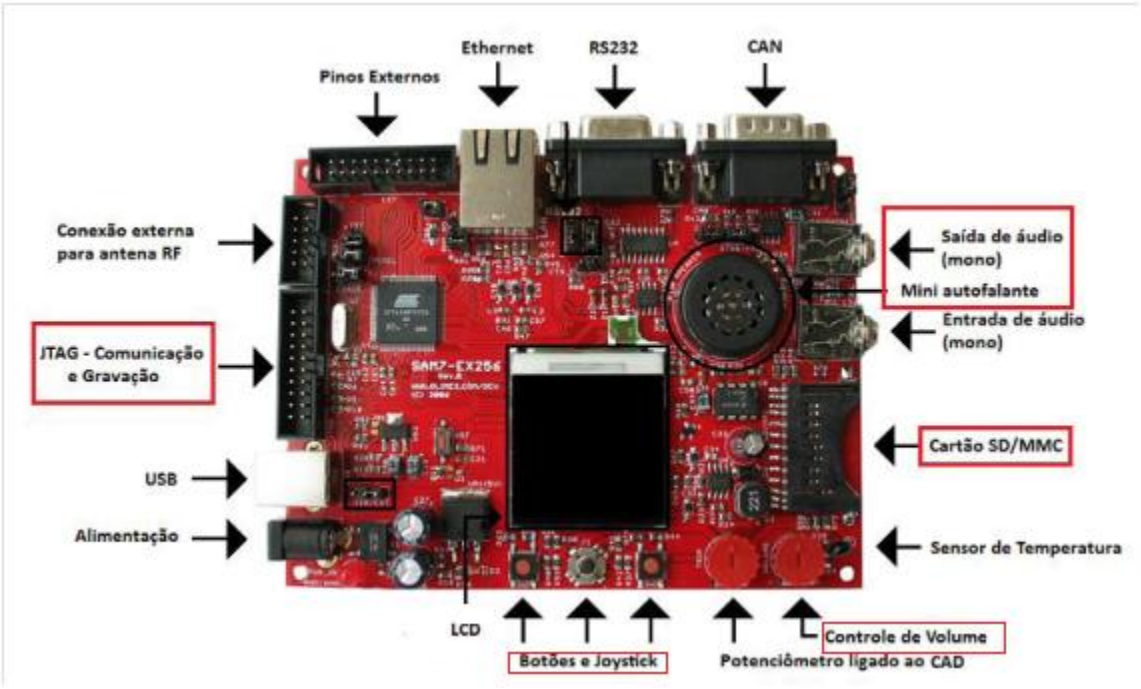

Figura 1 - Kit Olimex SAM7EX-256

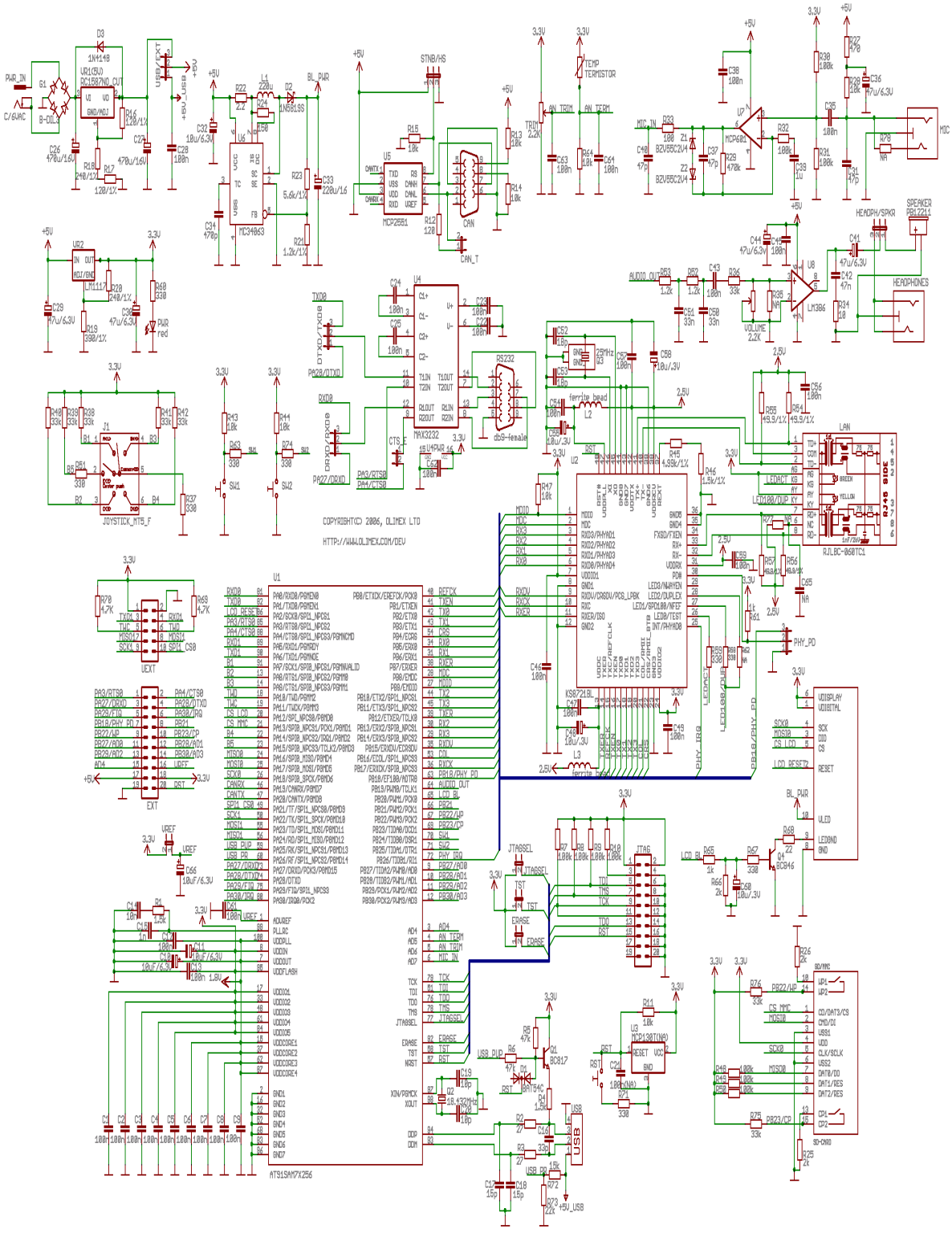

Figura 2 - Esquemático do kit SAM7EX-256

Circuito para saída de áudio:

A Saída do PWM está ligada à entrada do amplificador de áudio, como visto na figura abaixo, através do pino denominado AUDIO_OUT. É possível observar neste circuito a presença de um potenciômetro para o controle de volume e duas saídas, um *HEADPHONE* e outro *SPEAKER.* A escolha por qual saída utilizar é feita através do jumper denominado como HEADPH/SPKR.

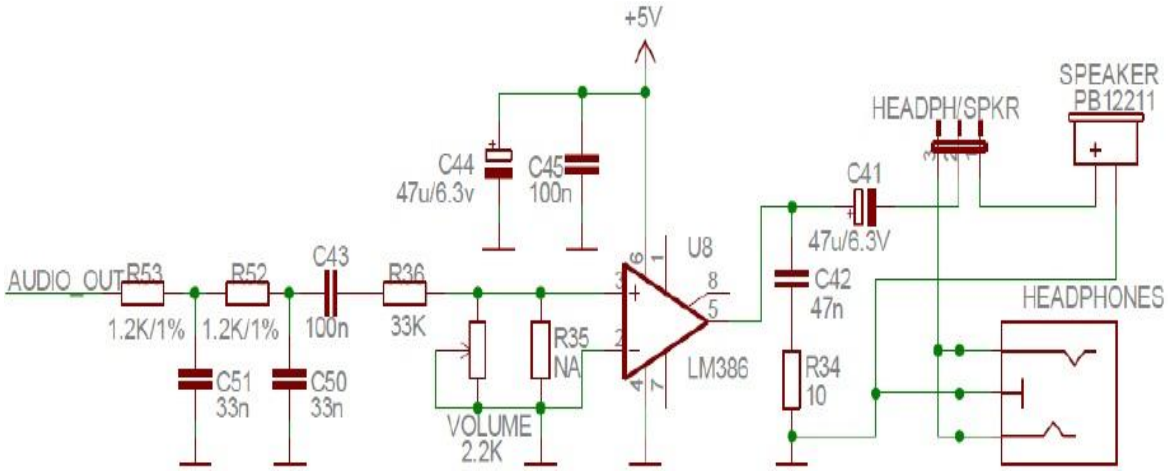

Figura 3 - Esquemático da Saída de áudio do kit

Circuito para *joystick:*

Na figura abaixo é possível observar o *joystick* e os botões liga/desliga. Nota-se que cada posição do *joystick* e que cada botão está ligado a uma porta distinta do microcontrolador e a ativação se dá em nível baixo, ou seja, se os botões não estão pressionados o microcontrolador lê a entrada em nível alto.

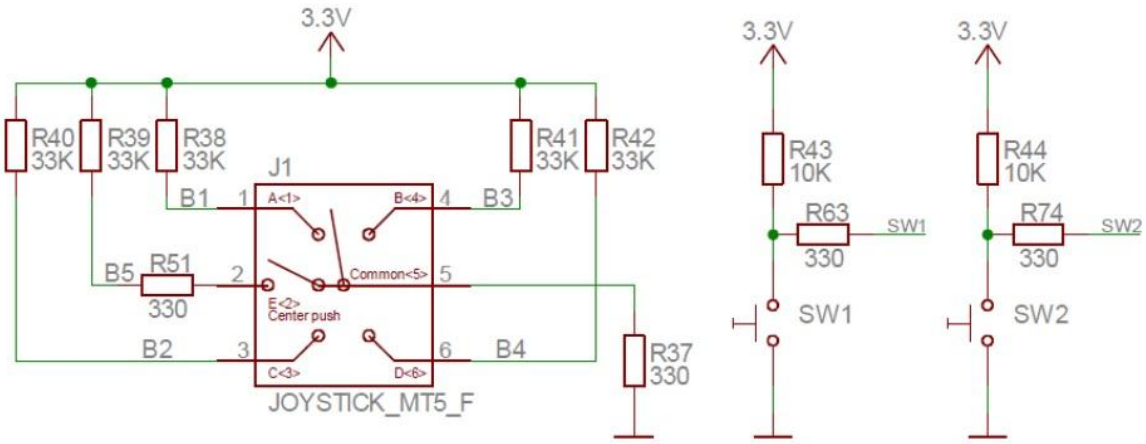

Figura 4 - Esquemático do joytisck

Circuito para o Cartão SD:

O pino 1 (CD/DAT3/CS) é utilizado para detecção do cartão e como terceiro sinal do barramento de transmissão de dados (DAT). O pino 2 (CMD/DI) é utilizado para envio de comandos (do kit para o cartão) e respostas (do cartão para o kit). Os pinos 7, 8 e 9 fazem parte do barramento de dados DAT.

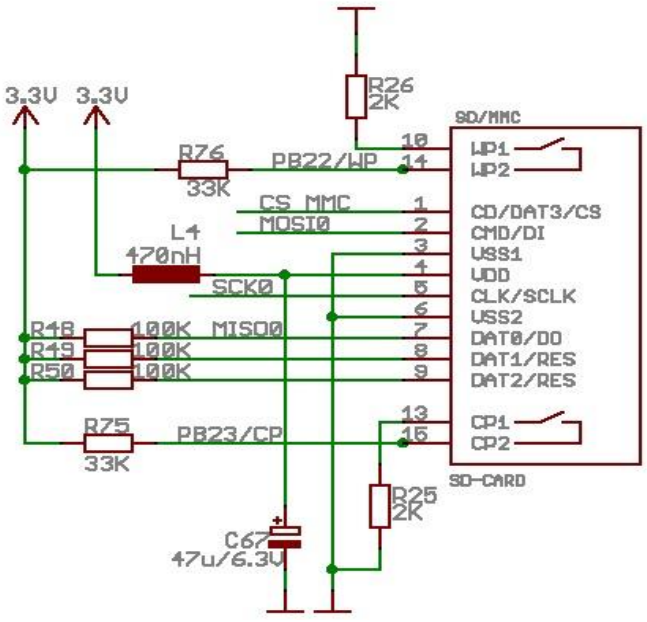

Figura 5 - Esquemático da entrada do cartão MMC/SD

2.3 Protocolo de comunicação SPI

A comunicação SPI (*Serial Peripheral Interface*) é um link de dados síncrono que possibilita a comunicação entre dispositivos. A interface SPI é essencialmente um registrador que transmite serialmente dados para outras interfaces SPI. Durante a transferência de dados, um sistema SPI se comporta como *Master* e pode transmitir dados para outros sistemas SPI, denominados *Slaves*.

Um sistema SPI consiste de quatro linhas de dados principais:

- ***Master Out Slave In* (MOSI):** linha de dado que interconecta a saída do registrador *Master* e a(s) entrada(s) do(s) registrador(es) *Slave*(s).
- ***Master In Slave Out* (MISO):** linha de dado que interconecta a entrada do registrador *Master* e a(s) saída(s) do(s) registrador(es) *Slave*(s).
- ***Serial Clock* (SPCK):** linha de controle gerada pelo *Master* e dirigida até *Slave* a fim de regular a transmissão dos bits.
- ***Slave Select* (NSS):** linha de controle que permite a ativação (ou não) por *hardware* dos *Slaves*.

Na figura abaixo é visualizado a distribuição dos pinos entre os dispositivos *master* e *slave*, no caso, três *slaves*.

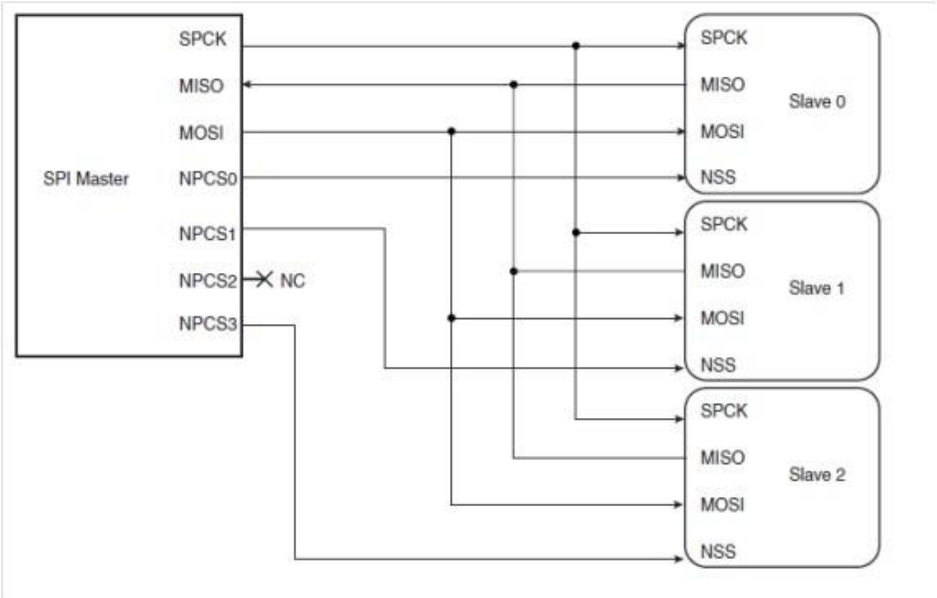

Figura 6 - Conexão *Master/Slave*

Para se trabalhar com a comunicação SPI, primeiramente existem três características principais a serem analisadas e programadas conforme a necessidade do usuário:

- Linhas de I/O: Os pinos utilizados podem ser multiplexados com as linhas das portas paralelas de I/O. O programador deve primeiro programar estas linhas para serem utilizadas como função SPI.
- Gerenciamento de Energia: O SPI pode ser alimentado pelo controlador PMC *(Power Management Controller*), assim este deve ser configurado antes de habilitar o *clock* SPI.
- Interrupção: A Interface SPI tem uma linha de interrupção conectada ao AIC (*Advanced Interrupt Controller*). Assim, para gerenciar a interrupção SPI exige-se a programação do AIC antes de utilizar a SPI.

Em se tratando de transferência de dados na comunicação SPI deve-se levar em conta a polaridade e as fases disponíveis na transmissão de dados. O clock, por exemplo, tem sua polaridade programada pelo bit CPOL e sua fase programada pelo bit NCPHA. Como cada um desses dois parâmetros possui dois possíveis estados, ao todo acabam existindo quatro combinações, assim, para cada *master*/*slave* uma combinação deve ser adotada.

Analisando a descrição funcional de um dispositivo SPI, observa-se dois tipos distintos de operação: modo master e modo *slave*. De forma resumida a operação em modo *Master* é programado escrevendo '1' no bit MSTR no registrador *Mode Register*. Os pinos NPCS0 a NPCS3 são todos configurados como saída e os pinos MISO torna-se entrada ao passo que MOSI torna-se saída do transmissor. Caso contrário, se o bit MSTR é escrito como '0' então o dispositivo fica em modo *Slave*.

Operação no Modo *Master* :

- *clock* operado pelo gerador de *baud rate*, este controla os dados transferidos do dispositivo *Slave* conectado no SPI *bus*.
- a transferência inicia quando o controlador escreve algum dado no SPI_TDR (*Transmit Data Register*), dado este transferido para o para o *Shift Register*

(ShiftR), ação indicada pelo bit TDRE (*Transmit Data Register Empty*) no *Registrador Stats Register* (SPI_SR), e alocado na linha MOSI.

- o bit TDRE indica também se um novo dado pode ser carregado no SPI_TDR, caso isso ocorra durante uma transmissão, o dado, obrigatoriamente, se mantém em espera até o encerramento da mesma.
- o final de uma transferência é indicado pela flag TXEMPTY.
- a transferência de um dado a partir do *Receive Data Register* (SPI_RDR) para o SPI_RDR é indicado pelo bit RDRF *(Receive Data Register Full*) no registrador SPI_SR, bit é zerado quando o dado recebido é lido.

Abaixo, pode ser visto o diagrama de bloco quando um dispositivo atua no modo *Master*.

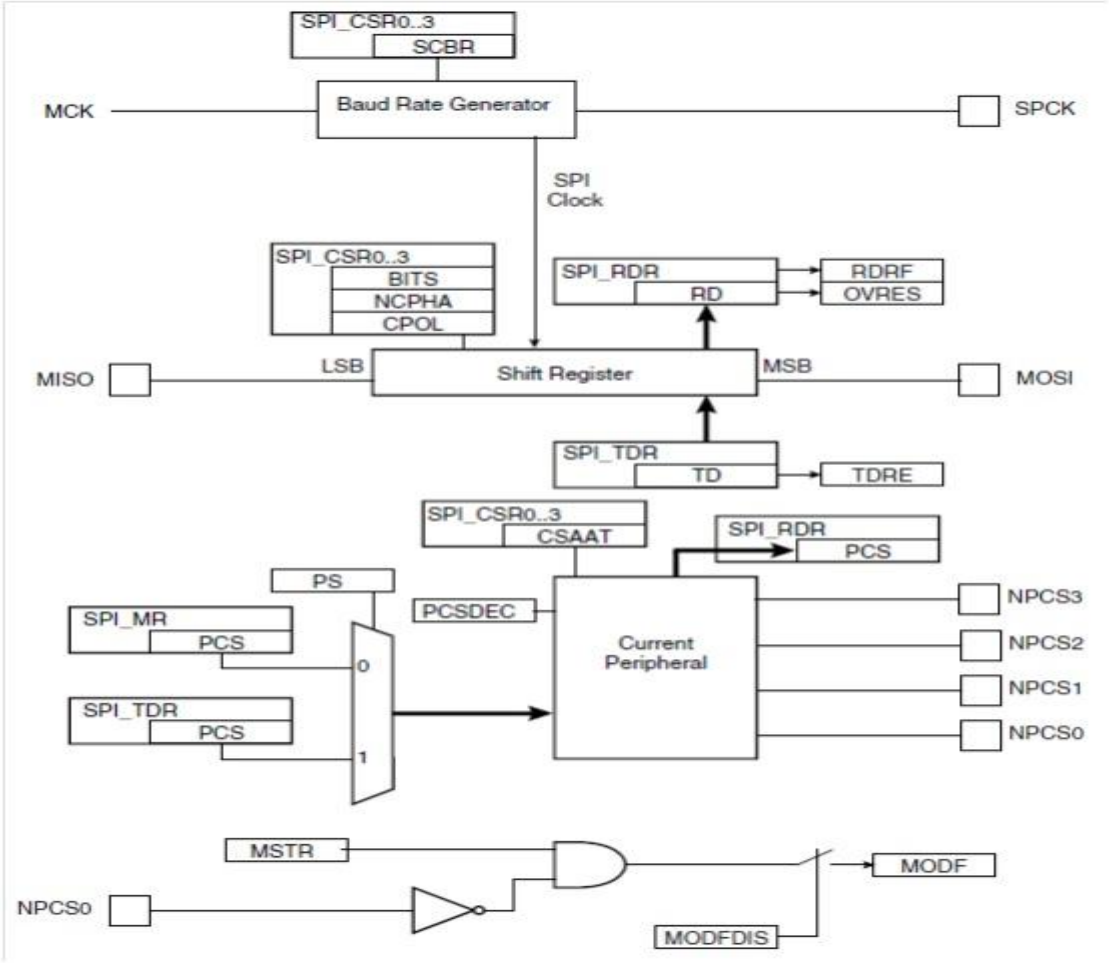

Figura 7 - Operação no modo *Master*

Operação no Modo *Slave* :

- os bits dos dados são processados de acordo com o *clock* fornecido pelo pino SPCK.
- o pino NSS precisa ser setado em 0 para ocorrer a seleção do dispositivo a o reconhecimento do clock vindo do *Master*, processando o número de bits definidos pelo campo BITS do *Chip Select Register* (SPI_CSR0).
- após os bits terem sido processados, o dado recebido é transmitido para o SPI_TRD e o bit RDRF é setado em 1.
- os registrados SPI_TDR, SPI_RDR e o *Shift Register* é tem atuação semelhante ao que ocorre no modo *Master*.

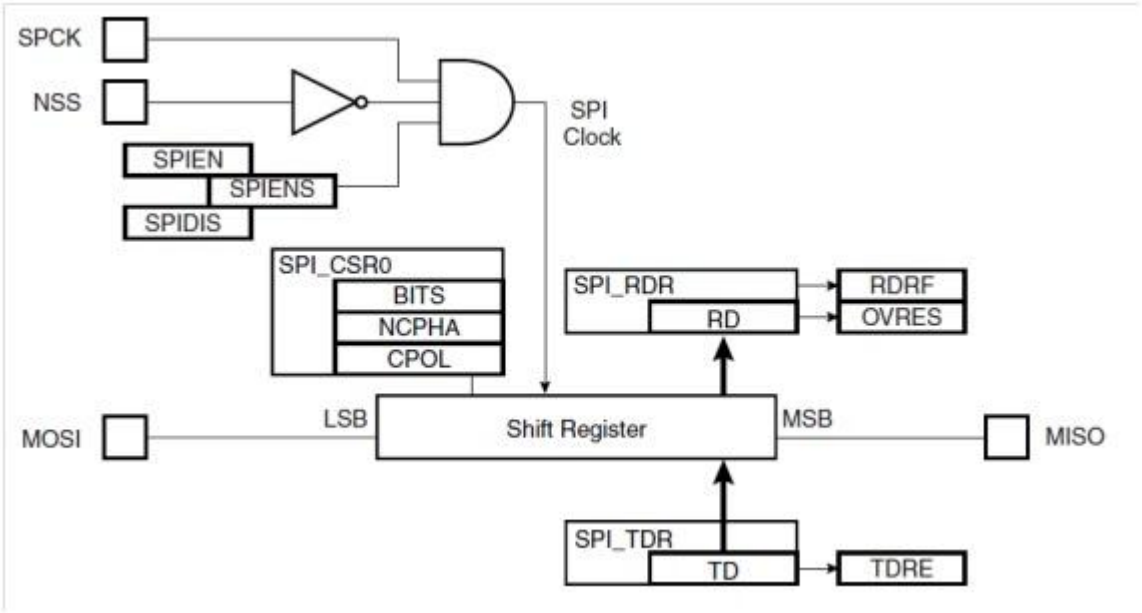

Figura 8 - Operação no modo *Slave*

2.4 Pulse Width Modulation (PWM)

PWM ou Modulação por largura de Pulso é um técnica para controlar a potência de um sinal transmitido através de uma série de pulsos com ciclo de trabalho (*duty cycle*) variável. O valor de tensão médio fornecido a uma carga é controlado por uma chave eletrônica que liga e desliga rapidamente. Quanto maior o tempo que a chave está ligada, maior é o ciclo de trabalho da onda e maior é a potência fornecida pela fonte de

alimentação. Assim, a potência média fornecida à carga é regulada pelo *duty cycle* do PWM (SEDRA, et al., 2000).

No AT91SAM7EX256 é possível programar tanto o período quanto o *duty cycle* do PWM e é possível determinar a polaridade do sinal de saída, ou seja, ele pode ser iniciado tanto em nível lógico 1 quanto em nível 0.

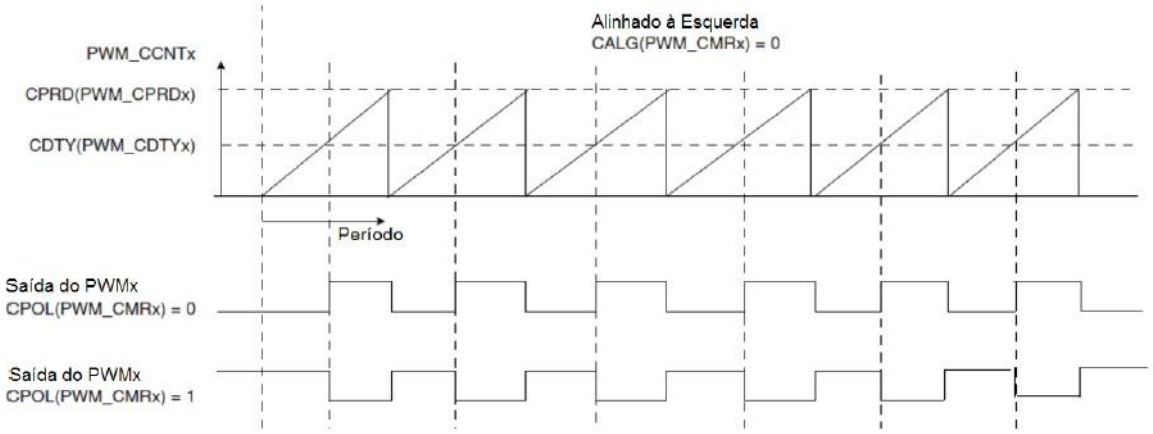

Figura 9 - Ciclos do PWM

Na figura abaixo é possível visualizar o diagrama de blocos da estrutura do PWM presente no microcontrolador.

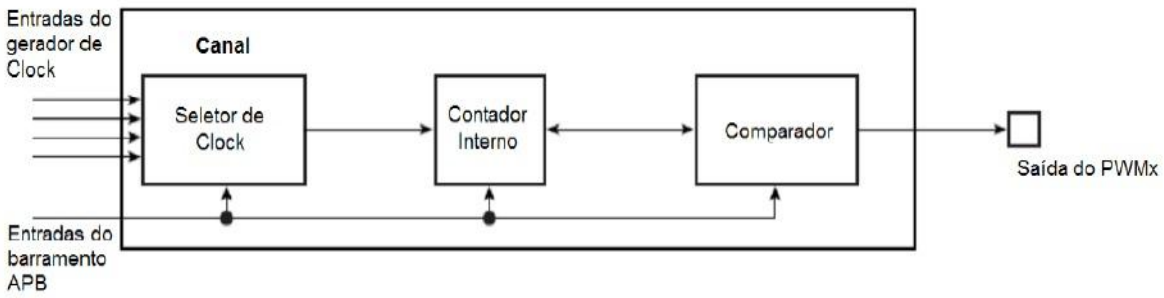

Figura 10 - Diagrama de blocos do PWM do microcontrolador

2.4 Cartão SD (*Secure Digital)*

Os cartões de memória *Secure Digital Card* ou SD *Card* são uma evolução da tecnologia *MultiMediaCard* (ou MMC, memória *flash* do tipo EEPROM desenvolvida pelas empresas Sandisk e Siemens AG). Adicionam capacidades de criptografia e gestão de direitos digitais (daí o *Secure*), para atender às exigências da indústria da música e uma trava para impedir alterações ou apagamento do conteúdo do cartão, assim como os disquetes de 3½".

Tornou-se o padrão de cartão de memória com melhor custo/benefício do mercado (ao lado do *Memory Stick*), desbancando o concorrente *Compact Flash*, devido a sua popularidade e portabilidade, e conta já com a adesão de grandes fabricantes como Canon, Kodak e Nikon que anteriormente utilizavam exclusivamente o padrão CF (sendo que seguem usando o CF apenas em suas câmeras profissionais). Além disso, está presente também em palmtops, celulares (nos modelos MiniSD, MicroSD e *Transflash*), sintetizadores MIDI, tocadores de MP3 portáteis e até em aparelhos de som automotivo.

Figura 11 - Cartão SD *(Secure Digital)*

Como qualquer outra tecnologia de memória flash, a maioria dos cartões SD são pré-formatado com um sistema de arquivo em cima de um MBR (*Master Boot Record*), esquema de partição. Os cartões SD são tipicamente formatados como FAT16, SDHC cartões como FAT32 , os cartões SDXC como exFAT . A utilização de FAT16 e FAT32 permite que os cartões sejam acessados em praticamente qualquer máquina com um dispositivo leitor de SD. Além disso, utilitários de manutenção padrão FAT (por exemplo, SCANDISK) pode ser usado para reparar ou recuperar dados corrompidos, e alguns

utilitários pode recuperar arquivos apagados, desde que não tenham sido substituídos. No entanto, porque o cartão aparece como um disco rígido removível para o sistema host, o cartão pode ser reformatado para qualquer sistema de arquivos suportados pelo sistema operacional.

Os cartões SD são simples dispositivos de bloco e não implica de modo algum qualquer layout de partição específica ou sistema de arquivos, ou seja, pode ser utilizado qualquer outro tipo de particionamento que não seja MBR ou sistemas de arquivos FAT.

Os cartões SD podem se comunicar em dois modos diferentes, dependendo da sua capacidade, são eles: SPI *Mode* (neste modo o cartão SD funciona basicamente igual ao cartão MMC (*Multi Media Car*d)), e *SD Mode*. O SPI *Mode* é obrigatório para todas as famílias SD.

- SPI: O SPI Mode comunica-se através de uma Interface Periférica Serial (SPI – *Serial Peripheral Interface*), que permite que vários dispositivos troquem dados em *full-duplex*. O SPI é um protocolo síncrono que permite um dispositivo mestre (*master*) iniciar a comunicação com um escravo (*slave*). Para prover o sincronismo, o sinal de *clock* (pino SCK) pode ser gerado somente pelo mestre, e este sinal controla quando os dados podem mudar e quando são válidos para leitura. Por ser síncrono, esse protocolo torna-se interessante para ser usado em um oscilador não tão estável, como um oscilador RC, pois a taxa de transferência varia apenas de acordo com as subidas e descidas do oscilador, não afetando a integridade dos dados. Por permitir vários escravos, o SPI precisa controlar qual deles está sendo acessado, e o faz com um sinal CS (*Chip Select*) ou SS (*Slave Select*), que não precisa ser conectado caso haja apenas um escravo[3].
- SD *Mode*: Utiliza pinos extras além de alguns pinos realocados . O modo SD é o modo padrão de transferência do cartão SD, onde o cartão transfere 4 bits por ciclo, a uma freqüência de até 50MHz, resultando em taxas de transferência de até 25MB/s (desde que os *chips* de memória usados acompanhem, naturalmente), (Carlos E. Morimoto, 2007)

As figuras a seguir mostram a diferença entre as pinagens para cada modo de transferência.

SD Mode

Pin	Name	Dir	Description
1	CD/DAT3	↔ PP)	Card Detect/Data Line 3
2	CMD	↔ PP)	Command Line
3	VSS1	—	Ground
4	Vdd	←	Voltage Supply [2.7v or 3.6v]
5	Clock	←	Clock
6	Vss2	—	Ground
7	DAT0	↔ PP)	Data Line 0
8	DAT1	↔ PP)	Data Line 1
9	DAT2	↔ PP)	Data Line 2

Note: Direction is memory card relative to host
PP) Output is Push-pull.

Figura 12 - Pinagem Cartão SD em SD *Mode*

SPI Mode

Pin	Name	Dir	Description
1	/CS	←	Chip Select/Slave Select (SS)
2	DI	← PP)	Master Out/Slave In (MOSI)
3	VSS	—	Supply voltage ground
4	VDD	←	Supply voltage
5	CLK	←	Clock (SCK)
6	VSS	—	Supply voltage ground
7	DO	↔ PP)	Master In/Slave Out (MISO)
8	IRQ		not connected or IRQ
9	NC		not connected

Note: Direction is memory card relative to host
PP) Push-pull.

Figura 13 - Pinagem Cartão SD em SPI *Mode*

2.5 Arquivo de Áudio formato WAV

O formato WAV (ou **WAVE**), forma curta de *WAVE form audio format,* é um formato-padrão de arquivo de áudio da Microsoft e IBM para armazenamento de áudio em PCs.

Apesar de um arquivo WAV poder conter áudio compactado, o formato mais comum de WAV contém áudio em formato de modulação de pulsos PCM (*pulse-code modulation*). O PCM usa um método de armazenamento de áudio não-comprimido (sem perda). Usuários profissionais podem usar o formato WAV para qualidade máxima de áudio. Áudio WAV pode ser editado e manipulado com relativa facilidade usando softwares

Por ser um formato sem compressão, o WAV caiu em popularidade, porém se o espaço em disco não for uma restrição e houver uma necessidade de som com alta qualidade, o formato WAV é muito indicado, pois apresenta uma estrutura simples e de fácil manipulação.

Ao contrário de formatos como FLAC, WAV geralmente não têm informações adicionais, como por exemplo, o nome da música, artista, álbum, ano, etc.

Um arquivo WAV é composto por três blocos principais: Cabeçalho (*Header*) com tamanho de 12 bytes, bloco “fmt” com tamanho de 24 bytes, e bloco “data” seu tamanho depende do tamanho da música. E cada bloco é dividido em vários sub-blocos.

A seguir será apresentada a estrutura de um arquivo WAV, descrevendo detalhadamente cada bloco e seus sub-blocos.

2.5.1 Bloco (*Header*)

Bloco que identifica o arquivo como formato WAV e apresenta o tamanho do arquivo.

Posição (byte)	**Tamanho (bytes)**	**Descrição**
0	4	Apresenta o identificador do bloco *Header*, com as letras “RIFF” em ASCII
4	4	Apresenta o tamanho do arquivo, desconsiderando os bytes do ID “RIFF” e os bytes que apresenta o

		tamanho do arquivo. Ou seja para descobrir o tamanho do arquivo original basta somar ao resultado 8 bytes. Tamanho arquivo = byte[4] + 256 * byte[5] + 65536 * byte[6]+ 16777216 * byte[7] + 8.
8	4	Apresenta o identificador com as letras “WAVE” em ASCII, identificando como um arquivo WAV.

Tabela 1 - Descrição do bloco *Header*

2.5.2 Bloco “fmt” (*subchunk1*)

Bloco que descreve o formato dos dados da música

Posição (byte)	**Tamanho (bytes)**	**Descrição**
12	4	Apresenta o identificador do segundo bloco, com as letras “fmt” em ASCII.
16	4	Apresenta o tamanho do resto do bloco “fmt”. Tamanho “fmt” = byte[16] + 256 * byte[17] + 65536 * byte[18]+ 16777216 * byte[19]

20	2	Apresenta se é do tipo PCM ou qual foi a forma de compressão. Verifique a tabela 3.
22	2	Apresenta o número de canais. Se é mono = (byte[22] = 0x01 e byte[23] = 0x00) ou estéreo = (byte[22] = 0x02 e byte[23] = 0x00).
26	4	Apresenta o valor da taxa de amostragem (*Sample Rate*). SR = byte[26] + 256 * byte[27] + 65536 * byte[28]+ 16777216 * byte[29]
30	4	Apresenta a taxa de bytes (*Byte Rate*). BR = byte[30] + 256 * byte[31] + 65536 * byte[32]+ 16777216 * byte[33]
32	2	Apresenta o número de bytes para uma amostra incluindo todos os canais. (*block Align*) BA = byte[32] + 256 * byte[33]

34	2	Apresenta o número de bits para cada amostra. (*Bits Per Sample*) BPS = byte[34] + 256 * byte[35]

Tabela 2 - Descrição bloco "fmt"

Tabela com as formas de compressão utilizadas para arquivos WAV

Format Code	Forma de Compressão
0x0001	WAVE format PCM
0x0003	WAVE format IEEE float
0x0006	WAVE format ALAW
0x0007	WAVE format MULAW
0xFFFE	WAVE format EXTENSIBLE

Tabela 3 - Possibilidades de compressão WAV

2.5.3 Bloco “data” (*subchunk* 2)

Bloco que possui os dados reais da música.

Posição (byte)	Tamanho (bytes)	Descrição

36	4	Apresenta o identificador do terceiro bloco, com as letras “data” em ASCII.
40	4	Apresenta o tamanho do resto do bloco “data”. Tamanho “data” = byte[40] + 256 * byte[41] + 65536 * byte[42]+ 16777216 * byte[43]
44	Tamanho Bloco	Apresenta os dados reais da música.

Tabela 4 - Descrição bloco "data"

A figura a seguir mostra o conteúdo hexadecimal e ASCII de um arquivo WAV, com taxa de bits por amostragem igual a 8, taxa de amostragem igual 8kHz e com apenas um canal.

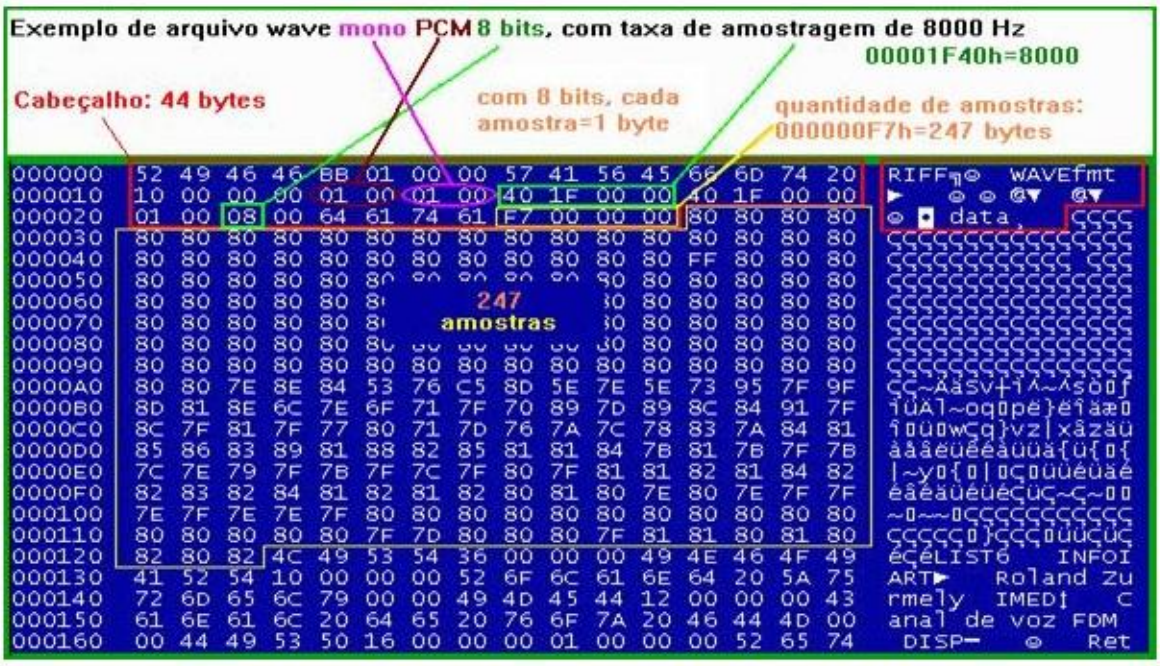

Figura 14 - Estrutua de um arquivo WAV em hexadecimal

No exemplo acima por se tratar de uma música com apenas um canal (mono), cada amostra (1*Byte*) é reproduzido a cada intervalo de tempo, definido pela taxa de amostragem. Se fosse uma música com dois canais (estéreo), seriam reproduzidas duas amostras (2bytes) a cada intervalo de tempo, a primeira amostra seria reproduzida no primeiro canal e a segunda amostra no segundo canal.

Um arquivo WAV de 8*bits* significa que o valor da amplitude do sinal, de cada amostra, pode ser representado por 256 valores, ou seja, podemos ter 127 valores positivos e 128 valores negativos. Onde FFh = 255 representa o valor máximo positivo, 00h = 0 representa o valor máximo negativo e 7Fh=128 representa o valor 0.

Já para um arquivo WAV de 16*bits*, podemos representar o valor da amplitude do sinal, de cada amostra, com 65536 valores. Tendo assim, 32767 valores positivos e 32768 de valores negativos. Diferentemente de um arquivo WAV de 8 bits, um arquivo WAV de 16bits utiliza-se a codificação complemento de 2 para representar o valor da amplitude do sinal. Ou seja, o valor do bit mais significativo representa se o sinal é positivo (bit mais significativo = 0) e negativo (bit mais significativo = 1).

- 0000h = 0, representa o valor 0
- 7FFFh = 32767, representa o valor máximo positivo
- 0001h = 1, representa o valor mínimo positivo
- 8000h = 32768, representa o valor máximo negativo
- FFFFh = 65535, representa o valor mínimo negativo.

CAPÍTULO 3 – Materiais e Métodos

3.1 Considerações Iniciais

Todo o desenvolvimento do projeto do reprodutor de música foi estruturado para o kit Olimex SAM7EX-256, utilizando as bibliotecas de código fornecidas pela própria empresa que fabrica o kit.

O desenvolvimento de programação foi realizado pela plataforma IDE Eclipse; e para comunicação entre o kit e a plataforma utilizou-se o conector JTAG.

Para realização deste projeto necessitou de três *softwares* auxiliares: Hex Editor, Dev-C e Conversor de áudio MP3 para WAV.

O Hex Editor é um *software* que possibilita editar e visualizar os dados em binário de qualquer arquivo. Este foi essencial para verificar se a decodificação dos arquivos WAV, foi realizada com sucesso.

O Dev-C, *software* de programação em linguagem C/C++, foi essencial para testar se a decodificação dos arquivos foi correta.

O Conversor de áudio, foi necessário para se obter arquivos WAV.

O sucesso e o perfeito funcionamento do projeto devem-se a utilização do *Timer* da interrupção. Esta ferramenta solucionou o problema de falta de memória durante o processo de reprodução de uma música, e a baixa velocidade de transmissão de dados entre o cartão SD e o microcontrolador. Já que o microcontrolador não possui uma memória interna capaz de armazenar dados de uma música inteira, e a velocidade de transmissão de dados não permitir que um dado da música seja buscado do cartão e seja reproduzido no intervalo de tempo correto. A interrupção fez com que conseguisse buscar dados do cartão e ao mesmo tempo reproduzi-los.

3.2 Concepção do projeto

Para o desenvolvimento deste projeto baseou-se nos funcionamentos básicos de um reprodutor de música portátil do mercado atual, o *Ipod Shuffle da empresa Apple*, reprodução de músicas armazenadas na memória e que possui os comandos de *Play*/*Pause,* Avanço e atraso de faixa, e controle de volume.

Figura 15 - Ipod Shuffle da marca Apple

No projeto realizado para representar os botões de *Play*, *Pause*, avanço e atraso de faixa, escolheu-se o *joystick* presente no kit. E para regular o volume utilizou-se o regulador de som já presente no kit. Diferentemente do *Ipod Shuffle* que possui um memória interna para armazenar os arquivos de áudio, para este projeto adotou-se a entrada de cartão SD/MMC presente no SAM7EX-256.

Os comandos no *joystick* ficaram da seguinte maneira:

- *Joystick* para cima = *play*
- *Joystick* para esquerda = *Atrasa faixa*
- *Joystick* para direita = *Avanca faixa*
- *Joystick* para baixp = *pause*

O microcontrolador ARM7TDMI ficou responsável por realizar a comunicação e o gerenciamento dos periféricos, decodificarem o arquivo WAV, controlar o tempo de reprodução da música e de gerar o sinal de áudio.

3.3 Geração do sinal de música

Primeiramente é necessário decodificar o arquivo WAV, para saber se os dados da música são referentes a 1 canal (mono) ou 2 canais (estéreo), qual é o valor da taxa de amostragem (*Sample Rate*), e qual é o valor da taxa de *bits* por amostragem.

A geração do sinal da música fica por conta do próprio microcontrolador, através do acesso dos *buffers* que possuem os dados lidos e já decodificados para a reprodução. A cada período um valor é alocado na saída do PWM, o período com que é amostrado os dados fica a cargo do *timer* da interrupção, que é configurado de acordo com o valor do *Sample Rate*. Quando o contador do *timer* estoura, chama-se a rotina de interrupção, e nela está presente a atribuição do valor do *buffer* na saída do PWM.

Caso a música possua dois canais, os valores dos dois canais são somados e divido por dois antes de serem atribuídos para a saída do PWM, já que o kit apresenta somente um PWM configurado para saída de áudio (ou alto-falante), ou seja, o kit apresenta somente um canal de saída (mono).

Caso a música possuir 16 *bits (2bytes)* por amostragem deve-se primeiro converter este valor para um número entre 0 e 1023, pois a saída do PWM somente aceita valores entre este intervalo, já que o conversor AD somente opera com 10*bits*. Não se esquecendo que quando a música possuir 16 *bits* por amostragem, os dados utilizam codificação complemento de 2.

Já para o caso a música possuir 8*bits(1Byte)* por amostragem não haverá necessidade de converter o valor para se adequar ao intervalo, pois ao iniciar o PWM, sua saída será configurada para receber somente valores entre 0 e 255.

O uso da interrupção para este projeto foi necessário, pois o microcontrolador possui uma SRAM de 64*kbytes,* o que torna impossível armazenar em *buffer*s todos os dados de uma música, que possuem em média de 6*Mbytes* a 9*Mbytes* de tamanho. Sendo assim, para contornar este problema foram construídos dois *buffers* de 25*kbytes* de tamanho, em que quando um deles está sendo preenchido, o outro está sendo alocado na saída do PWM na rotina de interrupção.

3.4 O *software*

O programa para o microcontrolador AT91SAM7EX256 foi criado em linguagem C, no ambiente de desenvolvimento IDE ECLIPSE. A seguir é mostrado o fluxograma que representa o software implementado para o projeto do reprodutor de música.

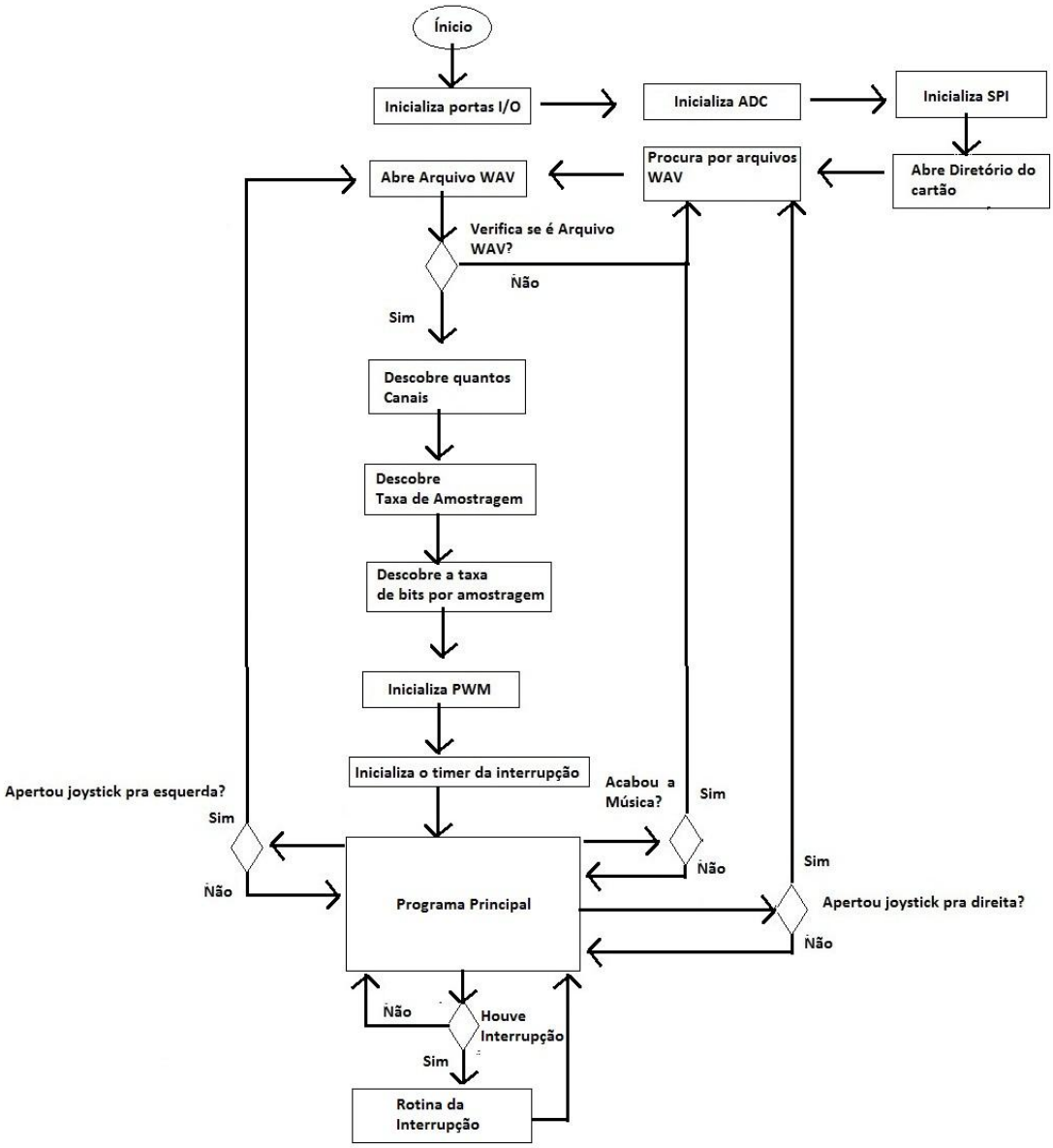

Figura 16 - Fluxograma do *software*

3.4.1 Detalhamento das rotinas

Inicializa porta I/O

Habilita o *clock* para as portas PIOA e PIOB, e defini quais portas são *inputs* ou *outputs.*

Inicializa ADC

Habilita o clock para a conversão AD e configura o conversor para operar com *trigger* ativo por *software* e resolução de 10 *bits.*

Inicializa SPI

Habilita a comunicação SPI que será utilizada para a comunicação com o cartão de memória. Nesta etapa são habilitados os pinos SPCK1, NPCS1, MISO1 e MOS1, é habilitado o clock para o SPI. Habilita a comunição como 8bits de trasnferência e clock da comunicação em 48kHz.

Abre o diretório do cartão.

Abre o diretório do cartão e aponta o ponteiro para o começo do diretório. Utiliza-se a função f_opendir encontrada na biblioteca “ff.h” (biblioteca disponibilizada no site da empresa Olimex).

f_opendir(*DIR* *, Nome Diretório).

DIR = ponteiro do tipo DIR para abrir o diretório.

Procura por arquivos WAV

Procura por arquivos WAV no diretório. Utiliza-se função f_*readdir(DIR *, finfo *)* também presente na biblioteca “ff.h”. Onde *DIR* é um ponteiro do tipo diretório, o mesmo que foi utilizado para abrir o diretório. E *finfo* é uma variável tipo struct que possui as informações do arquivo, como por exemplo, Nome e Tamanho do arquivo.

Abre o arquivo WAV

Se for para reproduzir a próxima faixa, abre o arquivo encontrado na rotina anterior. Se for para reproduzir a música anterior, abre o arquivo cujo nome está guardado.

Para abrir o arquivo utiliza-se a função f_open da biblioteca “ff.h”.

*f_open (FIL *fp, const char *path, BYTE mode)*.

fp = ponteiro que será direcionado para arquivo.

path=Nome do arquivo a ser aberto.

mode=modo que deseja abrir o arquivo (Leitura/Escrita, Criar/AbrirSeExiste).

Descobre quantos canais

Descobre com quantos canais a música deve ser reproduzida. Olhar Tabela 2 para saber qual posição em bytes fica os dados que representa o valor do canal.

Descobre a Taxa de Amostragem

Descobre o valor da taxa de amostragem. Olhar Tabela 2 para saber qual posição em bytes fica os dados que representa o valor da Taxa de Amostragem.

Descobre a taxa de bits por amostragem

Descobre o valor da taxa de bits. Olhar Tabela 2 para saber qual posição em bytes fica os dados que representa o valor da Taxa de bits por amostragem.

Inicializa PWM

Habilita o clock para o PWM e é atribuída a funcionalidade do pino à saída do PWM para executar esta função. São realizadas também as configurações referentes ao modo que o PWM irá operar, definindo valor do seu período e do *duty cycle.* Se a música possuir taxa de bits por amostragem igual 8 o seu período será configurado com valor 256 e se for 16 terá valor 1024. O valor do período é atribuído a porta PWMC_CPRDR.

Inicializa o timer da interrupção

Habilita a interrupção. E atribui o valor da taxa de amostragem para o *timer* da interrupção. O valor do *timer* é atribuído a porta AT91C_PITC_PIMR.

Programa Principal

Após realização de todas as rotinas de inicialização o programa principal começa a executar em *loop*, até que música seja reproduzida por inteira. Podendo ser interrompido se o usuário pressionar os botões direito ou esquerdo do *joystick.* Caso o usuário pressionar o botão esquerdo, será reproduzida a música anterior. Caso o usuário pressionar o botão direito, será reproduzido a próxima música, buscada pela rotina *Avancar().*

Dentro deste *loop* também se verifica se o usuário pressionou o botão para baixo (*pause*) do *joystick.* Caso aperte, não há reprodução de música e o programa fica aguardando até que o usuário aperte o botão pra cima (*play*) do joystick.

E é nesta etapa também que ocorre as interrupções do *Timer,* que a cada intervalo de tempo, definido pela taxa de amostragem, interrompe a rotina do programa principal para alocar uma amostra da música na saída do PWM.

Rotina de Interrupção

Nesta etapa onde ocorre a alocação de uma amostra da música na saída do PWM.

Primeiro é necessário verificar qual é o valor da taxa de bits por amostragem e verificar quantos canais a música possui. Pois dependendo desses valores é necessário efetuar mudanças no valor da amostra, antes de ser atribuído na saída do PWM. A seguir será descrito como deve efetuar as mudanças em cada caso.

- 8bits por amostragem e um canal

Neste caso não é necessário realizar nenhuma mudança no valor atribuído ao PWM. O próprio valor da amostra será alocado.

- 8 bits por amostragem e dois canais

Neste caso é necessário somar o valor da amostra de cada canal e dividir por dois, antes de atribuir o valor na saída do PWM. Pois a saída de áudio do kit está configurada apenas para um PWM.

-16bits por amostragem e um canal

Para este caso deve-se realizar a conversão do valor da amostra, já que se utiliza codificação em sinal-complemento de 2. E é necessário converter o valor para se adequar ao intervalo de valor que o PWM aceita na sua saída.

-16bits por amostragem e dois canais

Primeiramente deve-se realizar a mesma conversão do caso anterior para cada amostra de cada canal. Feito a conversão deve-se somar o valor das duas amostras e dividir por dois, e assim atribuir o resultado à saída do PWM.

CAPÍTULO 4 – Resultados e Conclusões

4.1 Resultados

Com auxílio de um osciloscópio analógico foi possível capturar a forma de onda do sinal sonoro gerado pelo PWM e então compará-lo com a forma de onda teórica simulada.

a- Senóide

Para efeito de comparação escolheu-se reproduzir uma forma de onda do tipo senóide com período de 4kHz, e amplitude de 128. A forma de onda foi representada por 40 pontos, mostrado a seguir. Cada ponto foi amostrado (atribuído na saída do PWM) com taxa de amostragem igual 160kHz.

128 148 167 186 203 218 231 242 249 254
256 254 249 242 231 218 203 186 167 148
128 107 88 69 52 37 24 13 6 1
0 1 6 13 24 37 52 69 88 107

Na figura 17 é possível observar o sinal da saída do PWM referente à senóide reproduzida. Pela imagem da tela do osciloscópio é possível observar que um período possui aproximadamente 1,25 divisões. Sendo cada divisão 0,2ms, isto determina que a freqüência da senóide seja de aproximadamente 4khz, confirmando assim que a reprodução do sinal foi realizada com sucesso.

Na figura 18 é possível observar a forma de onda simulada e notar que o resultado prático condiz com o esperado.

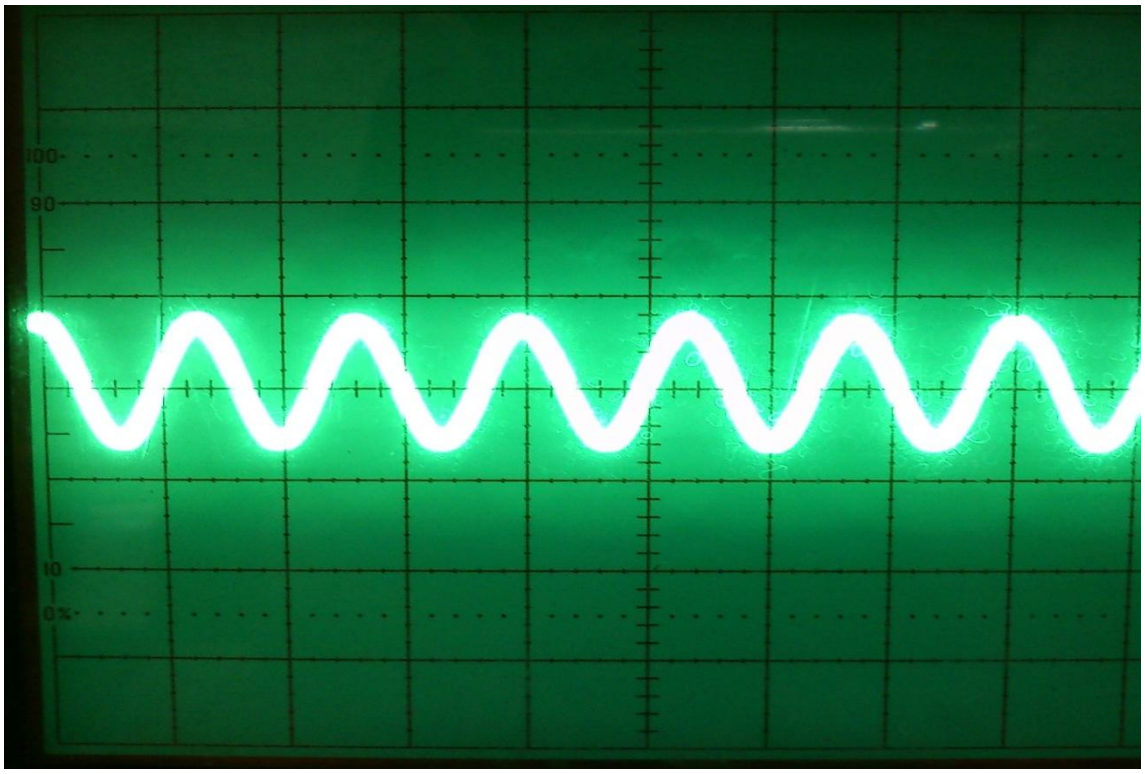

Figura 17 - 2,5V/Div 0.2ms/Div- Sinal na Saída do PWM (Sinal senoidal de frequência 4kHz)

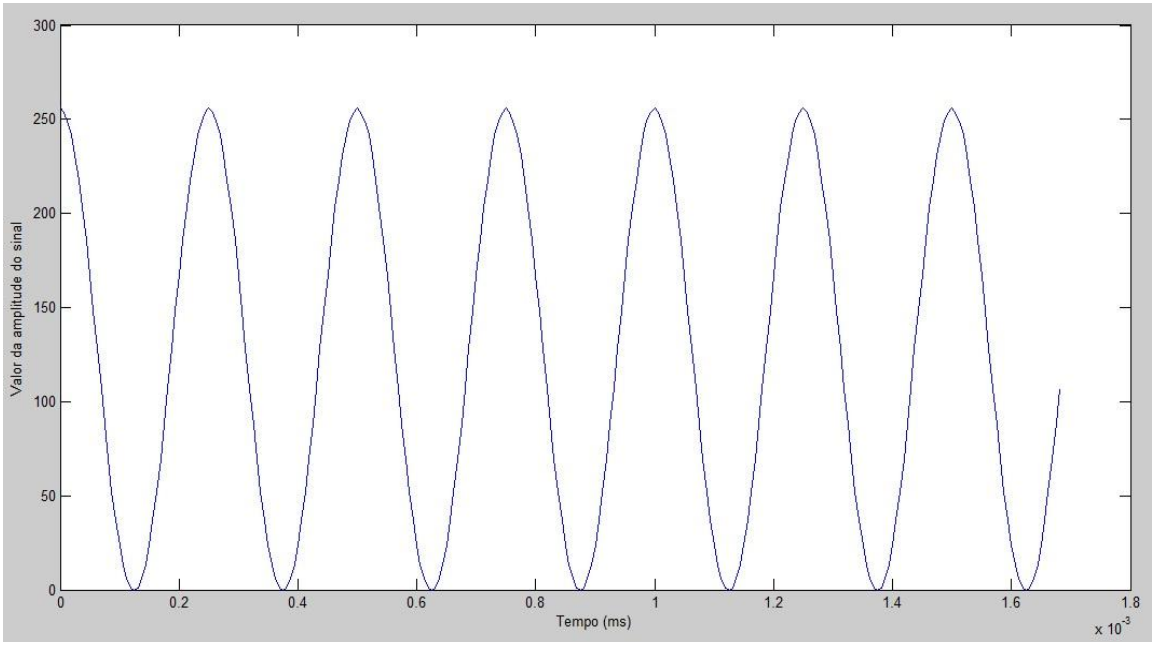

Figura 18 - Simulação da forma de onda do tipo senóide

b- Trecho de uma música

O sinal de áudio utilizado para efeito de comparação, foi um pequeno trecho extraído de um arquivo WAV com taxa de bits por amostragem igual a 16, taxa de amostragem igual a *8kHz* e possuindo somente um canal. A seguir é mostrado em hexadecimal o pequeno trecho utilizado, possuindo 65 amostras.

00 B2 00 B7 00 F6 00 28 00 32 00 0C 00 F6 00 EC
00 F6 00 FA 00 00 00 EC 00 E2 00 E9 00 F1 00 E2
00 23 00 30 00 52 00 32 00 CE 00 AD 00 B2 00 FF
00 3C 00 2D 00 00 00 E2 00 EA 00 FF 00 0F 00 FF
00 F6 00 E8 00 E9 00 F5 00 06 00 FF 00 E5 00 D3

00 23 00 58 00 23 00 B1 00 B9 00 F4 00 32 00 3F
00 14 00 FF 00 D5 00 F4 00 19 00 20 00 00 00 E2
00 D8 00 F1 00 0C 00 19 00 02 00 D0 00 C5 00 E7
00 28

Na Figura 19 é possível observar o sinal de saída do alto-falante referente ao trecho reproduzido acima. Pela imagem da tela do osciloscópio é possível perceber que a escala por divisão é de 1ms, isto determina que a cada divisão seja amostrado 8 amostras, já que a taxa de amostragem do som reproduzido é de 8kHz.

Na figura 20 é possível observar a forma de onda simulada, e notar que o sinal obtido na saída do alto-falante condiz com o esperado. Confirmando assim, que as configurações realizadas para obter a taxa de amostragem definida no cabeçalho do arquivo de áudio, foram realizadas com sucesso, juntamente com a interpretação dos seus dados.

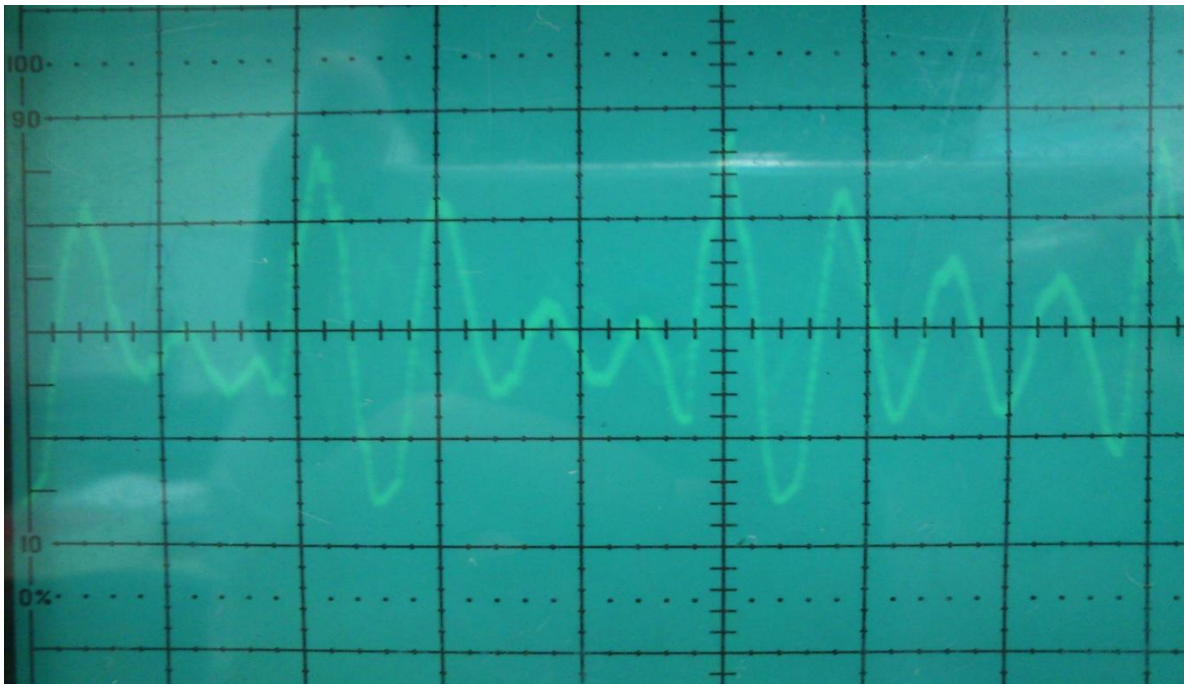

Figura 19 - 1V/Div 1ms/Div- Sinal na Saída do Auto-Falante (Sinal com taxa de amostragem igual 8kHz)

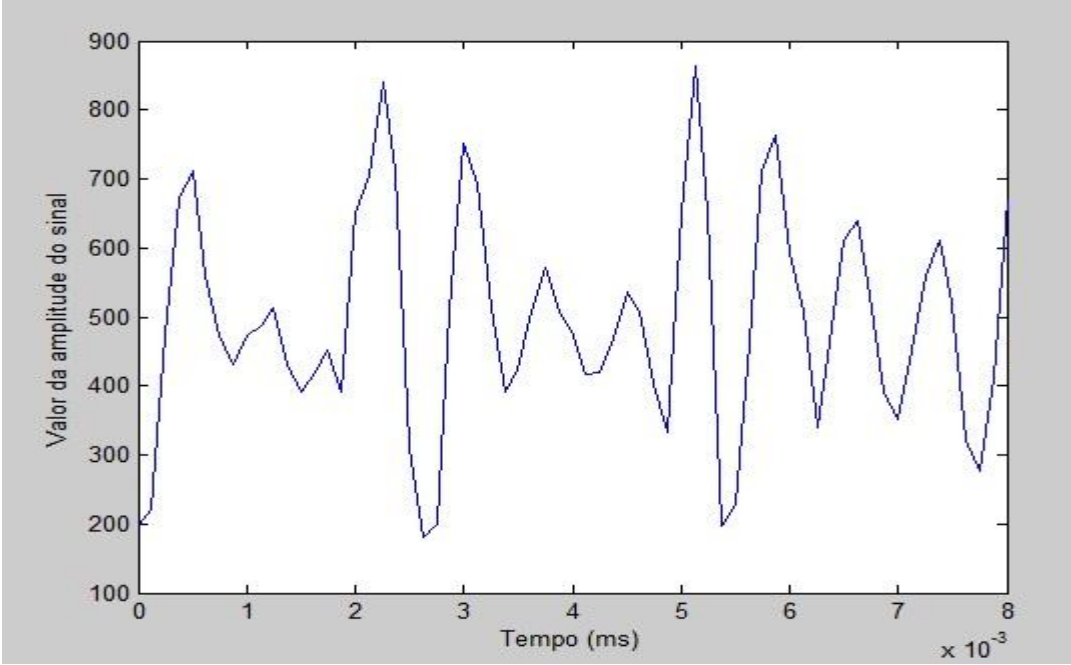

Figura 20 - Simulação do sinal de áudio referente ao trecho analisado

4.2 Conclusão

A utilização de interrupção para a reprodução da música trouxe resultados excelentes para o projeto, pois solucionou o problema de falta de memória para armazenar dados durante o processo de reprodução, além de garantir a precisão do intervalo de amostragem para cada amostra.

No desenvolvimento do projeto do reprodutor de música, foi possível trabalhar vários fundamentos, envolvendo desde a manipulação e interpretação de dados de um arquivo de áudio no formato WAV, até a aplicação de conceitos eletrônicos relacionados ao uso de microcontroladores.

Desenvolver um projeto utilizando um microcontrolador baseado em uma arquitetura tão bem sucedida e com aplicações tão atuais quanto a ARM é uma experiência que agrega muito conhecimento.

Foi possível observar que, mesmo tendo os resultados propostos pelo projeto, é possível agregar muitas outras funcionalidades utilizando-se os periféricos fornecidos pelo microcontrolador, como por exemplo, a comunicação Etherneth e USB.

O desenvolvimento de aplicações em microcontroladores é um exercício que estimula a busca por novas soluções e nos obriga a estar sempre agregando conhecimento, pois no ramo digital a evolução é constante e há sempre novos dispositivos com novas funcionalidades sendo lançados no mercado.

REFERÊNCIAS

ARM ltd 2011. The Architecture for de the digital world. ARM [Online] ARM ltd., 2011. www.arm.com

Atmel Corporation. 2009. AT91 ARM Thumb-based Microcontrollers. S.I.: Atmel Corporation, 2009.

PEREIRA, Fábio. 2007. *Tecnologia ARM: Microcontroladores de 32bits*. S.I. : Érica, 2007.

Wikipedia. 2011. *Wikipedia The Free Encyclopedia*. Wikipedia *The Free Encyclopedia.* [Online] 2011. http://en.wikipedia.org/.

Olimex ltd 2011. *Olimex Development boards and tools*, [Online] 2011. www.olimex.com

SEDRA, Adel S. e SMITH, K. C. 2000. *Microeletrônica*. São Paulo: Pearson Makron Books, 2000.

Prof. Peter Kabal, 2011. *Audio File Format Specifications*, [Online] 2011. http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html

Scott Wilson, 2003. *WAVE PCM soundfile format*, [Online] 2003. https://ccrma.stanford.edu/courses/422/projects/WaveFormat/

Scott Wilson, 2003. *The WAVE File Format*, [Online] 2003. http://www.lightlink.com/tjweber/StripWav/WAVE.html

Carlos E. Morimoto, 2007. *Secure Digital (SD),* [Online] 2007, http://www.hardware.com.br/termos/sd

APÊNDICE A – CÓDIGO DO PROJETO REPRODUTOR DE MÙSICA

```
/*Trabalho de Conclusão de Curso
 Nome Projeto WAV Player utilizando ARM7
 Aluno Fernando Takahashi 5910710
 Orientador Professor Evandro
 Software para decodificação de arquivos Wav
 */

//*****************************************************************
//*************************Includes********************************
#include "..\include\AT91SAM7X256.h"
#include "..\include\lib_AT91SAM7S256.h"
#include "common_definitions.h"
#include "isrsupport.h"
#include "adc.h"
#include "data.h"
#include "tff.h"
#include "diskio.h"
#include "system.h"
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
//*****************************************************************
//*************************Defines*********************************
#define __inline extern inline
//Definindo interrupcao
#define __ramfunc

#define AT91C_US_ASYNC_MODE ( AT91C_US_USMODE_NORMAL +
AT91C_US_NBSTOP_1_BIT + AT91C_US_PAR_NONE + AT91C_US_CHRL_8_BITS +
AT91C_US_CLKS_CLOCK )

#define PA0    ((unsigned int) 1 <<  0)
#define PA1    ((unsigned int) 1 <<  1)
#define PA2    ((unsigned int) 1 <<  2)

#define MCK            47923200   // MCK (PLLRC div by 2)
//definindo tamanho do Buffer que contem a musica
#define BUFF_SIZE 25000
//*****************************************************************
//******************Variaveis Globais**********************************
DIR dir;
FILINFO finfo;
volatile FIL plik;
FATFS fsst;
FIL fFile;
```

```
FRESULT res;
WORD br;
int long iTamanhoChunk2;
static int control_delay;
volatile unsigned int iIndeks=0, iBuf_1=1, iPlay=1, iRead=1, iAcabou = 0;
long int iSampleRate, iChannel, iBitPerSample;
//**********************************************************************
//*********************Struct*****************************************
typedef struct _Music {//Para dados da musica
        volatile unsigned char bufor_0[BUFF_SIZE];
} Music;
  Music List[2];
typedef struct _ListMusic {//Para lista de musicas
  char MusicName[20];
} ListMusic;
 ListMusic ListMusica[11];
//**********************************************************************
//***********Funcoes Auxiliares ****************************************
void ReiniciaVetores(){
  int i;
  for(i = 0; i < BUFF_SIZE; i++){
        List[0].bufor_0[i] = 0;
        List[1].bufor_0[i] = 0;
  }
}
void Delay(unsigned long h) {
        volatile unsigned long d;
        d = h;
        while (--d != 0);
}
void Atrasa(){
  Player(ListMusica[0].MusicName);
}
void Avancar()
{
        int n, res, j = 0;
        iPlay=0;
        do
        {       n=0;
                res = f_readdir(&dir, &finfo);
                if ((res != FR_OK) || !finfo.fname[0])
                {
                        f_opendir(&dir, ptr);
                        res = f_readdir(&dir, &finfo);
                }
                if(!(finfo.fattrib & AM_DIR)) while(finfo.fname[n++]);
        }while((finfo.fattrib & AM_DIR) || (finfo.fname[n-2]!='V') || (finfo.fname[n-3]!='A') ||
(finfo.fname[n-4]!='W') );
        f_close(&plik);
   strcpy(ListMusica[0].MusicName, &(finfo.fname[0]));
        Player(&(finfo.fname[0]));
```

```
}
void Read_Block(void){
  if(iRead == TRUE){
        if(iBuf_1 == TRUE)                                          //carrega o primeiro
buffer
        {
          f_read(&fFile, List[0].bufor_0, BUFF_SIZE, &br);
        }
        else                                                        //carrega o
segundo buffer
        {
          f_read(&fFile, List[1].bufor_0, BUFF_SIZE, &br);
        }
        iPlay = 1; iRead = 0;
  }
}
//**********************Configuração para PWM*************************
void PWMInit(int Periodo)
{
        pPWM->PWMC_DIS = 1;
        AT91C_BASE_PWMC->PWMC_IDR=0x1<<1|0x1<<2;
        AT91C_BASE_PWMC->PWMC_DIS=0x1<<1|0x1<<2;
        pPMC->PMC_PCER = 1<<2;
        pPMC->PMC_PCER = 1<<10;
        // Set second functionality of pin
        pPIOB->PIO_PDR = BIT19;
        pPIOB->PIO_ASR = BIT19;
        pPIOB->PIO_BSR = 0;
        pPWM_CH0->PWMC_CPRDR = Periodo;
        // Duty for PWM
        pPWM_CH0->PWMC_CDTYR = 1;
        // Enable PWM chanel
        pPWM->PWMC_ENA = 1;
}
//**********************Inicializar o SPI do cartão e o cartão***********
void power_on (void)
{
        pPIOA->PIO_PPUDR = CARD_INS_PIN | CARD_WP_PIN;
        pPIOA->PIO_ODR = CARD_INS_PIN | CARD_WP_PIN;
        pPIOA->PIO_PER = CARD_INS_PIN | CARD_WP_PIN;
        pPIOA->PIO_PDR = AT91C_PA16_SPI0_MISO | AT91C_PA17_SPI0_MOSI |
AT91C_PA18_SPI0_SPCK;
        pPIOA->PIO_ASR = AT91C_PA16_SPI0_MISO | AT91C_PA17_SPI0_MOSI |
AT91C_PA18_SPI0_SPCK;
        pPIOA->PIO_PER  = AT91C_PA13_SPI0_NPCS1; // habilita GPIO of CS-pin
        pPIOA->PIO_SODR = AT91C_PA13_SPI0_NPCS1;
        pPIOA->PIO_OER  = AT91C_PA13_SPI0_NPCS1;
       s_pPMC->PMC_PCER = 1 << AT91C_ID_SPI0;
        s_pSpi->SPI_CR      = 0x81;
        s_pSpi->SPI_CR      = 0x01;
        s_pPDC->PDC_PTCR = AT91C_PDC_TXTEN | AT91C_PDC_RXTEN;
```

```
          s_pSpi->SPI_PTCR = AT91C_PDC_TXTEN | AT91C_PDC_RXTEN;
        s_pSpi->SPI_MR=AT91C_SPI_MSTR | AT91C_SPI_PS_FIXED |
AT91C_SPI_MODFDIS;
        s_pSpi->SPI_CSR[1]  = 0x00001F02;
        s_pSpi->SPI_MR &= 0xFFF0FFFF;
        s_pSpi->SPI_MR |= 0xD0000;
}
//**********************Configuração para o conversor******************
void ADCInit(void)
{
        // Enable clock for interface
        pPMC->PMC_PCER = 1 << AT91C_ID_ADC;
        // Reset
        pADC->ADC_CR = 0x1;
        pADC->ADC_CR = 0x0;
        // Set maximum startup time and hold time
        pADC->ADC_MR = 0x0F1F0F00;
}
//************************Manipulação arquivo WAV***********************
/*----------------------------------
 ----------CONSTANTES----------------*/
//constantes True e False
int True = 1, False = 0;
//Constante Cabecalho normal
int iConstCabecalhoNormal = 16;
//Constante BitsPerSAmple = 8
int BitsPerSample_8 = 8, BitsPerSample_16 = 16;
//constantes format_code
int iConstPCM = 1, iConstIEEE = 2, iConstALAW = 3, iConstMULAW = 4,
iConstEXTENSIBLE = 5;
//constantes channels
int iConstMono = 1, iConstEstereo = 2;
//Funcao para busca de bytes ao longo do arquivo retornando uma vetor
unsigned char *Busca(int AiNumBytes) {
        unsigned char *cBusca;
        cBusca = (unsigned char *) malloc(AiNumBytes * sizeof(unsigned char));
        f_read(&fFile, cBusca, AiNumBytes, &br);
        return (cBusca);
}
//Funcao para busca de bytes ao longo do arquivo retornando uma char
unsigned char BuscaChar() {
        int iNumByte;
        unsigned char *cBusca;
        iNumByte = 1;
        cBusca = (unsigned char *) malloc(iNumByte * sizeof(unsigned char));
        f_read(&fFile, cBusca, iNumByte, &br);
        return (cBusca[0]);
}
//veriricar se é formato Wav, 0 a 3 bytes
int Verificar_ID_RIFF() {
        unsigned char *cRIFF;
```

```
	int iRIFF, iEhRIFF;
	iRIFF = 4;
	iEhRIFF = False;
	//chamar Busca, verificar se os primeiros 4 bytes sao RIFF.
	cRIFF = Busca(iRIFF);
	if (cRIFF[0] == 'R' && cRIFF[1] == 'I' && cRIFF[2] == 'F' && cRIFF[3]== 'F') {
		iEhRIFF = True;
	}
	return (iEhRIFF);
}
//Tamanho do arquivo 3 a 7 bytes, Tamanho = chunk + 8(RIFF + 4bytesChk)
int Tamanho_Arquivo() {
	unsigned char * cTamanhoArquivo;
	int iTamanho;
	long int iTamanhoArquivo;
	iTamanho = 4;
	//busca os bytes 3 a 7
	cTamanhoArquivo = Busca(iTamanho);
	//calculo do tamanho do arquivo
	iTamanhoArquivo = cTamanhoArquivo[0] + 256 * cTamanhoArquivo[1] + 65536
			* cTamanhoArquivo[2] + 16777216 * cTamanhoArquivo[3] + 8;
	return (iTamanhoArquivo);
}
//verifica se o id eh WAVE, 8 a 11 bytes
int Verificar_ID_WAVE() {
	unsigned char *cWAVE;
	int iWAVE, iEhWAVE;
	iWAVE = 4;
	iEhWAVE = False;
	//chamar Busca, verificar se os 4 bytes sao WAVE.
	cWAVE = Busca(iWAVE);
	if (cWAVE[0] == 'W' && cWAVE[1] == 'A' && cWAVE[2] == 'V' && cWAVE[3]== 'E')
{
		iEhWAVE = True;
	}
	return (iEhWAVE);
}
//verificar se id = fmt, bytes 12 a 15
int Verificar_ID_fmt() {
	unsigned char *cfmt;
	int ifmt, iEhfmt;
	ifmt = 4;
	iEhfmt = False;
	//chamar Busca, verificar se os 4 bytes sao fmt.
	cfmt = Busca(ifmt);
	if (cfmt[0] == 'f' && cfmt[1] == 'm' && cfmt[2] == 't' && cfmt[3] == ' ') {
		iEhfmt = True;
	}
	return (iEhfmt);
}
//calcular o tamanho do cabecalho, byte 16 a 19
```

```
int Chunk_Size1() {
	unsigned char * cChunkSize1;
	int iChunkSize1;
	long int iTamanhoChunkSize1;
	iChunkSize1 = 4;
	//busca os bytes 16 a 19
	cChunkSize1 = Busca(iChunkSize1);
	//calculo do tamanho do chunksize
	iTamanhoChunkSize1 = cChunkSize1[0] + 256 * cChunkSize1[1] + 65536
			* cChunkSize1[2] + 16777216 * cChunkSize1[3];
	return (iTamanhoChunkSize1);
}
//verificar se tem cabecalho estendido ou não! Cabecalho extendio > 16 bytes.
int Eh_Cabecalho_Extendido(int AiSizeChunk1) {
	int iEh_Cabecalho_Extendido;
	iEh_Cabecalho_Extendido = True;
	if (AiSizeChunk1 == iConstCabecalhoNormal) {
		iEh_Cabecalho_Extendido = False;
	}
	return (iEh_Cabecalho_Extendido);
}
//Verificar qual format_code, bytes 20 e 21
int Format_Code() {
	unsigned char * cFormatCode;
	int iByteFormatCode, iFormatCode;
	iByteFormatCode = 2;
	//busca os bytes 20 a 21
	cFormatCode = Busca(iByteFormatCode);
	//format PCM
	if (cFormatCode[1] == 0x00 && cFormatCode[0] == 0x01) {
		iFormatCode = iConstPCM;
	}//format IEEE
	else if (cFormatCode[1] == 0x00 && cFormatCode[0] == 0x03) {
		iFormatCode = iConstIEEE;
	}//format ALAW
	else if (cFormatCode[1] == 0x00 && cFormatCode[0] == 0x06) {
		iFormatCode = iConstALAW;
	}//format MULAW
	else if (cFormatCode[1] == 0x00 && cFormatCode[0] == 0x07) {
		iFormatCode = iConstMULAW;
	}//format Extensible
	else if (cFormatCode[1] == 0xFF && cFormatCode[0] == 0xFE) {
		iFormatCode = iConstEXTENSIBLE;
	}return (iFormatCode);
}
//verificar se é mono ou estero, bytes 22 e 23
int Channel() {
	unsigned char * cChannel;
	int iByteChannel;
	iByteChannel = 2;
	//busca os bytes 22 a 23
```

```
	cChannel = Busca(iByteChannel);
	//channel mono
	if (cChannel[1] == 0 && cChannel[0] == 1) {
		iChannel = iConstMono;
	}//channel estereo
	else if (cChannel[1] == 0 && cChannel[0] == 2) {
		iChannel = iConstEstereo;
	}return (iChannel);
}
//descobrir qual é o samplerate, bytes 24 a 27
int Sample_Rate() {
	unsigned char * cSampleRate;
	int iByteSampleRate;
	iByteSampleRate = 4;
	//busca os bytes 24 a 27
	cSampleRate = Busca(iByteSampleRate);
	//calculo do sample rate
	iSampleRate = cSampleRate[0] + 256 * cSampleRate[1] + 65536
			* cSampleRate[2] + 16777216 * cSampleRate[3];
	return (iSampleRate);
}
//descobrir ByteRate = SampleRate * NumChannels * BitsPerSample/8, bytes 28 a 31
int Byte_Rate() {
	unsigned char * cByteRate;
	int iByteByteRate;
	long int iByteRate;
	iByteByteRate = 4;
	//busca os bytes 28 a 31
	cByteRate = Busca(iByteByteRate);
	//calculo do sample rate
	iByteRate = cByteRate[0] + 256 * cByteRate[1] + 65536 * cByteRate[2]
			+ 16777216 * cByteRate[3];
	return (iByteRate);
}
//Calculo para saber BitsPerSample = ByteRate * 8 /(SampleRate * NumChannels)
int Calculo_Bits_Per_Sample(int AiNumChannels, int AiSampleRate, int AiByteRate) {
	int iBitsPerSample;
	iBitsPerSample = 8 * (AiByteRate / (AiSampleRate * AiNumChannels));
	return (iBitsPerSample);
}
//funcao para verificar se é BitsPerSAmple = 8 bits
int Verificar_Eh_8Bits(int AiBitsPerSample) {
	int iEh8Bits;
	iEh8Bits = False;
	if (AiBitsPerSample == BitsPerSample_8) {
		iEh8Bits = True;
	}
	return (iEh8Bits);
}
//descobrir qual é block align = BitsPerSample/nBits, bytes 32 e 33
int Block_Align() {
```

```
	unsigned char * cBlockAlign;
	int iByteBlockAlign;
	long int iBlockAlign;
	iByteBlockAlign = 2;
	//busca os bytes 32 a 33
	cBlockAlign = Busca(iByteBlockAlign);
	//block align
	iBlockAlign = cBlockAlign[0] + 256 * cBlockAlign[1];
	return (iBlockAlign);
}
//Descobri qual é BitsPerSample, bytes 34 e 35
int Bits_Per_Sample() {
	unsigned char * cBitsPerSample;
	int iByteBitsPerSample;
	long int iBitsPerSample;
	iByteBitsPerSample = 2;
	//busca os bytes 34 a 35
	cBitsPerSample = Busca(iByteBitsPerSample);
	//BitsPerSample
	iBitsPerSample = cBitsPerSample[0] + 256 * cBitsPerSample[1];
	return (iBitsPerSample);
}
//Se for cabecalho Extendido procurar byte a byte ate achar DATA
int Seek_ID_Data(int AiEhCabecalhoExtendido) {
	int iAchouData, i4byte, iPosicaoZero;
	unsigned char *cSeekData;
	i4byte = 4;
	iPosicaoZero = 0;
	if (AiEhCabecalhoExtendido == True) {
		//procurar byte a byte ate char ID data
		iAchouData = False;
		cSeekData = Busca(i4byte);
		while (iAchouData != True) {
			if (cSeekData[0] == 'd' && cSeekData[1] == 'a' && cSeekData[2]
					== 't' && cSeekData[3] == 'a') {
				iAchouData = True;
			} else {
				cSeekData[0] = cSeekData[1];
				cSeekData[1] = cSeekData[2];
				cSeekData[2] = cSeekData[3];
				cSeekData[3] = BuscaChar();
			}
		}
	}return (iAchouData);
}
//verificar ID = Data, byte 36 a 39
int Verificar_ID_Data() {
	unsigned char *cData;
	int iData, iEhData;
	iData = 4;
	iEhData = False;
```

```
	//chamar Busca, verificar se os primeiros 4 bytes sao DATA.
	cData = Busca(iData);
	if (cData[0] == 'd' && cData[1] == 'a' && cData[2] == 't' && cData[3]
			== 'a') {
		iEhData = True;
	}return (iEhData);
}
//calcular o tamanho do data, byte 40 a 43
long int Chunk_Size2() {
	unsigned char * cChunkSize2;
	int iChunkSize2;
	long int iTamanhoChunkSize2;
	iChunkSize2 = 4;
	//busca os bytes 40 a 43
	cChunkSize2 = Busca(iChunkSize2);
	//calculo do tamanho do chunksize
	iTamanhoChunkSize2 = cChunkSize2[0] + 256 * cChunkSize2[1] + 65536
			* cChunkSize2[2] + 16777216 * cChunkSize2[3];
	return (iTamanhoChunkSize2);
}
int ArquivoWAV(char * AfFilename) {
	int iChannel, iSampleRate, iByteRate, iTamanhoArquivo, iTamanhoChunk1,
			iEhCabExt, iCabData, iPodeTocar, iByteALer;
	unsigned char *teste;
	int *teste2;
	//Atencao nao altere a sequencia de chamada das funcoes, pois ha uma sequencia
certa para chamar cada funcao
	iPodeTocar = False;
	//primeiro devemos abrir o arquivo
	f_open(&fFile, AfFilename, FA_OPEN_EXISTING| FA_READ);
	//verificar se o arquivo é wav, verificar se o q primeiros bytes corresponde a RIFF
	if (Verificar_ID_RIFF() == True) {
		// calcular o tamanho do arquivo Tamanho = chunk + 8(RIFF + 4bytesChk)
		iTamanhoArquivo = Tamanho_Arquivo();
		// verificar se este possui o cabecacalho wav
		if (Verificar_ID_WAVE() == True) {
			// verificar se possui o cabecalho fmt (descreve formato dos dados
wav)
			if (Verificar_ID_fmt() == True) {
				//calcular o tamanho do chunk1
				iTamanhoChunk1 = Chunk_Size1();
				//verificar se possui cabecalho extendido (chunk1 > 16)
				iEhCabExt = Eh_Cabecalho_Extendido(iTamanhoChunk1);
				//verificar se o format code (tem quer igual a iConstPCM
senao e arquivo comprimido)
				if (Format_Code() == iConstPCM) {
					// chamar funcao para descobrir se e estereo ou mono
					iChannel = Channel();
					//descobrir qual é o sample rate
					iSampleRate = Sample_Rate();
					//descobrir o byte rate
```

```
                                        iByteRate = Byte_Rate();
                                        //descobrir qual e o block align
                                        Block_Align();
                                        //descobrir o bits_per_sample
                                        iBitPerSample = Bits_Per_Sample();
                                                if (iEhCabExt == True) {
                                                        iCabData = Seek_ID_Data(iEhCabExt);
                                                }//se nao for cabecalho extendido verificar se
tem cabecalho data
                                                else {
                                                        iCabData = Verificar_ID_Data();
                                                }//verificar se exite cabecalho data
                                                if (iCabData == True) {
                                                        //calcular chunk2
                                                        iTamanhoChunk2 = Chunk_Size2();
                                                        iPodeTocar = True;
                                                }
                                        }
                                }
                        }
                }
        }
  return iPodeTocar;
}
//*********************************************************************
//*****************Interrupcao*****************************************
__ramfunc void Interrupcao(void)
{
        AT91C_BASE_AIC->AIC_EOICR=0x0;
}
__ramfunc void Reproduz()                               //interrupacao para reproducao
{
        int dummy1, iAux = 0;
        float Aux = 0;
        dummy1=AT91C_BASE_PITC->PITC_PIVR;
        //verifica se e pra tocar
        if(iPlay == TRUE){
        //verifica se ja acabou a musica
          if(iTamanhoChunk2 > 0){
                //verifica qual e a taxa de bits por amostragem
                  if(iBitPerSample == BitsPerSample_16){
                          if(iBuf_1 == TRUE)
                          //toca pelo primeiro buffer
                           {
                         //Se for 2 canais somar as duas amostras, ja que possui somente
uma saida de PWM
                           for (int i = 0; i < iChannel; i++){
                         // como e 16 bits por amostragem, deve-se fazer a conversão, pois
utiliza complemento de 2
                             if (((List[1].bufor_0[iIndeks+1] >> 7) & 0x01) == 0x01){
                                 Aux = ((List[1].bufor_0[iIndeks+((iChannel * i)) ] + 256 *
                                 List[1].bufor_0[iIndeks + 1 + iChannel * i)])   - 32767) + Aux;
```

```
                        }else{
                            Aux = ((List[1].bufor_0[iIndeks + iChannel * i)] + 256 *
                            List[1].bufor_0[iIndeks + 1 + iChannel * i)])   + 32767) + Aux;
                             }
                    }
                   //deve-se dividir o valor de AUX por 64 antes de atribuir a saida do
PWM, pois o PWM somente aceita valores ate 1023
                     AT91C_BASE_PWMC->PWMC_CH[0].PWMC_CUPDR =
                            (int)((Aux/64) / iChannel);
                   //verifica se ja tocou tudo buffer
                   if (iIndeks == BUFF_SIZE){
                      iRead = 1;iIndeks = 0;iBuf_1 = 0;
                    }
                    }
                     else            //toca o segundo buffer
                     {
//Se for 2 canais somar as duas amostras, ja que possui somente uma saida de PWM
                      for (int i = 0; i < iChannel; i++){
                        if (((List[0].bufor_0[iIndeks+1] >> 7) & 0x01) == 0x01){
                          Aux = ((List[0].bufor_0[iIndeks+((iChannel * i)) ] + 256 *
                         List[0].bufor_0[iIndeks + 1 + iChannel * i)])   - 32767) + Aux;
                         }else{
                   // como e 16 bits por amostragem, deve-se fazer a conversão, pois
                   utiliza complemento de 2
                          Aux = ((List[0].bufor_0[iIndeks + iChannel * i)] + 256 *
                          List[0].bufor_0[iIndeks + 1 + iChannel * i)])   + 32767) + Aux;
                     }
                     }
//deve-se dividir o valor de AUX por 64 antes de atribuir a saida do PWM, pois o PWM
somente aceita valores ate 1023
                      AT91C_BASE_PWMC->PWMC_CH[0].PWMC_CUPDR =
                   (int)((Aux/64) / iChannel);
//verifica se ja tocou tudo buffer
                    if (iIndeks == BUFF_SIZE){
                      iRead = 1; iIndeks = 0; iBuf_1 = 1;
                     }
                   }
                      iTamanhoChunk2 = iTamanhoChunk2 - iChannel * 2;
                      iIndeks = iIndeks + iChannel * 2;
                 }
               //verica se e 8 bits por amostragem
                 else if(iBitPerSample == BitsPerSample_8){
               //reproduz pelo primeiro buffer
                   if(iBuf_1 == TRUE)
                   {
               //verifica se e estereo
                      if (iChannel == 2){
               //soma as duas amostras seguidas pois existe somente uma saida de PWM
                        Aux = List[1].bufor_0[iIndeks] + List[1].bufor_0[iIndeks + 1]
                       }else {
                         Aux = List[1].bufor_0[iIndeks];
```

```
                    }
                //deve-se dividir o valor de AUX pelo numero de canais
                     AT91C_BASE_PWMC->PWMC_CH[0].PWMC_CUPDR =
                          (int) (Aux / iChannel);
                //verifica se o buffer ja foi reproduzido
                 if (iIndeks == BUFF_SIZE){
                    iRead = 1;  iIndeks = 0; iBuf_1 = 0;
                  }
                }
                //reproduz o segundo buffer
                 else
                 {
                //verifica se e estereo
                  if (iChannel == 2){
                //soma as duas amostras seguidas pois existe somente uma saida de PWM
                     iAux = List[0].bufor_0[iIndeks] + List[0].bufor_0[iIndeks + 1]
                   }
                  else
                   {
                     iAux = List[0].bufor_0[iIndeks];
                   }
                //deve-se dividir o valor de AUX pelo numero de canais
                     AT91C_BASE_PWMC->PWMC_CH[0].PWMC_CUPDR =
                          (int) (iAux / iChannel);
                //verifica se o buffer ja foi reproduzido
                 if (iIndeks == BUFF_SIZE){
                   iRead = 1; iIndeks = 0; iBuf_1 = 1;
                  }
                  }
                   iTamanhoChunk2 = iTamanhoChunk2 - iChannel;
                  iIndeks = iIndeks + iChannel;
                  }
            }
            else{
              iAcabou = 1;
            }
          }
          AT91C_BASE_AIC->AIC_EOICR=0x0;
}
//*****************************************************************************
//***********Funcao para setar PIT (Timer da interrupcao)*********************
void InitPIT(unsigned long dwPeriod)
 {
  // desabilita PIT  interrupt
  AT91C_BASE_AIC->AIC_IDCR = (1 << AT91C_ID_SYS);
  // Ter Certeza que a interrupcao foi zerada
  AT91C_BASE_AIC->AIC_ICCR = (1 << AT91C_ID_SYS);
  // Limpar PIVR
  *AT91C_AIC_EOICR = *AT91C_PITC_PIVR;
  //Setar PIT
  *AT91C_PITC_PIMR          = AT91C_PITC_PITIEN | /* PIT
```

```
    Interrupt Habilitado */
    AT91C_PITC_PITEN | /* PIT
      Enable */
           ((MCK/dwPeriod)); /* Valor do Periodo */
   AT91C_BASE_AIC->AIC_SMR[AT91C_ID_SYS] =
          AT91C_AIC_SRCTYPE_INT_POSITIVE_EDGE | 7;
   //Determina onde é interrupcao
   AT91C_BASE_AIC->AIC_SVR[AT91C_ID_SYS] = (unsigned long) Reproduz;
   // Habilita a interrupt
   AT91C_BASE_AIC->AIC_IECR = (1 << AT91C_ID_SYS);
 }
//*****************************************************************************
//********************Configurar o ARM*****************************************
void Device_Init(void){
          AT91C_BASE_AIC->AIC_SPU  = (int) Interrupcao ;
          int dummy;
          AT91C_BASE_PMC->PMC_PCER=1<<2;
          AT91C_BASE_PMC->PMC_PCER=1<<10;
          AT91C_BASE_PWMC->PWMC_IDR=0x1<<1|0x1<<2;
          AT91C_BASE_PWMC->PWMC_DIS=0x1<<1|0x1<<2;
          AT91C_BASE_PIOA->PIO_ASR=PA1|PA2;
          AT91C_BASE_PIOA->PIO_PDR=PA1|PA2;
          pRSTC->RSTC_RCR = 0xA5000008; //Enable RESET
          pRSTC->RSTC_RMR = 0xA5000001;
          Delay(1000);
          SYSInitFrec(); // Freq init
          SYSInitPeriphery();
          Delay(1000); // Init periphery
          ADCInit();
          f_mount(0, &fsst);
          AT91C_BASE_RSTC->RSTC_RMR = AT91C_RSTC_URSTEN | (0x4<<8) |
(unsigned int)(0xA5<<24);
          AT91C_BASE_AIC->AIC_SPU  = (int) Interrupcao ;
}
//*****************************************************************************
//***********************Funcao para controlar a reproducao******
void Player(char * AfFilename){
//verifica se conseguiu abrir o arquivo corretamente
  if (ArquivoWAV(AfFilename) == TRUE){
          iRead = 1; iPlay = 0; iBuf_1 = 1; iAcabou = 0;
          //configura o PWM de acordo com o numero bits por amostra
          if (iBitsPerSampe == 8){
             PWMInit(256);
          }else{
             PWMInit(1023);
          }
          //inicializa o timer da interrupcao com o valor do sample rate
          InitPIT(iSampleRate);//16500
          while(1){
          //verifca se apertou botao pra cima
            if(!((pPIOA->PIO_PDSR) & BIT9)) {//iPlay
```

```
                while(!((pPIOA->PIO_PDSR) & BIT9)){
                  iPlay = 1;
                }
          }
        //vefirifica se apertou botao pra baixo
          if(!((pPIOA->PIO_PDSR) & BIT8)) {//pause
                while(!((pPIOA->PIO_PDSR) & BIT8)){
                  iPlay = 0;
           }
          }
        //verifica se apertou o botao pra direita
          if(!((pPIOA->PIO_PDSR) & BIT14)) {//next
            while(!((pPIOA->PIO_PDSR) & BIT14)){
                  iPlay = 0; iRead = 1; iTamanhoChunk2 = 0;
                }
            AT91C_BASE_AIC->AIC_IDCR = (1 << AT91C_ID_SYS);
            ReiniciaVetores();
            Avancar();
          }
        //verifica se apertou o botao pra esquerda
          if(!((pPIOA->PIO_PDSR) & BIT7)) {//previous
            while(!((pPIOA->PIO_PDSR) & BIT7)){
                  iPlay = 0; iRead = 1; iTamanhoChunk2 = 0;
                }
            AT91C_BASE_AIC->AIC_IDCR = (1 << AT91C_ID_SYS);
            ReiniciaVetores();
            Atrasa();
          }
          if (iAcabou == TRUE){
                AT91C_BASE_AIC->AIC_IDCR = (1 << AT91C_ID_SYS);
                iTamanhoChunk2 = 0;
                ReiniciaVetores();
                Avancar();
          }
        Read_Block();
        }
  }
  else{//se a musica na reproduzir tocar a proxima
        ReiniciaVetores();
        Avancar();
  }
}
//*************************************************************************
//*******************Programa Principal*********************************
int main(){
//Inicializa o microcontrolador
  Device_Init();
//inicializa o cartão SD e SPI
  power_on();
//abre o diretorio do arquivo
  int res = f_opendir(&dir, "NomeDoDiretorio");
```

```
//conseguiu abrir o diretorio
  if (res == FR_OK){
          Avancar();
          }
}
```