Centro Federal de Educação Tecnológica de Santa Catarina – CEFET-SC

Curso de Pós-Graduação em Desenvolvimento de Produtos Eletrônicos Digitais

Tocador de Músicas Monofônicas de Celular com Detecção de Movimento

Fábio Pereira

Florianópolis

2008

Fábio Pereira

Tocador de Músicas Monofônicas de Celular com Detecção de Movimento

Monografia apresentada ao curso de pós-graduação em desenvolvimento de produtos eletrônicos digitais como parcial à obtenção do título de especialista em desenvolvimento de produtos eletrônicos digitais.

Orientador:

Prof. Me. Francisco Édson Nogueira de Mélo

Florianópolis, SC

2008

Fábio Pereira

Monografia sob o título tocador de músicas monofônicas de celular com detecção de movimento, defendido por Fábio Pereira em 30 de maio de 2008 e aprovado pela banca examinadora constituída conforme abaixo:

_____

Prof. Francisco Édson Nogueira de Melo (Mestre)

_____

Prof. Marco Valério Miorim Villaça (Doutor)

_____

Prof Wilson Berckembrock Zapelini (Doutor)

FLORIANÓPOLIS/SC

2008

CEFET/SC

ATA DE DEFESA

ATA DA DEFESA DE MONOGRAFIA DA ESPECIALIZAÇÃO EM DESENVOLVIMENTO DE PRODUTOS ELETRÔNICOS

Aos 30 dias do mês de maio de 2008, com início às 14:00 e término às 15:30, na Unidade Florianópolis do Centro Federal de Educação Tecnológica de Santa Catarina, teve lugar a sessão pública da defesa de Monografia, sobre o tema

PLAYER DE RINGTONES COM SENSOR DE MOVIMENTO

para a obtenção do certificado de **especialista** do aluno

FÁBIO PEREIRA

A Banca foi constituída pelos seguintes membros: Prof. Msc. Francisco Edson Nogueira de Melo – Orientador, Prof. Dr. Marco Valério Miorim Villaça e Prof. Dr. Wilson Berckembrock Zapelini. O ato teve início com a apresentação da Banca pelo Presidente, Prof. Msc. Francisco Edson Nogueira de Melo que, a seguir, passou a palavra ao aluno para expor o seu trabalho. Na seqüência, os componentes da banca fizeram suas argüições, que foram respondidas pelo aluno. Ao término da defesa, a banca, após deliberação sigilosa, atribuiu o seguinte conceito: A (EXCELENTE) e, à vista desses resultados, o Presidente declarou encerrada a defesa, lavrando-se a presente ata que vai assinada pelos (as) professores (as), membros da banca examinadora, e que será entregue à Coordenadoria do Curso.

Florianópolis, 30 de maio de 2008.

Prof. Msc. Francisco Edson Nogueira de Melo (presidente) _____

Prof. Dr. Marco Valério Miorim Villaça _____

Prof. Dr. Wilson Berckembrock Zapelini _____

Fábio Pereira (aluno) _____

DEDICATÓRIA

Este trabalho é dedicado a todos os estudantes, professores e pesquisadores que atuam na pesquisa e desenvolvimento de produtos e projetos inovadores na área da tecnologia eletrônica.

AGRADECIMENTOS

Agradeço aos meus pais Pedro e Cristina, minha avó Juçá, à minha noiva Mariani, grande incentivadora de todas as horas, à Freescale USA, pela placa de demonstração utilizada nesta aplicação, aos professores e colegas do curso de pós-graduação do CEFET-SC e a todos os amigos e conhecidos pelas sugestões e idéias.

SUMÁRIO

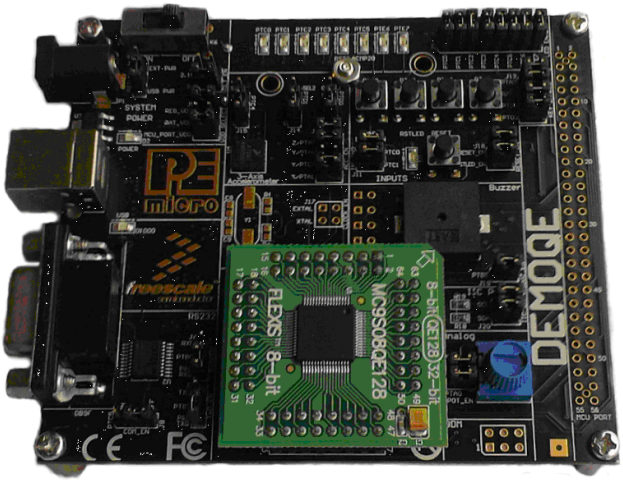
1. INTRODUÇÃO

Este trabalho visa a apresentar a implementação de um dispositivo capaz de executar músicas monofônicas de aparelhos de telefone celular (conhecidas como *ringtones*). Além de ser capaz de tocar as referidas músicas, o dispositivo apresentado inclui uma característica adicional, que é capacidade de detectar movimentos súbitos que são utilizados para provocar o avanço da seqüência de execução. Desta forma, um simples “chacoalhar” do dispositivo em uma determinada direção provoca o avanço da seqüência de execução.

O objetivo do projeto é demonstrar uma alternativa de baixo custo para um *player* ou *jukebox* eletrônico, encontrando aplicação nas áreas de entretenimento (jogos eletrônicos simples), telecomunicações (música de espera para sistemas telefônicos), etc.

O sistema foi implementado em uma placa de demonstração DEMO9S08QE128 da Freescale (baseada no microcontrolador MC9S08QE128). Esta placa inclui um *buzzer* piezoelétrico e um acelerômetro integrado, elementos vitais à aplicação em questão.

Figura 1: Foto do kit de demonstração utilizado na aplicação



2. DESENVOLVIMENTO

A implementação da presente aplicação requer o desenvolvimento de estruturas de código de forma que o microcontrolador realize as seguintes tarefas:

1. Geração de sinais audíveis, com freqüência e duração programáveis;
2. Decodificação das *strings* contendo o conteúdo musical a ser processado;
3. Leitura dos sinais de tensão analógica provenientes do acelerômetro. Estes sinais devem ser analisados em busca do padrão característico do movimento que se deseja detectar.

Antes de detalhar os elementos de *software* da aplicação, vejamos os elementos de *hardware*: o microcontrolador MC9S08QE128 e o acelerômetro MMA7260Q.

A escolha do microcontrolador e do acelerômetro obedeceu a alguns quesitos quais sejam:

1. Velocidade de operação do microcontrolador (especialmente para gerar satisfatoriamente os sinais de som);
2. Disponibilidade de periféricos (*timer* e conversor A/D);
3. Quantidade de memória FLASH disponível (especialmente para o armazenamento das músicas);
4. Quantidade de eixos de sensibilidade (no caso do acelerômetro).

O custo final não foi avaliado no presente caso, mas seria de suma importância em uma aplicação comercial. Ainda assim a presente aplicação, se desprovida do acelerômetro integrado, poderia ser facilmente compilada para outros microcontroladores como o MC9S08QD4, com 4 KiB de FLASH e que custa aproximadamente US$ 1,00. De fato, qualquer dispositivo da família HCS08 da Freescale poderia executar plenamente a aplicação, pois todos atendem aos requisitos mínimos de *hardware*. Somente a quantidade de músicas armazenadas é que iria variar de um dispositivo para outro.

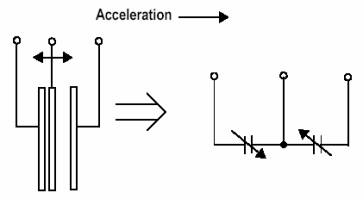
2.1. Acelerômetro MMA7260QT

Um acelerômetro, como o seu próprio nome indica, é um dispositivo capaz de medir acelerações (taxas de variação de velocidade). O acelerômetro utilizado no presente trabalho consiste num dispositivo semicondutor construído através da tecnologia conhecida como MEMS (*Micro Electro-Mechanical System* – sistema micro-eletromecânico).

O princípio de funcionamento de um acelerômetro MEMS é bastante simples: uma massa inercial (chamada *g-cell*) montada na pastilha semicondutora é suspensa por nanomolas. Esta massa movimenta-se diferentemente do restante da pastilha quando o *chip* é submetido a acelerações e desacelerações. Uma estrutura (similar a um pente) é utilizada para detectar a movimentação da massa inercial.

A figura 2 demonstra o funcionamento de um sensor deste tipo. A estrutura comporta-se como dois capacitores, onde as placas das extremidades são conectadas a pastilha do circuito integrado, enquanto que a placa central é conectada à massa inercial (*g-cell*).

Figura 2: Diagrama básico de um acelerômetro integrado



Fonte: *Datasheet* MMA7260QT (2007, p. 4)

Quando o *chip* encontra-se em repouso (nenhuma força atuando sobre a massa inercial), as duas capacitâncias são aproximadamente iguais. Ao submeter o *chip* a uma força (como por exemplo, a força da gravidade ou outro movimento qualquer), a massa desloca-se de forma diferente do restante do *chip*, provocando um desequilíbrio nas capacitâncias. Amplificadores e circuitos de conformação de sinal no interior do *chip* encarregam-se de gerar um sinal de tensão proporcional à aceleração detectada.

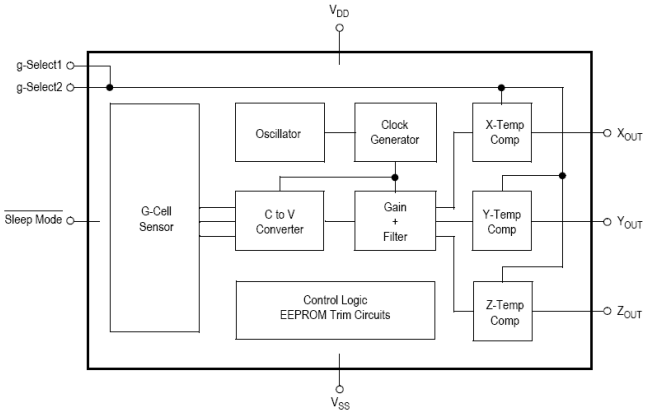
O MMA7260QT, fabricado pela Freescale, consiste em três acelerômetros integrados dentro do mesmo componente. Cada acelerômetro encontra-se montado de forma a medir um eixo do espaço tridimensional (eixo X, Y e Z).

Algumas das características do MMA7260QT:

- Baixa tensão de operação (de 2,2 a 3,6V);
- Baixo consumo de energia (500µA);
- Sensibilidade selecionável (1,5/2/4/6g);
- Alta sensibilidade (aproximadamente 800mV/g na escala de 1,5g);
- Filtros passa-baixa integrados ao circuito interno de condicionamento de sinal;
- Encapsulamento de pequenas dimensões (QFN de 16 pinos).

O diagrama em blocos do dispositivo encontra-se apresentado na figura 3.

Figura 3: Diagrama em blocos do MMA7260QT



Fonte: *Datasheet* MMA7260QT (2007, p. 2)

As faixas de sensibilidade selecionadas pelas entradas g-Select1 e g-Select2 estão apresentadas na tabela 1.

Tabela 1: Faixas de sensibilidade do MMA7260QT

g-select2	g-select1	Faixa g	Sensibilidade (mV/g)
0	0	+/- 1,5g	800
0	1	+/- 2g	600
1	0	+/- 4g	300
1	1	+/- 6g	200

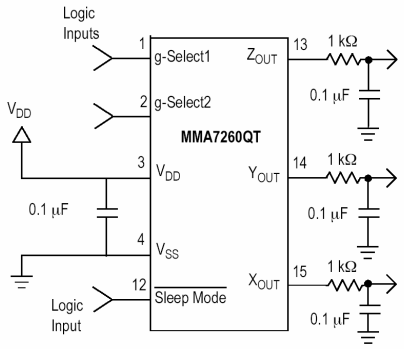
Fonte: *Datasheet* MMA7260QT (2007, p. 4)

Quando operando na escala de 1,5g com uma tensão de alimentação de 3,3V, a tensão de saída de cada eixo (X_{OUT}, Y_{OUT} e Z_{OUT}) é igual a 1,65V quando o respectivo acelerômetro está em repouso (nenhuma força atuando no sentido de deslocamento da massa inercial). Ao submeter um eixo a uma aceleração positiva

de 1g, a tensão de saída aumenta em 800mV, atingindo 2,45V. Da mesma forma, submetendo o eixo a uma aceleração negativa da ordem de -1g, a tensão de saída será igual a 0,85V.

A utilização do acelerômetro é bastante simples: configura-se a sensibilidade (através dos pinos g-Select1 e g-Select2), desativa-se o sinal Sleep Mode e efetua-se a leitura das tensões analógicas nas saídas X_{OUT}, Y_{OUT} e Z_{OUT}. Devido à utilização de filtros capacitivos comutados (*switched capacitor*) nos circuitos de saída do acelerômetro, o fabricante recomenda a montagem externa de filtros passa-baixas do tipo RC conforme demonstra a figura 4.

Figura 4: Conexões recomendadas pelo fabricante



Fonte: *Datasheet* MMA7260QT (2007, p. 5)

O sinal analógico proveniente das saídas X_{OUT}, Y_{OUT} e Z_{OUT} é aplicado aos pinos PTA1, PTA6 e PTA7 respectivamente. Estes pinos correspondem às entradas ADP1, ADP7 e ADP9 do conversor analógico-digital interno de 12 *bits*.

2.2. Microcontrolador MC9S08QE128

O microcontrolador MC9S08QE128 é um membro recente da família HCS08 de microcontroladores de 8 *bits* da Freescale.

Estes microcontroladores incluem uma CPU de 8 *bits* (evolução dos microcontroladores HC05 e HC08) com as seguintes características:

- Arquitetura CISC com instruções de largura variável (1 a 4 *bytes*);
- Velocidades de *clock* de até 50 MHz;
- Instruções e modos de endereçamento otimizados para aplicações escritas em C;

- Otimizações e modos de baixo consumo voltados para aplicações a bateria;
- *Hardware* interno de depuração com facilidades como: *breakpoints* condicionais e complexos, *tracing*, *profiling*, cobertura de código, etc. Este sistema de depuração é totalmente não-intrusivo e transparente à aplicação, permitindo inclusive a monitoração em tempo real das variáveis e conteúdo da memória do microcontrolador, mesmo com a aplicação em execução em velocidade máxima;

Os microcontroladores MC9S08QE128 incluem ainda funcionalidades adicionais como:

- Controlador de memória (MMU) que permite o endereçamento de quantidades de memória maiores que as suportadas pela largura do barramento de endereços (que é de 16 *bits*). A MMU utiliza o conceito de paginação de memória, onde uma área de 16 KiB da área de 64 KiB da CPU endereça uma área especial apontada pelos registradores da MMU. Além disso, a MMU inclui registradores que otimizam acessos seqüênciais e indexados às regiões paginadas da memória. Duas novas instruções (CALL e RTC) foram adicionadas ao conjunto de instruções dos HCS08 para permitir o correto salvamento e recuperação do apontador atual de página;
- Três *timers* de 16 *bits* (TPM1, TPM2 e TPM3). Cada *timer* possui um conjunto de três canais de captura/comparação/PWM, exceto o TPM3 que possui seis canais;
- Conversor A/D de 12 *bits* com 24 canais;
- 54 pinos de E/S (dos quais 17 possuem capacidades de interrupção);

Além destas funcionalidades, os *chips* incluem muitas outras (interfaces SCI, I^2C, SPI, relógio de tempo real, comparadores analógicos, etc.) que não são diretamente utilizadas nesta aplicação e por isso não serão abordadas no presente trabalho.

Das capacidades citadas acima, apenas duas são essenciais à aplicação em questão: o conversor A/D (para a leitura do acelerômetro) e os *timers* (encarregados da geração da temporização necessária à produção dos sinais de áudio).

Vamos iniciar nosso estudo pelo temporizador (*timer*), responsável pela geração da base de tempo do sinal de áudio.

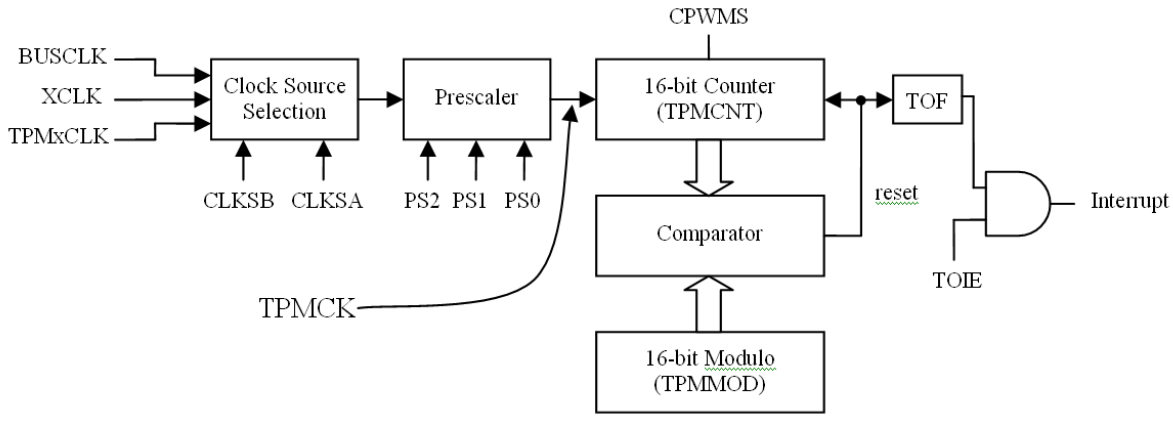
A aplicação apresentada no presente trabalho não visa à obtenção de uma elevada fidelidade quanto à qualidade do sinal de áudio gerado. Ao invés disso, optou-se por um sistema simples e barato onde um *buzzer* piezoelétrico reproduz sinais de áudio gerados através de ondas quadradas que excitam o mesmo fazendo com que vibre na mesma freqüência do sinal excitador. Variando-se a freqüência do sinal excitador podemos variar a freqüência do sinal de áudio.

2.2.1. Temporizador de 16 *Bits* e Geração de Sons

O microcontrolador MC9S08QE128 inclui um conjunto de 4 *timers*, sendo um dedicado a interrupções periódicas (RTC) e três *timers* de uso geral (TPM). O RTC (*Real Time Clock*) consiste basicamente em um contador de 8 *bits* dedicado a tarefas de temporização e geração periódica de interrupções. Os TPMs são *timers* de uso geral de 16 *bits* que podem ser utilizados não só para temporizações, mas também para geração de sinais (inclusive PWM) e medições de período.

A figura 5 apresenta o diagrama em blocos de um temporizador TPM. Como podemos observar, há um circuito de seleção da fonte de *clock*, um prescaler que realiza a pré-divisão do sinal de *clock* selecionado, por um fator programável (entre 1, 2, 4, 8, 16, 32, 64 ou 128), o contador de 16 *bits* (TPMCNT), um comparador digital (que permite configurar a contagem máxima do contador principal) e circuitos de interrupção.

Figura 5: Diagrama em blocos de um temporizador TPM



Fonte: HCS08 Unleashed (2008, p. 236)

A freqüência de incremento do contador principal do TPM (TPMCNT) pode ser determinada através da seguinte fórmula:

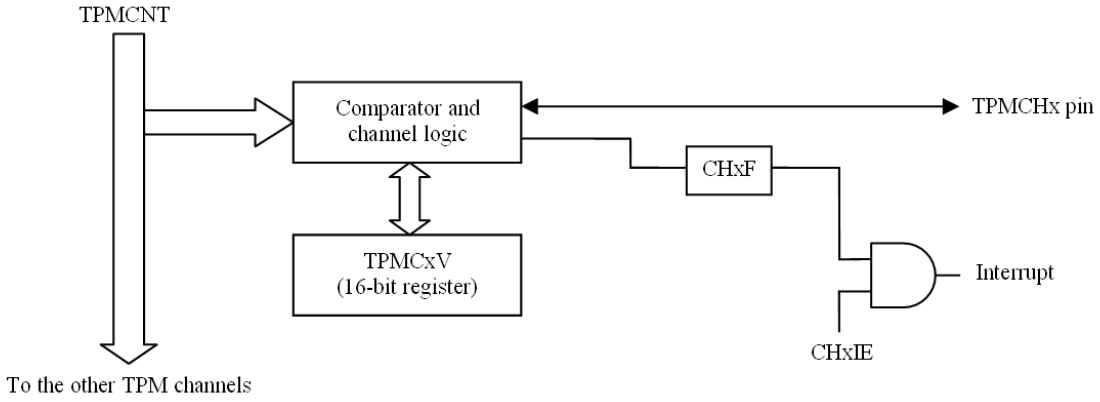
$$F_{TPMCK} = \frac{F_{Source}}{Prescaler}$$

No caso presente, a freqüência da fonte (F_{Source}) é a própria freqüência de barramento (F_{bus}) do MCU, ou seja, 16MHz. Utilizamos um fator de divisão igual a 4 para o *prescaler*, resultando em que a freqüência de incremento do TPM seja igual a 4MHz.

Cada um dos TPMs inclui um conjunto de canais dedicados a tarefas de captura, comparação e geração de PWM. No caso do MC9S08QE128, o TPM1 e TPM2 possuem três canais e o TPM3 possui 6 canais.

A figura 6 apresenta o diagrama em blocos de um canal típico de um TPM, que é composto por um circuito de comparação digital, um registrador de captura/comparação do canal, lógica de controle de pino e lógica de interrupção do canal.

Figura 6: Diagrama em blocos de um canal de um TPM



Fonte: HCS08 Unleashed (2008, p. 239)

A aplicação apresentada no presente trabalho faz uso de um canal de uma unidade TPM para a geração dos sinais de áudio.

Existem duas formas para se gerar uma forma de onda digital através do TPM: configurando-se um canal para geração de um sinal PWM e variando-se a freqüência do sinal, ou através do modo de comparação.

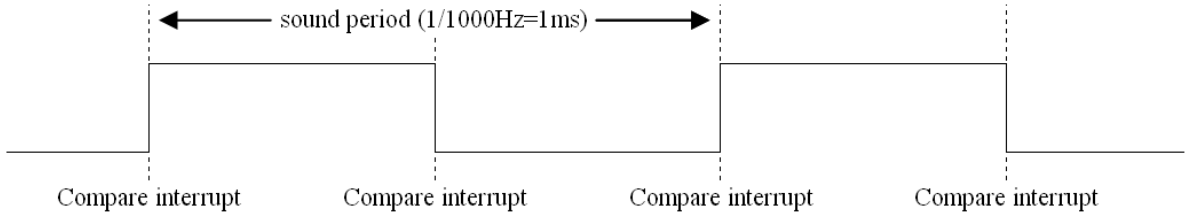
Na presente aplicação, optamos pela utilização do modo de comparação, pois o mesmo permite exercitar as facilidades de comparação do canal além de permitir que se utilize os canais restantes para outras operações.

A utilização do modo de comparação na geração de sinais baseia-se no seguinte princípio: quando operando no modo de comparação, o comparador digital do canal compara continuamente a contagem do contador principal (TPMCNT) com o valor armazenado no registrador de canal (TPMCxV). Ao detectar uma coincidência, um evento é gerado. Este evento pode ser a modificação do estado do pino do canal e/ou a geração de uma interrupção. No presente caso, configuramos o canal de forma que a saída do mesmo tenha o seu estado invertido a cada comparação bem sucedida.

Desta forma, podemos observar que para gerar um sinal periódico com ciclo ativo de 50%, é necessário realizar duas comparações por ciclo, cada uma com um período igual a metade do tempo total de um ciclo do sinal.

A figura 7 ilustra esta situação, demonstrando os eventos relacionados à geração de um sinal de 1kHz. Neste caso, é necessário que ocorra uma comparação a cada 500µs.

Figura 7: Geração de forma de onda com o modo de comparação



Fonte: HCS08 Unleashed (2008, p. 256)

Para produzir comparações como as demonstradas acima, é necessário configurar o canal para que após cada comparação, outra seja agendada para um período igual a metade do período do sinal.

A seguir temos a listagem das funções de interrupção responsáveis pela geração do som e controle da duração do mesmo.

A primeira listagem mostra a função de tratamento da interrupção do canal 0 do TPM 1. Este canal é configurado para operar no modo de comparação, gerando uma

interrupção a cada 1ms. Ele possui duas funções básicas: controlar a duração da nota sendo tocada e controlar a amostragem do acelerômetro.

A duração da nota em execução é controlada através da variável duration_timer. A mesma é automaticamente decrementada pela interrupção até atingir o valor zero (a função de execução da música monitora este evento). A amostragem do acelerômetro é controlada pela variável accel_timer. Foi utilizada uma taxa de amostragem de 50ms para o acelerômetro (20 leituras por segundo).

```
// Interrupção de estouro do TPM1: controla a duração do som
void interrupt VectorNumber_Vtpm1ch0 tpm1ch0_isr(void)
{
  TPM1C0SC_CH0F = 0;                    // apaga o flag de interrupção do TPM1
  TPM1C0V += 4000;                      // próxima comparação em 1ms
  if (duration_timer) duration_timer--; // decrementa o contador de duração se >0
  else                                  // se o contador for igual a zero
  {
    TPM1C1SC = 0;                       // desabilita o canal TPM1CH1 (desliga o som)
    sound_playing = 0;                  // apaga o flag (nenhum som está em execução)
  }
  if (accel_timer) accel_timer--;       // temporizador de amostragem do acelerômetro
  else
  {
    last_y = y_axis;              // amostra o eixo y
    accel_timer = 50;             // próxima amostra em 50ms
  }
}
```

A função de tratamento de interrupção do canal 1 do TPM 1 controla a geração do som propriamente dito. Este canal é configurado para operar no modo de comparação, gerando uma interrupção a cada comparação e invertendo o estado do pino do canal (pino PTB5).

É importante ressaltar a importância da variável channel_reload_value, que contém um valor correspondente à metade do período do sinal de áudio a ser gerado. Este valor é somado ao registrador do canal de forma que uma nova comparação aconteça em um período igual a um semiciclo do sinal.

```
// Interrupção de comparação do canal 1 do TPM1: controla a geração do som
// o estado pino do canal é automaticamente invertido a cada comparação
void interrupt VectorNumber_Vtpm1ch1 tpm1ch1_compare(void)
{
  TPM1C1SC_CH1F = 0;                    // apaga o flag de interrupção
  TPM1C1V += channel_reload_value;      // próxima comparação
}
```

A geração do sinal de áudio é controlada pela função sound() que recebe dois parâmetros de chamada: a freqüência do sinal em Hertz e a duração do som em milisegundos.

A função então calcula o valor de recarga do canal 1 (channel_reload_value), seta a variável de controle de duração (duration_timer) e configura o canal 1 para operação no modo de comparação com interrupção e troca de estado do pino.

```
// Função para geração de som através do canal 1 do TPM1 (pino PTB5): freq em Hz e dur
// em ms
void sound(unsigned int freq, unsigned int dur)
{
  while (sound_playing);      // se um som está tocando, aguarda o seu término
  // o valor de recarga do canal é igual à metade do período do sinal
  channel_reload_value = (4000000/freq)/2;
  duration_timer = dur;       // seta a duração do som
  // configura o canal 1 para o modo de comparação com inversão automática do pino
  TPM1C1V = channel_reload_value; // seta a primeira comparação
  TPM1C1SC = bCHIE | TPM_COMPARE_TOGGLE;
  sound_playing = 1;          // seta o flag de som em execução
}
```

Antes de encerrarmos esta explanação sobre o algoritmo de geração dos sons, é importante apresentar a *array* que contém as freqüências das notas musicais utilizadas no programa.

```
// Array com a freqüência das notas musicais
const unsigned int note[4][12] =
{// C    C#    D     D#    E     F     F#    G     G#    A     A#    B
   262,  277,  294,  311,  330,  349,  370,  392,  415,  440,  466,  494, // 4ª. oitava
   523,  554,  587,  622,  659,  698,  740,  784,  830,  880,  932,  988, // 5ª. oitava
  1047, 1109, 1175, 1244, 1319, 1397, 1480, 1568, 1660, 1760, 1865, 1976, // 6ª. oitava
  2093, 2218, 2349, 2489, 2637, 2794, 2960, 3136, 3320, 3520, 3728, 3951  // 7ª. oitava
};
```

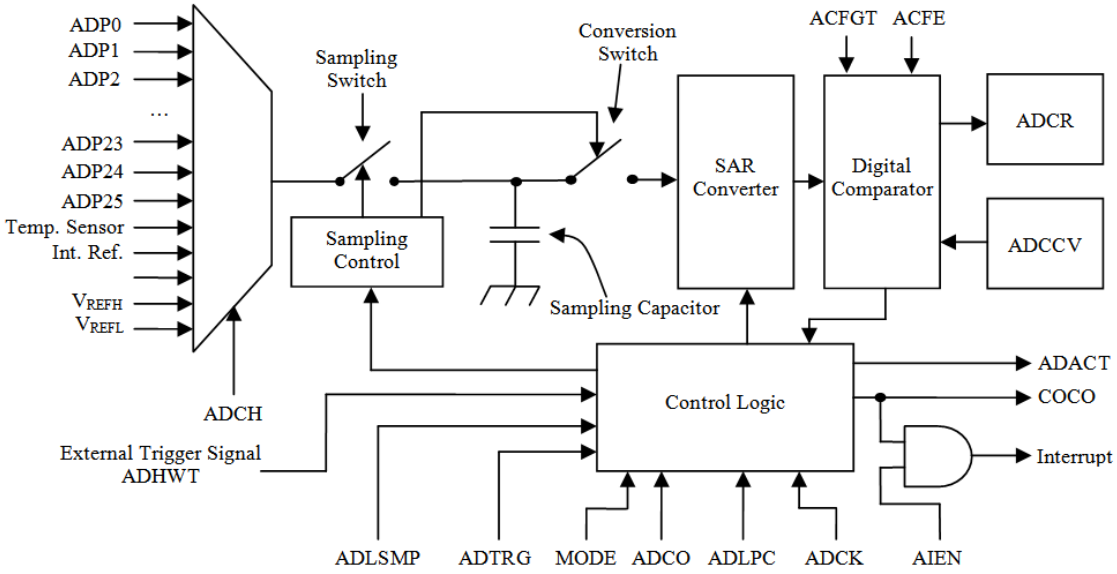
A *array* bidimensional contém as freqüências das diversas notas e oitavas na seqüência. O *parser* RTTTL apresentado na seqüência efetua a leitura e decodificação da música e chama a função sound() passando como parâmetro um índice para a *array* note.

2.2.2. Conversor A/D

Outro módulo importante na implementação da presente aplicação é o conversor analógico/digital ou ADC. O módulo conversor A/D dos microcontroladores MC9S08QE128 consiste num conversor A/D do tipo SAR (*Sucessive Aproximation Register* – registrador de aproximações sucessivas), um circuito de entrada de

amostragem-e-retenção (*sample-and-hold*), um multiplexador analógico de 24 canais, comparador digital e circuitos lógicos de controle e apoio.

Figura 8: Diagrama em blocos do conversor A/D



Fonte: HCS08 Unleashed (2008, p. 269)

O conversor A/D destes microcontroladores é de utilização bastante simples. Para configurá-lo, basta selecionar a fonte de clock, o modo de operação, selecionar o canal desejado e efetuar a leitura do registrador de resultado (ADCR) depois de terminada a conversão (o que é sinalizado pelo bit COCO).

O comparador digital integrado, embora não utilizado nesta aplicação, permite diminuir a carga da CPU, somente gerando interrupções quando os resultados são maiores ou menores que um valor previamente programado.

O conversor é configurado para operar apartir da freqüência de barramento (16MHz) dividida por 16, o que resulta em um clock de 1MHz. Considerando que cada conversão necessita de 40 ciclos de clock do ADC, a taxa de conversão é de 25 mil amostras por segundo (25 ksps).

O código da função de tratamento da interrupção do conversor encontra-se abaixo. Trata-se de uma média aritmética das últimas oito amostras lidas do conversor.

```
// Interrupção do conversor A/D
void interrupt VectorNumber_Vadc adc_isr(void)
{
  y_avg = (y_avg + ADCR) - y_axis;     // faz a média das leituras do acelerômetro
  y_axis = y_avg >> 3;                 // o eixo Y atual é igual a média/8
```

Em seguida é realizada uma comparação da amostra atual de saída do eixo Y e a amostra anterior (lida 50ms antes). Caso seja detectada uma aceleração no sentido –Y precedida de outra maior no sentido +Y, temos a um deslocamento com parada súbita que caracteriza o movimento que desejamos detectar. Neste caso o *flag* de avanço da música é setado.

```
    // se for detectada uma aceleração em +Y seguida por outra em -Y,
    // seta o flag para pular a música
    if (y_axis<800 && last_y>3000) next_song_flag = 1;
}
```

2.3. O Formato RTTTL

A aplicação apresentada neste trabalho utiliza o formato RTTTL (*Ringtone Text Transfer Language* – linguagem de texto para transferência de *ringtones*) para codificação das notas musicais componentes das músicas.

O formato RTTTL foi originalmente desenvolvido pela Nokia para facilitar a personalização e intercâmbio de campainhas musicais monofônicas entre os usuários de aparelhos telefônicos celulares.

Uma música RTTTL é definida através de uma *string* ASCII contendo três partes básicas: título, configuração padrão e notas. As diferentes seções são separadas entre si por dois pontos.

A primeira seção (título) descreve o nome da campainha. O padrão recomenda um tamanho máximo de 10 caracteres (não são permitidos caracteres especiais e de pontuação).

A segunda seção descreve os padrões utilizados pela música. São três os principais padrões definidos pelo formato:

1. Duração: este campo especifica a duração padrão das notas musicais. Os valores possíveis são os seguintes: 1, 2, 4, 8, 16, 32 ou 64 e referem-se ao tempo de duração da nota (entre 1/1 e 1/64 do tempo de uma nota). O especificador da duração obedece ao seguinte formato: d=x, onde x pode assumir um dos valores permitidos. Caso não seja especificada uma duração padrão, adota-se ¼ de duração.

2. Oitava: este campo especifica a oitava padrão utilizada na música. O formato RTTTL permite até quatro oitavas: 4ª. (A4 ou LA = 440 Hz), 5ª. (A5 ou LA = 880 Hz), 6ª. (A6 ou LA = 1760 Hz) ou 7ª. (A7 ou LA = 3520 Hz). O especificador da oitava obedece ao seguinte formato: o=x, onde x pode assumir um dos seguintes valores: 4, 5, 6 ou 7. Caso a oitava padrão não seja especificada, adota-se o=6.
3. Tempo: este campo especifica o número de batidas por minuto (BPM) da música. Com base neste parâmetro é que se calcula a duração de cada nota musical. Por exemplo: uma composição feita para execução a 63 BPM implica que sejam tocadas 63 notas em um minuto, ou seja, cada nota possui uma duração de 60/63 = 0,952 segundos. Caso este parâmetro seja omitido, deve-se adotar o padrão de 63 BPM.

A última seção contém a seqüência de notas que compõem a música. O padrão RTTTL utiliza a notação natural inglesa para representação das notas musicais: c = Dó, c# = Dó sustenido, d = Ré, d# = Ré sustenido, e = Mi, f = Fá, f# = Fá sustenido, g = Sol, g# = Sol sustenido, a = Lá, a# = Lá sustenido e b = Si.

Cada nota pode opcionalmente ser precedida de um número que indica a duração da mesma e seguida por um número que indica a oitava. Quando a duração ou oitava não são informadas, utilizam-se os valores padrões definidos na seção anterior.

Opcionalmente, uma nota pode ainda ser seguida por um ponto, indicando que a mesma deve ter a sua duração aumentada em 50%.

Uma típica melodia RTTTL encontra-se apresentada abaixo:

```
GirlFromIpanema:d=4,o=5,b=160:g.,8e,8e,d,g.,8e,e,8e,8d,g.,e,e,8d,g,8g,8e,e,8e,8d,f,d,d,
8d,8c,e,c,c,8c,a#4,2c
```

Podemos diferenciar as três seções que compõem a melodia:

Nome: GirlFromIpanema (Garota de Ipanema).

Padrões: d=4, o=5 e b=160.

Notas: “g.,8e,8e,d,g.,8e,e,8e,8d,g.,e,e,8d,g,8g,8e,e,8e,8d,f,d,d,8d,8c,e,c,c,

8c,a#4,2c”.

2.3.1. O *Parser* RTTTL

O *parser* RTTTL encontra-se apresentado abaixo. Basicamente o mesmo efetua a leitura de uma *string*, procurando pelas seções e caracteres de controle previstos no formato RTTTL. Inicialmente são lidos os valores padrão da música e em seguida o *parser* varre a composição, executando as notas ou pausas na seqüência.

```
// Esta função toca uma música de acordo com a string RTTTL passada como argumento
void play_song(char *song)
{
  unsigned char temp_duration, temp_octave, current_note, dot_flag;
  unsigned int calc_duration;
  duration = 4;                 // duração padrão = 4/4 = 1 batida
  tempo = 63;                   // tempo padrão = 63 bpm
  octave = 6;                   // oitava padrão = 6th
  while (*song != ':') song++;  // encontra o primeiro ':'
  song++;                       // pula o ':'
  while (*song!=':')            // repete até encontrar outro ':'
  {
    if (*song == 'd')           // se for uma configuração de duração
    {
      duration = 0;             // seta a duração para zero (temporariamente)
      song++;                   // avança para o próximo caractere
      while (*song == '=') song++;  // pula o '='
      while (*song == ' ') song++;  // pula eventuais espaços
      // se o caractere for um número, seta a duração
      if (*song>='0' && *song<='9') duration = *song - '0';
      song++;                   // avança para o próximo caractere
      // se o próximo caractere também for um número, a duração tem dois dígitos
      if (*song>='0' && *song<='9')
      { // multiplica a duração atual por dez e soma o valor do caractere
        duration = duration*10 + (*song - '0');
        song++;                 // avança para o próximo caractere
      }
      while (*song == ',') song++;  // pula a ','
    }
    if (*song == 'o')           // se for uma configuração de oitava
    {
      octave = 0;               // seta temporariamente a oitava para zero
      song++;                   // avança para o próximo caractere
      while (*song == '=') song++;  // pula o '='
      while (*song == ' ') song++;  // pula os espaços
      // se o caractere for um número, seta a oitava
      if (*song>='0' && *song<='9') octave = *song - '0';
      song++;                   // avança para o próximo caractere
      while (*song == ',') song++;  // pula a ','
```

```
  }
  if (*song == 'b')           // se for uma configuração de tempo (batidas por minuto)
  {
    tempo = 0;                // seta temporariamente tempo para zero
    song++;                   // avança para o próximo caractere
    while (*song == '=') song++;  // pula '='
    while (*song == ' ') song++;  // pula os espaços
    // agora lê o tempo (até três dígitos)
    if (*song>='0' && *song<='9') tempo = *song - '0';
    song++;                   // avança para o próximo caractere
    if (*song>='0' && *song<='9')
    {
      tempo = tempo*10 + (*song - '0'); // tempo tem dois dígitos ...
      song++;                 // avança para o próximo caractere
      if (*song>='0' && *song<='9')
      {
        tempo = tempo*10 + (*song - '0'); // tempo tem três dígitos ...
        song++;               // avança para o próximo caractere
      }
    }
    while (*song == ',') song++;  // pula a ','
  }
  while (*song == ',') song++;    // pula a ','
}
song++;                       // avança para o próximo caractere
```

A partir deste ponto, o *parser* passa a buscar as notas musicais:

```
// leitura das notas musicais
while (*song)                 // repete até encontrar um caractere nulo
{
  current_note = 255;         // nota padrão é pausa
  temp_octave = octave;       // seta a oitava atual como a oitava padrão
  temp_duration = duration;   // seta a duração atual como a duração padrão
  dot_flag = 0;               // apaga o flag de detecção do ponto
  // procura um prefixo de nota
  if (*song>='0' && *song<='9')
  {
    temp_duration = *song - '0';
    song++;
    if (*song>='0' && *song<='9')
    {
      temp_duration = temp_duration*10 + (*song - '0');
      song++;
    }
  }
```

A decodificação das notas musicais é realizada pelo *switch* a seguir. A cada nota da escala é atribuído um índice na *array* de notas.

```
  // procura uma nota
  switch (*song)
  {
```

```
  case 'c': current_note = 0; break;    // C (do)
  case 'd': current_note = 2; break;    // D (re)
  case 'e': current_note = 4; break;    // E (mi)
  case 'f': current_note = 5; break;    // F (fa)
  case 'g': current_note = 7; break;    // G (sol)
  case 'a': current_note = 9; break;    // A (la)
  case 'b': current_note = 11; break;   // B (si)
  case 'p': current_note = 255; break;  // pausa
}
song++;                     // avança para o próximo caractere
```

Os intervalos são preenchidos com as notas ditas sustenidas (seguidas do caractere #), como por exemplo, C# (Dó sustenido) que recebe o índice 1 mais adiante no código.

```
// procura por um # seguindo a nota
if (*song=='#')
{
  current_note++;          // incrementa a nota (A->A#, C->C#, D->D#, F->F#, G->G#)
  song++;                   // avança para o próximo caractere
}
```

O ponto “.” indica que a nota deve ter uma duração 50% maior que o valor padrão.

```
// procura por um '.' (estende a duração atual em 50%)
if (*song=='.')
{
  dot_flag = 1;             // se um '.' for encontrado, seta o flag
  song++;                   // avança para o próximo caractere
}
```

Ainda é necessário verificar se a nota contém um sufixo (que indica a oitava da mesma).

```
// procura por um sufixo de oitava
if (*song>='0' && *song<='9')
{
  temp_octave = *song - '0';// seta a oitava temporária
  song++;                   // avança para o próximo caractere
}
if (*song=='.')             // um ponto também pode seguir a oitava (???)
{
  dot_flag = 1;             // se um '.' for encontrado, seta o flag
  song++;                   // avança para o próximo caractere
}
while (*song == ',') song++;    // pula a ','
```

Uma vez decodificada a nota, o *parser* calcula a duração da mesma (baseado na duração especificada e no tempo da música).

```
      // calcula a duração da nota
      calc_duration = (60000/tempo)/(temp_duration);
      calc_duration *= 4;         // uma nota tem duração de quatro batidas
```

Caso um ponto tenha sido detectado para esta nota, a duração é aumentada em 50% (o que equivale a multiplicar a duração por 1,5). Para realizar a multiplicação por 1,5, multiplicamos a duração por 3 e dividimos por 2.

```
      // se o flag de ponto estiver ativo, aumenta a duração da nota em 50%
      if (dot_flag) calc_duration = (calc_duration*3)/2;
```

Utilizamos o valor 255 para indicar que a nota não deve ser tocada e que ao contrário, consiste apenas em uma pausa.

```
      // se a nota não for uma pausa, toca a nota usando a função sound
      if (current_note<255) sound(note[temp_octave-4][current_note],calc_duration);
      else
      { // se for uma pausa, aguarda pela duração especificada
        duration_timer = calc_duration;
        sound_playing = 1;
      }
```

O *flag* sound_playing é setado pela função sound() ao iniciar a geração do som e apagado após a duração especificada. A função play() aguarda este *flag* ser apagado antes de continuar com a execução. Desta forma, evita-se que uma nova nota interrompa uma nota que esteja sendo tocada.

```
      while (sound_playing);      // aguarda o fim da nota/pausa atual
      if (next_song_flag) break;  // se houve movimentação, pula para a próxima música
    }
  }
```

3. CONCLUSÃO

O presente texto apresentou um sistema de *hardware* e *software* bastante simples e eficiente, capaz de executar melodias monofônicas com uma boa qualidade de áudio.

A aplicação foi desenhada de forma a ocupar poucos recursos do microcontrolador (apenas dois canais de comparação, dois pinos de E/S e um canal do conversor A/D) e caso se opte por não utilizar o acelerômetro, os requisitos são ainda menores (dois canais de comparação de timer e um pino de saída).

A ocupação de memória também é pequena, permitindo que a aplicação possa ser executada por microcontroladores de baixo custo com 4KiB ou menos de memória FLASH.

Outro aspecto interessante abordado nesta aplicação é a utilização de um acelerômetro para detectação de padrões simples de movimento. Este é um tema que está em franca ascenção e que deverá estar cada vez mais presente em dispositivos de uso cotidiano.

A capacidade de reproduzir melodias padronizadas segundo o formato RTTTL também é um diferencial da aplicação, pois permite utilizar um enorme acervo de melodias disponíveis para download através da internet.

No que tange à construção da aplicação em si, restam ainda diversas possibilidades de aprimoramento e modificação da mesma:

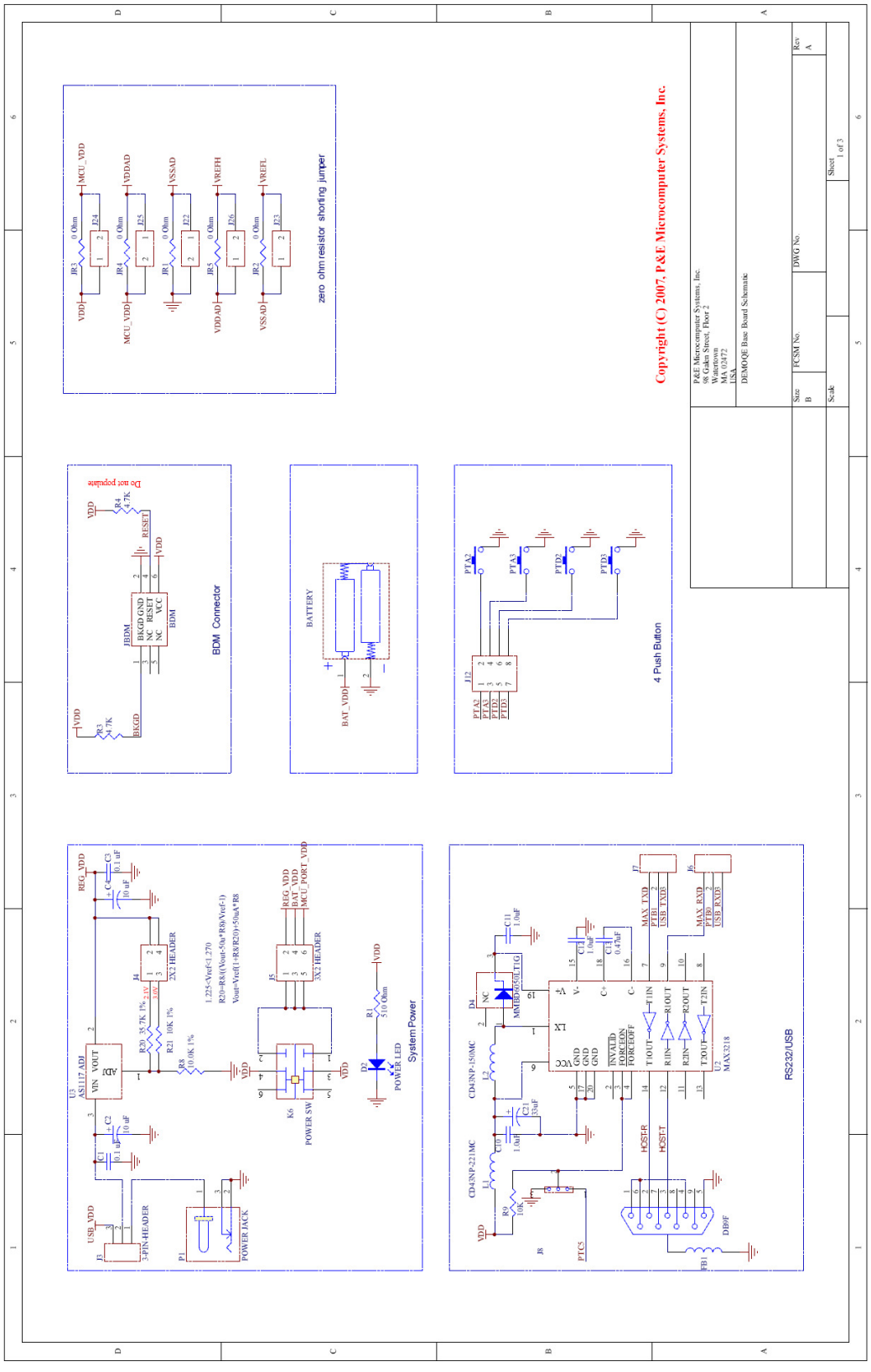
1. Utilização do modo de PWM na geração dos sons: no modo de PWM o timer pode operar de forma mais independente da CPU, pois as comparações de contagem são feitas por hardware;
2. Otimização e modificação do mesmo para utilização de outras plataformas (PICs, MSPs, AVRs, ARMs, etc);
3. Armazenamento das melodias em memória serial externa (I^2C ou SPI): isto permitia armazenar uma grande quantidade de melodias e ainda assim utilizar um microcontrolador pequeno e barato;
4. Implementação de comunicação serial para viabilizar a modificação e troca das melodias armazenadas em memória;

5. Modificação do algoritmo de detecção de movimento para torná-lo menos susceptível a falsas detecções.

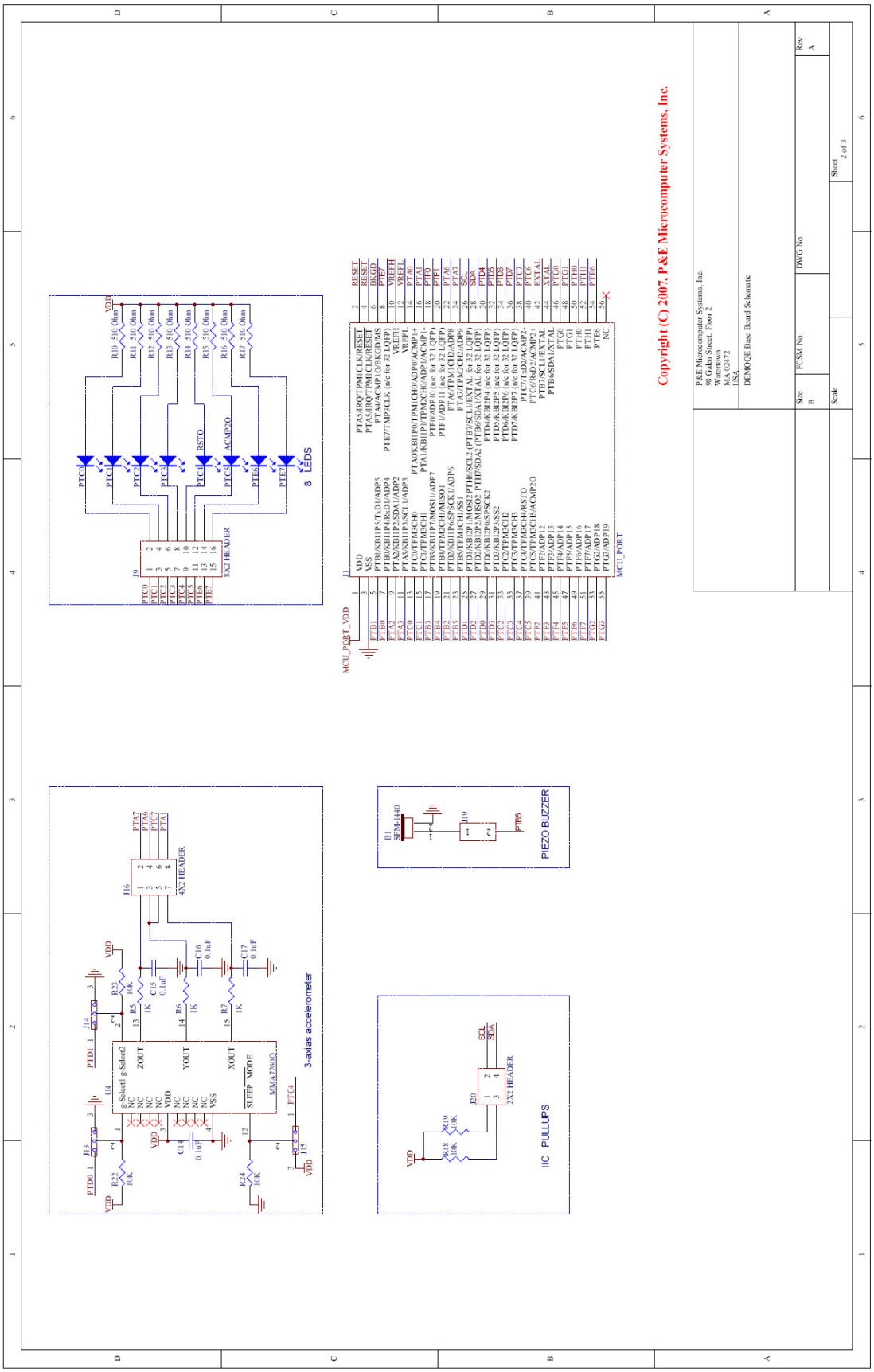
Também devemos registrar que a utilização da linguagem C, apesar do eventual acréscimo no consumo de memória, permite um enorme grau de portabilidade, sem o qual não seria viável (sem um grande esforço) portar a aplicação para outros microcontroladores, tanto dentro da linha HCS08 quanto fora da mesma.

4. ANEXOS

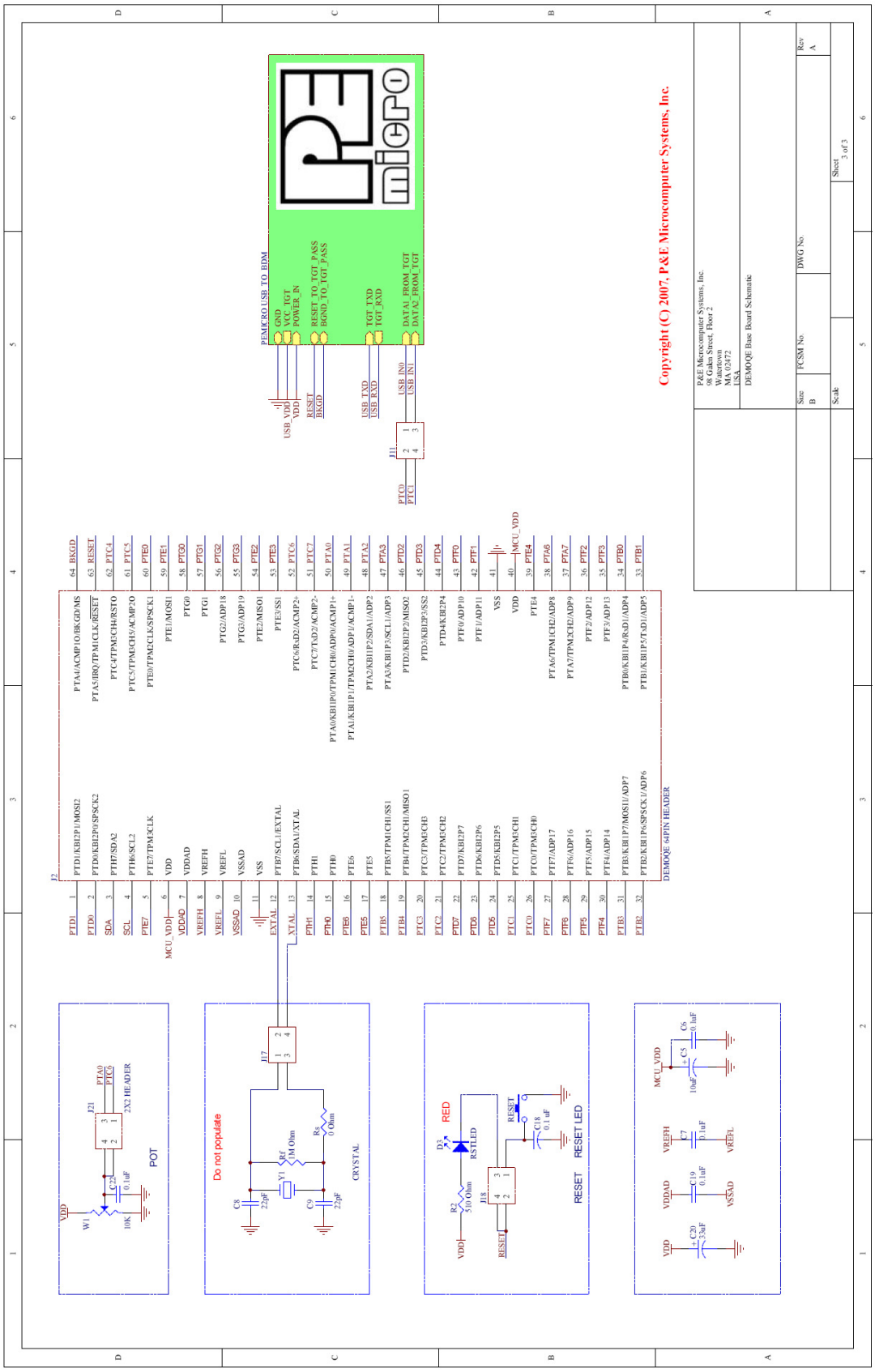
4.1. Esquemático da placa DEMO9S08QE128 (parte 1)



4.2. Esquemático da placa DEMO9S08QE128 (parte 2)



4.3. Esquemático da placa DEMO9S08QE128 (parte 3)



4.4. Listagem do Programa

```
/*
 PTA1 - XOUT
 PTA6 - YOUT
 PTA7 - ZOUT
 PTC4 - MMA7260 seleção do modo sleep (0 = sleep, 1 = operação normal)
 PTD0 - MMA7260 entrada g-select1
 PTD1 - MMA7260 entrada g-select2
 PTB5 - buzzer (saída TPM1CH1)
 ADC:
 - medição do acelerômetro (PTA1, PTA6 e PTA7)
 TPM1:
 - canal 0: controle da duração do som e amostragem do acelerômetro
 - canal 1: geração do som
*/

#include <hidef.h>
#include "derivative.h"
#include "hcs08.h"        // Header especial (do livro HCS08 Unleashed)

#define ACCEL_ON  PTCD_PTCD4

// Músicas RTTTL
const char *rtttl_library[]=
{
{"U2Newyears:d=8,o=6,b=125:a5,a5,c,4a5,a5,a5,a5,c,c,e,4c,c,c,e,e,e,g,4e,e,e,a5,e,e,g,e,e,e,
e,e,a5,a5,c,a5,a5,a5,a5,c,c,e,c,c,c,e,e,e,g,e,e,e,2a5"},
{"MissionImp:d=4,o=6,b=150:16d5,16d#5,16d5,16d#5,16d5,16d#5,16d5,16d5,16d#5,16e5,16f5,
16f#5,16g5,8g5,4p,8g5,4p,8a#5,8p,8c6,8p,8g5,4p,8g5,4p,8f5,8p,8p,8g5,4p,4p,8a#5,8p,8c6,8p,
8g5,4p,4p,8f5,8p,8f#5,8p,8a#5,8g5,1d5"},
{"ET:d=2,o=6,b=200:d,a,8g,8f#,8e,8f#,d,1a5,b5,b,8a,8g#,8f#,8g#,e,1c#7,e,d7,8c#7,8b,8a,8g,f,
d.,16d,16c#,16d,16e,f,d,d7,1c#7"},
{"Batman:d=8,o=5,b=160:16a,16g#,16g,16f#,16f,16f#,16g,16g#,4a.,p,d,d,c#,c#,c,c,c#,c#,d,d,
c#,c#,c,c,c#,c#,d,d,c#,c#,c,c,c#,c#,g6,p,4g6"},
{"Axelf:d=8,o=5,b=160:4f#,a.,f#,16f#,a#,f#,e,4f#,c6.,f#,16f#,d6,c#6,a,f#,c#6,f#6,16f#,e,
16e,c#,g#,4f#."},
{"Hogans:d=16,o=6,b=45:f5.,g#5.,c#.,f.,f#,32g#,32f#.,32f.,8d#.,f#,32g#,32f#.,32f.,d#.,g#5.,
c#,32c,32c#.,32a#5.,8g#5.,f5.,g#5.,c#.,f5.,32f#5.,a#5.,32f#5.,d#.,f#.,32f.,g#.,32f.,c#.,
d#.,8c#."},
{"Jamesbond:d=4,o=5,b=320:c,8d,8d,d,2d,c,c,c,c,8d#,8d#,2d#,d,d,d,c,8d,8d,d,2d,c,c,c,c,8d#,
8d#,d#,2d#,d,c#,c,c6,1b.,g,f,1g."},
{"Pinkpanther:d=16,o=5,b=160:8d#,8e,2p,8f#,8g,2p,8d#,8e,p,8f#,8g,p,8c6,8b,p,8d#,8e,p,8b,
2a#,2p,a,g,e,d,2e"},
{"Countdown:d=4,o=5,b=125:p,8p,16b,16a,b,e,p,8p,16c6,16b,8c6,8b,a,p,8p,16c6,16b,c6,e,p,8p,
16a,16g,8a,8g,8f#,8a,g.,16f#,16g,a.,16g,16a,8b,8a,8g,8f#,e,c6,2b.,16b,16c6,16b,16a,1b"},
{"Adamsfamily:d=4,o=5,b=160:8c,f,8a,f,8c,b4,2g,8f,e,8g,e,8e4,a4,2f,8c,f,8a,f,8c,b4,2g,8f,e,
8c,d,8e,1f,8c,8d,8e,8f,1p,8d,8e,8f#,8g,1p,8d,8e,8f#,8g,p,8d,8e,8f#,8g,p,8c,8d,8e,8f"},
{"TheSimpsons:d=4,o=5,b=160:c.6,e6,f#6,8a6,g.6,e6,c6,8a,8f#,8f#,8f#,2g,8p,8p,8f#,8f#,8f#,
8g,a#.,8c6,8c6,8c6,c6"},
{"Indiana:d=4,o=5,b=250:e,8p,8f,8g,8p,1c6,8p.,d,8p,8e,1f,p.,g,8p,8a,8b,8p,1f6,p,a,8p,8b,
2c6,2d6,2e6,e,8p,8f,8g,8p,1c6,p,d6,8p,8e6,1f.6,g,8p,8g,e.6,8p,d6,8p,8g,e.6,8p,d6,8p,8g,f.6,
8p,e6,8p,8d6,2c6"},
{"GirlFromIpanema:d=4,o=5,b=160:g.,8e,8e,d,g.,8e,e,8e,8d,g.,e,e,8d,g,8g,8e,e,8e,8d,f,d,d,
8d,8c,e,c,c,8c,a#4,2c"},
{"TakeOnMe:d=4,o=4,b=160:8f#5,8f#5,8f#5,8d5,8p,8b,8p,8e5,8p,8e5,8p,8e5,8g#5,8g#5,8a5,8b5,
8a5,8a5,8a5,8e5,8p,8d5,8p,8f#5,8p,8f#5,8p,8f#5,8e5,8e5,8f#5,8e5,8f#5,8f#5,8f#5,8d5,8p,8b,
```

```
8p,8e5,8p,8e5,8p,8e5,8g#5,8g#5,8a5,8b5,8a5,8a5,8a5,8e5,8p,8d5,8p,8f#5,8p,8f#5,8p,8f#5,8e5,
8e5"},
{"She:d=16,o=5,b=63:32b,4c6,c6,c6,8b,8d6,8c6.,8b,a,32b,2c6,p,c6,c6,b,8d6,8c6,8b,a,4c6.,8c6,
b,32c6,4d6,8c6.,b,8a,8g,8f.,e,f,e,32f,2g,8p"},
{"Barbiegirl:d=4,o=5,b=125:8g#,8e,8g#,8c#6,a,p,8f#,8d#,8f#,8b,g#,8f#,8e,p,8e,8c#,f#,c#,p,
8f#,8e,g#,f#"},
{"Entertainer:d=4,o=5,b=140:8d,8d#,8e,c6,8e,c6,8e,2c.6,8c6,8d6,8d#6,8e6,8c6,8d6,e6,8b,d6,
2c6,p,8d,8d#,8e,c6,8e,c6,8e,2c.6,8p,8a,8g,8f#,8a,8c6,e6,8d6,8c6,8a,2d6"},
{"Xfiles:d=4,o=5,b=125:e,b,a,b,d6,2b.,1p,e,b,a,b,e6,2b.,1p,g6,f#6,e6,d6,e6,2b.,1p,g6,f#6,
e6,d6,f#6,2b.,1p,e,b,a,b,d6,2b.,1p,e,b,a,b,e6,2b.,1p,e6,2b."},
{"Autumn:d=8,o=6,b=125:a,a,a,a#,4a,a,a#,a,a,a,a#,4a,a,a#,a,16g,16a,a#,a,g.,16p,a,a,a,a#,4a,
a,a#,a,a,a,a#,4a,a,a#,a,16g,16a,a#,a,g."},
{"Spring:d=16,o=6,b=125:8e,8g#,8g#,8g#,f#,e,4b.,b,a,8g#,8g#,8g#,f#,e,4b.,b,a,8g#,a,b,8a,
8g#,8f#,8d#,4b.,8e,8g#,8g#,8g#,f#,e,4b.,b,a,8g#,8g#,8g#,f#,e,4b.,b,a,8g#,a,b,8a,8g#,8f#,
8d#,4b."},
};

// Array com a freqüência das notas musicais
const unsigned int note[4][12] =
{// C    C#    D     D#    E     F     F#    G     G#    A     A#    B
   262,  277,  294,  311,  330,  349,  370,  392,  415,  440,  466,  494, // 4ª. oitava
   523,  554,  587,  622,  659,  698,  740,  784,  830,  880,  932,  988, // 5ª. oitava
  1047, 1109, 1175, 1244, 1319, 1397, 1480, 1568, 1660, 1760, 1865, 1976, // 6ª. oitava
  2093, 2218, 2349, 2489, 2637, 2794, 2960, 3136, 3320, 3520, 3728, 3951  // 7ª. oitava
};

unsigned int channel_reload_value, duration_timer;
unsigned char accel_timer;
char sound_playing=0;
char duration, octave;
unsigned int tempo;
unsigned int y_axis, y_avg;
char next_song_flag = 0;
unsigned int last_y;

// Interrupção de estouro do TPM1: controla a duração do som
void interrupt VectorNumber_Vtpm1ch0 tpm1ch0_isr(void)
{
  TPM1C0SC_CH0F = 0;                    // apaga o flag de interrupção do TPM1
  TPM1C0V += 4000;                      // próxima comparação em 1ms
  if (duration_timer) duration_timer--; // decrementa o contador de duração se ele for >0
  else                                  // se o contador for igual a zero
  {
    TPM1C1SC = 0;                       // desabilita o canal TPM1CH1 (desliga o som)
    sound_playing = 0;                  // apaga o flag (nenhum som está em execução)
  }
  if (accel_timer) accel_timer--;       // temporizador de amostragem do acelerômetro
  else
  {
    last_y = y_axis;                    // amostra o eixo y
    accel_timer = 50;                   // próxima amostra em 50ms
  }
}

// Interrupção de comparação do canal 1 do TPM1: controla a geração do som
// o estado pino do canal é automaticamente invertido a cada comparação
void interrupt VectorNumber_Vtpm1ch1 tpm1ch1_compare(void)
```

```
{
  TPM1C1SC_CH1F = 0;                    // apaga o flag de interrupção
  TPM1C1V += channel_reload_value;      // próxima comparação
}

// Função para geração de som através do canal 1 do TPM1 (pino PTB5): freq em Hz e dur em ms
void sound(unsigned int freq, unsigned int dur)
{
  while (sound_playing);      // se um som está tocando, aguarda o seu término
  // o valor de recarga do canal é igual a metade do período do sinal
  channel_reload_value = (4000000/freq)/2;
  duration_timer = dur;       // seta a duração do som
  // configura o canal 1 para o modo de comparação com inversão automática do pino
  TPM1C1V = channel_reload_value; // seta a primeira comparação
  TPM1C1SC = bCHIE | TPM_COMPARE_TOGGLE;
  sound_playing = 1;          // seta o flag de som em execução
}

// Esta função toca uma música de acordo com a string RTTTL passada como argumento
void play_song(char *song)
{
  unsigned char temp_duration, temp_octave, current_note, dot_flag;
  unsigned int calc_duration;
  duration = 4;                 // duração padrão = 4/4 = 1 batida
  tempo = 63;                   // tempo padrão = 63 bpm
  octave = 6;                   // oitava padrão = 6th
  while (*song != ':') song++;  // encontra o primeiro ':'
  song++;                       // pula o ':'
  while (*song!=':')            // repete até encontrar outro ':'
  {
    if (*song == 'd')           // se for uma configuração de duração
    {
      duration = 0;             // seta a duração para zero (temporariamente)
      song++;                   // avança para o próximo caractere
      while (*song == '=') song++;  // pula o '='
      while (*song == ' ') song++;  // pula eventuais espaços
      // se o caractere for um número, seta a duração
      if (*song>='0' && *song<='9') duration = *song - '0';
      song++;                   // avança para o próximo caractere
      // se o próximo caractere também for um número, trata-se de uma duração de dois dígitos
      if (*song>='0' && *song<='9')
      { // multiplica a duração atual por dez e soma o valor do caractere
        duration = duration*10 + (*song - '0');
        song++;                 // avança para o próximo caractere
      }
      while (*song == ',') song++;  // pula a ','
    }
    if (*song == 'o')           // se for uma configuração de oitava
    {
      octave = 0;               // seta temporariamente a oitava para zero
      song++;                   // avança para o próximo caractere
      while (*song == '=') song++;  // pula o '='
      while (*song == ' ') song++;  // pula os espaços
      // se o caractere for um número, seta a oitava
      if (*song>='0' && *song<='9') octave = *song - '0';
      song++;                   // avança para o próximo caractere
      while (*song == ',') song++;  // pula a ','
    }
```

```
  if (*song == 'b')           // se for uma configuração de tempo (batidas por minuto)
  {
    tempo = 0;                // seta temporariamente tempo para zero
    song++;                   // avança para o próximo caractere
    while (*song == '=') song++;  // pula '='
    while (*song == ' ') song++;  // pula os espaços
    // agora lê o tempo (até três dígitos)
    if (*song>='0' && *song<='9') tempo = *song - '0';
    song++;                   // avança para o próximo caractere
    if (*song>='0' && *song<='9')
    {
      tempo = tempo*10 + (*song - '0'); // tempo tem dois dígitos ...
      song++;                 // avança para o próximo caractere
      if (*song>='0' && *song<='9')
      {
        tempo = tempo*10 + (*song - '0'); // tempo tem três dígitos ...
        song++;               // avança para o próximo caractere
      }
    }
    while (*song == ',') song++;  // pula a ','
  }
  while (*song == ',') song++;    // pula a ','
}
song++;                       // avança para o próximo caractere
// leitura das notas musicais
while (*song)                 // repete até encontrar um caractere nulo
{
  current_note = 255;         // nota padrão é pausa
  temp_octave = octave;       // seta a oitava atual como a oitava padrão
  temp_duration = duration;   // seta a duração atual como a duração padrão
  dot_flag = 0;               // apaga o flag de detecção do ponto
  // procura um prefixo de nota
  if (*song>='0' && *song<='9')
  {
    temp_duration = *song - '0';
    song++;
    if (*song>='0' && *song<='9')
    {
      temp_duration = temp_duration*10 + (*song - '0');
      song++;
    }
  }
  // procura uma nota
  switch (*song)
  {
    case 'c': current_note = 0; break;    // C (do)
    case 'd': current_note = 2; break;    // D (re)
    case 'e': current_note = 4; break;    // E (mi)
    case 'f': current_note = 5; break;    // F (fa)
    case 'g': current_note = 7; break;    // G (sol)
    case 'a': current_note = 9; break;    // A (la)
    case 'b': current_note = 11; break;   // B (si)
    case 'p': current_note = 255; break;  // pausa
  }
  song++;                     // avança para o próximo caractere
  // procura por um # seguindo a nota
  if (*song=='#')
```

```
    {
      current_note++;           // incrementa a note (A->A#, C->C#, D->D#, F->F#, G->G#)
      song++;                   // avança para o próximo caractere
    }
    // procura por um '.' (estende a duração atual em 50%)
    if (*song=='.')
    {
      dot_flag = 1;             // se um '.' for encontrado, seta o flag
      song++;                   // avança para o próximo caractere
    }
    // procura por um sufixo de oitava
    if (*song>='0' && *song<='9')
    {
      temp_octave = *song - '0';// seta a oitava temporária
      song++;                   // avança para o próximo caractere
    }
    if (*song=='.')             // um ponto também pode seguir a oitava (???)
    {
      dot_flag = 1;             // se um '.' for encontrado, seta o flag
      song++;                   // avança para o próximo caractere
    }
    while (*song == ',') song++;    // pula a ','
    // calcula a duração da nota
    calc_duration = (60000/tempo)/(temp_duration);
    calc_duration *= 4;         // uma nota tem duração de quatro batidas
    // se o flag de ponto estiver ativo, aumenta a duração da nota em 50%
    if (dot_flag) calc_duration = (calc_duration*3)/2;
    // se a nota não for uma pausa, toca a nota usando a função sound
    if (current_note<255) sound(note[temp_octave-4][current_note],calc_duration);
    else
    { // se for uma pausa, aguarda pela duração especificada
      duration_timer = calc_duration;
      sound_playing = 1;
    }
    while (sound_playing);      // aguarda o fim da nota/pausa atual
    if (next_song_flag) break;  // se uma movimentação foi detectada, pula para a próxima música
  }
}

// Interrupção do conversor A/D
void interrupt VectorNumber_Vadc adc_isr(void)
{
  y_avg = (y_avg + ADCR) - y_axis;   // faz a média das leituras do acelerômetro
  y_axis = y_avg >> 3;               // o eixo Y atual é igual a média/8
  // se for detectada uma aceleração em +Y seguida por outra em -Y, seta o flag para pular a música
  if (y_axis<800 && last_y>3000) next_song_flag = 1;
}

void main(void)
{
  unsigned char song_sel;
  SOPT1 = bBKGDPE;                   // habilita o pino de depuração
  ICSSC = DCO_MID | NVFTRIM;         // calibração do FTRIM e DCO na faixa média
  ICSTRM = NVICSTRM;                 // calibração do DCO
  ICSC1 = ICS_FLL | bIREFS;          // seleciona modo FEI (ICSOUT = DCOOUT = 1024 * IRCLK)
  ICSC2 = BDIV_1;                    // ICSOUT = DCOOUT / 1
  // BUSCLK = 16MHz
  TPM1SC = TPM_BUSCLK | TPM_DIV4;    // TPMCK = 4MHz
```

```
  // configura o canal 0 para modo de comparação (somente interrupção), interrupções a cada 1ms
  TPM1C0V = 3999;
  TPM1C0SC = bCHIE | TPM_COMPARE_INT;
  // Configuração do ADC: habilita amostragem longa, modo de 12 bits, ADICLK = 11b, ADCK = ADCLK/8
  // ADC sampling rate = 25ksps
  ADCCFG = bADLSMP | ADC_12BITS | ADC_INTCLK | ADIV_8;
  APCTL2 = ADPC8;                    // ADP8 no modo analógico
  // ativa a conversão do canal 8 no modo contínuo com interrupção ativada
  ADCSC1 = bAIEN | ADCH8 | bADCO;
  PTDD = 0;                          // seleciona sensibilidade de 1,5g
  PTDDD = BIT_0 | BIT_1;             // PTD0 e PTD1 como saídas
  PTBDS = BIT_5;                     // ativa modo drive strengh para o pino PTB5 (buzzer)
  PTCDD = BIT_4;                     // PTC4 como saída
  y_avg = 0;
  y_axis = 0;
  accel_timer = 50;                  // amostragem do acelerômetro
  ACCEL_ON = 1;                      // liga o acelerômetro
  EnableInterrupts;                  // habilita as interrupções (CCR:I=0)
  while(1)
  {
    for (song_sel=0;song_sel<20;song_sel++)
    {
      duration_timer = 250;
      sound_playing = 1;
      while (sound_playing);         // aguarda 250ms antes da próxima música
      play_song(rtttl_library[song_sel]); // toca a música
      // aguarda 2 segundos antes da próxima música
      duration_timer = 2000;
      sound_playing = 1;
      // avança para a próxima música se passados 2 segundos ou movimento detectado
      while (sound_playing && !next_song_flag);
      next_song_flag = 0;
    }
  }
}
```

5. REFERÊNCIAS BIBLIOGRÁFICAS

1. PEREIRA, F. **HCS08 Unleashed: Designer's Guide to the HCS08 Microcontrollers**. USA: Booksurge, 2008.
2. _____. **Microcontroladores HCS08: Teoria e Prática**. Brasil: Érica, 2005.
3. GANSSLE, J. **The Firmware Handbook: The Definitive Guide to Embedded Firmware Design and Applications**. USA: Elsevier, 2004.
4. PREDKO, M. **Handbook of Microcontrollers**. USA: McGraw-Hill, 1999.
5. FREESCALE. **HCS08 Family Reference Manual HCS08RMv1/D Rev.2**. USA: Freescale, 2007.
6. _____. **MC9S08QE128 Data sheet Rev 3**. USA: Freescale, 2007.
7. _____. **MMA7260QT Data sheet Rev 3**. USA: Freescale, 2007.
8. WIKIPEDIA – Enciclopédia Eletrônica Livre – Ring Tone Transfer Language. Disponível em:

   <http://en.wikipedia.org/wiki/Ring_Tone_Transfer_Language>. Acesso em 05 de maio de 2008.
9. BEYONDLOGIC – Generating Ring Tones on your PIC16F87x Microcontroller. Disponível em:

   http://www.beyondlogic.org/pic/ringtones.htm. Acesso em 05 de maio de 2008.