Instituto de Engenharia de Sistemas
e Tecnologia de Informação

Universidade Federal de Itajubá

Estudo e implementação de sistemas de tempo real para microcontroladores de 8 bits.

RELATÓRIO FINAL

PIBIC – FAPEMIG
2011/2012

Aluno: César Augusto Marcelino dos Santos
Matrícula: 15800
Curso: Engenharia da Computação
Orientador: Prof. Msc. Rodrigo Maximiano Antunes de Almeida
Co-Orientador:

Período: Março de 2011 a Fevereiro de 2012

Resumo

O presente trabalho de iniciação científica visa desenvolver um sistema de tempo real (STR) para microcontroladores PIC, especificamente na família 18F.

Inicialmente será realizada uma revisão sobre linguagem C, voltada para sistemas embarcados, e arquitetura de microcontroladores. Em seguida serão estudados os periféricos disponíveis, suas especificações e configurações e inter-relações no desenvolvimento de um sistema de tempo real. A revisão de STR se dará especificamente em sistemas de baixo custo para microcontroladores.

Após a revisão bibliográfica o projeto será encaminhado utilizando um hardware próprio para desenvolvimento do software. Nesta placa será realizado os testes de estabilidade e atuação multitarefa, contemplando um display de LCD, comunicação serial, 4 displays de 7 segmentos e um teclado matricial de 16 teclas. Como objetivo espera-se configurar e instalar corretamente um STR a ser escolhido, na placa, criando, conforme as necessidades, os drivers específicos para os dispositivos em uso.

Este projeto visa aumentar o conhecimento do aluno para sistemas embarcados e o desenvolvimento de uma plataforma de software disponível para futuros projetos.

Palavras – Chave

- Programação Embarcada
- Sistemas de Tempo Real
- Microcontroladores

Sumário

Índice de Figuras

1 Introdução

A programação para sistemas embarcados exige uma série de cuidados especiais, pois estes sistemas geralmente possuem restrições de memória e processamento. Por se tratar de sistemas com funções específicas, as rotinas e técnicas de programação diferem daquelas usadas para projetos de aplicativos para desktop's(BARROS 2002)(PACK 2004).

É necessário conhecer o hardware que será utilizado pois cada microprocessador possui uma arquitetura diferente, com quantidade e tipos de instruções diversos. Este é um dos fatores que mais aumenta a complexidade no desenvolvimento de aplicações para sistemas embarcados.

Por outro lado, os programadores voltados para aplicações em desktop´s possuem uma vantagem significativa: o sistema operacional. Ele opera como um tradutor, realizando a interface entre o código da aplicação e o código de interface do dispositivo, mais conhecido como drivers.

Este trabalho se concentra no desenvolvimento das camadas de "Drivers" e "Sistema Operacional" da Figura 1. A camada de "Aplicação" não será desenvolvida por não ser objeto de estudo deste trabalho.

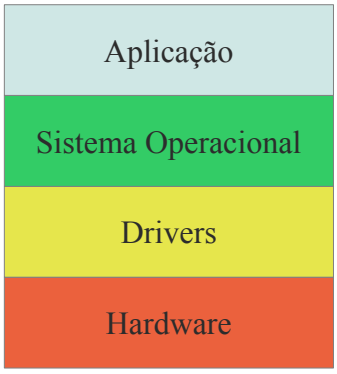


Figura 1: Camadas de abstração de um sistema operacional

A revisão de literatura se concentrará nos conceitos de programação necessários para o desenvolvimento de um kernel de um sistema de tempo real. Neste ponto haverá também uma revisão sobre o "Hardware" disponível na placa de desenvolvimento. Esta revisão é necessária para o desenvolvimento dos drivers.

Para simplificar a implementação do código, o desenvolvimento se dará em duas frentes: o projeto do kernel e o projeto da controladora de drivers. Para que estes operem corretamente, a comunicação será feita através da passagem de mensagens, feitas sob forma de parâmetros de funções.

Por exemplo: a aplicação será executada sobre o kernel, através de processos que serão adicionados dinâmicamente ao sistema. Para acessar o driver a aplicação fará uma requisição ao kernel que, por sua vez, acionará a controladora de drivers. Esta possui uma lista das funções que aquele drive pode ou não executar.

1.1 Objetivos e Justificativa

Este trabalho tem por objetivo a definição e implementação de um sistema de tempo real focado para microcontroladores da família PIC18F da Microchip. Este sistema deve contemplar drivers para os dispositivos disponíveis na placa de desenvolvimento utilizada, de modo que o sistema desenvolvido possa ser utilizado posteriormente como facilitador no desenvolvimento de aplicações.

É necessário também que o sistema apresente um bom nível de segurança, garantindo o correto funcionamento dos dispositivos, sendo capaz de se recuperar de uma situação adversa através da ferramenta *watchdog* disponível no microprocessador. Esta ferramente permite que o sistema consiga retornar ao seu estado normal mesmo quando algum erro de software acontecer como *loops* infinitos, *stack over/underflow* ou até mesmo *deadlocks* provenientes de recursos compartilhados mau dimensionados.

O desenvolvimento deste projeto permitirá ao grupo de engenharia biomédica desenvolver novos projetos de modo mais rápido e com maior segurança. A opção pelo microcontrolador PIC18F4550 se deu devido ao *know-how* existente no grupo, que já desenvolveu vários projetos nos últimos quatro anos em parceria com a empresa de equipamentos médicos FANEN.

Outro fator motivacional para o desenvolvimento deste projeto é a possibilidade de geração de material escrito que possa ser utilizado de referência para futuros estudos nas áreas de sistemas embarcados ou como ponto de partida para aplicações utilizando a plataforma de hardware escolhida.

Dentre os STR *opensource* disponíveis e pesquisados até o momento,

apenas um é compatível com a família de processadores PIC18F4550, o FreeRTOS (BARRY 2010). Apesar disto este não oferece total compatibilidade com o processador, sendo necessário alguns ajustes.

2 Revisão de literatura

2.1 Sistemas Embarcados

Sistemas embarcados são sistemas microprocessados onde o computador é completamente dedicado ao dispositivo ou sistema que ele controla. De modo diferente dos computadores de propósito geral, como PC´s de mesa ou laptops, um sistema embarcado opera apenas um conjunto de tarefas predefinidas, geralmente com pré-requisitos específicos. Já que este é um sistema dedicado a tarefas específicas, pode-se otimizar o projeto reduzindo tamanho, recursos computacionais e custo do produto(OLIVEIRA 2006).

O desenvolvimento de sistemas embarcados é atualmente uma área fundamental dentro da eletrônica pois o custo de sistemas programados já atingiu o mesmo patamar de custo de sistemas dedicados de processamento(ROWE 2002). Deste modo é possível desenvolver produtos com alto grau de personalização com custos extremamente baixos. Uma das maiores vantagens trazidas pelo software embarcado é a facilidade de atualização e correção de erros, reduzindo o tempo de projeto e aumentando a quantidade de subprodutos gerados(BARR 1999).

O problema deste tipo de abordagem é a garantia da segurança, estabilidade e robustez do programa desenvolvido(RUSHBY 2001). Para garantir que o projeto atinja os níveis necessários de segurança, o projetista deve se utilizar de ferramentas de gerenciamento de projeto, baterias de testes e dispositivos de segurança desenvolvidos exatamente para este fim (LUTZ 1993).

No projeto de sistemas dedicados em geral é necessário operar diversas tarefas simultaneamente (PUSCHNER 1990)(CHEN 1990). Entre as arquiteturas disponíveis para multitarefa estão os sistemas de tempo real. Este tipo de arquitetura permite ao projetista um bom desempenho no chaveamento das tarefas a serem executadas e ainda sim garantindo segurança (JAHANIAN 1986).

Com o intuito de reduzir ainda mais o custo dos projetos de sistemas embarcados, diversos fabricantes começaram a desenvolver chips com as unidades de processamento, memória e diversos periféricos num mesmo encapsulamento. Apesar do custo por chip aumentar, no quadro geral ele é reduzido. Isto acontece

por causa da diminuição da área necessária na placa de fenolite, facilidade de roteamento das trilhas e diminuição da quantidade de chips diferentes numa mesma placa.

2.2 Sistemas de Tempo Real

Na medida em que o uso de sistemas computacionais aumenta, as aplicações com requisitos de tempo real tornam-se cada vez mais comuns. Essas aplicações variam muito em sua complexidade e em relação às necessidades de garantia no atendimento de restrições temporais.

Entre os sistemas mais simples, estão os controladores embutidos em utilidades domésticas, tais como lavadoras de roupa e videocassetes. Na outra extremidade estão os sistemas militares de defesa, os sistemas de controle de plantas industriais (químicas e nucleares) e o controle de tráfego aéreo e ferroviário.

Algumas aplicações de tempo real apresentam restrições de tempo mais rigorosas do que outras; entre esses, encontram-se os sistemas responsáveis pelo monitoramento de pacientes em hospitais, sistemas de supervisão e controle em plantas industriais e os sistemas embarcados em robôs e veículos.

Entre aplicações que não apresentam restrições tão críticas, podemos citar os videogames, as teleconferências através da Internet e as aplicações de multimídia em geral. Todas essas aplicações que apresentam a característica de estarem sujeitas a restrições temporais, são agrupados no que é usualmente identificado como Sistemas de Tempo Real(FARINES 2000).

Os sistemas operacionais ou núcleos de tempo real, que gerenciam interrupções e tarefas e permitem a programação de temporizadores e de "timeouts", são, para muitos programadores, suficientes para a construção de sistemas de tempo real(RENAUX 1989).

2.3 Kernel

Em ciência da computação, o kernel é a camada de software de um sistema responsável por implementar a interface e gerenciar o hardware e a aplicação. Os recursos de hardware mais críticos a serem gerenciados são o processador, a memória e os drivers de entrada/saída (I/O), conforme pode ser visto pela Figura 2.

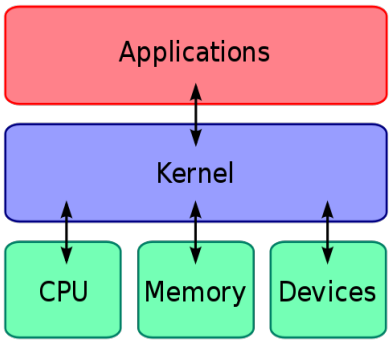


Figura 2: Interfaceamento realizado pelo kernel

Outra função comumente desempenhada pelo kernel é o gerenciamento de processos. Tal tarefa possui maior importância no contexto de sistemas embarcados, no qual, em geral, os processos possuem limitações de tempo de execução.

Quando não há kernel, a responsabilidade de organizar processos, hardware e aplicações é do programador.

Em geral, um kernel possui três principais responsabilidades:

1) Gerenciar e coordenar a execução dos processos através de algum critério

Tal critério pode ser o tempo máximo de execução, prioridade, criticidade de um evento, sequência de execução, entre outros. É ele que diferencia um kernel dito preemptivo (no qual cada processo possui tempo máximo para ser executado. Se este limite for ultrapassado, o processo seguinte é iniciado, e quando este finaliza ou estoura a faixa de tempo, o anterior volta a ser executado no ponto em que foi interrompido) de um cooperativo (cada processo é executado completamente, de modo que um novo processo é chamado somente quando o anterior finalizar). Sendo responsável por gerenciar processos, o kernel deve possuir funções que permitem incluir novos processos ou remover antigos.

Como cada processo utiliza, internamente, uma certa quantidade de memória para suas variáveis, o kernel deve gerenciá-la. Esta é a segunda responsabilidade do kernel.

2) Manusear a memória disponível e coordenar o acesso dos processos a ela

O kernel deve, também, ser capaz de informar ao processo quando uma função de alocação de memória não for executada.

Além da memória, os processos precisam acessar os recursos de entrada e

saída do computador/microcontrolador como portas seriais, displays LCD, teclados, etc. A responsabilidade de permitir ou negar o acesso dos processos aos dispositivos de hardware é do kernel. Esta é a sua terceira responsabilidade.

3) Intermediar a comunicação entre os drivers de hardware e os processos

O kernel deve fornecer uma API (Application Programming Interface, traduzida literalmente como Interface de Programação de Aplicativos) para a qual os processos podem seguramente acessar a informação disponível no hardware, tanto para leitura como para escrita.

Construir o próprio kernel pode facilitar o desenvolvimento das aplicações e ainda assim fornecer ao desenvolvedor total controle sobre a código gerado.

Com uma arquitetura de execução única (single-loop), é necessário re-testar quase tudo a cada vez que se reutiliza o código. Quando o kernel está totalmente testado, não há problema no reuso. Até mesmo aplicações possuem uma maior chance de reaproveitamento em kernels que mantêm a camada de abstração de hardware, independente da mudança de um chip.

Quando se planeja utilizar um kernel no desenvolvimento de um novo sistema, sempre considerar todas as alternativas, tanto pagas quanto gratuitas, mesmo que em opções caseiras sejam escolhidas inicialmente um projeto gratuito como base. Tanto OpenRTOS e BRTOS são muito pequenos (BRTOS possui somente 7 arquivos de código, e somente um é dependente de hardware) e suas licenças são mais flexíveis, de modo que é permitido fechar o código-fonte.

Uma grande fonte de informação é o kernel do Linux (www.kernel.org). São mais de 10kB de linhas de código adicionadas diariamente.

2.3.1 Alternativas

Há muitas opções para mudar de um sistema kernel-less para um com kernel. Soluções pagas possuem benefícios, especialmente pelo suporte técnico oferecido. A seguir, algumas opções e suas descrições.

Windows Embedded Compact® é a versão Windows para pequenos computadores e sistemas embarcados. É um sistema operacional modular de tempo real com um kernel que pode ser executado com menos de 1 MB de memória. Está disponível para arquiteturas de processadores ARM, MIPS, SuperH e x86. O código-fonte está disponível para modificações.

VxWorks® é um sistema operacional de tempo real. Foi portado e otimizado para sistemas embarcados, incluindo a família de processadores x86, MIPS, PowerPC, Freescale ColdFirem Intel i960, SPARC, SH-4 e ARM. Em sua versão compacta (totalmente estática), possui um tamanho de apenas 36 kB.

X RTOS® é um kernel focado principalmente em sistemas aprofundadamente embarcados com severas restrições temporais e de recursos computacionais. Fornece suporte a processadores ARM e PowerPC.

FreeRTOS consiste em um kernel de três ou quatro arquivos na linguagem C (possui poucas funções em linguagem Assembly, inseridas quando necessárias). SafeRTOS é baseado em seu código, mas possui documentação atualizada, testada e auditada para permitir seu uso em aplicações IEC 61508, relacionadas à segurança.

BRTOS é um sistema operacional leve, preemptivo de tempo real, desenvolvido para microcontroladores de pequeno porte. Permite um scheduler preemptivo, semáforos, mutex, caixas e fila de mensagens. Foi escrito, em sua maior parte, na linguagem C, com poucos trechos em Assembly. Há ports para Coldfire V1, HCS08, RX600, MSP430, ATMEGA328/168 e Microchip PIC18. Pode ser compilado para somente 2 kB de memória programada e por volta de 100 bytes de RAM.

2.3.2 Kernel monolítico x microkernel

Em uma das mais famosas discussões a respeito do assunto, Andy Tanenbaum e Linus Torvalds debateram a respeito (TANEMBAUM 2012). O primeiro, defendendo arquiteturas microkernel, exemplificando o sistema MINIX, enquanto o outro se mostrava a favor de uma estrutura monolítica, comentando a respeito do LINUX. As argumentações foram de alta qualidade, analisando diversas nuances necessárias a um bom sistema operacional, mas a conclusão se mostrou como uma diferença de filosofia dos criadores.

Um kernel monolítico é definido com um único arquivo, núcleo, que possui todas as funções e procedimentos necessários para um sistema operar, ou seja, operações sobre dispositivos de entrada/saída, gerenciamento de memória e processos seriam todos realizados por uma única central. Em oposição, o microkernel possui a funcionalidade de distribuir e repassar as tarefas às demais unidades, como controladoras de drivers, ou seja, gerenciar os processos, e quando

estes necessitassem de outro recurso, a unidade responsável seria ativada.

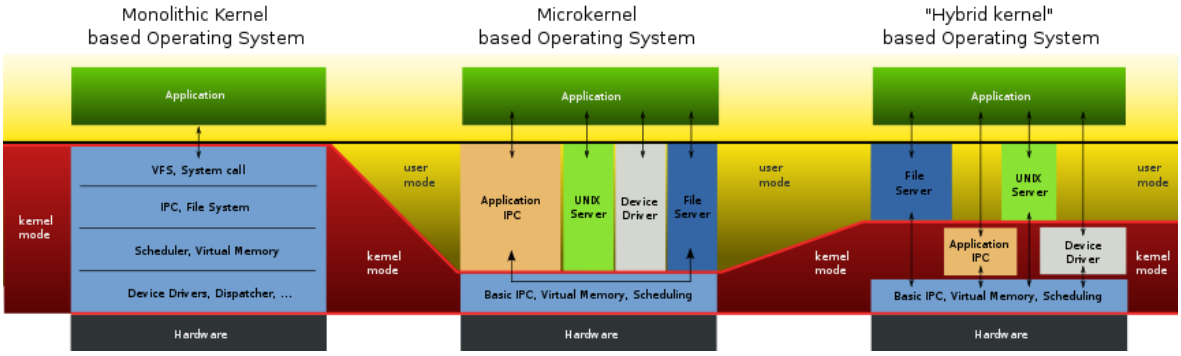


Figura 3: Diferenças entre Kernel Monolítico e Microkernel

A principal diferença entre estas arquiteturas está na quantidade de funções implementadas dentro do espaço do kernel. Utilizando uma visão minimalista, microkernels tendem a utilizar menos recursos da CPU. Pelo fato dos drivers de dispositivos estarem no espaço do usuário, microkernels são menos suscetíveis a erros de drivers. Além disso, é fácil manter um microkernel pelo pequeno tamanho de seu código-fonte, geralmente inferior a 10.000 linhas.

2.4 Conceitos de programação para embarcados

O desenvolvimento do kernel requer um certo conhecimento aprofundado em programação e em questões que interligam hardware e software. Algumas destas questões serão posteriormente explicadas.

2.4.1 Ponteiros de função

Em algumas situações, deseja-se que um programa escolha qual função executar, por exemplo um editor de imagens: utilizar as funções blur (borrar) ou sharpen (aguçar) em uma figura. Declarando as funções:

```
image Blur(image nImg){
    // Implementação da função
}

image Sharpen(image nImg){
    // Implementação da função
}
```

Pode-se construir a engine do editor de imagem:

```
image imageEditorEngine(image nImg, int option){
    image temp;
    switch(option){
        case 1:
            temp = Sharpen(nImg);
        break;
        case 2:
            temp = Blur(nImg);
        break;
    }
    return temp;
 }
```

Para adicionar outras funções, é necessário modificar o código da engine. Em geral, modificar o código significa mais testes e mais erros.

Outra opção é tornar a engine um pouco mais genérica utilizando ponteiros de funções. Sua sintaxe é a seguinte:

```
// definindo um ponteiro de função
typedef tipoDeRetorno (*nomeDoPonteiro)(tipoDoArgumento argumento);

// chamando uma função via ponteiro de função
variavelDeSelecaoDaFuncao = (*nomeDoPonteiro)(argumentoDeEntrada);
```

É uma solução mais refinada do que a utilizada anteriormente. Sendo assim, a nova engine seria:

```
// declaração de ponteiro de função
typedef image (*ptrFunc)(image nImg);

// engine do editor de imagens
image imageEditorEngine(ptrFunc function, image nImg){
    image temp;
    temp = (*function)(nImg);
    return temp;
}
```

A partir do código, pode-se notar que a função recebe um ponteiro de função como parâmetro. Deste modo não é necessário se preocupar em adicionar novas funcionalidades à aplicação, pois o código principal permanecerá intacto. Um dos inconvenientes é que todas as funções agora necessitam da mesma “assinatura”, ou seja, devem receber o mesmo tipo de parâmetro na mesma ordem e a variável de retorno deve ser do mesmo tipo.

Utilizando o conceito de ponteiros de função, é possível utilizar as funções Blur e Sharpen de uma maneira mais fácil:

```
//...
    image nImage = getCameraImage();
    nImage = imageEditorEngine(Blur, nImagem);
    nImage = imageEditorEngine(Sharpen, nImagem);
//...
```

As funções são passadas como se fossem variáveis. Por ser essencialmente um ponteiro, deve-se dereferenciar a variável antes de se utilizar a função:

```
temp = (*function)(nImg);
```

Pode-se também armazenar a função passada como parâmetro como uma variável convencional. Deste modo pode-se chamar a função posteriormente no programa (somente o ponteiro é armazenado; nenhum código é realmente copiado).

A sintaxe de declaração de um ponteiro de função é, de certa forma, complexa. Normalmente se utilizam typedef's para tornar a implementação mais clara.

2.4.2 Structs

Structs são variáveis compostas. Através delas é possível agrupar vários tipos de informação e manipulá-las como se fossem uma única variável. Podem ser comparadas a vetores, mas cada posição pode armazenar um tipo diferente de variável. Um exemplo:

```
typedef struct{
    unsigned short int age;
    char name[51];
    float weight;
}people; // declaração de uma struct

void main(void){
    struct people myself = {26, "Rodrigo", 70.5};

    // utilizando cada variável da struct
    printf("Age:  %d\n", myself.age);
    printf("Name:  %s\n", myself.name);
    printf("Weight: %f\n", myself.weight);

    return 0;
}
```

Para criar um kernel funcional, é necessário agregar mais informação sobre cada processo, e isto será feito através de structs. Por ora, ponteiros de função são suficientes. Quando mais informação for necessária (como um identificador de processor ou prioridade), serão adicionadas posteriormente à struct do processo.

```
// declaração de ponteiro de função
typedef char(*ptrFunc)(void);

// struct do processo
typedef struct {
    ptrFunc function;
} process;
```

Deve-se notar agora que cada processo deve retortar um tipo char, e tal situação será utilizada como condição de retorno, indicando sucesso ou falha.

2.4.3 Buffers circulares

Buffers são espaços de memória com o propósito de armazenar dados temporários, os quais podem ser implementados um simples vetor com dois índices, com um indicando o início da lista e o outro seu final.

O principal problema com esta implementação é definir quando o vetor está cheio ou vazio, pois em ambos os casos, os índices de início e fim apontam para o mesmo lugar.

Há, ao menos, quatro alternativas em como resolver este problema. Para manter a simplicidade do sistema, o último slot sempre estará vazio. Neste caso, se start e end forem iguais, a lista está vazia.

Abaixo, um exemplo de como circular por todo o vetor um número infinito de vezes:

```
#define CB_SIZE 10
int circular_buffer[CB_SIZE];
int index=0;
for(;;){
    // utilizar o buffer para alguma finalidade
    circular_buffer[index] = index;
    // incrementa o índice
    index = (index+1)%CB_SIZE;
}
```

Para adicionar um elemento ao buffer (evitando estouro de faixa), pode-se implementar a função deste modo:

```
#define CB_SIZE 10
int circular_buffer[CB_SIZE];
int start=0;
int end =0;

char AddBuff(int newData){
    // checa se há espaço para inserir um número
    if ( ((end+1)%CB_SIZE) != start){
        circular_buffer[end] = newData;
        end = (end+1)%CB_SIZE;
        return SUCCESS;
    }
    return FAIL;
}
```

2.4.4 Condições de temporização

Na grande maioria dos sistemas embarcados, é necessário garantir que a função será executada com uma certa frequência. Alguns sistemas podem até falhar se estas deadlines não são cumpridas.

Há pelo menos três condições que necessitam ser satisfeitas para que se implemente condições de temporização no kernel:

1. Deve haver um tick (evento de instante) que ocorre com uma frequência precisa;
2. O kernel deve ser informado da frequência de execução necessária para cada processo;
3. A soma da duração dos processos deve se encaixar com o tempo ocioso do processador.

A primeira condição pode ser facilmente satisfeita se houver um timer interno disponível que possa gerar uma interrupção. Tal fato é verdadeiro para a grande maioria dos microcontroladores. Não há necessidade de uma rotina de interrupção dedicada.

Para a segunda condição é necessário que se adicione a informação desejada na struct do processo. Adicionando duas variáveis inteiras, com a primeira indicando o período com o qual haverá repetição (caso retorne REPEAT), e a outra é uma variável interna, na qual o kernel armazena o tempo restante antes de chamar a função.

```
// struct do processo
typedef struct {
    ptrFunc function;
    int period;
    int start;
} process;
```

A terceira condição depende inteiramente do sistema em si. Supondo um sistema cuja função AtualizaDisplay() necessita ser chamada em um intervalo de 5ms. Se este tempo de execução desta função for superior a 5ms, é impossível garantir o intervalo de execução. Outro quesito válido a ser considerado é a respeito do tipo do context switcher, sendo este preemptivo ou cooperativo. Em um sistema cooperativo, o processo deve finalizar sua execução antes que outro possa ser processado pela CPU. Já em um caso preemptivo, o kernel pode parar a execução de um processo a qualquer momento para executar outro processo. Se um sistema não se adequa no tempo disponível, há três opções: trocar o processador por um mais rápido, otimizar o tempo de execução dos processos ou redefinir as frequências necessárias para cada processo.

Quando se está implementando as condições temporais, um problema pode vir à tona. Supondo dois processos P1 e P2. O primeiro é agendado para ocorrer após 10 segundos a partir de agora, e o segundo para daqui a 50 segundos. O temporizador utilizado é não-sinalizado de 16 bits (com valores de 0 a 65.535)

contando em milissegundos, marcando neste instante 45.5 segundos (now_ms = 45.535).

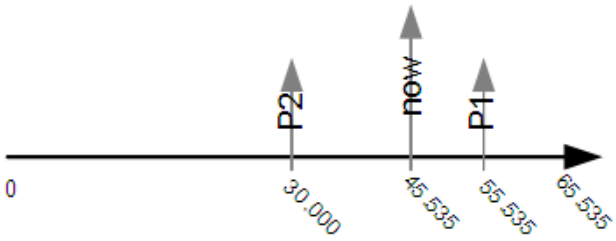


Figura 4: Visualização dos processos em fila, em relação ao tempo

Através da figura, o processo P2 foi corretamente agendado como P2.start = now_ms + 50.000 = 30.000; a variável now_ms será incrementada até o valor 55.535, quando o processo P1 se iniciará (com o correto delay de 10 segundos). A variável now_ms continuará até que se atinja o valor 65.535 e então retorna a zero.

Quando ocorre o estouro de faixa, exatamente 20 segundos se passaram do início (65.535 – 45.535 = 20.000 ms). P2 necessitou de 50 segundos de delay. É necessário aguardar por mais 30 segundos antes que possa ser requisitado, que é exatamente o que acontece quando now_ms atinge o valor 30.000.

O problema para se utilizar um número finito para medir tempo pode aparecer quando dois processos podem ser chamados em um pequeno espaço de tempo, ou mesmo simultaneamente.

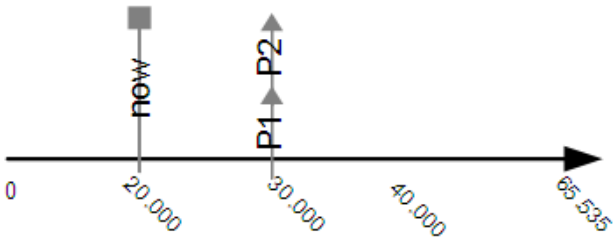


Figura 5: Processos adicionados para iniciarem no mesmo instante

Supondo que agora P1 e P2 estão agendados para ocorrer exatamente em now_ms = 30.000. Se P1 é chamado primeiro e leva 10 segundos para ser executado, tem-se a seguinte linha do tempo:

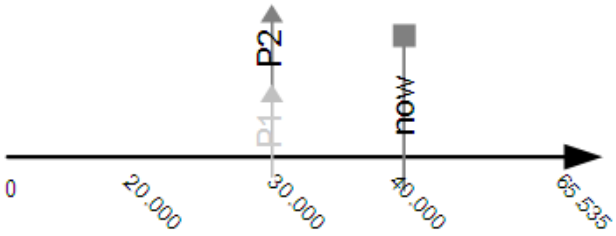


Figura 6: Definindo prioridade de processamento na fila

Questiona-se: a partir da linha do tempo (que é a única informação disponível para o kernel), o processo P2 já foi executado ou foi agendado para ocorrer daqui a 50.535 ms, a partir de agora?

Para resolver esta situação, há duas opções:

1. Criar uma flag para cada processo, indicando quando já foi passado pelo contador de tempo. Deste modo é possível visualizar se um processo anterior ao contador está atrasado ou se foi agendado para ocorrer posteriormente.
2. Criar um contador que decrementa a cada clock do kernel. A execução do processo ocorrerá quando seu contador atingir o valor zero.

A segunda opção gera maior overhead, pois é necessário decrementar todos os contadores dos processos. Por outro lado, se for permitido que o contador assuma valores negativos (de acordo com o que for decrementando), pode-se examinar por quanto tempo o processo está aguardando. Com esta informação pode-se tomar uma providência para se evitar starvation. Uma opção é criar um sistema de prioridade e promover o processo caso permaneça muito tempo aguardando.

2.4.5 Pointeiros para void

Para desenvolver a controladora de drivers de dispositivos, deve-se criar uma "central de distribuição de chamadas", o qual será responsável por receber um pedido da aplicação, via kernel, e redirecioná-la para o correto driver do dispositivo. O problema surge quando se planeja em quantos parâmetros a função deve receber: um representando qual driver é requisitado, outro representando qual a função do driver será chamada e uma quantidade indeterminada de parâmetros necessários a serem passados para o driver. Como criar tal função?

Tal ação pode ser feita através de ponteiros para void.

```
char * name = "Paulo";
double weight = 87.5;
unsigned int children = 3;

void print(int option; void *parameter){
    switch(option){
        case 0:
            printf("%s",*((char*)parameter));
        break;
        case 1:
            printf("%f",*((double*)parameter));
        break;
        case 2:
            printf("%d",*((unsigned int*)parameter));
        break;
    }
}

void main (void){
    print(0, &name);
    print(1, &weight);
    print(2, &children);
}
```

Através do exemplo acima é possível notar como receber diferentes tipos utilizando a mesma função.

2.5 Periféricos disponíveis

2.5.1 Barramento de LEDs

Existe na placa utilizada um barramento de 8 bits, de modo que cada linha possui um LED associado. Este barramento está ligado diretamente com a porta D do microcontrolador e operam sob lógica invertida (são ativos em nível lógico 0).

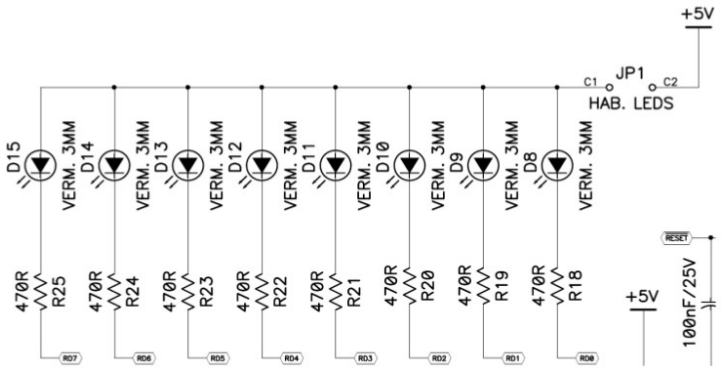


Figura 7: Diagrama elétrico do barramento de leds

2.5.2 Displays de 7 Segmentos multiplexados

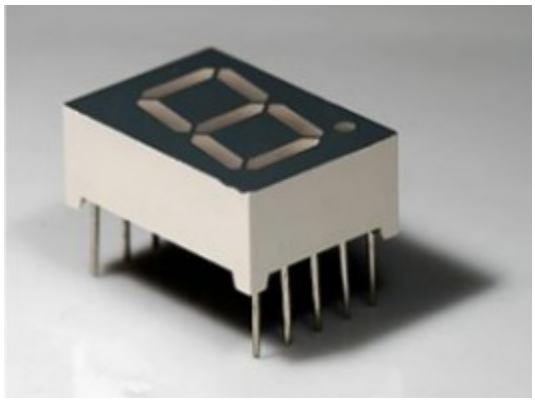


Figura 8: Display de 7 segmentos.

http://commons.wikimedia.org/wiki/File:Seven_segment_02_Pengo.jpg - Peter Halasz

Os displays de 7 segmentos são componentes opto-eletrônicos utilizados para apresentar informações para o usuário, comumente utilizados em marcadores de tempo (relógios, cronômetros, entre outros). Estes displays podem gerar dez algarismos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Além destes é possível representar algumas letras de modo não ambíguo: as maiúsculas A, C, E, F, H, J, L, P, S, U, Z e as minúsculas: a, b, c, d, h, i, n, o, r, t, u.

Os displays podem ser do tipo catodo ou anodo comum, indicando qual o tipo de ligação dos LEDs. Na placa de desenvolvimento utilizada são encontrados 4 displays de catodo comum. Na Figura 9 pode-se visualizar o esquema elétrico e a disposição física de cada LED no componente.

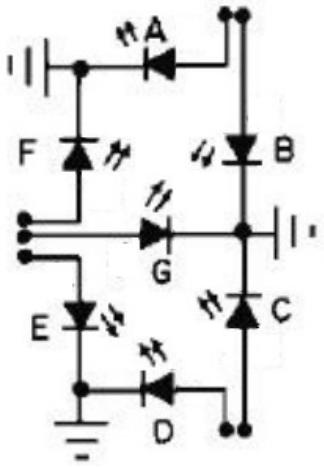


Figura 9: Diagrama elétrico para display de 7 segmentos com anodo comum.

http://www.hobbyprojects.com/the_diode/seven_segment_display.html

Pela figura nota-se que, para se exibir o número 2 no display é necessário acender os leds a, b, g, e, d. Se for utilizado um display com catodo comum, é necessário colocar um nível alto para ligar o LED, ou seja, liga com valor 1 (um) e desliga com valor 0 (zero). Na tabela da Figura 10 são apresentados os valores em binário e em hexadecimal para cada representação alfanumérica. Dentre as letras disponíveis estão apresentadas apenas os caracteres A, b, C, d, E, F. Estes foram escolhidos por serem os mais utilizados para apresentar valores em hexadecimal nos displays. Neste trabalho será utilizada a ordem direta apresentada.

	Ordem inversa								Ordem direta							
Display	a	b	c	d	e	f	g	Hex (0abcdefg)	g	f	e	d	c	b	a	Hex (0gfedcba)
0	1	1	1	1	1	1	0	7E	0	1	1	1	1	1	1	3F
1	0	1	1	0	0	0	0	30	0	0	0	0	1	1	0	06
2	1	1	0	1	1	0	1	6D	1	0	1	1	0	1	1	5B
3	1	1	1	1	0	0	1	79	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	33	1	1	0	0	1	1	0	66
5	1	0	1	1	0	1	1	5B	1	1	0	1	1	0	1	6D
6	1	0	1	1	1	1	1	5F	1	1	1	1	1	0	1	7D
7	1	1	1	0	0	0	0	70	0	0	0	0	1	1	1	07
8	1	1	1	1	1	1	1	7F	1	1	1	1	1	1	1	7F
9	1	1	1	1	0	1	1	7B	1	1	0	1	1	1	1	6F
A	1	1	1	0	1	1	1	77	1	1	1	0	1	1	1	77
b	1	0	1	1	1	1	0	5E	1	1	1	1	1	0	0	7C
C	1	0	0	1	1	1	1	4F	0	1	1	1	0	0	1	39
d	0	1	1	1	1	0	1	3D	1	0	1	1	1	1	0	5E
E	1	0	0	1	1	1	1	4F	1	1	1	1	0	0	1	79
F	1	0	0	0	1	1	1	47	1	1	1	0	0	0	1	71

Figura 10: Conversão binário-hexadecimal para display de 7 segmentos

A seguir, o esquema elétrico das conexões de cada display na placa.

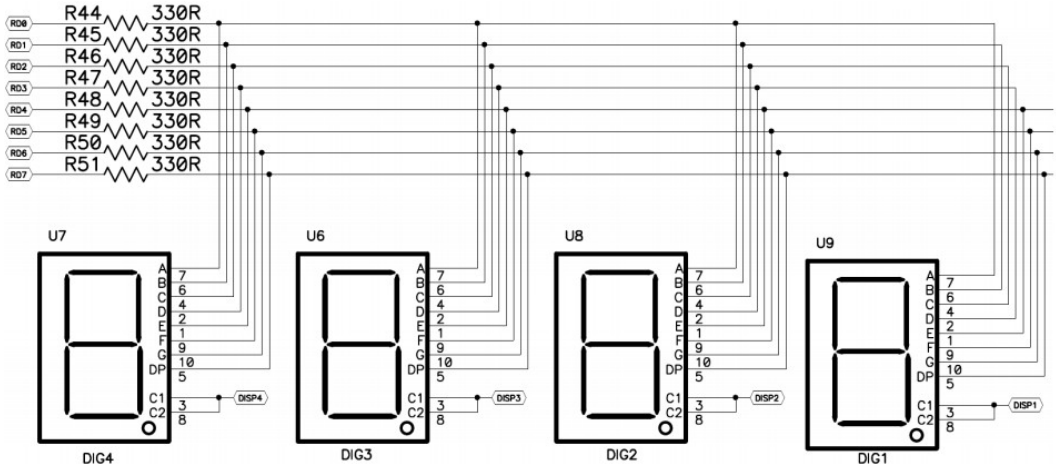


Figura 11: Ligação de 4 displays de 7 segmentos multiplexados

O esquema elétrico mostra que os leds a, b, c, d, e, f, g e dp (*decimal point*), de cada display estão conectados à porta D do microcontrolador. As linhas de catodo comum e as usadas para desativar e ativar os displays (*Disp1*, *Disp2*, *Disp3 e Disp4*) estão conectadas respectivamente aos pinos RA5, RA2, RE0 e RE2.

Cada display exige 7 ou 8 terminais de controle, caso também seja utilizado o ponto decimal. Para utilizar 4 displays, por exemplo um relógio com dois dígitos para horas e dois para minutos, precisaríamos de 32 terminais de saída, o que pode ser um custo muito alto para o projeto.

Uma técnica que pode ser utilizada é a multiplexação dos displays. Esta técnica leva em conta um efeito biológico denominado percepção retiniana. O olho humano é incapaz de perceber mudanças mais rápidas que 1/30 (s). Outro fator importante é que as imagens mais claras ficam gravadas na retina devido ao tempo que leva para sensibilizar e dessensibilizar as células (bastonetes). Deste modo, é possível ligar e desligar rapidamente o display, mantendo a imagem contínua na retina. Ao ligar cada display, um por vez e sequencialmente, de maneira suficientemente rápida, causa a impressão que todos estão ligados. A frequência de chaveamento deve ser mais rápida que 30Hz.

A Figura 11 apresenta o circuito com 4 displays multiplexados, de modo que os terminais iguais estão ligados juntos e os terminais de catodo comum estão, cada um, ligados a uma saída diferente. Com esta arquitetura, reduz-se a quantidade de terminais necessários de 32 para 12, uma economia significativa. Porém, há um custo: o sistema se torna mais complexo pois não é possível ligar dois displays ao mesmo tempo.

2.5.3 Driver de Display LCD

O display de LCD utilizado possui duas linhas por 16 colunas de caracteres, compatível com HD44780 (HITACHI 2012), que é um padrão muito utilizado na indústria.

Figura 12: Display Alfanumérico LCD 2x16

Possuem 14 ou 16 terminais, dependendo se existe ou não retro-iluminação (*backlight*), como observado na tabela abaixo:

Pino	**Função**	**Pino**	**Função**	**Pino**	**Função**
1	Terra	7	Bit de Dados 0	12	Bit de Dados 5
2	VCC (+5 V)	8	Bit de Dados 1	13	Bit de Dados 6
3	Ajuste do contraste	9	Bit de Dados 2	14	Bit de Dados 7
4	Seleção de registro (RS)	10	Bit de Dados 3	15	Backlight VCC
5	Leitura/Escrita (RW)	11	Bit de Dados 4	16	Backlight Terra
6	Clock, Habilitar (EN)				

Este tipo de display possui, integrado, um microcontrolador para realizar as rotinas de manutenção do estado do display, controle da luminosidade e interface com o restante do sistema eletrônico. A comunicação é realizada através de um barramento paralelo de 8 bits, por onde são enviados os dados/comandos.

O terminal RS indica ao display se o barramento contém um comando a ser executado (0) ou uma informação para ser exibida (1). O terminal RW indica ao display se ele receberá um valor (0) ou se estamos requisitando alguma informação (1). O terminal EN indica ao display que ele pode ler/executar o que está no barramento de dados.

As informações são enviadas através da codificação ASCII, sendo que os caracteres de 0 a 127 são padronizados. Os caracteres de 128 a 255 dependem do fabricante do display, de acordo com a ROM utilizada pelo mesmo. É possível também criar algumas representações, símbolos definidos pelo usuário e armazenar na memória interna do display.

Figura 13: Caracteres disponíveis para ROM A00.

http://www.sparkfun.com/datasheets/LCD/HD44780.pdf - Datasheet Hitachi

Figura 14: Caracteres disponíveis para ROM A02.

http://www.sparkfun.com/datasheets/LCD/HD44780.pdf - Datasheet Hitachi

Os comandos reconhecidos pelo display são apresentados:

Instrução	RS	RW	Barramento de dados (bit)								Tempo
			7	6	5	4	3	2	1	0	
Limpa todo o display e configura o endereço para 0.	0	0	0	0	0	0	0	0	0	1	37 us
Configura o endereço para 0. Retorna o display para o início se houve alguma operação de shift.	0	0	0	0	0	0	0	0	1	-	1.52 ms
Configura a movimentação do cursor e o modo de shift do display	0	0	0	0	0	0	0	1	ID	S	37 us
Configura o display (D) inteiro para desligado ou ligado, cursor (C) ligado ou desligado e "blinking" (B) do cursor.	0	0	0	0	0	0	1	D	C	B	37 us
Move o cursor e/ou o display sem alterar o conteúdo	0	0	0	0	0	1	SC	RL	-	-	37 us
Configura o tamanho da palavra (DL), número de linhas (N) e fonte dos caracteres (F)	0	0	0	0	1	DL	N	F	-	-	37 us
Desloca o cursor para a posição desejada: linha e coluna.	0	0	1	X	0	0	Coluna				37 us
Verifica se o display está disponível ou se esta ocupado com alguma operação interna.	0	1	BF	-	-	-	-	-	-	-	10 us

Definições das opções

ID: 1 – Incrementa, 0 – Decrementa	N: 1 – 2 linhas, 0 – 1 linha
S: 1 – O display acompanha o deslocamento	F: 1 – 5x10 pontos, 0 – 5x8 pontos
SC: 1 – Desloca o display, 0 – Desloca o cursor	BF: 1 – Ocupado, 0 – Disponível
RL: 1 – Move para direita, 0 – Move para esquerda	X: 1 – 2a linha, 0 – 1a linha
DL: 1 – 8 bits, 0 – 4 bits	Coluna: nible que indica a coluna

Figura 15: Lista de comandos aceitos pelo LCD.

http://www.sparkfun.com/datasheets/LCD/HD44780.pdf - Datasheet Hitachi (modificado)

Os terminais de dados estão ligados à porta D, juntamente com o display de 7 segmentos e o barramento de LED's. Para estes dispositivos funcionarem em conjunto é necessário multiplexá-los no tempo. Os terminais de controle estão ligados à porta E conforme o esquema apresentado na Figura 16.

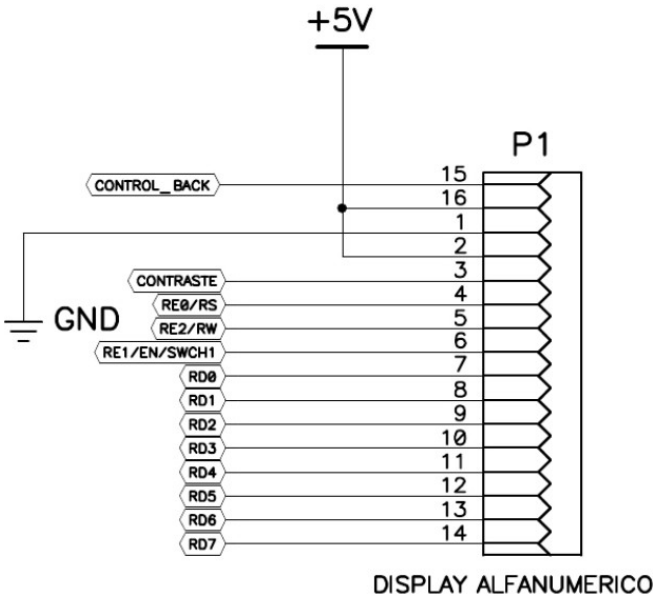


Figura 16: Esquemático de ligação do display de LCD

2.5.4 Conversor Análogo-Digital

Um conversor de sinal analógico para digital é um circuito capaz de transformar um valor de tensão em uma informação binária. O circuito utilizado possui precisão de 10 bits, ou seja, é capaz de sentir uma variação de praticamente um milésimo de excursão máxima do sinal, mais precisamente, 2^{10} ou 1024 valores diferentes.

A conversão A/D é muito utilizada na leitura de sensores. Todo sensor é baseado em um transdutor, sendo este um elemento capaz de transformar um tipo de grandeza em outro, como energia elétrica em luminosa, assim como uma lâmpada. Para a eletrônica, o interesse está na utilização de transdutores cuja saída seja uma variação de tensão, corrente ou resistência, sendo um simples exemplo o potenciômetro, cujo ângulo de rotação influencia em seu valor de resistência.

Outro exemplo de sensor é o LM35 (Figura 17), utilizado para monitoramento de temperatura. Abaixo é apresentado seu diagrama de blocos. O diodo é utilizado como unidade sensora de temperatura. Quando polarizado através de uma corrente constante, havendo mudança de temperatura, a tensão sobre o diodo muda. Os dois amplificadores e as respectivas realimentações estão inseridas no circuito para amplificar e estabilizar as variações de tensão.

Figura 17: Circuito integrado LM35

Para ajustar a precisão de um sinal digital advindo de uma conversão, a utilização de comparadores é a abordagem mais simples. Cada um possui um nível diferente de tensão de referência. Estes níveis são escolhidos de forma que a representação binária faça sentido. Por exemplo, utilizando 2 bits para uma variação de sinal analógico de 0 a 5 Volt, gera-se a seguinte tabela:

Representação binária com 2 bits	Valor em tensão
00	0.625v
01	1.875v
10	3.125v
11	4.375v

O circuito eletrônico responsável pelas comparações pode ser visualizado na Figura 18, sendo ele conhecido como conversor análogo-digital tipo flash, pois cada nível de tensão possui seu próprio comparador.

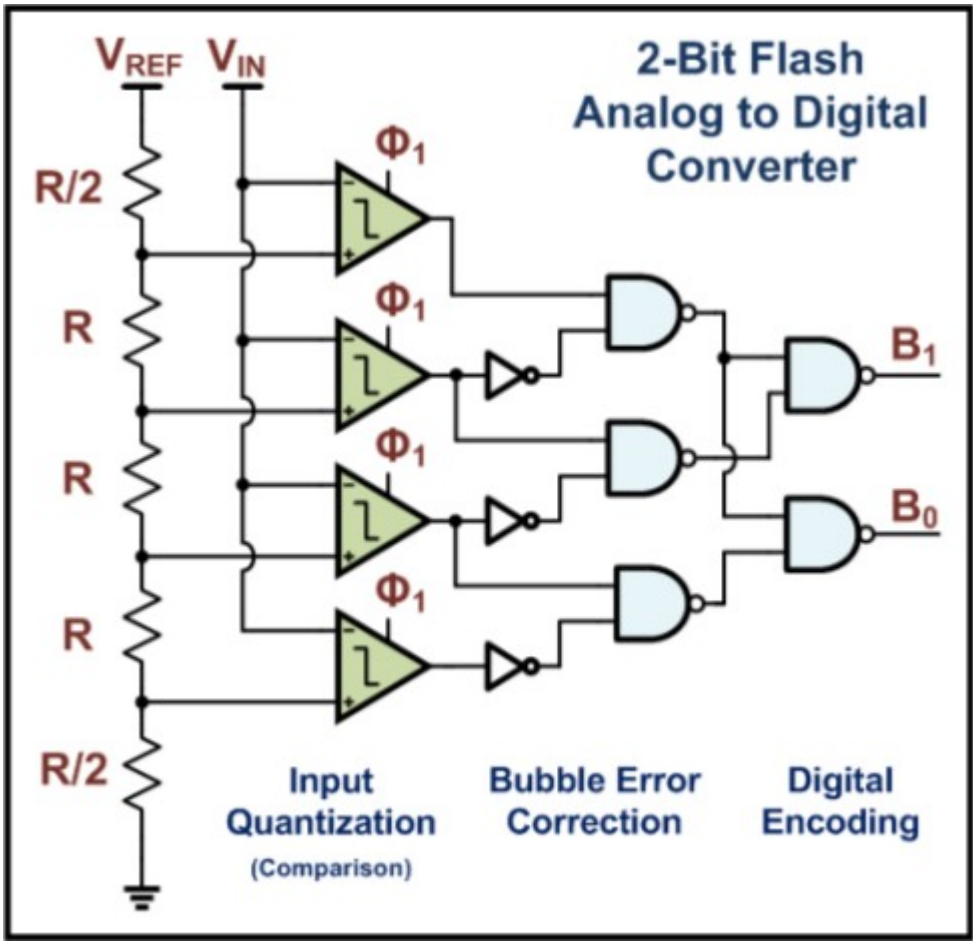


Figura 18: Conversor análogo-digital de 2 bits.

http://en.wikipedia.org/wiki/File:Flash_ADC.png - Jon Guerber

2.5.5 Comunicação Serial

Em geral a comunicação entre dois dispositivos eletrônicos é realizada de modo serial, isto é, as informações são passadas bit a bit, do transmissor para o receptor. Este tipo possui algumas vantagens em relação à comunicação paralela, na qual a palavra (*byte*) é enviada inteiramente, e não fracionada.

A primeira vantagem é a simplificação do hardware, pois os dados são enviados um a um, sendo possível montar um sistema com apenas um fio de comunicação. A segunda é a maior taxa de transmissão, o que à primeira vista é inconsistente, pois com um mesmo ciclo de *clock* a comunicação paralela envia mais de um bit, enquanto a serial apenas um. Este fato acontece para frequências muito altas, possivelmente gerando atraso entre um fio e outro se os cabos da comunicação paralela possuírem qualquer diferença. Além disso, existe o problema do *crosstalking*, de modo que o campo magnético gerado por um cabo induz uma

pequena tensão no outro, impedindo uma correta comunicação. Estes problemas aumentam com a frequência, limitando assim a máxima transferência possível pelo barramento paralelo. Este é o motivo que levou os projetistas de hardware a desenvolverem o protocolo SATA, em detrimento ao IDE/ATA, para comunicação entre o HD e a placa mãe conforme pode ser visto na Figura 19.

Protocolo	*Taxa (Mbit/s)*	*Taxa (Mb/s)*
ATA 33 (Ultra DMA)	264	33
ATA 66 (Ultra DMA)	528	66
ATA 100 (Ultra DMA)	800	100
ATA 133 (Ultra DMA)	1064	133
Serial ATA (SATA-150)	1200	150
Serial ATA 2 (SATA-300)	2400	300
Serial ATA 3 (SATA-600)	4800	600

Figura 19: Taxas de transmissão para diferentes protocolos

O protocolo de comunicação serial mais conhecido é o RS232 (*Recommended Standard* 232), muito utilizado para comunicação entre dispositivos que transmitem ou recebem pouca quantidade de informações. É um dos protocolos mais antigos, utilizado pela primeira vez em 1962 para máquinas de escrever eletromecânicas. O padrão RS232 revisão C é datado de 1969. Em 1986 aparece a revisão D pela EIA (*Electronic Industries Alliance*). A versão atual do protocolo é datada de 1997 pela TIA (*Telecommunications Industry Association*), chamada de TIA-232-F.

O procedimento de envio de um valor pela serial através do padrão RS232 pode ser visto como uma operação de *bit-shift*. Por exemplo, a letra K: em ASCII é codificada como 76_{10} e em binário como 11010010_2. Antes de iniciar a transmissão dos bits, é enviado um bit de começo, indicando que haverá transmissão a partir daquele instante. Após isso o bit menos significativo é enviado para a saída do microcontrolador, operação realizada pela maioria dos dispositivos. Realiza-se então um *shift* para a direita e o novo bit menos significativo é enviado. Esta operação é realizada oito vezes. Após esse procedimento envia-se um bit de parada, que pode ter a duração de um ou dois bits.

A Figura 20 apresenta o sinal elétrico enviado ao longo do tempo para a letra K. A região em branco, que se estende entre +3 V e -3 V, indica a região de tensão na qual o sinal não está definido. Caso a tensão lida esteja nestes limiares, seja devido aos ruídos ou outros problemas, o sistema de recepção não entenderá a

mensagem e os dados podem ser corrompidos.

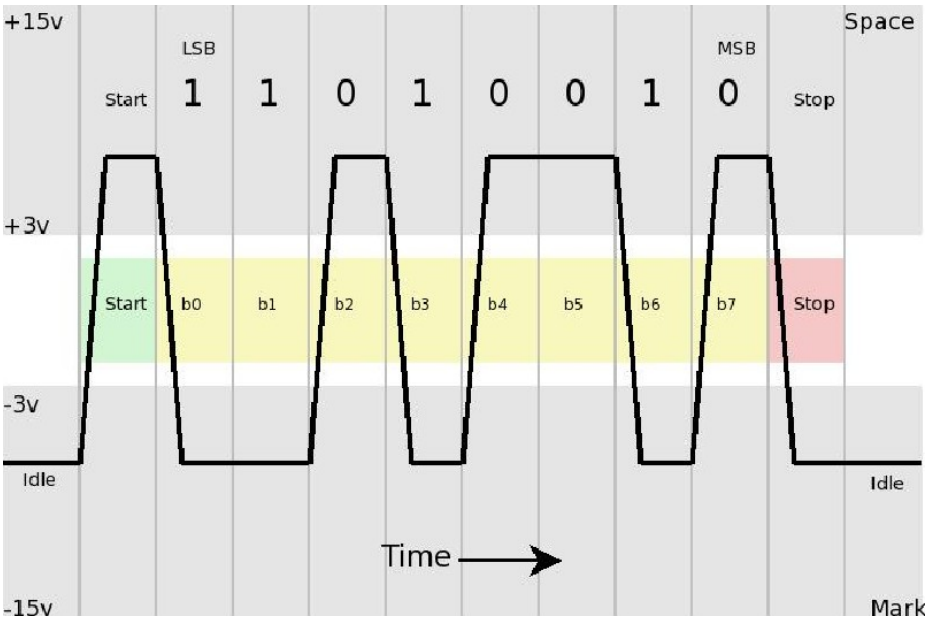


Figura 20: Sinal serializado para transmissão em RS232

2.5.6 Teclado Matricial

Para realizar a leitura de uma tecla é necessário criar um circuito que realize a leitura de um sinal elétrico para o valor zero e outro para o valor um. Os níveis de tensão associados dependem muito dos circuitos envolvidos. Os níveis mais comuns são os compatíveis com TTL, de modo que o zero lógico é representado por 0 V e o um lógico é representado por 5 V.

Uma maneira de se obter este funcionamento é com o uso de uma chave ligada ao VCC e um *pull-down* ou uma chave ligada ao terra (GND) e um *pull-up*.

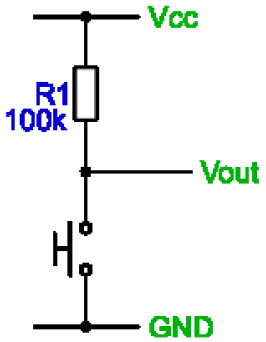


Figura 21: Circuito de leitura de chave.

http://www.labbookpages.co.uk/electronics/debounce.html - Dr. Andrew Greensted

Pela Figura 21 nota-se que a tensão de saída é igual a VCC quando a chave está desligada, pois não existe corrente circulando no circuito, portanto a queda de

tensão em R1 é zero. Quando a chave é pressionada, uma corrente flui de VCC para o terra, passando por R1, e como não existe nenhuma outra resistência no circuito, toda a tensão se concentra em R1 e a tensão de saída torna-se zero.

Apesar do funcionamento aparentemente simples, este tipo de circuito apresenta um problema de oscilação do sinal no momento em que a tecla é pressionada. Esta oscilação é conhecida como *bouncing* (Figura 22).

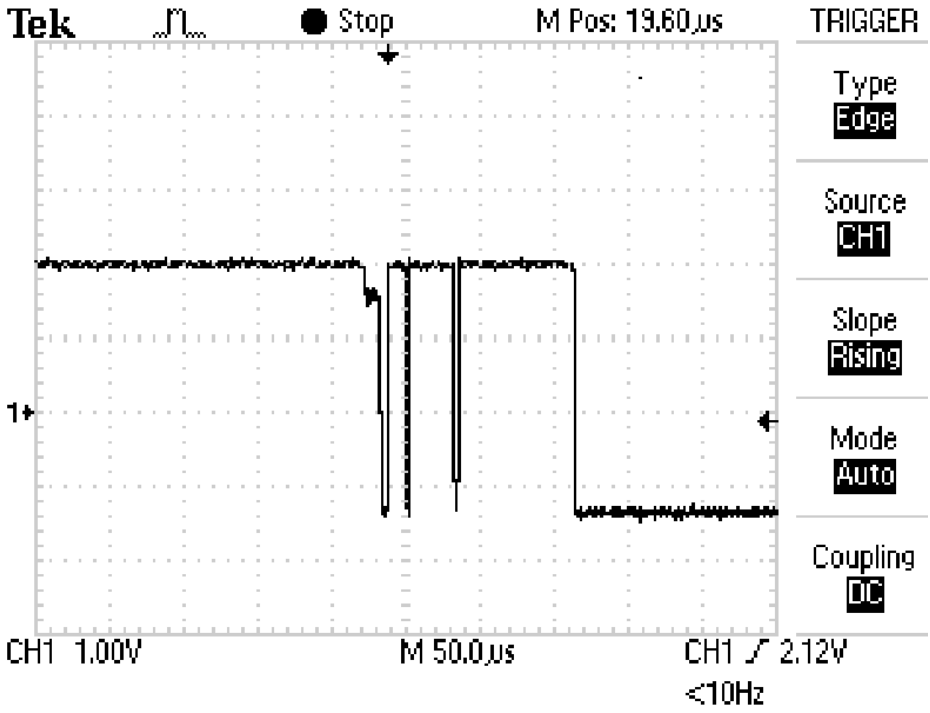


Figura 22: Oscilação do sinal no momento do chaveamento.

http://www.labbookpages.co.uk/electronics/debounce.html - Dr. Andrew Greensted

Estas oscilações indevidas podem gerar acionamentos acidentais, causando mau funcionamento do programa. Para evitar isso, pode-se utilizar técnicas de *debounce*, por hardware ou software.

A opção de *debounce* por hardware pode ser visualizada na Figura 23.

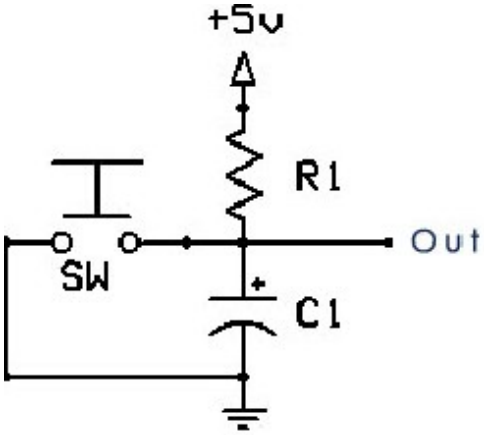


Figura 23: Circuito de debounce.

http://www.ikalogic.com/debouncing.php - Ibrahim Kamal

Neste circuito, o capacitor desempenha o papel de amortecedor do sinal. Um circuito com um resistor e um capacitor possui um tempo de atraso para o sinal, e este é o tempo necessário para carregar o capacitor. Deste modo as alterações rápidas no sinal, devido à oscilação mecânica da chave, são filtradas e não ocorre o problema dos chaveamentos indevidos conforme pode ser visto na Figura 24.

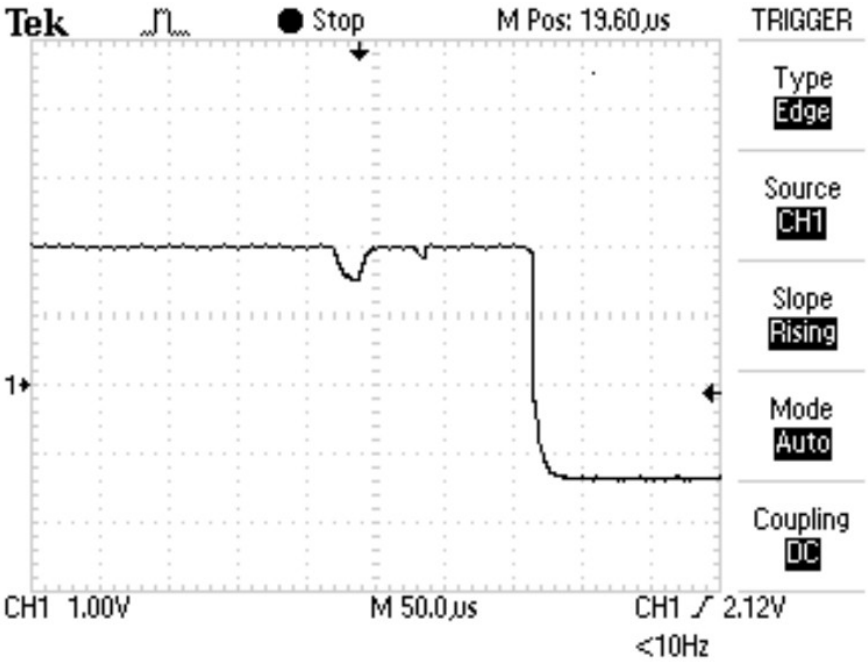


Figura 24: Utilização de filtro RC para debounce do sinal.

http://www.labbookpages.co.uk/electronics/debounce.html – A. Greensted (modificado)

Notar que o nível do sinal filtrado não chega a zero em nenhum momento, devido à constante de tempo do filtro RC ser maior que o período de *debounce*.

Em um projeto com poucas teclas é comum que cada tecla/botão seja

conectada separadamente a um terminal do microcontrolador. Para um teclado maior, é possível que o microcontrolador não possua terminais disponíveis em quantidade suficiente. Do mesmo modo que no caso dos displays, é possível multiplexar as chaves, aumentando a quantidade de chaves por terminal. Novamente, este ganho, em termos de hardware, aumenta a complexidade para o software e juntamente com este, o custo em termos de tempo de computação.

Uma das técnicas mais eficientes para a geração deste teclado é o arranjo em formato matricial. Com esta configuração é possível, com N terminais, ler até $(N/2)^2$ chaves. Conforme a Figura 25, cada chave pode ser identificada unicamente pela sua posição (linha, coluna).

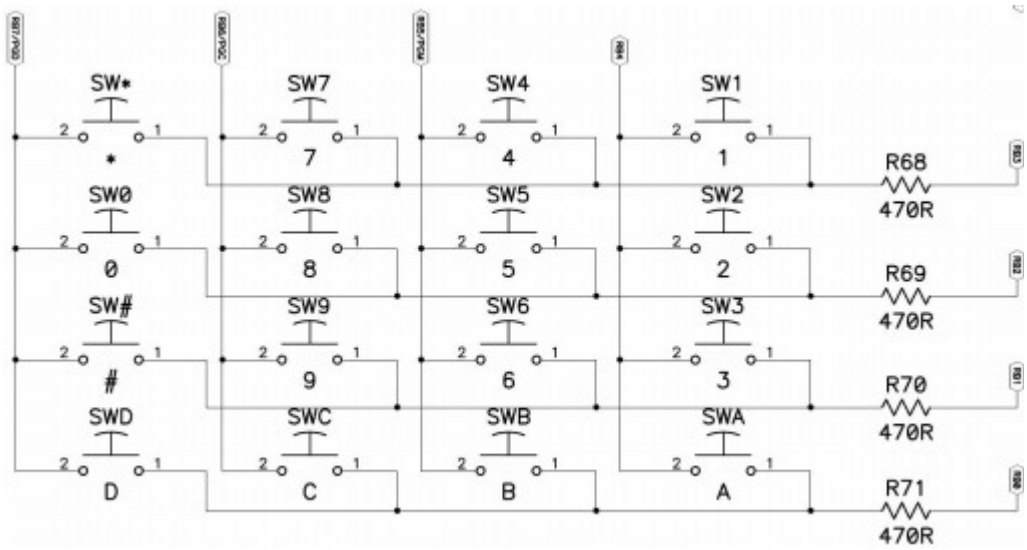


Figura 25: Teclado em arranjo matricial

3 Desenvolvimento

3.1 Decisões para desenvolvimento de um kernel

O desenvolvedor deve considerar alguns quesitos antes de iniciar o desenvolvimento de seu kernel:

Gerenciamento de dispositivos I/O: Como o kernel deve implementar a interface do dispositivo? No interior do kernel? Utilizando drivers de dispositivos? Utilizará uma controladora de driver à parte ou estará implícita nas atividades do kernel? O acesso direto entre aplicação e driver será possível? Em qual caso? Em casos de dispositivos hot-plug, como o kernel carregará dinamicamente o driver?

Gerência de processos: A troca de contexto do kernel será cooperativa ou preemptiva? Como um processo pode se comunicar com outros? Uma fila de mensagens será implementada? Possuirá uma memória compartilhada? Como controlar seu acesso? Semáforos estarão disponíveis? Há necessidade de se implementar checagem de prioridade dos processos?

Segurança do sistema: Há algum meio de segurança de hardware a ser utilizada (watchdog, memória protegida)? Será utilizada proteção hierárquica? Caso seja, a CPU fornece MMU (Memory Management Unit, traduzido livremente para Unidade de Gerenciamento de Memória) ou o desempenho pode diminuir com a checagem de proteção do software? O sistema deve tentar fechar e reiniciar um processo sem resposta automaticamente?

Tais decisões devem ser tomadas cuidadosamente, pois algumas delas não podem ser mudadas sem uma reescrita completa do código-fonte, enquanto outras podem ser postergadas durante o desenvolvimento do projeto.

Neste projeto, será utilizado um microkernel cooperativo, simples, não-preemptivo, sem memória compartilhada, com uma controladora de drivers dos dispositivos, com o intuito de isolar os driver dos dispositivos do kernel. Os processos serão agendados baseados nas necessidades de frequência de

execução.

3.2 Primeira implementação

Neste primeiro exemplo, será criada a parte principal do kernel. Deve haver uma maneira de armazenar quais funções são necessárias para serem executadas e em qual ordem, e para concluir esta tarefa, serão utilizados ponteiros de funções:

```
// declaração de ponteiro de função
typedef void(*ptrFunc)(void);
// “pool” de processos
static ptrFunc pool[4];
```

Os processos serão do tipo ptrFunc, i.e. não recebem qualquer tipo de parâmetro e não retornam nada.

Cada processo será representado por uma função. Por exemplo:

```
void tst1(void){
    printf("Process 1\n");
}
void tst2(void) {
    printf("Process 2\n");
}
void tst3(void) {
    printf("Process 3\n");
}
```

Estes processos somente escrevem o próprio nome na tela de console/saída padrão.

O kernel propriamente dito possui três funções: uma para se inicializar, outra para adicionar novos processos no pool de processos e um terceiro para sua execução. É esperado que este kernel jamais cesse sua execução, então utiliza-se um loop infinito na função de execução.

```
// variáveis internas do kernel
ptrFunc pool[4];
int end;
// protótipos das funções do kernel
void kernelInit(void);
void kernelAddProc(ptrFunc newFunc);
void kernelLoop(void);
// implementação das funções do kernel
void kernelInit(void){
    end = 0;
}
void kernelAddProc(ptrFunc newFunc){
    if (end <4){
        pool[end] = newFunc;
        end++;
    }
}
void kernelLoop(void){
    int i;
  for(;;){
        for(i=0; i<end;i++){
            (*pool[i])();
        }
    }
}
```

Neste primeiro exemplo, o kernel somente executa as funções que lhe são dadas, na ordem em que são chamadas. Não há outro tipo de controle, sendo o tamanho do pool de processos definido estaticamente.

Para utilizar o kernel deve-se seguir os três seguintes passos: inicializar o kernel, adicionar os processos desejados e executá-lo.

```
void main(void){
    kernelInit();
    kernelAddProc(tst1);
    kernelAddProc(tst2);
    kernelAddProc(tst3);
    kernelLoop();
}
```

3.3 Segunda implementação

Há quatro importantes alterações nesta versão: o pool de processos agora é implementado como um buffer circular, o processo é representado como uma struct (composto somente pelo ponteiro de função do processo) e todas as funções retornam um código de erro/sucesso.

A última modificação é que os processos podem informar ao kernel que “desejam” serem reagendados. O kernel, então, readiciona o processo ao pool de processos.

```
// código de retorno
#define SUCCESS  0
#define FAIL   1
#define REPEAT  2

// declaração de ponteiro de função
typedef char(*ptrFunc)(void);

// struct do processo
typedef struct {
    ptrFunc function;
} process;

// informações do kernel
#define POOL_SIZE 4
process pool[SLOT_SIZE];
char start;
char end;

// funções do kernel
char kernelInit(void);
char kernelAddProc(process newProc);
void KernelLoop(void);
```

A grande mudança no uso do kernel é que passa a ser necessário inserir uma struct de processo na função AddProc ao invés de somente o ponteiro de função. Perceba que cada função de processo retorna somente se foi corretamente finalizada ou se deseja ser reagendada.

```
char tst1(void){
    printf("Process 1\n");
    return REPEAT;
}
char tst2(void){
    printf("Process 2\n");
    return SUCCESS;
}

char tst3(void){
    printf("Process 3\n");
    return REPEAT;
}

void main(void){
    // declarando os processos
    process p1 = {tst1};
    process p2 = {tst2};
    process p3 = {tst3};
    kernelInit();
    // agora é possível testar se o processo foi corretamente adicionado
    if (kernelAddProc(p1) == SUCCESS){
        printf("1st process added\n");
    }
    if (kernelAddProc(p2) == SUCCESS){
        printf("2nd process added\n");
    }
    if (kernelAddProc(p3) == SUCCESS){
        printf("3rd process added\n");
    }
    kernelLoop();
}
```

A função de execução dos processos (kernelLoop()) do kernel é a que mais possui modificações. Agora é necessário checar se a função executada deseja ser reagendada e age de acordo com tal pedido.

```
void kernelLoop(void){
    int i=0;
    for(;;){
        // Há algum processo a ser executado?
        if (start != end){
            printf("Ite. %d, Slot. %d: ", i, start);
            // executa a primeira função e checa se há necessidade de
               reagendá-la
            if ( (*(pool[start].Func))() == REPEAT){
                //reagendando
                kernelAddProc(pool[start]);
            }
            // preparando para ir para o próximo processo
            start = (start+1)%POOL_SIZE;
            i++; // somente para fins de depuração
        }
    }
}
```

A função AddProc() deve checar se existe ao menos dois slots disponíveis no buffer (lembre-se de que a última posição deve estar disponível todo o tempo) e

inserir o processo.

```
char kernelAddProc(process newProc){
    // checando a existência de espaço livre
    if ( ((end+1)%SLOT_SIZE) != start){
        pool[end] = newProc;
        end = (end+1)%POOL_SIZE;
        return SUCCESS;
    }
    return FAIL;
}
```

A rotina de inicialização somente configura as variáveis start e end para a primeira posição.

```
char kernelInit(void){
    start = 0;
    end = 0;
    return SUCCESS;
}
```

A seguir, é apresentada a saída do programa principal para as primeiras dez iterações.

```
----------------------------
1st process added
2nd process added
3rd process added
Ite. 0, Slot. 0: Process 1
Ite. 1, Slot. 1: Process 2
Ite. 2, Slot. 2: Process 3
Ite. 3, Slot. 3: Process 1
Ite. 4, Slot. 0: Process 3
Ite. 5, Slot. 1: Process 1
Ite. 6, Slot. 2: Process 3
Ite. 7, Slot. 3: Process 1
Ite. 8, Slot. 0: Process 3
Ite. 9, Slot. 1: Process 1
...
----------------------------
```

Note que somente os processos 1 e 3 estão repetindo, como esperado. Perceba também que o pool está circulando pelos slots 0, 1, 2 e 3 naturalmente. Para o usuário o pool de processos aparenta ser “infinito” enquanto não há mais funções do que slots.

3.4 Terceira Implementação

Desta vez serão adicionados os componentes temporais ao kernel. Cada processo possui um campo chamado start, o qual é decrementado com o passar do tempo. Quando atinge o valor zero (ou negativo), a função é chamada.

A função kernelLoop() será responsável por encontrar o processo que está próximo de ser executado baseado em seu contador start. É necessário navegar por

todo o pool para realizar esta busca. Quando o próximo processo a ser executado é encontrado, sua posição é trocada com a primeira no pool. Após isso, o restante do tempo é gasto aguardando o processo estar pronto para ser executado.

O tempo, aparentemente sem utilidade, é utilizado para sincronizar todos os eventos e é uma boa oportunidade para deixar o sistema em modo de economia de energia.

```
void kernelLoop(void){
    unsigned char j;
    unsigned char next;
    process tempProc;
    for(;;){
        if (start != end){
            // Detectando o processo com menor “start”
            j = (start+1)%SLOT_SIZE;
            next = start;
            while(j!=end){
                // por acaso o processo seguinte possui um tempo menor?
                if (pool[j].start < pool[next].start){
                    next = j;
                }
                // pega o seguinte no buffer circular
                j = (j+1)%SLOT_SIZE;
            }
            // trocando posições no pool
            tempProc = pool[next];
            pool[next] = pool[start];
            pool[start] = tempProc.;
            while(pool[start].start>0){
                // ótimo momento para entrar em
                // modo de economia de energia
            }
            // checando se é necessário ocorrer repetição
            if ( (*(pool[ini].function))() == REPEAT ){
                AddProc(&(vetProc[ini]));
            }
            // próximo processo
            ini = (ini+1)%SLOT_SIZE;
        }
    }
}
```

Agora a rotina de tratamento de interrupção: deve decrementar o campo start de todos os processos.

```
void interrupt_service_routine(void) interrupt 1{
unsigned char i;
    i = ini;
    while(i!=fim){
        if((pool[i].start)>(MIN_INT)){
            pool[i].start--;
        }
        i = (i+1)%SLOT_SIZE;
    }
}
```

A função AddProc() será a responsável por inicializar o processo com um

valor adequado nos campos da struct.

```
char AddProc(process newProc){
    // buscando espaços livres
    if ( ((end+1)%SLOT_SIZE) != start){
        pool[end] = newProc;
        // incrementando o timer de start com o período
        pool[end].start += newProc.period;
        end = (end+1)%SLOT_SIZE;
        return SUCCESS;
    }
    return FAIL;
}
```

Ao invés de resetar o contador start adiciona-se o período a ele. Isto foi feito porque quando a função inicia, seu contador continua decrementando. Se uma função necessitar ser executada em um intervalo de 5ms e gastar 1ms executando, quando é finalizada deseja-se reagendá-la para ser executada 4ms à frente, 5ms de período menos 1ms do contador negativo (start), e não 5ms à frente.

3.5 Implementação final do kernel

Este é o kernel completo apresentado nos passos anteriores, como visto na Figura 26. Para que seja processado na placa de desenvolvimento serão utilizadas bibliotecas auxiliares: uma para atuar sobre interrupções (“drvInterrupt.c” e “drvInterrupt.h”), outra para operar sobre o timer (“drvTimer.c” e “drvTimer.h”), mais uma para configurar os registros do microcontrolador (“config.h”) e uma última com informações dos registros especiais (“basico.h”).

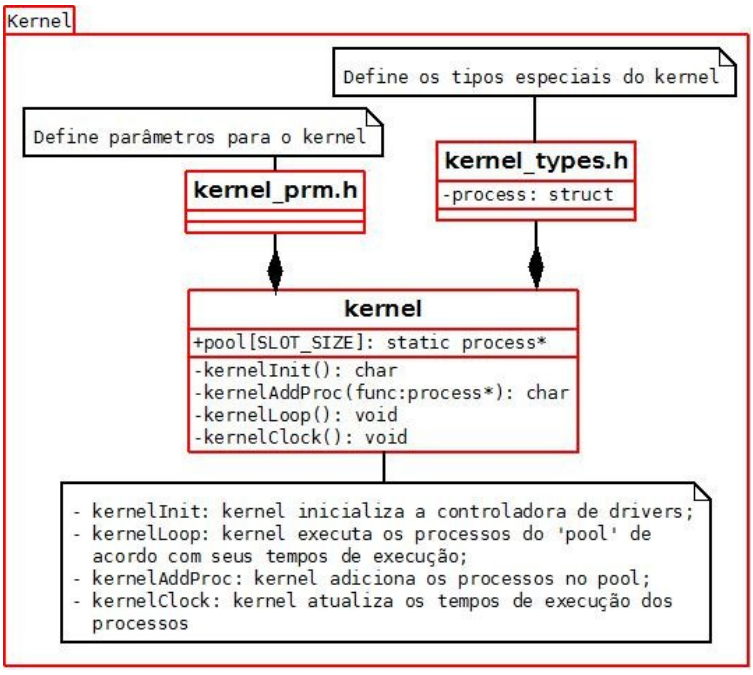


Figura 26: Diagrama da estrutura do kernel

```
//drvInterrupt.h (trecho)
char initInterrupt(void* parameters);

//drvTimer.h (trecho)
char isTimerEnd(void* parameters);
char resetTimer(void* parameters);
char waitTimer(void* parameters);
char initTimer(void* parameters);

//basico.h (trecho)
#define OK      0
#define FAIL    1
#define REPEAT  2
```

Para que funcione com requisições temporais é necessário realizar algumas operações em intervalos de tempo fixo, principalmente decrementando o contador start dos processos. Estes passos foram agrupados em uma função: KernelClock(). O usuário somente precisa chamar esta função de sua própria função de interrupção por timer.

```
//atualiza os tempos de execução dos processos
void kernelClock(void){
    unsigned char i;
    i = start;
    while(i!=end){
        if((pool[i]->start)>(MIN_INT)){
            pool[i]->start--;
        }
        i = (i+1)%SLOT_SIZE;
    }
}
```

A outra função do kernel permanece como mostrada a seguir:

```
//adiciona os processos no pool
char kernelAddProc(process* func){
    // adiciona processo somente se houver espaço livre
    //o fim nunca pode coincidir com o inicio
    if ( ((end+1)%SLOT_SIZE) != start){
        //adiciona o novo processo e agenda para executar imediatamente
        func->start += func->period;
        pool[end] = func;

        end = (end+1)%SLOT_SIZE;
        return OK; //sucesso
    }
    return FAIL; //falha
}
//inicializa o kernel em conjunto com a controladora de drivers
char kernelInit(void){
    BitSet(OSCCON, 7) ; //'Idle mode' em caso de instrução SLEEP
    initCtrDrv();
    start = 0;
    end = 0;
    return SUCCESS;
}
//executa os processos do 'pool' de acordo com seus tempos de execução
void kernelLoop(void){
    unsigned char j;
    unsigned char next;
```

```
        process* tempProc;
        for(;;){
            if (start != end){
                //Procura a próxima função a ser executada com base no tempo
                j = (start+1)%SLOT_SIZE;
                next = start;
                while(j!=end){
                    if (pool[j].start < pool[next].start){
                        next = j;
                    }
                    //para poder incrementar e ciclar o contador
                    j = (j+1)%SLOT_SIZE;
                }
                //troca e coloca o processo com menor tempo como o próximo
                tempProc = pool[next];
                pool[next] = pool[start];
                pool[start] = tempProc;
                while(pool[start].start>0){
                    //coloca a cpu em modo de economia de energia
                    _asm
                        SLEEP
                    _endasm;
                }
                //retorna se precisa repetir novamente ou não
                switch (pool[start]->function()) {
                    case REPEAT:
                        kernelAddProc(pool[start]);
                    break;
                    case FAIL:
                        break;
                    default: ;
                }
                //próxima função
                start = (start + 1) % SLOT_SIZE;
            }
        }
    }
```

Para declarar a função de interrupção no compilador SDCC deve-se adicionar o trecho “interrupt 1” ao final do nome da função (maiores detalhes serão explicados nos próximos tópicos). Como mencionado, somente reseta o timer e chama KernelClock().

```
// Interrupção
void isr1(void) interrupt 1{
    timerReset(1000); // reseta com 1ms
    kernelClock();
}
```

Para utilizar o kernel deve-se chamar sua função de inicialização, adicionar os processos com sua frequência de execução e executar a função kernelLoop().

```
// Piscar led 1
char tst1(void) {
    BitFlp(PORTD,0);
    return REPEAT;
}
// Piscar led 2
char tst2(void) {
    BitFlp(PORTD,1);
    return REPEAT;
}
// Piscar led 3
char tst3(void) {
    BitFlp(PORTD,2);
    return REPEAT;
}
void main(void){
    // declarando os processos
    process p1 = {tst1,0,100};
    process p2 = {tst2,0,1000};
    process p3 = {tst3,0,10000};
    kernelInit();
    kernelAddProc(p1);
    kernelAddProc(p2);
    kernelAddProc(p3);
    kernelLoop();
}
```

3.6 Controladora de dispositivos

Para que os drivers sejam isolados do kernel (consequentemente das aplicações), será criada uma controladora de drivers dos dispositivos. Esta será responsável por carregar o driver e transmitir os pedidos recebidos do kernel para o driver correto.

3.6.1 Padrão de controladora de dispositivos

Todos os kernels apresentam um certo padrão na construção de suas controladoras. Esta padronização é fundamental para o sistema. Somente por possuir uma interface padrão, o kernel pode se comunicar com um driver o qual ele desconhece, em tempo de compilação.

Com o intuito de simplificar o uso de ponteiros, serão utilizados typedef's. ptrFuncDriver é um ponteiro de função que retorna char (código de erro/sucesso) e recebe um ponteiro para void como parâmetro. É utilizado para chamar cada função do driver, já que todos devem possuir a seguinte assinatura:

```
typedef char(*ptrFuncDrv)(void *parameters);
```

A struct de um driver, de acordo com a Figura 27, é composta pelo seu identificador, um vetor de ponteiros de ptrFuncDriver (representado como um ponteiro) e uma função especial, também do tipo ptrFuncDriver. Esta última é

responsável por inicializar o driver, uma vez que é carregado.

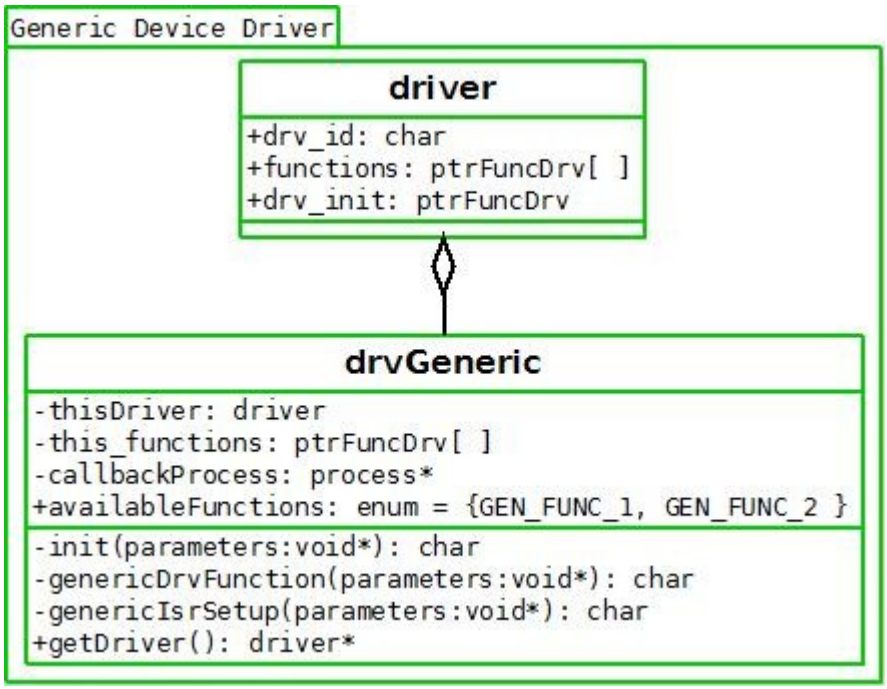


Figura 27: Diagrama da estrutura de um driver genérico

```
typedef struct {
    char drv_id;
    ptrFuncDrv *functions;
    ptrFuncDrv drv_init;
} driver;
```

Para que a controladora de drivers dos dispositivos acesse os drivers dos dispositivos, é necessário receber um ponteiro para a struct do driver. Ao invés de realizar uma ligação estática, os links serão configurados com um ponteiro de função o qual, quando chamado, retorna o driver do dispositivo desejado.

```
typedef driver* (*ptrGetDrv)(void);
```

Um driver genérico deve, então, implementar ao menos duas funções: init() e getDriver().

```
driver* getGenericDriver(void) {
    thisDriver.drv_init = initGeneric;
    thisDriver.func_ptr = (ptrFuncDrv*) &this_functions;
    return &thisDriver;
}
```

A função getDriver() permite que a função de inicialização (init()) e as demais funções do driver sejam acessadas pela controladora.

```
char initGeneric(void *parameters) {
    /* Configura os registros de hardware necessários para a operação
     * desejada do driver */
    // ...

    // Referencia os identificadores de funções do driver
    // ...

    // Inicializa variáveis internas
    // ...

    return OK;
}
```

Também deve conter uma struct de driver e um vetor de ponteiros, com cada posição apontando para cada função que implementa. Outro elemento necessário é um enumerado que define as posições de cada ponteiro de função no vetor, atuando como um rótulo para cada um deles.

```
static driver thisDriver;
static ptrFuncDrv this_functions[GEN_END];
enum {
    GEN_FUNC1,
    GEN_FUNC2,
    GEN_FUNC3,
    GEN_END
};
```

Para este projeto, foi padronizado que cada driver possui um *header* contendo um enumerado com os rótulos das funções, assim como a função getDriver().

3.6.2 Mecanismo da controladora

A controladora de drivers dos dispositivos precisará, ao menos, conhecer todos os driver disponíveis, neste caso, através de um vetor de ptrGetDriver, no qual cada posição contém o ponteiro para a função do driver que retorna sua struct. A posição na qual os ponteiros estão armazenados no vetor é definida por um enumerado, auxiliando na identificação do ponteiro para seu respectivo driver.

A estrutura da controladora pode ser vista na Figura 28.

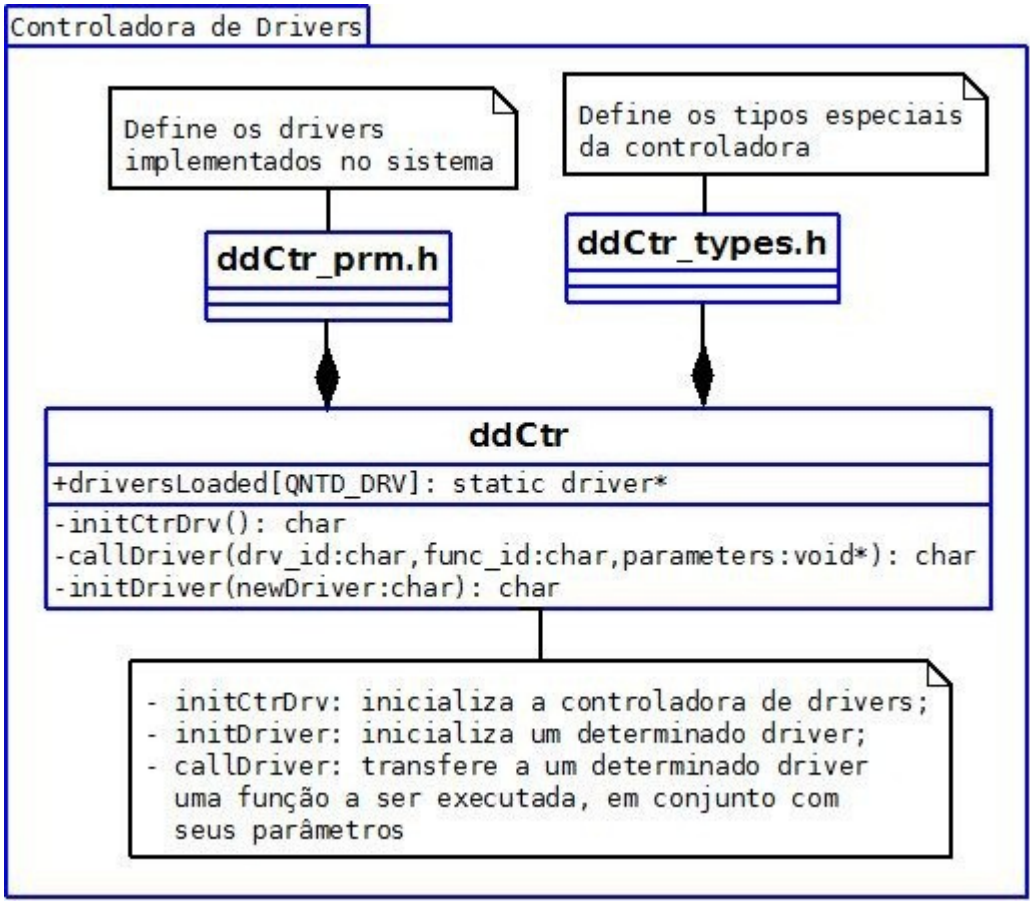


Figura 28: Diagrama da estrutura da controladora de drivers

```
// é necessário para incluir todos os arquivos de drivers
#include "drvInterrupt.h"
#include "drvTimer.h"
#include "drvLcd.h"
// este enumerado auxilia o desenvolvedor/usuário a acessar os drivers
enum {
    DRV_INTERRUPT,
    DRV_TIMER,
    DRV_LCD,
    DRV_END // DRV_END deve sempre ser o último, para facilitar o
            // controle da quantidade de drivers
};
// as funções para obter os drivers devem ser inseridas na mesma
// ordem que o enumerado acima
static ptrGetDrv drvInitVect[DRV_END] = {
    getInterruptDriver,
    getTimerDriver,
    getLCDDriver
};
```

A controladora de drivers de dispositivos possui: um vetor de drivers e um contador, indicando quantos drivers foram carregados até o momento. Possui somente três funções: uma para inicializar as variáveis de intervalo, outra para carregar o driver e a terceira para checar os comandos do kernel para o driver correto.

Carregar um driver é bem direto. Se o driver DRV_INTERRUPT necessita ser carregado, basta acessar a lista de drivers disponíveis e realizar o pedido ao driver de interrupção. Então é chamada sua rotina de inicialização e o driver passa a ser armazenado na lista de carregados. Se não houver espaço para outro driver, a função retorna um erro.

```
char initDriver(char newDriver) {
    char resp = FAIL;
    if(qntDrvLoaded < QNTD_DRV) {
        driversLoaded[qntDrvLoaded] = drvGetFunc[newDriver]();
        resp = driversLoaded[qntDrvLoaded]->drv_init(&newDriver);
        qntDrvLoaded++;
    }
    return resp;
}
```

A rotina que chama o driver percorre a lista de drivers carregados para identificar o driver correto. Quando é casado o padrão, a função procurada é chamada e os parâmetros são passados como um ponteiro para void (não são sabidos quais os parâmetros).

```
char callDriver(char drv_id, char func_id, void *parameters) {
    char i;
    for (i = 0; i < qntDrvLoaded; i++) {
        if (drv_id == driversLoaded[i]->drv_id) {
            return driversLoaded[i]->func_ptr[func_id](parameters);
        }
    }
    return DRV_FUNC_NOT_FOUND;
 }
```

3.6.3 Utilizando os mecanismos da controladora

Para que se use a engine da controladora necessita-se incluir seu header no arquivo principal e fazer uso de seus enumerados definidos em cada arquivo de driver para acessar o hardware.

A função LCD_CARACTER no driver DRV_LCD envia um caracter (codificado em ASCII) para o LCD interligado ao microcontrolador. Se houver necessidade de modificar o LCD ou mudar o port ao qual está conectado, a aplicação será mantida intacta, de modo que o desenvolvedor precisa somente alterar o driver.

```
void main(void) {
    // inicialização do sistema
    // o kernel também inicia a função de inicialização da controladora
    kernelInit();
    initDriver(DRV_LCD);
    callDriver(DRV_LCD, LCD_CARACTER, 'U');
    callDriver(DRV_LCD, LCD_CARACTER, 'n');
    callDriver(DRV_LCD, LCD_CARACTER, 'i');
    callDriver(DRV_LCD, LCD_CARACTER, 'f');
    callDriver(DRV_LCD, LCD_CARACTER, 'e');
    callDriver(DRV_LCD, LCD_CARACTER, 'i');
}
```

3.6.4 Camada de abstração da interrupção

Há algumas soluções interessantes que auxiliam a aplicação a manter seu alto nível enquanto permanece interagindo com o hardware. Uma possibilidade é esconder a rotina de interrupção dentro do driver e ainda assim permitir ao desenvolvedor da aplicação definir seu comportamento.

```
// definindo o tipo do ponteiro de função
// para utilizá-lo como interrupção
typedef void (*intFunc)(void);
// armazena aqui o ponteiro da rotina de tratamento de interrupção
static intFunc thisInterrupt;
char setInterruptFunc(void *parameters) {
    thisInterrupt = (intFunc) parameters;
    return OK;
}
```

O driver de interrupção armazenará um ponteiro dentro de si. Este ponteiro pode ser alterado através das função setInterruptFunc(). A atual função de interrupção será passada como parâmetro.

Também no interior do arquivo está a definição para o compilador, que indica qual função é responsável por chamar a interrupção.

```
// modo do compilador SDCC
void isr(void) interrupt 1{
    thisInterrupt();
}
// modo do compilador C18
void isr (void){
    thisInterrupt();
}
#pragma code highvector=0x08
void highvector(void){
    _asm goto isr _endasm
}
#pragma code
```

Utilizando o ponteiro para armazenar a ISR (Interrupt Service Routine, traduzida como Rotina de Tratamento de Interrupção), os detalhes de programação de baixo nível do compilador tornam-se ocultos para a aplicação.

```
void timerISR(void) {
  callDriver(DRV_TIMER, TMR_RESET, 1000);
  kernelClock();
}
void main (void){
  kernelInit();

  initDriver(DRV_TIMER);
  initDriver(DRV_INTERRUPT);

  callDriver(DRV_TIMER, TMR_START, 0);
  // habilitando todas as interrupções
  callDriver(DRV_TIMER, TMR_INT_EN, 0);
  callDriver(DRV_INTERRUPT, INT_TIMER_SET, (void*)timerISR);
  callDriver(DRV_INTERRUPT, INT_ENABLE, 0);

  kernelLoop();
}
```

3.6.5 Callback do driver

Em alguns processos de I/O existe a necessidade de aguardar uma resposta de hardware, geralmente através da mudança de um bit de status que, como consequência, gera uma interrupção. A leitura do conversor A/D é um bom exemplo dessa situação: após ativada sua inicialização, o conversor começa a processar o sinal lido (o tempo de processamento do módulo A/D varia de acordo com seu modelo e configuração). Após finalizado o processo, o bit de status do conversor é alterado e uma interrupção ocorre. Sem um tratamento correto, o kernel ficaria ocioso, esperando a resposta do conversor, causando perda de tempo e de processamento do microcontrolador.

A técnica de *callback* permite ao sistema continuar funcionando mesmo à espera da resposta de um componente de hardware. Sendo assim, pode-se chamar o driver pedindo para se iniciar seu trabalho, passar a função (*callback*) que deve ser executada após o término e dar continuidade ao fluxo. Deste modo economiza-se tempo de processamento do microcontrolador enquanto procura-se receber o resultado o mais rápido possível.

Para que isso se cumpra, o driver deve ser capaz de gerar uma interrupção no sistema.

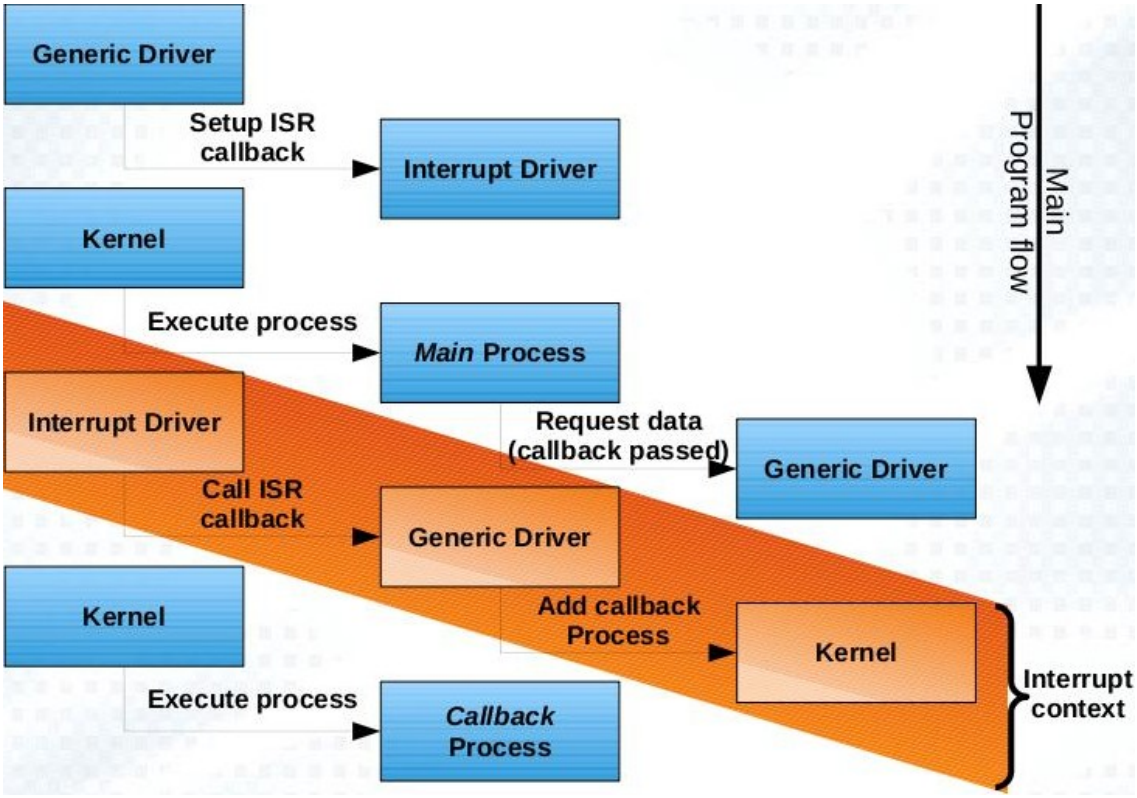


Figura 29: Processo de callback

Para explicar esta técnica, supõe-se um driver hipotético *drvCallback*, que conterá um arquivo *header* e um de implementação. Em termos de hardware, ele realiza um processo de aquisição de dados e os processa, o que gera lentidão no sistema, se comparada a velocidade do processo deste driver com os demais. Primeiramente, o driver deve informar ao driver de interrupção (*callback* à ISR) deve ser chamada quando a interrupção desejada é gerada.

```
// Trecho de implementação genérica ao arquivo do driver de callback
/* chamado em tempo de configuração para permitir a
 * interrupção do driver e configurar o callback da
 * rotina de tratamento de interrupção (ISR) */
char enableCallbackInterrupt(void* parameters){
    callDriver(DRV_INTERRUPT, INT_CALLBACK_SET, (void*)callbackISR);
    /* código necessário para desativar a flag do hardware,
     * sinalizando que pode ser gerada uma interrupção por parte
     * do periférico */
    //...

    return OK;
}

// drvInterrupt.c (trecho)
// armazena o ponteiro da função de interrupção
typedef void (*intFunc)(void);
static intFunc callbackInterrupt;

/* função para configurar a rotina de tratamento de interrupção (ISR)
 * do driver para uso posterior */
char setCallbackInt(void *parameters) {
    callbackInterrupt = (intFunc)parameters;
    return OK;
}
```

Quando a aplicação realiza uma requisição de dados para o driver, deve-se passar um processo de retorno. Este é o processo que será adicionado ao *pool* posteriormente, quando a interrupção ocorrer. Este processo é armazenado dentro do driver. O driver faz a configuração dos periféricos e inicia a conversão/aquisição.

```
// main.c (trecho)
// Processo chamado pelo kernel
char callbackDriver_func(void) {
    // criando um processo de callback
    static process proc_callback = {callback, 0, 0};
    callDriver(DRV_CALLBACK, CALLBACK_FUNC, &proc_callback);
    return REPEAT;
}

// Trecho de implementação genérica ao arquivo do driver de callback
/* função chamada pelo processo callback_func
 * via controladora de driver */
char CallbackFunction(void* parameters){
    callBack = parameters;
    /* configura os registros para que o periférico inicie seu trabalho,
     * ou seja, é a primeira parte do trabalho deste hardware */
    // ...

    return OK;
}
```

Quando a interrupção desejada ocorre, o driver de interrupção chama a rotina de serviço de interrupção que foi pré-definida pelo driver. O driver realiza todos os procedimentos necessários (copiar dados, definir flags, etc) para que os dados estejam disponíveis fora do contexto de interrupção. Essas funções devem ser

rapidamente executadas para evitar *starvation* no modo normal/aplicação.

```
// drvInterrupt.c (trecho)
// função de interrupção
void isr(void) interrupt 1 {
    if (BitTst(INTCON, 2)) {  // estouro de faixa do Timer
        timerInterrupt();
    }
    // finalização do processo do periférico
    /* testa a flag do periférico, para checar se foi feito um
     * pedido de requisição de interrupção */
    if (...) {
        /* chamando callback da ISR armazenada no ponteiro de função
         * do driver hipotético */
        // ...

        callbackInterrupt();
    }
}
// drvCallback.c (trecho)
// função de callback da ISR do periférico
void adcISR(void){
    // segunda parte do trabalho deste hardware é implementada aqui
    // ...

    kernelAddProc(callBack);
}
```

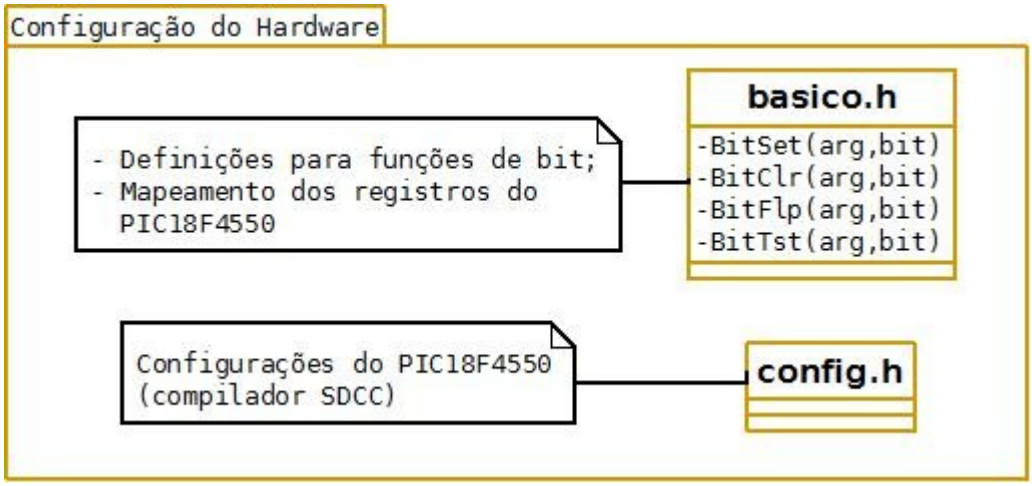
Quando o processo de *callback* se torna o próximo no *pool* de processos, o kernel garante seu compartilhamento do tempo de processamento. Dentro do processo de *callback*, pode-se dedicar maior tempo nas tarefas que necessitam do processador, como filtragem de sinais, armazenamento permanente de dados, etc.

```
// main.c (trecho)
// função de callback inicializada a partir do kernel
char drvCallback_callback(void) {
    unsigned int resp;

    // recebendo o valor processado
    callDriver(DRV_CALLBACK,CALLBACK_LAST_VALUE,&resp);
    return OK;
}
```

3.7 Arquivos de configuração

Além do kernel, drivers, controladora e o programa principal, dois arquivos são necessários para fazer o STR funcionar plenamente: um arquivo de configuração do microcontrolador, definindo como atuará em relação a sinais elétricos externos e como o hardware reagirá a códigos que realizem leitura/escrita em seus periféricos (“config.h”) e outro que defina os registros especiais e ports (“basico.h”). Neste último, além disso, foram definidas operações que agem diretamente sobre bits e códigos de retorno de processos.



Maiores detalhes desses programas nos anexos 7.1 e 7.2 .

3.8 Criação de drivers

A partir das explicações da controladora de drivers, uma prévia do que o driver necessita já foi dada: um identificador, uma função de inicialização e um vetor de demais funções. Além disso, deve-se atentar às funcionalidades que cada driver deve possuir, ou seja, quando um driver gerar uma interrupção no microcontrolador, quais configurações devem ser feitas? Após as configurações, uma função de callback deve ser chamada? Como devo tratar as interrupções?

A seguir, serão detalhados os periféricos utilizados da placa de desenvolvimento, assim como as nuances de implementação de cada driver criado.

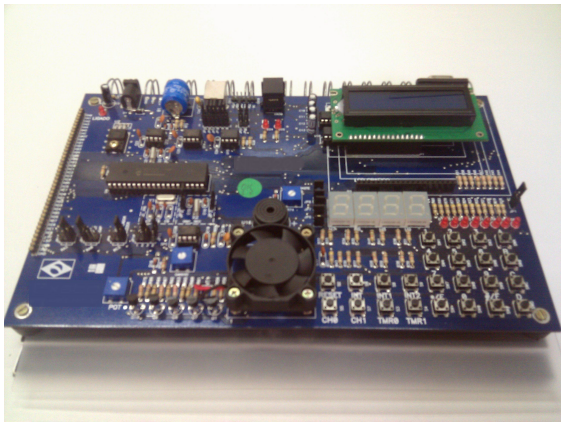


Figura 30: Plataforma de desenvolvimento usada no projeto

3.8.1 Driver do barramento de LEDs

O esquema de implementação do driver do barramento de LEDs pode ser visto na Figura 31.

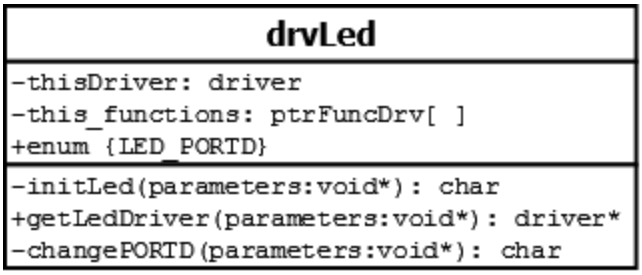


Figura 31: Esquema de implementação do driver do barramento de LEDs

```
//apenas retorna o "objeto" do driver
driver* getLedDriver(void);
// define os identificadores das funções do driver
enum {
    LED_PORTD,
    LED_END
};
```

No arquivo de implementação, deve haver, no mínimo: uma variável para o driver se auto-referenciar, um vetor com as funções do driver, uma funcionalidade do driver, função de inicialização do driver e uma função de auto-referência, que é utilizada pela controladora, já explicados anteriormente.

```
// variável utilizada para auto-referência do driver
static driver thisDriver;

// vetor de funções do driver
static ptrFuncDrv this_functions[LED_END];

// acende ou apaga os LED's selecionados, de acordo com os
// parâmetros recebidos
char changePORTD(void *parameters) {
    PORTD = *((char*) parameters);
    return OK;
}

// inicialização do driver
char initLed(void *parameters) {
    // Configura os pinos da porta D como saída
    TRISD = 0x00;
    thisDriver.drv_id = *((char*) parameters);
    this_functions[LED_PORTD] = changePORTD;
    return OK;
}

// função de auto-referência
driver* getLedDriver(void) {
    thisDriver.drv_init = initLed;
    thisDriver.func_ptr = (ptrFuncDrv*) &this_functions;
    return &thisDriver;
}
```

A variável thisDriver é utilizada pela função getDriver(), implementada em todos os drivers deste sistema. Para a função init(), mesmo possuindo detalhes de implementação comuns aos demais drivers, é nesta função que os registros e portas

do microcontrolador são configurados, para que o driver possa operar corretamente. Para este caso, os pinos da porta D são configurados como saída, para que os LED's possam ser acesos.

Para o driver de LED, este possui uma única funcionalidade implementada: acender/apagar seus LED's. A função recebe um parâmetro, a priori, desconhecido. Entretanto, espera-se bom uso por parte do programador-usuário deste sistema, e o mesmo deve passar quais LED's ficarão acesos e quais se apagarão.

Todas as funções de driver receberão, por padrão, um número indeterminado de parâmetros, genericamente representados como void*. Deste modo, como na função do LED, espera-se que o programador-usuário faça bom uso das funcionalidades do driver e envie os parâmetros corretos. É responsabilidade da função realizar o correto tratamento da informação que lhe chega.

Em alguns casos, não serão utilizados parâmetros nas funcionalidades, entretanto, por padronização, todas elas possuirão como parâmetro void*.

3.8.2 Driver de Displays de 7 Segmentos

O esquema de implementação do driver de displays de 7 segmentos pode ser visto na Figura 32.

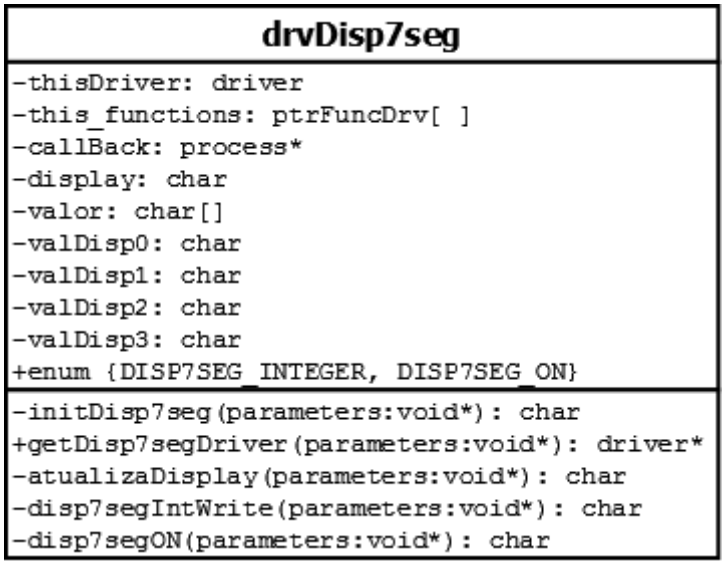


Figura 32: Esquema de implementação do driver de displays de 7 segmentos

Como explicado anteriormente, deve ser feito o controle dos displays de 7 segmentos um a um devido à multiplexação do circuito. O controle de qual display será ligado é feito através do transistor que permite a ligação do catodo comum ao terra, ou o anodo comum ao VCC (depende do tipo de dispositivo). Para o correto

funcionamento não basta acender os LEDs corretos para formar o número, mas seguir um algoritmo mais complexo: colocar no barramento de dados o valor a ser mostrado no display X, ligar o display X através da linha de comando, esperar um tempo adequado para evitar *flicker*(efeito de cintilação, ou seja, pisca constantemente a uma taxa perceptível ao olho humano), desligar o display e escolher o próximo display (X+1). Esse processo é repetido para cada um dos componentes, sendo repetido ciclicamente.

```
// drvDisp7Seg.h (trecho)
enum {
    DISP7SEG_INTEGER,
    DISP7SEG_ON,
    DISP7SEG_END
} ;
```

O driver dos display de 7 segmentos disponibiliza duas funções para uso externo: *DISP7SEG_INTEGER* que simplesmente guarda o valor de um número inteiro a ser escrito nos displays e *DISP7SEG_ON,* função que ativa os displays exibindo valores guardados previamente.

As funções de interface citadas acima são mapeadas internamente como: *Disp7segIntWrite() e disp7segON() / atualizaDisplay(),* respectivamente.

```
char Disp7segIntWrite(void *parameters) {
    int valTemp = (unsigned int)parameters ;

    if (valTemp < 10000) {
        valDisp3 = valTemp / 1000 ;
        valTemp %= 1000 ;
        valDisp2 = valTemp / 100 ;
        valTemp %= 100 ;
        valDisp1 = valTemp / 10 ;
        valTemp %= 10 ;
        valDisp0 = valTemp ;
        return OK;
    }

    return FAIL;
}
```

Para exibir um número inteiro no display é necessário distribuir os dígitos entre eles pela ordem de grandeza. Pode-se realizar essa distribuição utilizando uma técnica simples: o valor inteiro é divido por 1000 (milhar) e a resposta inteira é guardada na variável interna referente ao display do milhar. O resto é utilizado em uma nova divisão, agora por 100 (centena) e, como no primeiro caso, a parte inteira da resposta é armazenada em sua variável correspondente, e novamente divide-se o resto por uma ordem abaixo. O procedimento segue semelhante até finalmente atingir as unidades. Os valores armazenados nas variáveis internas serão

posteriormente apresentados nos displays correspondentes.

```
char atualizaDisplay(void)
{
    //desliga todos os displays
    PORTA &= 0b11011011;
    PORTE &= 0b11111010;
    //desliga todos os segmentos (LED's)
    PORTD = 0x00;
    // liga apenas um display por vez
    switch (display) {
        // display 0
        case 0:
                PORTD = valor[valDisp0];
                PORTA |= 0b00100000 ;
                display = 1;
        break;
        // display 1
        case 1:
                PORTD = valor[valDisp1];
                PORTA |= 0b00000100 ;
                display = 2;
        break;
        // display 2
        case 2:
                PORTD = valor[valDisp2];
                PORTE |= 0b00000001 ;
                display = 3;
        break;
        // display 3
        case 3:
                PORTD = valor[valDisp3];
                PORTE |= 0b00000100 ;
                display = 0;
        break;
        // acesso indevido: retorna ao display 0
        default:
                display = 0;
        break;
    }
    return REPEAT ;
}
```

A função *atualizaDisplay* é a responsável por, efetivamente, exibir os dados nos displays. Inicialmente todos os LEDs e displays são desativados. Em seguida coloca-se o valor correspondente a cada display na linha de dados multiplexada, e então, o display apropriado é ligado. Por fim, é setado na variável *display* qual deve ser o próximo a ser ativado.

A complexidade adicional de implementação devido à técnica de multiplexação está praticamente inteira na função *atualizaDisplay(),* faltando apenas os detalhes de temporazição que são resolvidos por *disp7segON()*.

```
static process proc_multiplexDisp7seg = {atualizaDisplay, 7, 0} ;

char disp7segON(void *parameters) {
    kernelAddProc(&proc_multiplexDisp7seg) ;
    return OK ;
}
```

Esta é responsável por escalonar a função de exibição dos dados nos displays de maneira a garantir o efeito de todos ligados. Para isso, cria-se um processo de período fixo que é adicionado ao pool de processos do kernel pela função *KernelAddProc().*

3.8.3 Driver de Display LCD

O esquema de implementação do driver de display LCD pode ser visto na Figura 33.

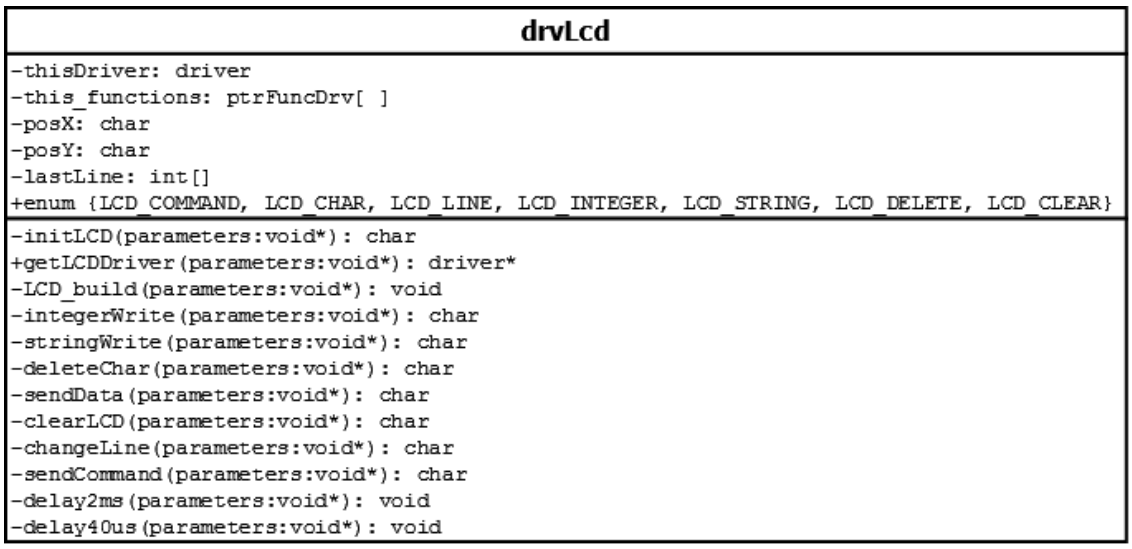


Figura 33: Esquema de implementação do driver de display LCD

Conforme explicado anteriormente e demonstrado na Figura 16, os terminais do display LCD utilizam duas portas: uma para dados e outra para controle. Destes terminais, os que serão trabalhos são os 8 bits de dados e 3 bits de controle (RS, RW e EN).

```
//drvLcd.h (trecho)
//apenas retorna o "objeto" do driver
driver* getLCDDriver(void);

// define os identificadores das funções do driver
enum {
    LCD_COMMAND,
    LCD_CHAR,
    LCD_LINE,
    LCD_INTEGER,
    LCD_STRING,
    LCD_DELETE,
    LCD_CLEAR,
    LCD_END
};
```

Através da tabela da Figura 15, diversas funções foram implementadas, como observadas pelo trecho extraído do *header* do driver.

```
//drvLcd.c (trecho)
//Definições do LCD
#define RS 0 // pino de seleção de registro do LCD
#define EN 1 // pino de habilitação do LCD, para futura leitura/escrita
#define RW 2 // pino de leitura/escrita do LCD

// número de colunas disponível no display LCD
#define COLUMNS 16

// caracteres escritos em uma linha, de acordo com o número de colunas
static unsigned int lastLine[COLUMNS] ;

// linha da posição atual do cursor
static unsigned char posX ;

// coluna da posição atual do cursor
static unsigned char posY ;

// funções locais
// gera um delay na resposta do LCD de 40us
void Delay40us(void) {
    // para um cristal de 8 MHz, cada instrução gasta 0.5 us: 40/0.5 = 80
instruções
    unsigned char i;
    for (i = 0; i < 25; i++); //3u + 3u * 25 = 78 us
}

// gera um delay na resposta do LCD de 2ms
void Delay2ms(void) {
    unsigned char i;
    for (i = 0; i < 200; i++) {
        Delay40us();
    }
}
```

As funções criadas anteriormente são utilizadas por causa do atraso gerado no processamento das funções do LCD, de modo que certos comandos necessitam de esperas menores, e outros maiores. A primeira gera um atraso de, aproximadamente, 40µs. Para tal, cria-se um laço: gastam-se 3µs para a declaração

do *for* e mais 3µs para cada iteração ocorrida nele. Sendo assim, um tempo médio de 78µs. Para atrasos maiores, basta se utilizar da repetição da função anterior, que é o caso de *Delay2ms()*.

```
// envia um comando ao display LCD
char sendCommand(void *parameters) {
    // comando
    BitClr(PORTE, RS);

    // habilita escrita
    BitClr(PORTE, RW);

    // insere a informação na linha de comunicação do display LCD
    PORTD = (unsigned char) parameters;

    // habilita leitura
    BitSet(PORTE, EN);
    Delay2ms();

    //termina leitura
    BitClr(PORTE, EN);

    //deixa em nivel baixo
    BitClr(PORTE, RS);

    //deixa em nivel baixo
    BitClr(PORTE, RW);
    Delay40us();

    return OK;
}
```

As informações enviadas ao microcontrolador podem ser, basicamente, divididas em dois grupos: comando e dados. Um comando seria a configuração de suas características, seu comportamento, por exemplo, definição dos caracteres exibidos, existência de cursor piscante ou mesmo inserção de um novo símbolo na memória interna.

Para tal, deve-se configurar os pinos de controle (situados na porta E), primeiramente informando que dados serão enviados ao LCD, ou seja, se trata de escrita (RW) e informando que se trata de um comando, e não um dado (RS). Assim, é possível inserir uma sequência de bits que realizem o comando desejado na porta D. Como já explicado anteriormente, espera-se que o programador-usuário envie os parâmetros corretos, que sejam corretamente interpretados ao serem convertidos para *char* não-sinalizado.

Com os dados prontos para serem lidos pelo microcontrolador do LCD, este necessita ser habilitado (EN). Esta operação necessita ser feita neste momento para garantir que sejam lidos os dados corretos, evitando que ruidos eletromagnéticos ou

informações anteriores sejam incorretamente captadas, e para garantir que sejam recebidas, aguardam-se alguns ciclos de clock.

Com o comando já recebido, reconfigura os pinos de controle do LCD, de modo que o mesmo não receba mais dados da porta D. Entretanto, o tempo passado até então não é suficiente para garantir a execução do comando enviado, então se aguarda mais alguns instantes.

```
// inicialização do driver
char initLCD(void *parameters) {
    thisDriver.drv_id = *(char*) parameters;
    this_functions[LCD_COMMAND] = sendCommand;
    this_functions[LCD_CHAR] = sendData;
    this_functions[LCD_INTEGER] = integerWrite;
    this_functions[LCD_LINE] = changeLine;
    this_functions[LCD_STRING] = stringWrite ;
    this_functions[LCD_DELETE] = deleteChar ;
    this_functions[LCD_CLEAR] = clearLCD ;

    // garante inicialização do LCD
    Delay2ms();
    Delay2ms();
    Delay2ms();
    Delay2ms();
    Delay2ms();

    // configurações de direção dos terminais
    BitClr(TRISE, RS);
    BitClr(TRISE, EN);
    BitClr(TRISE, RW);
    TRISD = 0x00;
    ADCON1 = 0b00001110;

    // Configura o display:
    // 8bits, 2 linhas, caracteres formados por células 5x8
    sendCommand(0x38);
    // modo incremental: insere caracter, vai para a célula seguinte
    sendCommand(0x06);
    // backlight ligada; cursor piscante ligado
    sendCommand(0x0F);
    // zera o buffer do display LCD
    sendCommand(0x03);
    // apaga todos os caracteres e símbolos escritos no LCD
    sendCommand(0x01);
    // posição inicial: linha 0, coluna 0
    sendCommand(0x80);

    posX = 0 ;
    posY = 0 ;

    return OK;
}
```

Como já padronizado anteriormente, para inicializar um driver, primeiramente são passados o identificador do mesmo, assim como rótulos para suas funções. Como é um periférico mais lento que o microcontrolador, deve-se aguardar um

tempo para garantir que o mesmo seja carregado corretamente.

Então, os registros que utiliza são configurados como pinos de saída, ou seja, informação será enviada a eles. No caso do barramento de dados, ou seja, a porta D, a mesma deve ser configurada para transferir e receber dados digitais, através da configuração do rótulo ADCON1.

Na sequência, o comportamento do LCD é definido através do envio de comandos internos do microcontrolador associado a ele, e então, as variáveis referentes às posições de linha e coluna corrente do display, respectivamente *posX* e *posY*, são inicializadas com zero, significando que o caracter a ser exibido estará na primeira linha, primeira coluna.

```
// apaga todos os caracteres e símbolos escritos no LCD
char clearLCD() {
    sendCommand(0x01) ;
    return OK ;
}

// pula uma linha no LCD
char changeLine(void *parameters) {
    if ((unsigned char) parameters == 1) {
        sendCommand(0x80);
    }
    if ((unsigned char) parameters == 2) {
        sendCommand(0xC0);
    }
    return OK;
}
```

As funções de limpeza de tela e troca de linha são utilizações diretas de comandos internos do LCD. A primeira elimina qualquer símbolo impresso no display, enquanto a outra altera a linha corrente de escrita, ou seja, dependendo do argumento passado, a informação será reproduzida na primeira ou na segunda linha). Apesar do parâmetro ser um número, não há necessidade do mesmo ser do tipo inteiro (*int*).

```
// envia um caracter para o display LCD
char sendData(void *parameters) {
    unsigned char character = (unsigned char) parameters ;
    unsigned char i ;

    if (posX > 15) {
        if (posY == 1) {
            posY = 0 ;
            posX = 0 ;
            clearLCD() ;
            changeLine(1) ;

            for (i = 0; i < COLUMNS; i++) {
                sendData(lastLine[i]) ;
            }

        }

        posY = 1 ;
        posX = 0 ;
        changeLine(2) ;
    }

    //dados
    BitSet(PORTE, RS);

    // habilita escrita
    BitClr(PORTE, RW);
    PORTD = character ;

    //habilita leitura
    BitSet(PORTE, EN);
    Delay40us();

    //termina leitura
    BitClr(PORTE, EN);

    //deixa em nivel baixo
    BitClr(PORTE, RS);

    //deixa em nivel baixo
    BitClr(PORTE, RW);
    Delay40us();

    if (posY == 1) {
        lastLine[posX] = character ;
    }
    posX++ ;

    return OK;
}
```

Para enviar um dado (um único símbolo a ser reproduzido na tela do display), é necessário que o programador-usuário passe por parâmetro a informação a ser exibida. Com a análise das variáveis *posX* e *posY* definidas anteriormente no driver, que são, respectivamente, a linha e a coluna correntes do cursor do LCD, é possível checar se os caracteres ainda podem ser exibidos na linha corrente, se já

preencheram todas as colunas da primeira linha ou todas as linhas. No primeiro caso, basta atualizar a coluna corrente. No segundo caso, deve-se utilizar a função para ir à linha seguinte e atualizar as informações em *posX* e *posY*. No último caso, deve-se retornar o cursor para a primeira linha, eliminar os símbolos desenhados na tela e repassar todos os caracteres da segunda para a primeira linha.

Com o display preparado para exibir a informação, os registros de controle devem ser atualizados, com processo análogo à função *sendCommand()*. Sendo assim, deve-se determinar que a informação a ser inserida é um dado (RS) e a operação corrente no barramento do LCD é de escrita (RW), e com isso, o caracter passado por parâmetro pode ser inserido no barramento de dados, ou seja, porta D. Na sequência, o display deve ser habilitado (EN) e a informação lida. Para garantir o sucesso desta operação, deve-se aguardar um tempo.

Com a informação já exibida no display, deve-se desabilitá-lo. Além disso, a coluna da linha corrente deve ser atualizada e o símbolo armazenado no vetor de caracteres da linha corrente, sendo este utilizado para inserir caracteres da segunda para a primeira linha, como explicitado anteriormente.

```
// apaga um caracter do display LCD
char deleteChar() {
    unsigned char pos ;

    //posiciona o cursor uma posição anterior à inicial
    posX -= 1 ;
    if (posY == 1) {
        pos = 0xC0 + posX ;
    }
    else {
        pos = 0x80 + posX ;
    }
    sendCommand(pos) ;

    //escreve um caracter vazio no display LCD
    sendData(' ') ;

    //posiciona novamente o cursor uma posição anterior à inicial
    posX -= 1 ;
    sendCommand(pos) ;

    return OK ;
}
```

Para apagar um caracter do display, haveriam dois modos: reescrever a linha corrente, exceto o último caracter, ou substituir o símbolo a ser removido por um caracter em branco. A segunda opção requer menos processamento, logo, menos tempo.

Sendo assim, analisa-se a posição corrente do cursor do LCD. Com isso,

envia-se um comando para retornar uma posição, e atualiza a coluna corrente na variável interna *posX*. Na sequência, escreve-se o caracter em branco ' ' através da função *sendData()*. Entretanto, a mesma atualiza a posição da coluna, pois um dado foi efetivamente enviado e exibido, então deve-se retornar o cursor uma posição e atualizar novamente *posX*.

```
// envia um número para o display LCD
char integerWrite(void *parameters) {
    sendData(((unsigned int) parameters / 10000) % 10 + 48);
    sendData(((unsigned int) parameters / 1000) % 10 + 48);
    sendData(((unsigned int) parameters / 100) % 10 + 48);
    sendData(((unsigned int) parameters / 10) % 10 + 48);
    sendData(((unsigned int) parameters) % 10 + 48);
    return OK;
}
```

Para a exibição de um número inteiro não-negativo no display (entre 0 e 65.535), deve-se realizar a conversão do código ASCII, e enviar um número de cada vez, caso possuam mais de um caracter.

```
// envia um conjunto de caracteres para o display LCD
char stringWrite(void *parameters) {
    unsigned char i = 0 ;
    unsigned char* str = (unsigned char*)parameters ;
    while (str[i]) {
        sendData(str[i]) ;
        i++ ;
    }
    return OK ;
}
```

Para o envio de uma cadeia de caracteres para o display, basta enviar os símbolos um a um, através de um laço contendo a função *sendData()*.

```
// cria um símbolo e o escreve no display LCD
void LCD_build(void){
    char i;
    // cada linha é representada por um caracter
    char unifei[48] = {0x01,0x03,0x03,0x0E,0x1C,0x18,0x08,0x08, //0,0
                       0x11,0x1F,0x00,0x01,0x1F,0x12,0x14,0x1F, //0,1
                       0x10,0x18,0x18,0x0E,0x07,0x03,0x02,0x02, //0,2
                       0x08,0x18,0x1C,0x0E,0x03,0x03,0x01,0x00, //1,0
                       0x11,0x1F,0x00,0x01,0x1F,0x12,0x14,0x1F, //1,1
                       0x02,0x03,0x07,0x0E,0x18,0x18,0x10,0x00  //1,2
};

    // primeira posição da memória interna
    sendCommand(0x40);

    for(i=0;i<48;i++)
    {
        sendData(unifei[i]);
    }

    sendCommand(0x80);
}
```

Por último, a função *LCD_build()* possui o intuito de criar um símbolo. Cada caracter possui células dispostas em uma matriz 8x5. Sendo assim, cada caracter é o resultado do preenchimento ou não de cada uma destas células. Deste modo, para gravar um símbolo como mostrado na Figura 34, deve-se selecionar a memória, enviar um a um os elementos da matriz até que se preencha as células desejadas.

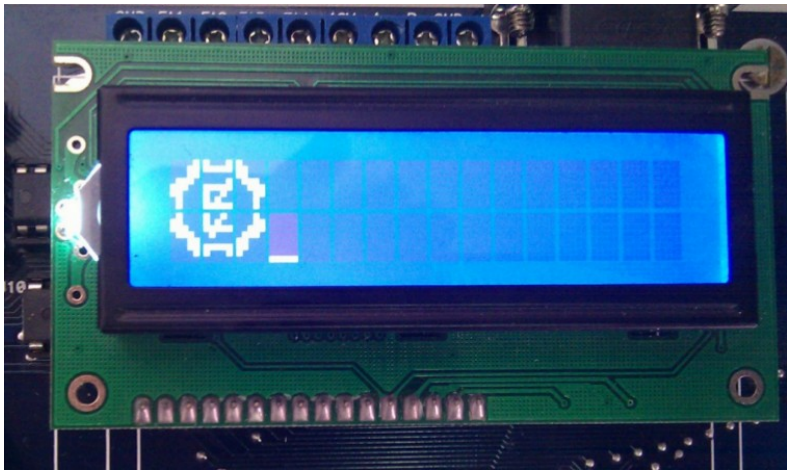


Figura 34: Criação e exibição de uma figura no display LCD

3.8.4 Driver de Interrupção

O esquema de implementação do driver de interrupção pode ser visto na Figura 35.

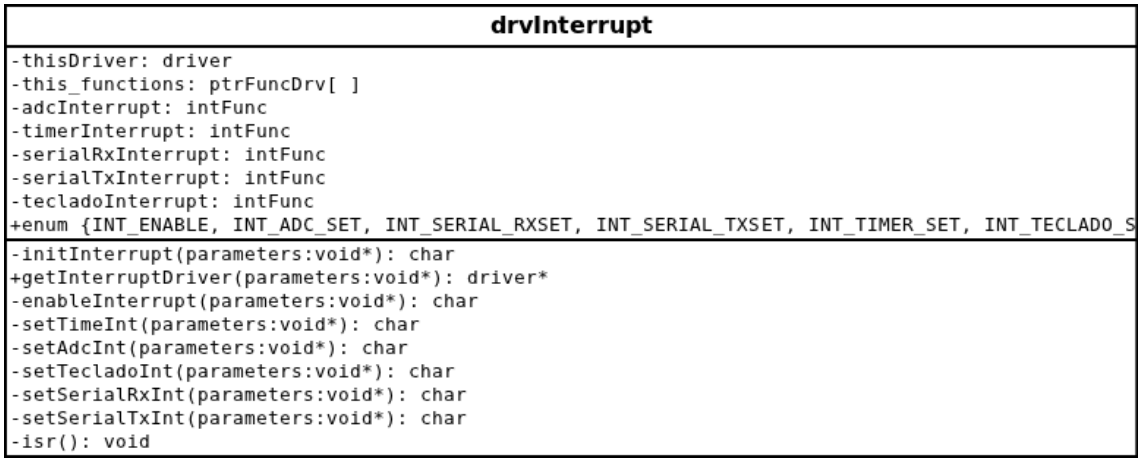


Figura 35: Esquema de implementação do driver de interrupção

Como citado anteriormente, a utilização de interrupções pode causar ganhos significativos ao sistema, principalmente pelo fato do processamento continuar mesmo à espera de um evento externo.

Todos os drivers que se utilizam da técnica de *callback* necessitam interagir de alguma forma com este driver.

Dos dispositivos estudados até agora os que geram interrupção são: porta serial (quando há informação disponível em RCREG ou quando o buffer de transmissão TXREG estiver disponível), conversor A/D (quando o resultado da conversão estiver disponível para leitura), porta B (algum dos bits configurados como entrada altera seu valor) e Timer0 (overflow em seu contador).

```
// drvInterrupt.c (trecho)
// permite que ocorram interrupções
char enableInterrupts(void *parameters) {
    BitSet(INTCON, 7); //habilita todas as interrupções globais
    BitSet(INTCON, 6); //habilita todas as interrupções de periféricos
    return OK;
}
```

O trecho acima configura dois bits de controle geral das interrupções, possibilitando ou não seu uso. O primeiro bit tem o controle de todas as interrupções e o segundo apenas das relacionadas a periféricos. Em outro momento, deve-se configurar cada interrupção individualmente.

Cada driver ligado a uma interrupção possui um ponteiro de função. Este será utilizado para executar a rotina de tratamento de interrupção correspondente.

```
// cada ponteiro de função de interrupção deve ser inserida
static intFunc adcInterrupt;
static intFunc timerInterrupt;
static intFunc serialRxInterrupt ;
static intFunc serialTxInterrupt ;
static intFunc tecladoInterrupt ;
```

Como em tempo de compilação o driver de interrupção ainda não sabe quais

serão as rotinas de tratamento de interrupção, existe a necessidade de uma interface que possibilite, em tempo de execução, informar ao driver qual função deve ser chamada. Isso é possível através das funções *setNomeDriverInt(),* que carregam os ponteiros de função com as rotinas de tratamento passadas pelos próprios drivers.

```
/* rotinas de configuração das funções de retorno
 * de interrupção dos drivers */
char setTimerInt(void *parameters) {
    timerInterrupt = (intFunc)parameters;
    return OK;
}

char setAdcInt(void *parameters) {
    adcInterrupt = (intFunc)parameters;
    return OK;
}

char setTecladoInt(void *parameters) {
    tecladoInterrupt = (intFunc)parameters ;
    return OK ;
}

char setSerialRxInt(void *parameters) {
    serialRxInterrupt = (intFunc)parameters;
    return OK;
}

char setSerialTxInt(void *parameters) {
    serialTxInterrupt = (intFunc)parameters;
    return OK;
}
```

Assim como os demais drivers estudados até o momento, o driver de interrupção também possui uma função de inicialização. Nela, as funções de interface são mapeadas para funções internas, o microcontrolador é configurado através do bit IPEN para não diferenciar interrupção em ordem prioridade, e todas as interrupções são desativadas para que possam ser posteriormente ativadas uma a uma.

```
// inicialização do driver
char initInterrupt(void *parameters) {
    thisDrivers.drv_id = *((char*)parameters);
    this_functions[INT_ADC_SET] = setAdcInt;
    this_functions[INT_TIMER_SET] = setTimerInt;
    this_functions[INT_SERIAL_RXSET] = setSerialRxInt;
    this_functions[INT_SERIAL_TXSET] = setSerialTxInt;
    this_functions[INT_TECLADO_SET] = setTecladoInt ;
    this_functions[INT_ENABLE] = enableInterrupts;

    // desabilita IPEN (modo de compatibilidade)
    BitClr(RCON, 7);

    // desabilita as interrupções de todos os dispositivos
    PIE1 = 0x00;

    // desabilita as interrupções de todos os dispositivos
    INTCON = 0x00;

    return OK;
}
```

Para gerenciar as interrupções, deve-se criar uma rotina que verifique o tipo de interrupção que ocorreu e tomar as providências necessárias. A maneira de declarar que uma função será responsável pelo tratamento das interrupções depende do compilador. No caso do SDCC basta inserir a expressão *interrupt 1* após o nome da função.

```
// drvInterrupt.c (trecho)
// rotina de atendimento de interrupções
void isr(void) interrupt 1 {

    if (BitTst(INTCON, 2)) {
        timerInterrupt();
    } // estouro de faixa do Timer

    if (BitTst(PIE1, 5) && BitTst(PIR1, 5)) {
        BitClr(PIR1, 5) ;
        serialRxInterrupt();
    } // recepção de dados via serial

    if (BitTst(PIE1, 4) && BitTst(PIR1, 4)) {
        BitClr(PIR1, 4) ;
        serialTxInterrupt();
    } // envio de dados via serial

    if (BitTst(INTCON, 3) && BitTst(INTCON, 0)) {
        BitClr(INTCON, 0) ;
        PORTD++;
        tecladoInterrupt() ;
    } // tecla do teclado matricial pressionada

    if (BitTst(PIE1, 6) && BitTst(PIR1, 6)) {
        adcInterrupt();
    } // término da conversão de dados no conversor A/D
}
```

A identificação de qual interrupção ocorreu é realizada com base em bits de

status descritos no datasheet do microncontrolador. É testado se a interrupção está individualmente ativada e se foi ela que realmente ocorreu. Em caso de resposta verdadeira ao teste, a função de tratamento respectiva é invocada.

3.8.5 Driver de Timer

O esquema de implementação do driver de Timer pode ser visto na Figura 36.

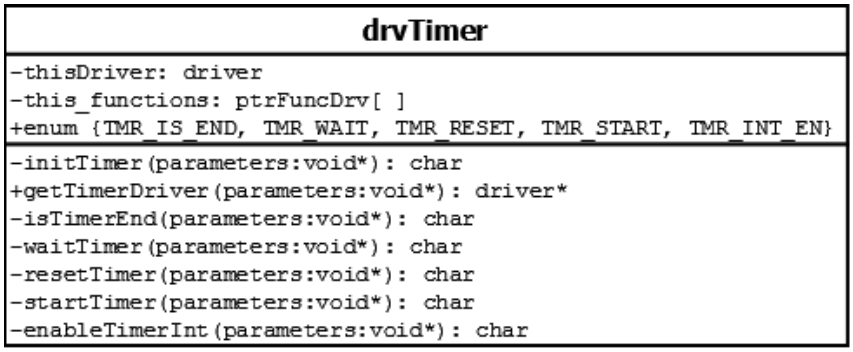


Figura 36: Esquema de implementação do driver de timer

Nos microcontroladores existem estruturas próprias para realizar a contagem de tempo, denominadas Timers. São essenciais para, por exemplo, criar um delay de um determinado tempo, ao invés de contarmos instruções, como foi feito no driver de LCD. Escolhendo um valor de tempo para contar, inicializam-se as variáveis internas necessárias do PIC e espera-se ocorrer um *overflow* (conhecido como estouro de faixa. Toda variável digital possui um valor máximo, por exemplo 255 para uma do tipo *unsigned char*. Quando está em seu valor limite e é acrescida de 1, seu valor passa a ser zero e acontece o estouro) na contagem do timer.

Deste modo, é possível realizar a contagem de tempo para a execução dos processos do kernel, através das funções anteriormente utilizadas.

```
// drvTimer.h (trecho)
enum {
    TMR_IS_END,
    TMR_WAIT,
    TMR_RESET,
    TMR_START,
    TMR_INT_EN,
    TMR_END
};

// drvTimer.c (trecho)
// inicialização do driver
char initTimer(void *parameters) {
    thisDriver.drv_id = *(char*) parameters;
    this_functions[TMR_WAIT] = waitTimer;
    this_functions[TMR_IS_END] = isTimerEnd;
    this_functions[TMR_RESET] = resetTimer;
    this_functions[TMR_START] = startTimer;
    this_functions[TMR_INT_EN] = enableTimerInt;

    // configura Timer 0 sem 'prescaler'
    T0CON = 0b00001000;

    return OK;
}
```

A função de inicialização do Timer basicamente configura um dos quatro Timers existentes no PIC18F4550, neste caso, o Timer0. Através do registro T0CON, define-se que não haverá *prescaler*, ou seja, o período utilizado pelo clock do sistema, seja ele interno ou externo, será o mesmo para contar os *ticks* do Timer0.

```
// liga a interrupção para o Timer 0
/* Por padronização, funções de drivers recebem parâmetros,
 * mesmo que não os utilizem. */
char enableTimerInt(void *parameters) {
    BitSet(INTCON, 5);
    return OK;
}

//liga o Timer0
char startTimer(void *parameters) {
    BitSet(T0CON, 7);
    return OK;
}
```

Para que o Timer0 possa gerar interrupção a cada *overflow* ocorrido em seu contador interno, deve-se ativar seu registro INTCON, e para que ele seja ativado, o mesmo deve ser feito ao seu registro T0CON. Lembrando que ajustar um registro não necessita configurar cada um de seus bits, mas somente aqueles cujo propósito é buscado.

```
// paralisa temporariamente a contagem do Timer 0
char waitTimer(void *parameters) {
    while (!BitTst(INTCON, 2));
    return OK;
}
```

Um meio de paralisar temporariamente a contagem de tempo é impedir o andamento das instruções do sistema até que se estoure o contador pois, nesta condição, o Timer0 gera interrupção. Este método foi aplicado através da estrutura acima utilizada, analisando o estado do registro INTCON.

```
// checa se o Timer 0 chegou ao fim
char isTimerEnd(void *parameters) {
    (*(char *) parameters) = BitTst(INTCON, 2);
    return OK;
}
```

Através da passagem de um argumento, pode ser checado se o Timer0 estourou, novamente pela análise de Timer0. O parâmetro passado deve ser previamente alocado, pois será gravada a informação diretamente em seu endereço de memória e atuará, na prática, como uma variável booleana.

```
// reinicializa o Timer 0, sendo o parâmetro de entrada dado em [ms]
char resetTimer(void *parameters) {
    // para placa com 8MHz, então 1 ms = 2 ciclos
    unsigned int ciclos = ((unsigned int)parameters) * 2;

    /* overflow acontece com 2^15 - 1 = 65535 (máximo valor atingido por
     * uma variável do tipo 'unsigned int') */
    ciclos = 65534 - ciclos;

    //subtrai tempo de overhead (experimental)
    ciclos -= 14;

    // salva os 8 bits mais significativos
    TMR0H = (ciclos >> 8);

    // salva os 8 bits menos significativos
    TMR0L = (char)(ciclos & 0x00FF);

    //limpa a flag de overflow
    BitClr(INTCON, 2);

    return OK;
}
```

A última função permite reajustar o tempo corrente do Timer0, de acordo com o parâmetro enviado. Para facilitar os cálculos, foi convertido o tempo em número de ciclos. Na sequência, o número de ciclos de espera desejado é descontado do quanto falta para o timer estourar. Por fim, é necessário limpar a flag, de modo que a interrupção não aconteça imediatamente após o término da função.

3.8.6 *Driver de Conversor A/D*

O esquema de implementação do driver do conversor A/D pode ser visto na .

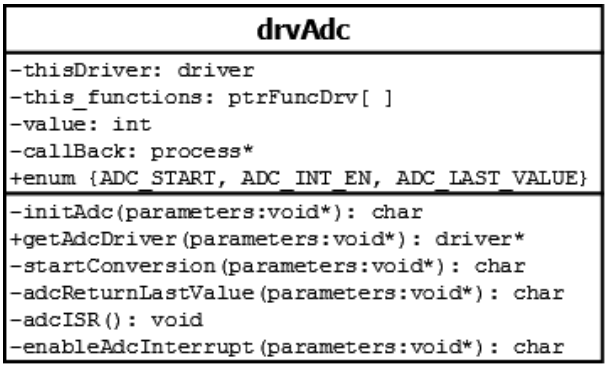


Figura 37: Esquema de implementação do driver do conversor A/D

Toda conversão leva um determinado tempo que depende da arquitetura utilizada, da qualidade do conversor e de sua precisão. Para que o microcontrolador realize corretamente a conversão é necessário seguir esta sequência: configurar o conversor, iniciar a conversão, monitorar o final da conversão e ler o valor.

```
// drvAdc.h (trecho)
enum {
    ADC_START,
    ADC_INT_EN,
    ADC_LAST_VALUE,
    ADC_END
};
```

As funções disponíveis para utilização do kernel são somente três: iniciar conversão A/D, permitir geração de interrupção por parte do conversor e leitura do último valor convertido.

```
// drvAdc.c (trecho)
// resposta da chamada de outro driver
static process *callBack;

// valor armazenado da última conversão A/D
static unsigned int value;
```

Além das variáveis internas já comuns para os drivers, o A/D se utilizará de um temporário para armazenar os valores convertidos e um importante elemento: um processo de *callback*. Através dele, pode-se processar apenas a informação essencial, sem prejudicar o andamento da fila de processos do kernel devido ao conversor ser mais lento, como já explicado anteriormente.

```
// inicialização do driver
char initAdc(void *parameters) {
    thisDrivers.drv_id = *((char*) parameters);
    this_functions[ADC_START] = startConversion;
    this_functions[ADC_INT_EN] = enableAdcInterrupt;
    this_functions[ADC_LAST_VALUE] = adcReturnLastValue;

    // configura o bit 0 como entrada
    BitSet(TRISA,0);

    //seleciona o canal 0 e liga o conversor A/D
    ADCON0 = 0b00000001;

    /* apenas o pino AN0 será analogico, e a referencia sera
     * baseada na fonte */
    ADCON1 = 0b00001110;

    // FOSC/32. Alinhamento à direita e tempo de conv = 12 TAD
    ADCON2 = 0b10101010;

    // desativa a interrupção do conversor A/D
    BitClr(PIE1, 6) ;

    return OK;
}
```

Para configurar o conversor A/D, os registros correspondentes devem ser definidos. O bit 0 da porta A deve ser configurado como entrada pois é neste terminal que se encontra o sensor de temperatura da placa. Por conveniência, será utilizado o canal 0 como padrão para a conversão análogo-digital, sendo configurado via registro ADCON0 (este modelo de PIC possui 12 canais, ou seja, 12 circuitos de conversão multiplexados entre si, de modo que deve ser selecionado um por vez), entretanto, se necessário, este registro deve ser reconfigurado para outros canais.

Além disso, a fonte de alimentação do circuito será utilizada como referência para a conversão, para que os demais pinos sejam configurados como entrada/saída digitais, exceto AN0, que será a entrada analógica, como é mostrada na configuração do registro ADCON1.

Por fim, ADCON2 é ajustado de forma a justificar a conversão à direita, ou seja, os bits 0 a 7 estarão no registro ADRESL, enquanto os bits 8 e 9 estarão em ADRESH. Além disso, 12 ciclos de T_{AD} (tempo de aquisição) são oferecidos para que se realize a captação do sinal analógico, e um T_{AD} foi definido como 32 avos do período de oscilação (clock) do circuito do microcontrolador, que foi configurado para operar sob 8 MHz. A Figura 38 demonstra o resultado desta configuração:

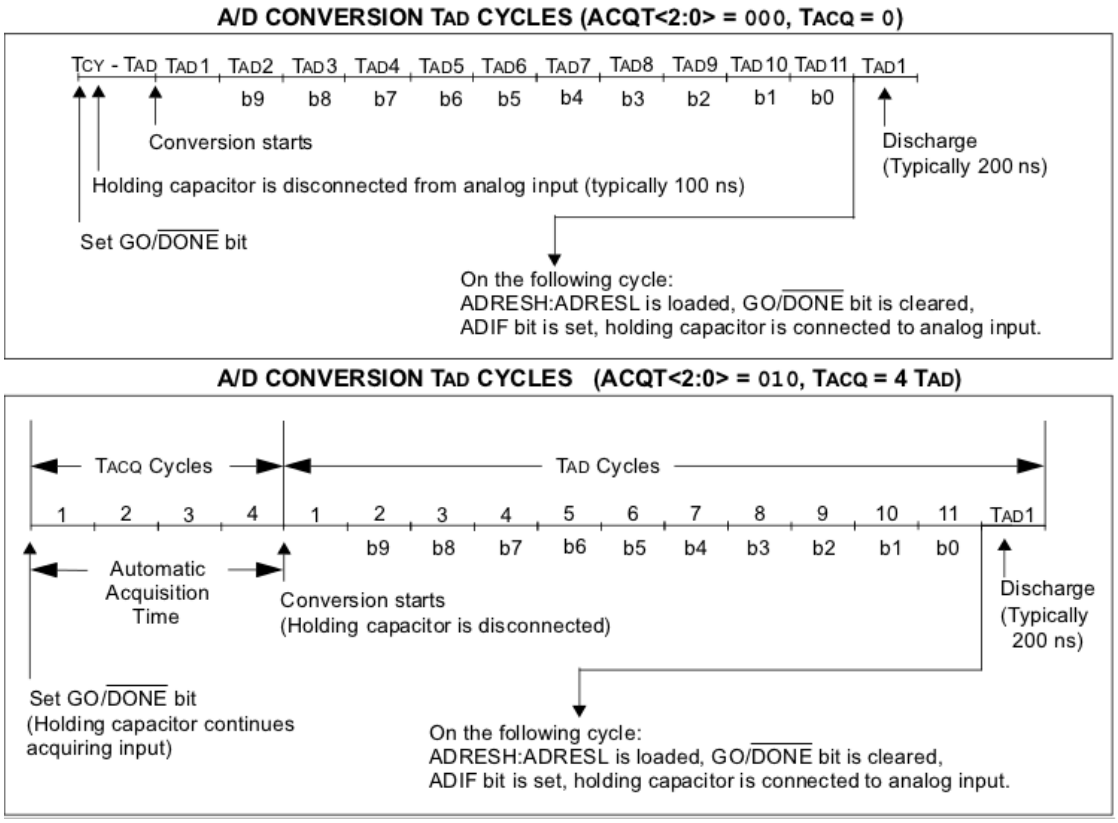


Figura 38: Diferenças de configuração de tempo de aquisição

```
// inicializa a conversão da informação analógica em digital
char startConversion(void* parameters){
    /* recebe a função de retorno de interrupção a ser
     * executada após esta, pelo programa principal */
    callBack = parameters;

    // inicia conversão
    ADCON0 |= 0b00000010;

    return OK;
}
```

A conversão do sinal analógico em formato digital ocorre via técnica de *callback*, de modo que um endereço de processo (definido no programa principal) é passado como parâmetro e inserido na variável interna do driver do conversor A/D. Em seguida, é feito o pedido de início de conversão, através do ajuste de um dos bits do registro ADCON0. Quando o conversor finalizar seu trabalho, o mesmo gerará uma interrupção no microcontrolador.

```
// rotina de tratamento de interrupção do conversor A/D
void adcISR(void){
    value = ADRESH;
    value <<= 8;
    value += ADRESL;
    BitClr(PIR1,6);
    kernelAddProc(callBack);
}
// permite que o conversor A/D gere interrupções
char enableAdcInterrupt(void* parameters){
    callDriver(DRV_INTERRUPT,INT_ADC_SET,(void*)adcISR);
    BitSet(PIE1, 6) ;
    return OK;
}
```

Após a geração da interrupção, é feita uma chamada ao driver de interrupção para que execute a rotina de tratamento de interrupção do conversor análogo-digital quando for feito um pedido e também um ajuste no registro PIE1 (*Peripheral Interrupt Enable*). Deste modo, o pedido feito via interrupção será atendido.

A rotina de tratamento de interrupção será responsável por salvar a última informação convertida na variável local *value*, de modo a ajustar corretamente a posição dos 10 bits na mesma. Na sequência, é necessário desativar o *flag* de interrupção por parte do conversor A/D, através do registro PIR1, para que uma próxima leitura possa ser feita, caso requisitada.

Como o tratamento destes dados não é necessariamente uma prioridade ao sistema, é feita a adição de um processo de *callback* na fila de processos do kernel, que atua como uma continuação da conversão A/D.

```
// retorna na variável recebida o resultado da última conversão A/D
char adcReturnLastValue(void* parameters){
    (*(unsigned int*)parameters) = value;
    return OK;
}
```

Esta função possui o intuito de somente transferir o último valor convertido à variável enviada como parâmetro, que deve ser previamente alocado, pois será gravada a informação diretamente em seu endereço de memória.

3.8.7 Driver de Comunicação Serial

O esquema de implementação do driver de comunicação serial pode ser visto na Figura 39.

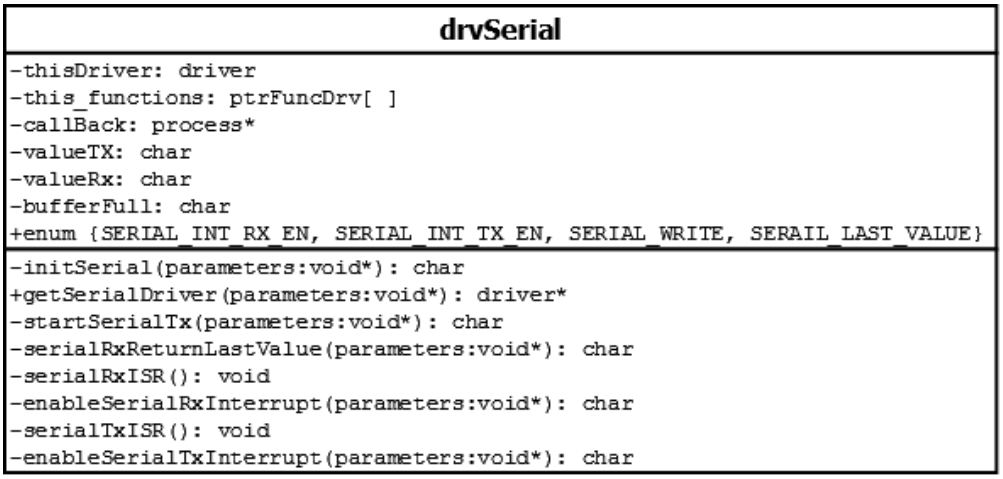


Figura 39: Esquema de implementação do driver de comunicação serial

Para o correto funcionamento do protocolo, deve-se garantir compatibilidade no nível físico (do hardware) e lógico (no software). Para o hardware basta compatibilizar o tipo de conector, os níveis de tensão e a pinagem dos conectores. Para o nível de software, é necessário definir a codificação utilizada (ASCII, UTF-8, entre outros), especificar o fluxo de caracteres (quantidade de bits por caractere, tamanho do bit de start/stop, paridade) e a taxa de transmissão desejada, de modo que estas configurações são realizadas através de 5 registros: TXSTA, RCSTA, BAUDCON, SPBRGH e SPBRG.

Os registros TXSTA e RCSTA são responsáveis por configurar o meio de transmissão: presença/ausência de bit de parada, tamanho da palavra de um caractere e transmissão síncrona/assíncrona. O registro BAUDCON é responsável por configurar o controle de velocidade de transmissão. Os registros SPBRGH e SPBRG representam os *bytes* mais alto e mais baixo da palavra de 16 bits que indica a taxa de transmissão, sendo esta calculada segundo a Figura 40.

Bits de Configuração			Precisão	Taxa de transmissão
TXSTA:4	BAUDCON:3	TXSTA:2		
0	0	0	8bits	$F_{232} = \frac{F_{OSC}}{[64 * (n+1)]}$
0	0	1	8bits	$F_{232} = \frac{F_{OSC}}{[16 * (n+1)]}$
0	1	0	16bits	
0	1	1	16bits	$F_{232} = \frac{F_{OSC}}{[4 * (n+1)]}$
1	0	x	8bits	
1	1	x	16bits	
x – não importa, n – valor do par SPBRGH:SPBRG				

Figura 40: Cálculo do valor da taxa de transmissão da porta serial

Como visto, existem três formulas diferentes para calcular a taxa de transmissão. A melhor maneira de configurar a taxa de transmissão da porta serial é verificar qual dos métodos gera o menor erro para uma dada taxa de transmissão.

Por exemplo, deseja-se gerar uma taxa de transmissão de 57,6 kbps, e a frequência disponível é um cristal de 8MHz. Usando as três formulas, definem-se os seguintes valores:

- $n_1 = 1$; $F_{232} = 62.500$, erro = -7,64%
- $n_2 = 8$, $F_{232} = 55.555$, erro = 3,63%
- $n_3 = 32$, $F_{232} = 57.142$, erro = 0,79%

A equação que gera o menor erro é a terceira. Para este projeto, será utilizada uma comunicação assíncrona, e pela tabela obtém-se que os bits de configuração devem ser: TXSTA(4) = 0, BAUDCON(3) = 1 e TXSTA(2) = 1. A seguir, todo o processo de configuração da porta serial RS232 e o mapeamento de funções.

```
// inicialização do driver
char initSerial(void* parameters) {
    thisDriver.drv_id = *((char*) parameters) ;
    this_functions[SERIAL_WRITE] = startSerialTx ;
    this_functions[SERIAL_INT_RX_EN] = enableSerialRxInterrupt ;
    this_functions[SERIAL_INT_TX_EN] = enableSerialTxInterrupt ;
    this_functions[SERIAL_LAST_VALUE] = serialRxReturnLastValue ;

    // configura a serial:
    // transmissão de dados da serial
    TXSTA = 0b00101100 ;

    // recepção de dados da serial
    RCSTA = 0b10010000 ;

    // sistema de velocidade da serial
    BAUDCON = 0b00001000 ;
    // velocidade de transmissão para 57600bps
    SPBRGH = 0x00 ;
    SPBRG = 0x42 ;

    //pino de recepção de dados
    BitSet(TRISC,6) ;

    // pino de envio de dados
    BitClr(TRISC,7) ;

    // indica que não há informação no buffer
    bufferFull = 0 ;

    /* inicializa, por padrão, as interrupção geradas
     * pelo driver desativadas */
    BitClr(PIE1, 5) ;
    BitClr(PIE1, 4) ;

    return OK ;
}
```

O procedimento de serialização dos bits é feito de maneira automática pelo hardware. Enquanto ele realiza este processo, não se pode mexer no registro que armazena o *byte* a ser enviado, por isso deve-se verificar se o registro está disponível. Isto é feito através do bit 4 do registro PIR. Quando este valor estiver em 1 basta escrever o valor desejado no registro TXREG.

```
// valor armazenado da última transmissão de dados via serial
static unsigned char valueTx ;

// indica se o buffer da serial está cheio ou não
static unsigned char bufferFull ;

char startSerialTx(void *parameters) {
    /* verifica se existe algum dado que ainda não foi
     * escrito na serial */
    if (bufferFull == 0) {
        valueTx = (unsigned char)parameters ;

        // indica que existe um dado a ser escrito na serial
        bufferFull = 1 ;

        // ativa interrupção de transmissão EUSART
        BitClr(PIE1, 4) ;

        return OK ;
    }
    return FAIL ;
}
```

No mapeamento de funções, pode-se perceber que a função invocada quando se deseja transmitir um dado é a *startSerialTx().* Nela é verificado se ainda existe algum dado que não foi transmitido, analisando-se o valor da variável *bufferFull*. No caso dela ser igual a 1*,* o buffer está cheio e a função retorna falha, e se for 0*,* o buffer está vazio e pode-se preparar a escrita. Primeiramente, o valor a ser escrito é guardado na variável interna *valueTx* que será usada posteriormente. Em seguida, *bufferFull* é carregada com o valor 1, indicando que existe um dado a ser escrito, e por fim, a interrupção de transmissão EUSART é ativada. Esta ocorre quando o microcontrolador está pronto para transmitir um dado. Na maioria dos casos ela ocorre assim que é ativada, e então, o fluxo é desviado para a rotina de tratamento.

```
/* rotina de tratamento de interrupção da transmissão
 * de dados da serial */
void serialTxISR() {
    // desativa interrupção de transmissão EUSART
    BitClr(PIE1, 4) ;

    // escreve no buffer de transmissão EUSART
    TXREG = valueTx ;

    // indica que não existe dado a ser escrito na serial
    bufferFull = 0 ;
}
```

A *serialTxISR()* é a responsável por efetivamente escrever o dado a ser transmitido no buffer de transmissão EUSART do microcontrolador. A interrupção de transmissão é então desativada pois, ao fim do processo de serialização interno e

transmissão do dado, a CPU estaria novamente pronta para novamente transmitir, o que ocasionaria uma interrupção indesejada.

O processo de desserialização também é realizado de modo automático pelo hardware do dispositivo. Assim que um *byte* for recebido e estiver disponível, o sistema seta o bit 5 do registro PIR1, tornando possível a leitura do valor disponível no registro RCREG.

A utilização de interrupções mais uma vez facilita e melhora o desempenho do sistema. O programador-usuário pode ativar a interrupção de recepção de dado serial com a utilização do seguinte código:

```
callDriver(DRV_SERIAL, SERIAL_INT_RX_EN,
           (void*) &proc_serialRx_callback) ;
```

Este trecho de uma aplicação mostra como ativar a interrupção serialRX já passando sua função de callback. Em nível de driver, o código acima invoca a função *enableSerialRxInterrupt()*.

```
// habilita recepção de dados serial por interrupção
char enableSerialRxInterrupt(void* parameters) {
    /* configura a função de retorno para interrupção
     * de recepção EUSART */
    callDriver(DRV_INTERRUPT, INT_SERIAL_RXSET,
               (void*)serialRxISR) ;

    /* configura a função de retorno para o programa principal
     * em caso de um tratamento de interrupção */
    callBack = parameters ;

    //ativa interrupção de recepção EUSART
    BitSet(PIE1, 5) ;

    return OK ;
}
```

A rotina de tratamento é passada ao driver de interrupção, a função de callback é carregada em uma variável interna e, finalmente, a interrupção de transmissão EUSART pode ser ativada.

```
// valor armazenado da última recepção de dados via serial
static unsigned char valueRx ;

// lê o buffer de recepção EUSART
void serialRxISR(void* parameters) {
    valueRx = RCREG ;
    kernelAddProc(callBack) ;
}
```

Quando ocorrer uma interrupção de transmissão EUSART, *serialRxISR()* será invocada, o dado recebido é, então, guardado em uma variável interna, *valueRx*, para ser acessado posteriormente, e a função de callback é escalonada no pool de processos do kernel.

Para que a aplicação tenha acesso ao dado recebido deve-se utilizar a seguinte linha de código:

```
callDriver(DRV_SERIAL, SERIAL_LAST_VALUE, &[resp]) ;
```

O dado será recebido na variável *resp* (uma variável qualquer). O procedimento interno ao driver é realizado pela função descrita abaixo.

```
/* retorna o valor lido na serial em uma variável
 * passada como parâmetro */
char serialRxReturnLastValue(void* parameters) {
    (*(unsigned char*)parameters) = valueRx ;
    return OK ;
}
```

A função *serialRxReturnLastValue()* apenas carrega no endereço passado como parâmetro o último valor recebido e guardado anteriormente em *valueRx*.

3.8.8 Driver de Teclado Matricial

O esquema de implementação do driver de teclado matricial pode ser visto na Figura 41.

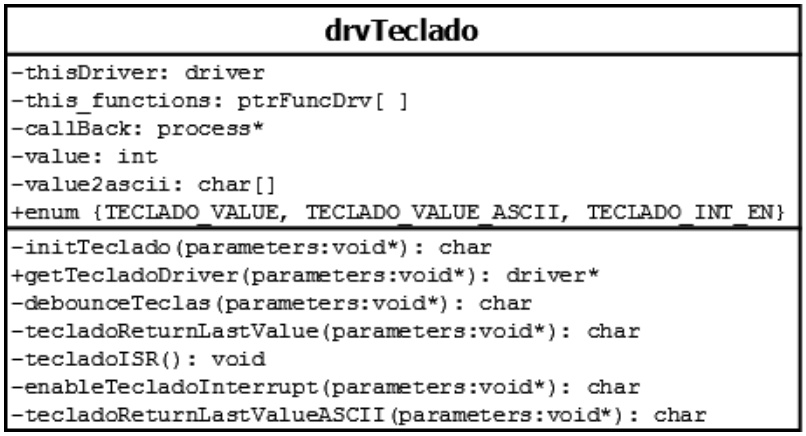


Figura 41: Esquema de implementação do driver de teclado matricial

O *debounce* por software em geral é utilizado em situações nas quais se deseja aumentar a robustez de uma entrada que já possua um *debounce* por hardware ou reduzir o custo da placa utilizando apenas a solução por software. A grande desvantagem deste tipo de sistema é inserir um atraso na detecção da informação.

Para realizar o *debounce* por software, precisa-se ter uma noção do tempo que a chave precisa para estabilizar. Da Figura 22, tem-se que este tempo, para uma determinada chave é de aproximadamente 150µs. Um ciclo de clock do sistema em questão é de 0,56µs. Antes de utilizar o valor lido na porta em questão, deve-se aguardar 300 ciclos de clock após alguma mudança para ter certeza que o sinal se

estabilizou, ou seja, a fase de *bouncing* acabou.

O *debounce* por software é realizado através da função *debounceTeclas()*.

```
// implementação da função de 'debounce'
char debounceTeclas(void) {
    unsigned char i, j ;
    unsigned char cont = 0x00 ;
    unsigned char tempo = MIN_TIME_DEBOUNCE ;
    volatile unsigned int valorNovo = 0x0000 ;
    volatile unsigned int valorAntigo = value;
   //código filtro debounce
}
```

A variável temporária *cont* é usada como um contador que indica quantas passagens no filtro de *debounce* ocorreram, se o valor nas teclas ficar oscilando por muito tempo, *cont* chegará ao valor de *MAX_TIME_DEBOUNCE*, indicando que o hardware de processamento não consegue distinguir com exatidão o status do teclado, retornando assim, uma falha. Já a variável *tempo* indicará se o valor pode ser considerado correto. As demais variáveis serão comentadas mais à frente.

O filtro *debounce* consiste em verificar várias vezes a entrada do teclado em um curto espaço de tempo, procurando garantir que o valor lido não é apenas um ruído. A cada ciclo de verificação, o status do teclado é colocado em uma variável auxiliar, *valorNovo*, e então comparado com o valor obtido no ciclo anterior, *valorAntigo*. Se o valor for o mesmo, há grandes chances desse valor ser realmente o valor correto. A variável que determina quantas vezes o status do teclado deve ser verificado para ser considerado aceitável é a variável *tempo*, iniciada com o valor da constante *MIN_TIME_DEBOUNCE. A* cada nova leitura do teclado, se *valorAntigo* continuar igual a *valorNovo*, *tempo* será decrementado até chegar a zero.

```
while ((tempo > 0) && (cont < MAX_TIME_DEBOUNCE)) {
    {
        /* código que retorna o status do teclado na variável
         * valorNovo. Esse trecho será comentado mais a frente. */
    }

    /* rotina de 'debounce' propriamente dita: a cada chamada
     * o tempo é decrementado em 1 unidade */
     if (valorAntigo == valorNovo) {
         tempo-- ;
     }
     else {
         tempo = MIN_TIME_DEBOUNCE ;
         valorAntigo = valorNovo ;
         cont++ ;
     }
}
```

Com o valor lido sendo considerado correto, basta analisar se esse valor já não era o valor do teclado, indicando que não houve nenhuma modificação. Caso

contrário, houve uma mudança no status do teclado e a função de *callback* do mesmo deve ser escalonada.

```
if (tempo == 0) {
    if (value != valorAntigo){
        value = valorAntigo ;
        if (value) {
            kernelAddProc(callBack) ;
        }
    }

     /* são necessárias duas leituras à porta B, para que haja
      * estabilidade na leitura do sinal */
    PORTB &= 0xf0 ;
    PORTB &= 0xf0 ;

/* desativa a flag de interrupção, possivelmente ativada
 * durante a oscilação do sinal no pressionamento da tecla */
    BitClr(INTCON, 0) ;

    //ativa a interrupção por mudança de estado na Porta B
    BitSet(INTCON, 3) ;
}
else
{
    return FAIL ;
}
return OK ;
```

Se o valor de *tempo* não for zero, é indicação de que ocorreu algum problema na fase de *debounce* e o driver retorna falha.

Notar que, no código, o contador é iniciado com o valor *MIN_TIME_DEBOUNCE*. Através da análise do assembler pode-se saber que cada ciclo de conferência do sinal possui 14 instruções. Assim é necessário que o sinal permaneça com o mesmo valor durante 308 ciclos para que a variável *valAtual* receba o valor da porta B. Estes valores podem ser determinados empiricamente através de testes com osciloscópios.

De acordo com o esquemático do teclado da Figura 25, os terminais ligados às linhas serão configurados como entrada, que servirão para ler os valores das teclas. Os terminais ligados às colunas serão configurados como saídas, fornecendo energia para as chaves. A leitura é realizada então por um processo conhecido como varredura: liga-se uma coluna por vez e verifica-se quais chaves daquela coluna estão ligadas.

```
for (i = 0; i < 4; i++) {
    // desliga todas as colunas
    PORTB |= 0x0F ;
    // liga a coluna correspondente
    PORTB &= ~(1<<i) ;
    // teste para cada bit, atualiza a variável
    for (j = 0; j < 4; j++) {
        if (!BitTst(PORTB,j+4)) {
            BitSet(valorNovo,(i*4)+j) ;
        }
        else {
            BitClr(valorNovo,(i*4)+j) ;
        }
    }
}
```

Cada tecla/terminal do *PORTB* é testada e seu valor colocado na variável de status (*valorNovo*).

```
char tecladoReturnLastValue(void* parameters) {
    (*(unsigned int*)parameters) = value ;
    return OK ;
}
```

Esta função retorna o último status verificado da matriz de teclas a partir do valor salvo na variável interna *value*.

Com o intuito de fornecer praticidade ao programador-usuário que utiliza o driver do teclado, foi criada uma função que retorna o valor lido da matriz de tecla já em código ASCII, ou seja, pronta para ser exibida em um LCD, por exemplo.

```
/* retorna na variável recebida o resultado da última leitura
 * do teclado, em código ASCII */
char tecladoReturnLastValueASCII(void* parameters) {
    unsigned char i ;
    for (i = 0; i < 16; i++) {
        if (BitTst(value, i)) {
            (*(unsigned char*)parameters) = value2ascii[i] ;
            return OK ;
        }
    }
    /* se nenhuma tecla foi pressionada, então
     * não há código ASCII */
    return FAIL ;
}
```

A função *tecladoReturnLastValueASCII()* é a responsável por fazer a conversão para ASCII. A implementação foi simplificada pela utilização de um vetor que indica qual o valor ASCII correspondente a cada valor de *value*.

```
// retorna o caracter da última tecla pressionada
static unsigned char value2ascii[] = {'A', 'B', 'C', 'D', '3',
        '6', '9', 'F', '2', '5', '8', '0', '1', '4', '7', 'E'} ;
```

4 Resultados obtidos

Foi desenvolvido com sucesso neste trabalho um kernel de um sistema operacional de tempo real, onde o escalonador se preocupa com os requisitos de tempo requisitados pelos processos.

Para o sistema proposto, o correto atendimento das requisições temporais depende da quantidade e complexidade dos processos. Não é possível garantir deterministicamente o atendimento de uma ou mais funções, o que torna o sistema proposto mais similar a um *soft-realtime* do que um *hard-realtime*.

O sistema apresenta três grandes divisões na implementação: os drivers, o kernel e a aplicação. O correto funcionamento destas depende da comunicação entre as partes, feitas neste trabalho através da passagem de mensagens por ponteiros de void. Este método apresentou bons resultados, mas exige maior cuidado do programador na utilização das interfaces.

Pode-se portanto notar que é uma estrutura completamente entrelaçada, de modo que nenhuma das partes opera ou oferece utilidade por si só. A estrutura desenvolvida pode ser visualizada na Figura 42. Esta estrutura sintetiza as partes e o funcionamento do sistema como um todo.

Em termos de hardware, o consumo de memória aleatória (RAM) e memória de armazenamento (Flash ROM) é exibido na tabela a seguir:

	RAM (bytes)	**ROM (Kbytes)**	**Linhas de código**
Kernel (basico.h, config.h, kernel_types.h, kernel_prm.h, kernel.h, kernel.c)	461	2,24	158
Controladora de drivers (ddCtr_types.h, ddCtr_prm.h, ddCtr.h, ddCtr.c)	154	0,64	76
Drivers	384	11,84	448

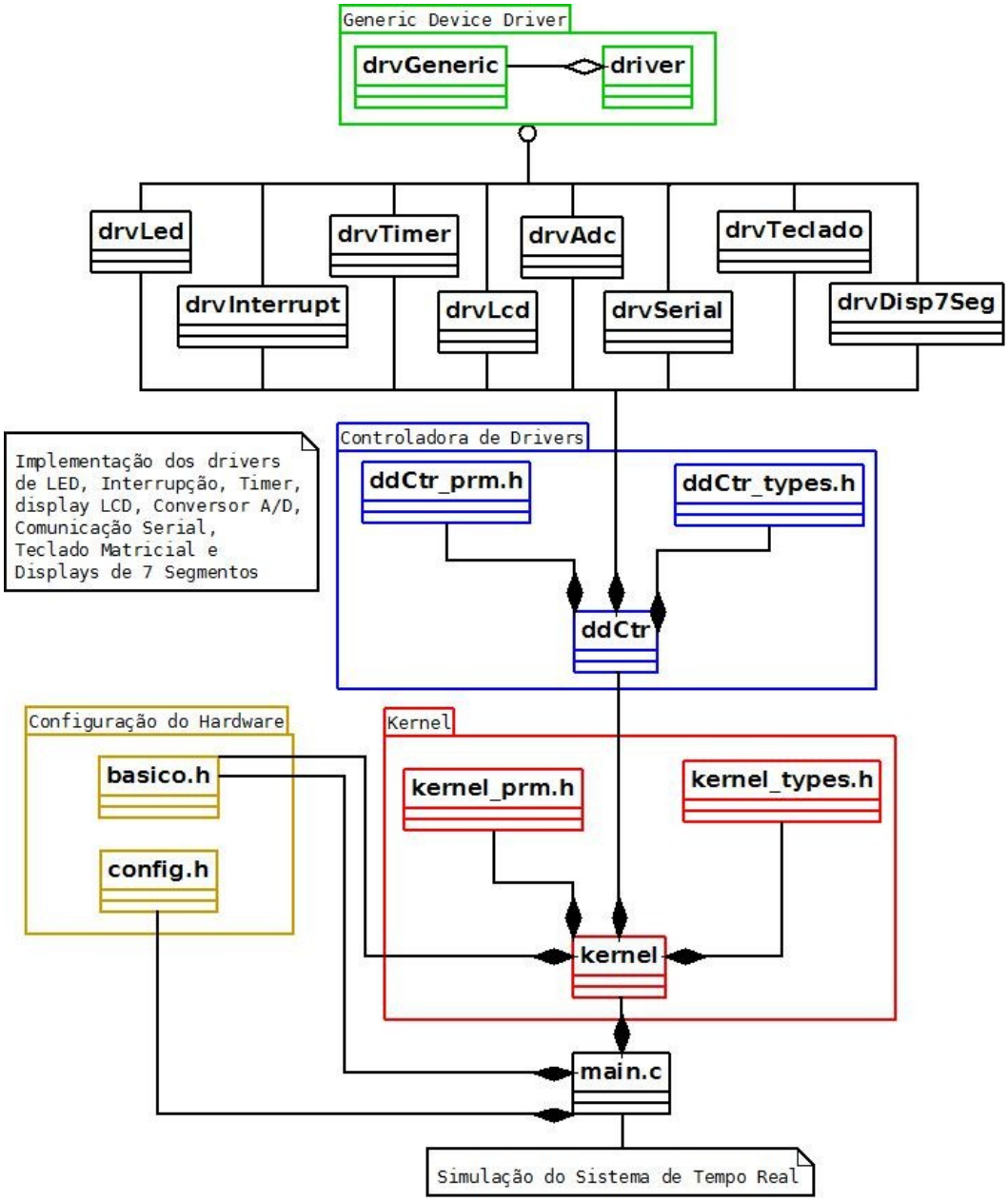


Figura 42: Diagrama do sistema completo

Resultado já esperado, o Kernel é o maior consumidor de memória RAM, como visto na Figura 43, pois é o responsável por gerar o dinamismo do sistema de tempo real, visto que insere e organiza os processos em buffer, além de colocá-los para execução.

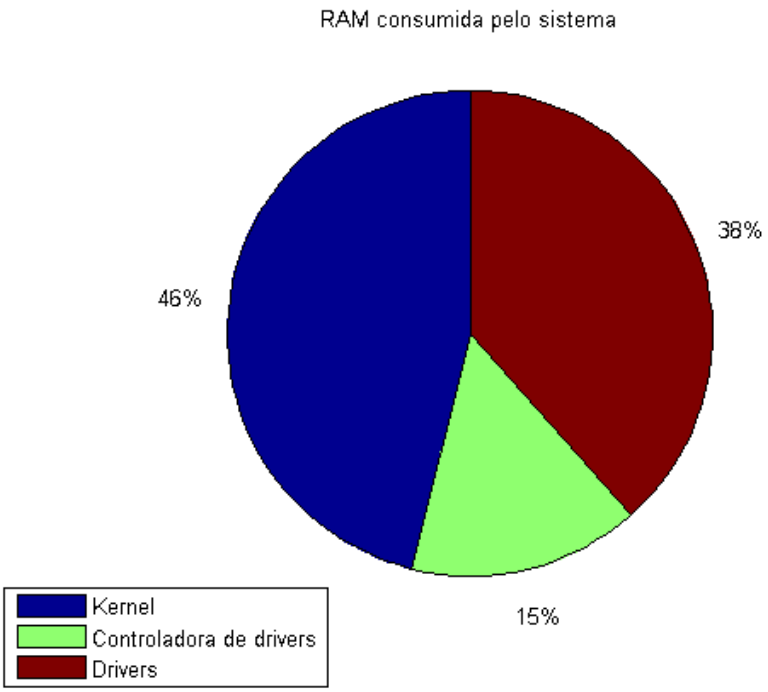


Figura 43: Memória RAM consumida pelo sistema

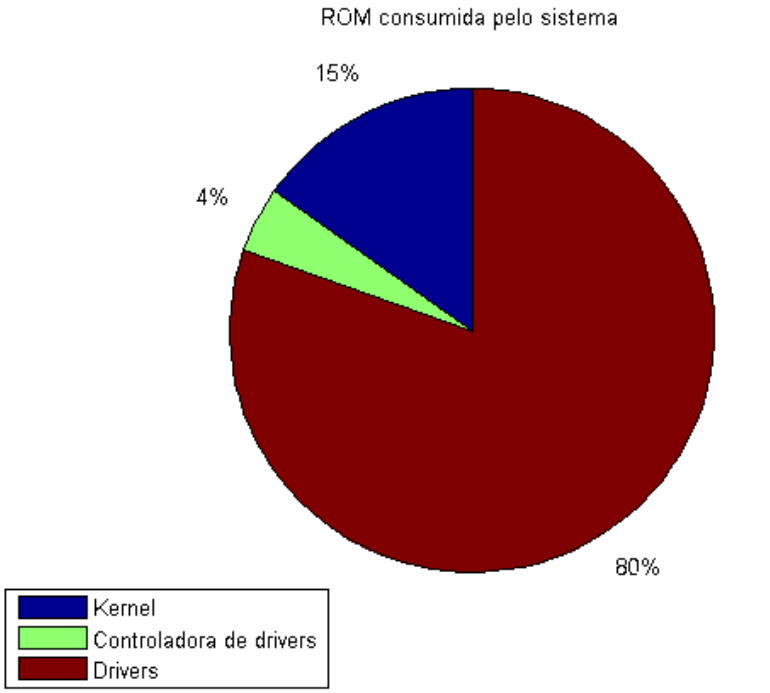


Figura 44: Memória ROM consumida pelo sistema

No caso da memória Flash do microcontrolador, a maior parte é ocupada pelos drivers, sendo demonstrado no gráfico da Figura 44, visto que necessitam de maior detalhamento de suas especificações, por se tratar da definição do comportamento de um hardware, ou seja, maior complexidade para descrição. Tal caractertística se reflete na quantidade de linhas de código necessárias para suas

implementações, como é melhor explicitado na tabela abaixo:

Arquivo	**Linhas de código**
ddCtr_prm.h	33
ddCtr_types.h	10
ddCtr.c	25
basico.h	55
config.h	15
ddCtr.h	8
kernel_types.h	9
kernel_prm.h	4
kernel.h	11
kernel.c	64
drvAdc.h	13
drvDisp7seg.h	12
drvInterrupt.h	15
drvTimer.h	14
drvTeclado.h	15
drvSerial.h	13
drvLed.h	10
drvLcd.h	16
drvTimer.c	43
drvTeclado.c	96
drvSerial.c	65
drvLed.c	18
drvLcd.c	160
drvInterrupt.c	73
drvDisp7seg.c	76
drvAdc.c	43

5 Conclusão

Como trabalhos futuros propomos algumas alterações no sistema desenvolvido. Estas alterações visam melhorar a performance do sistema, apresentar uma interface mais amigável ao programador ou melhorar os requisitos de *realtime* das aplicações.

A primeira é transformar o kernel do sistema, antes cooperativo, para um sistema preemptivo prioritário, de modo que processos de baixa prioridade pudessem ser interrompidos para que os de alta prioridade não sofressem atraso.

A segunda alteração é a melhoria do driver da comunicação serial. Propõe-se a criação de um buffer interno que armazenaria constantemente os últimos dados que foram transferidos ou recebidos e, deste modo, não seria necessário constantemente habilitar/desabilitar o driver. Isto diminuiria o overhead da reconfiguração do registro de interrupção do driver mas exigiria mais memória do sistema. A viabilidade da alteração é atrelada ao hardware disponível. Uma opção seria ainda permitir as duas opções em código, onde o uso seria condicionado por diretivas de pré-compilação.

A terceira alteração se refere ao teclado matricial. Ao invés de gerar uma interrupção a cada pressionamento de tecla, pode-se trabalhar sob *pooling*, ou seja, faz-se uma varredura do estado de todas as teclas a cada intervalo de tempo pré-determinado. Tal técnica gera um maior *overhead* e aumenta a latência do sistema, no entretanto possui uma implementação mais simples e intuitiva, diminuindo a dependência do sistema frente ao hardware de interrupções. O mais interessante é disponibilizar as duas implementações para escolha do programador.

6 Referências Bibliográficas

BARR, M., Programing Embedded Systems in C and C++. O'Reilly, 1999.

BARRET P., Embedded Systems. Design and Applications with the 68HC12 and HCS12, Prentice Hall, 2004.

BARROS E., CAVALCANTE S., Introdução aos Sistemas Embarcados, UFPE, 2002

BARRY, R, FreeRTOS an open source real time operating system, http://www.freertos.org/ acessado em 19/10/2010.

CHEN M.I., LIN K.J., Dynamic priority ceilings: A concurrency control protocol for real-time systems, Real-Time Systems, Springer – 1990.

FARINES, J. M., FRAGA, J. D., AND OLIVEIRA, R. S., Sistemas de Tempo Real. Escola de Computação 2000.

GOVERNO FEDERAL, Guia Livre - Referência de Migração para Software livre, versão 1.0, disponível no site http://www.governoeletronico.gov.br/anexos/guia-livre-versao-1.0, acessado em 13/08/2010.

HITACHI, Datasheet do controlador de display de LCD HD44780, disponível no site http://www.sparkfun.com/datasheets/LCD/HD44780.pdf, acessado em 13/03/2012.

JAHANIAN F., MOK A.K.L., Safety analysis of timing properties in real-time systems, IEEE Transactions on software engineering, 1986

LUTZ R.R., Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems, IEEE Transactions on Software Engineering, 1993

OLIVEIRA A.S., ANDRADE F.S., Sistemas embarcados - Hardware e firmware na prática, Editora: Érica, ISBN-10: 8536501057, 2006

PEREIRA F., Microcontroladores PIC – Programação em C, ISBN: 978-85-7194-935-5, 7ª Edição, Editora Érica 2009.

PUSCHNER P., KOZA C., Calculating the maximum execution time of real-time programs, Real-Time Systems, Springer – 1989.

RENAUX D., BRAGA A., KAWAMURA A., PERF: Um Ambiente Para Avaliação Temporal de Sistemas em Tempo Real, II Workshop de Sistemas em Tempo Real. 1989

ROWE A., ROSENBERG C., NOURBAKHSH I., A low cost embedded color vision system, Proceedings of IROS 2002

RUSHBY J., Bus architectures for safety-critical embedded systems, Embedded Software, Springer – 2001

SILVA J.D.S, Uma Plataforma para Sistemas Embarcados: Desenvolvimento e avaliação de desempenho de um Processador RTL e uma Cache L1 usando SystemC, Relatório Final de Graduação - Bacharel em Ciências UFRN, 2006

STALLINGS, W. Arquitetura e organização de computadores, Projeto para o desempenho, Editora Pearson-Prentice Hall 5a edição, 2002.

TANEMBAUM, A. TORVALDS, L.B., LINUX is obsolete, comp.os.minix, disponível no site http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html, acessado em 13/03/2012.

7 Anexos

7.1 Arquivo de definição de funções sobre bits e registros

```
//-------------------------------------------------------------------
// basico.h -> definições de funções sobre bits e registros do PIC18F4550
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//           rodrigomax at unifei.edu.br
//-------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//-------------------------------------------------------------------

#ifndef BASICO_H
    #define BASICO_H

    //códigos de retorno de funções
    enum {
        OK,
        FAIL,
        REPEAT,
        DRV_FUNC_NOT_FOUND
    };

    //define ponteiro vazio
    #define NULL    0

    //tamanhos maximos
    #define MIN_INT     -30000

    //funções de bit
    #define BitSet(arg,bit) ((arg) |= (1<<bit))
    #define BitClr(arg,bit) ((arg) &= ~(1<<bit))
    #define BitFlp(arg,bit) ((arg) ^= (1<<bit))
    #define BitTst(arg,bit) ((arg) & (1<<bit))
```

```
    //defines para registros especiais
    #define OSCCON  (*(volatile __near unsigned char*)0xFD3)
    #define PORTA   (*(volatile __near unsigned char*)0xF80)
    #define PORTB   (*(volatile __near unsigned char*)0xF81)
    #define PORTC   (*(volatile __near unsigned char*)0xF82)
    #define PORTD   (*(volatile __near unsigned char*)0xF83)
    #define PORTE   (*(volatile __near unsigned char*)0xF84)

    #define TRISA   (*(volatile __near unsigned char*)0xF92)
    #define TRISB   (*(volatile __near unsigned char*)0xF93)
    #define TRISC   (*(volatile __near unsigned char*)0xF94)
    #define TRISD   (*(volatile __near unsigned char*)0xF95)
    #define TRISE   (*(volatile __near unsigned char*)0xF96)

    #define INTCON  (*(volatile __near unsigned char*)0xFF2)
    #define INTCON2 (*(volatile __near unsigned char*)0xFF1)
    #define PIE1    (*(volatile __near unsigned char*)0xF9D)
    #define PIR1    (*(volatile __near unsigned char*)0xF9E)

    #define TMR0L   (*(volatile __near unsigned char*)0xFD6)
    #define TMR0H   (*(volatile __near unsigned char*)0xFD7)
    #define T0CON   (*(volatile __near unsigned char*)0xFD5)

    #define SPPCON  (*(volatile __near unsigned char*)0xF65)
    #define SPPCFG  (*(volatile __near unsigned char*)0xF63)
    #define ADCON2  (*(volatile __near unsigned char*)0xFC0)
    #define ADCON1  (*(volatile __near unsigned char*)0xFC1)
    #define ADCON0  (*(volatile __near unsigned char*)0xFC2)
    #define ADRESL  (*(volatile __near unsigned char*)0xFC3)
    #define ADRESH  (*(volatile __near unsigned char*)0xFC4)

    #define RCSTA   (*(volatile __near unsigned char*)0xFAB)
    #define TXSTA   (*(volatile __near unsigned char*)0xFAC)
    #define TXREG   (*(volatile __near unsigned char*)0xFAD)
    #define RCREG   (*(volatile __near unsigned char*)0xFAE)
    #define SPBRG   (*(volatile __near unsigned char*)0xFAF)
    #define SPBRGH  (*(volatile __near unsigned char*)0xFB0)
    #define BAUDCON (*(volatile __near unsigned char*)0xFB8)

    #define RCON    (*(volatile __near unsigned char*)0xFD0)
    #define WDTCON  (*(volatile __near unsigned char*)0xFD1)
    #define T2CON   (*(volatile __near unsigned char*)0xFCA)
    #define PR2     (*(volatile __near unsigned char*)0xFCB)
    #define CCP2CON (*(volatile __near unsigned char*)0xFBA)
    #define CCPR2L  (*(volatile __near unsigned char*)0xFBB)
    #define CCP1CON (*(volatile __near unsigned char*)0xFBD)
    #define CCPR1L  (*(volatile __near unsigned char*)0xFBE)

#endif
```

7.2 Arquivo de configuração dos registros do microcontrolador

```
//-------------------------------------------------------------------------
//   config.h -> configurações do microcontrolador
//   PIC18F4550 (compilador SDCC)
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//           rodrigomax at unifei.edu.br
//-------------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//-------------------------------------------------------------------------

#ifndef CONFIG_H
    #define CONFIG_H

//configurações do microcontrolador
code char at 0x300000 CONFIG1L = 0x00 ; // sem 'prescale' no PLL
code char at 0x300001 CONFIG1H = 0x0C ; // oscilador 'HS'
code char at 0x300002 CONFIG2L = 0x1F ; // Power-up ativo e Brown-out
                          // ativado em configuração mínima

code char at 0x300003 CONFIG2H = 0b00011110 ; // controle do 'Watchdog'
                          // via bit 'SWDTEN' e 'postscale' 1:256

code char at 0x300005 CONFIG3H = 0x83 ; // Master Clear Reset ativo,
                          // RE3 desabilitado

code char at 0x300006 CONFIG4L = 0b10000000 ; // programção em
                          // baixa tensão desativada

code char at 0x300008 CONFIG5L = 0x0F ; // proteção de código desativada
code char at 0x300009 CONFIG5H = 0xC0 ; // proteção de código da
                          // EEPROM desativada

code char at 0x30000A CONFIG6L = 0x0F ; // proteção contra escrita
                          // na EEPROM de 0x800 até 0x7fff desativada
code char at 0x30000B CONFIG6H = 0xE0 ; // proteção contra escrita
                          // na EEPROM de 0x0 até 0x7ff desativada

code char at 0x30000C CONFIG7L = 0x0F ; // proteção contra leitura
                          // na EEPROM de 0x800 até 0x7fff desativada
code char at 0x30000D CONFIG7H = 0x40 ; // proteção contra escrita
                          // na EEPROM de 0x0 até 0x7ff desativada

#endif
```

7.3 Arquivos de implementação do kernel

7.4 kernel_prm.h

```
//---------------------------------------------------------------------
//   kernel_prm.h -> header com parâmetros do kernel
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//          rodrigomax at unifei.edu.br
//---------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//---------------------------------------------------------------------

#ifndef kernel_prm_h
    #define kernel_prm_h
    // a quantidade a seguir deve ser 1 a mais que a máxima desejada
    #define SLOT_SIZE 20 // número máximo de funções de drivers
                         // que podem ser inseridas no 'pool'
#endif // kernel_prm_h
```

7.4.1 kernel_types.h

```
//------------------------------------------------------------------------
//   kernel_types.h -> definições de tipos especiais para o kernel
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//           rodrigomax at unifei.edu.br
//------------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//------------------------------------------------------------------------
#ifndef kernel_types_h
    #define kernel_types_h

    //declaracao do tipo ponteiro para funcao
    typedef char(*ptrFunc)(void);

    //estrutura do processo
    typedef struct {
        ptrFunc function;
        unsigned int period;
        signed int start;
    } process;

#endif // kernel_types_h
```

7.4.2 kernel.h

```
//------------------------------------------------------------------------
//   kernel.h -> header das funções para gerenciamento do kernel
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//           rodrigomax at unifei.edu.br
//------------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//------------------------------------------------------------------------

#ifndef KERNEL_H
    #define KERNEL_H

    #include "basico.h"
    #include "kernel_prm.h"
    #include "kernel_types.h"
    #include "ddCtr.h"

    //funcoes do kernel
    char kernelInit(void);
    char kernelAddProc(process *func);
    void kernelLoop(void);
    void kernelClock(void);
#endif //KERNEL_H
```

7.4.3 kernel.c

```
//------------------------------------------------------------------------
//   kernel.c -> funções para gerenciamento do kernel
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//           rodrigomax at unifei.edu.br
//------------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//------------------------------------------------------------------------

/* De acordo com o manual do SDCC
 * (http://sdcc.sourceforge.net/doc/sdccman.html/node183.html):
 * Não pode ser realizada passagem de parâmetro por valor
 * em struct e unions, somente passagem por referência!
*/
#include "kernel.h"

//variáveis do kernel
static process* pool[SLOT_SIZE]; //'pool' de processos
static volatile unsigned char start; // posição inicial
                        // do 'pool' de processos
static volatile unsigned char end; //posição final do 'pool' de processos

//inicializa o kernel em conjunto com a controladora de drivers
char kernelInit(void) {
    BitSet(OSCCON, 7) ; //'Idle mode' em caso de instrução SLEEP
    initCtrDrv();
    start = 0;
    end = 0;
    return OK;
}

// executa os processos do 'pool' de acordo com seus tempos de execução
void kernelLoop(void) {
    unsigned char j;
    unsigned char next;
    process *tempProc;
    for (;;) {
        if (start != end) {
            // procura a próxima função a ser executada com base no tempo
            j = (start + 1) % SLOT_SIZE;
            next = start;
            while (j != end) {
                if ((pool[j]->start) < (pool[next]->start)) {
                    next = j;
                }
                j = (j + 1) % SLOT_SIZE;
              // para poder incrementar e ciclar o contador
            }
            // troca e coloca o processo com menor tempo como o próximo
            tempProc = pool[next];
            pool[next] = pool[start];
            pool[start] = tempProc;
```

```
            while ((pool[start]->start) > 0) {
                // coloca a cpu em modo de economia de energia
                _asm
                    SLEEP
                _endasm ;
            }

            // retorna se precisa repetir novamente ou não
            switch (pool[start]->function()) {
                case REPEAT:
                    kernelAddProc(pool[start]);
                    break;
                case FAIL:
                    break;
                default:;
            }
            // próxima função
            start = (start + 1) % SLOT_SIZE;
        }
    }
}
// adiciona os processos no pool
char kernelAddProc(process *func) {
    // adiciona processo somente se houver espaço livre
    // o fim nunca pode coincidir com o inicio
    if (((end + 1) % SLOT_SIZE) != start) {
        // adiciona o novo processo e agenda para executar imediatamente
        func->start += func->period;
        pool[end] = func;

        end = (end + 1) % SLOT_SIZE;
        return OK; //sucesso
    }
    return FAIL; //falha
}

// atualiza os tempos de execução dos processos
void kernelClock(void) {
    unsigned char i;
    i = start;
    while (i != end) {
        if ((pool[i]->start)>(MIN_INT)) {
            pool[i]->start--;
        }
        i = (i + 1) % SLOT_SIZE;
    }
}
```

7.5 Arquivos de implementação da controladora de drivers

7.5.1 ddCtr_prm.h

```
//---------------------------------------------------------------------------
//   ddCtr_prm.h -> parâmetros da controladora, e drivers implementados
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//           rodrigomax at unifei.edu.br
//---------------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//---------------------------------------------------------------------------
#ifndef ddCtr_prm_h
    #define ddCtr_prm_h

    // quantidade máxima de drivers disponíveis para o
    // sistema simultaneamente
    #define QNTD_DRV 20

    // é necessário para incluir todos os arquivos de drivers
    #include "drvLed.h"
    #include "drvInterrupt.h"
    #include "drvTimer.h"
    #include "drvLcd.h"
    #include "drvAdc.h"
    #include "drvSerial.h"
    #include "drvTeclado.h"
    #include "drvDisp7seg.h"

    //este enumerado auxilia o desenvolvedor/usuário a acessar os drivers
    enum {
        DRV_LED, DRV_INTERRUPT, DRV_TIMER, DRV_LCD, DRV_ADC, DRV_SERIAL,
        DRV_TECLADO, DRV_DISP7SEG, DRV_END
    };
/*  DRV_END  deve  sempre  ser  o  último,  para  facilitar  o  controle  da
quantidade de drivers */
// as funções para obter os drivers devem ser inseridas
// na mesma ordem que o enumerado acima
    static ptrGetDrv drvGetFunc[DRV_END] = {
        getLedDriver,
        getInterruptDriver,
        getTimerDriver,
        getLCDDriver,
        getAdcDriver,
        getSerialDriver,
        getTecladoDriver,
        getDisp7segDriver
    };

#endif // ddCtr_prm_h
```

7.5.2 ddCtr_types.h

```
//---------------------------------------------------------------------
//   ddCtr_types.h -> definições de tipos especiais
//                da controladora de drivers
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//          rodrigomax at unifei.edu.br
//---------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//---------------------------------------------------------------------

#ifndef ddCtr_types_h
    #define ddCtr_types_h

    typedef char(*ptrFuncDrv)(void *parameters);

    //estrutura do driver
    typedef struct {
        char drv_id; // identificador do driver
        ptrFuncDrv drv_init; // ponteiro da função de
                    // inicialização do driver
        ptrFuncDrv *func_ptr; // vetor dos demais ponteiros
                     // de funções do driver
    } driver;

    // definição de um ponteiro de função para drivers
    // (com número indefinido de parâmetros)
    typedef driver* (*ptrGetDrv)(void);

#endif // ddCtr_types_h
```

7.5.3 ddCtr.h

```
//---------------------------------------------------------------------
//   ddCtr.h -> header das funções de gerenciamento dos
//              drivers de dispositivos
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//          rodrigomax at unifei.edu.br
//---------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//---------------------------------------------------------------------

#ifndef ddCtr_h
    #define ddCtr_h

    #include "ddCtr_prm.h"
    #include "ddCtr_types.h"
    #include "basico.h"
    char initCtrDrv(void);
    char initDriver(char newDriver);
    char callDriver(char drv_id, char func_id, void *parameters);

#endif // ddCtr_h
```

7.5.4 ddCtr.c

```
//---------------------------------------------------------------------
//   ddCtr.c -> funções de gerenciamento dos drivers de dispositivos
//   Autor:  Rodrigo Maximiano Antunes de Almeida
//           rodrigomax at unifei.edu.br
//---------------------------------------------------------------------
//   This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; version 2 of the License.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//---------------------------------------------------------------------

#include "ddCtr.h"

static driver* driversLoaded[QNTD_DRV]; //vetor com os drivers iniciados
static char qntDrvLoaded;

//inicializa a controladora de drivers
char initCtrDrv(void) {
    qntDrvLoaded = 0;
    return OK;
}

//inicializa um determinado driver
char initDriver(char newDriver) {
    char resp = FAIL;
    if (qntDrvLoaded < QNTD_DRV) {
        driversLoaded[qntDrvLoaded] = drvGetFunc[newDriver]();
        resp = driversLoaded[qntDrvLoaded]->drv_init(&newDriver);
        qntDrvLoaded++;
    }
    return resp;
}

// transfere a um determinado driver uma função
// a ser executada, em conjunto com seus parâmetros
char callDriver(char drv_id, char func_id, void *parameters) {
    char i;

    for (i = 0; i < qntDrvLoaded; i++) {
        if (drv_id == driversLoaded[i]->drv_id) {
            return driversLoaded[i]->func_ptr[func_id](parameters);
        }
    }
    return DRV_FUNC_NOT_FOUND;
}
```

_____

Prof. Msc. Rodrigo Maximiano Antunes de Almeida
Orientador

_____

César Augusto Marcelino dos Santos
Orientado

Itajubá, 30 de março de 2012.