Capítulo 7

Estrutura de Dados

Autora: Wu Shin-Ting

Entendemos como uma **estrutura de dados** não só a forma como os dados são organizados como também um conjunto de operações definidas sobre estes dados. Embora a memória seja um recurso crítico nos sistemas embarcados, em muitas situações é comum sacrificá-la em favor do tempo de processamento. Estruturas de dados adequadamente selecionadas podem proporcionar uma forma mais eficiente na manipulação dos dados por um programa e podem propiciar a elaboração de algoritmos mais complexos e flexíveis.

Vimos no Capítulo 4 que a linguagem C suporta diferentes tipos de dados através dos quais podemos alocar, de forma diferenciada, pequenos espaços de mamória estritamente necessários aos dados da nossa aplicação. Mostramos também como encapsular um conjunto pequeno de dados de variados tipos via *struct*, e como organizar sequencialmente um conjunto maior de dados de mesma natureza através de *arrays*. No Capítulo 6 falamos sobre duas estruturas de dados mais elaboradas no processamento de interrupções implementadas no nosso microcontrolador. A primeira é a Tabela de Vetores de Interrupção utilizada para armazenar os endereços das rotinas de serviço e a segunda é a pilha utilizada para salvar o estado do contexto corrente (modo *Thread* ou modo *Handler*) antes do chaveamento para o modo *Handler*. Neste capítulo vamos mostrtar como podemos implementar tais estruturas com os tipos de dados nativos da linguagem C e apresentar mais três estruturas de dados comumente encontradas em sistemas embarcados [1].

7.1 Tabela de Desvios

Tabela de desvios (*jump table* ou *branch table*) é uma alternativa para desviar o fluxo de controle corrente para uma rotina sem fazer uso de comandos de desvio, como **if-then-else** ou **switch**. Usualmente ela é implementada como um **arranjo de funções** em linguagem de alto nível ou um arranjo de instruções de desvio incondicional em linguagem de montagem, de forma que se pode desviar para o fluxo de controle de uma nova função **operando diretamente sobre os índices do arranjo**, como mostra o seguinte trecho de código em C. Foi declarado no código um arranjo “funcoes” de três endereços de funções, “func_a”, “func_b” e “func_c”, de froma que, para acessar cada uma destas três funções, só precisamos variar o índice do arranjo. As três funções correspondem, respectivamente, aos elementos funcoes[0], funcoes[1] e funcoes[2].

```
void func_a();
void func_b();
void func_c();

void (*const funcoes[3])(void) = {func_a, func_b, func_c};;

int variavel;
void func_a() {
        variavel = 10;
        return;
}

void func_b() {
        variavel = 20;
        return;
}

void func_c() {
        variavel = 30;
        return;
}
```

Manipular índices é muito mais eficiente do que processar comandos de desvio. Podemos, portanto, ter um ganho no desempenho do sistema com uso de tabela de desvios. Além disso, ao tratarmos os endereços das funções como elementos (dados) de um arranjo, ganhamos a flexibilidade de modificá-lo em tempo de execução sem precisar recompilar o código. Vale observar que no trecho do código dado, o conteúdo do arranjo foi declarado com o qualificador **const**; portanto, não é modificável durante o tempo de execução.

Se compararmos o arranjo “funcoes” com o arranjo “InterruptVector” mostrada na Figura 5 do Capítulo 6, é fácil perceber a similaridade na sintaxe dos dois arranjos. O índice de cada rotina de serviço no “InterruptVector” corresponde exatamente ao número de exceção da interrupção/exceção.

7.2 Pilhas

As **pilhas** (*stacks*) são também conhecidas como uma estrutura de dados linear LIFO (last-in, first out), onde o último elemento inserido será o primeiro elemento a ser retirado. Usualmente o índice do elemento do topo da pilha é armazenado numa variável para facilitar o seu processamento. Quando se precisa processar outros elementos inseridos, é comum usar o endereço do topo da pilha como endereço-base corrigido pelo deslocamento (*offset*) entre o topo da pilha e o endereço do elemento de interesse. Duas operações são associadas à manipulação de uma pilha: **empilhar** (*push*) e **desempilhar** (*pop*).

O seguinte trecho de código em C demonstra uma implementação simples da estrutura de uma pilha com uso de *struct* e arranjo de tamanho fixo alocado dinamicamente. Observe que para evitar o uso de “-1” para indicar que a pilha está vazia, o código foi implementado de tal forma que o índice do topo da pilha deve ser subtraído de 1 para obter o índice efetivo do elemento que está no topo da pilha.

```
#include <stdlib.h>
```

```c
struct pilha {
	unsigned int topo;
	unsigned int top;
	int *elementos;
};

void cria_pilha (struct pilha *p, unsigned int n)
{
	p->topo = 0;
	p->top = n;
	if (n == 0)
		p->elementos = NULL;
	else
		p->elementos = (int *)malloc(sizeof(int)*n);
	return;
}

uint8_t empilha (struct pilha *p, int v)
{
	if (p->topo == p->top) return 0;
	p->elementos[p->topo++] = v;
	return 1;
}

int desempilha (struct pilha *p)
{
	int v;
	if (p->topo) {
		v = p->elementos[p->topo-1];
		p->topo--;
		return v;
	} else {
		return 0xffffffff;
	}
}

int topo (struct pilha *p)
{
   return(p->elementos[p->topo-1]);
}

uint8_t pilha_cheia (struct pilha *p)
{
	if (p->topo == p->top) return 1;
	return 0;
}

uint8_t pilha_vazia (struct pilha *p)
{
	if (p->topo == 0) return 1;
	return 0;
}

int tamanho (struct pilha *p)
{
    return (p->top);
}

void deleta_pilha (struct pilha *p)
```

```
{
    if (p->elementos)
        free((void *)p->elementos);
    p->top = p->topo = 0;
    p->elementos = NULL;
}
```

Pilha é uma estrutura de dados utilizada pelo processador para salvar os dados de um contexto quando ocorre o desvio do fluxo de controle numa chamada de rotina ou no atendimento de uma exceção. Cada vez que se chaveia de um contexto para o outro, é empilhado automaticamente o conteúdo dos 8 registradores como mostra Figura 1. Quando ocorrem múltiplos chaveamentos, são empilhados múltiplos blocos de forma que o contexto mais recente fique no topo da pilha. Com isso, facilita a recuperação do contexto no retorno de cada exceção, da mais recente para a mais antiga. Basta ir desempilhando o bloco que estiver no topo à medida que for retornando de uma exceção.

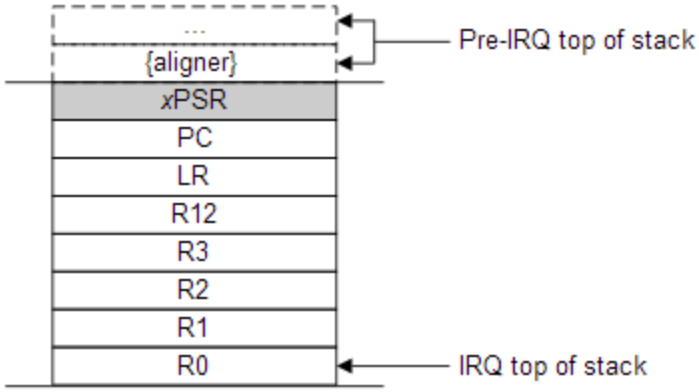

Figura 1: Empilhamento dos estados de cada contexto no processamento de interrupções aninhadas (Fonte: [2])

7.3 Filas

As **filas** (*queues*) são estruturas de dados lineares FIFO (first-in, first-out), onde o primeiro elemento inserido será o primeiro elemento a ser removido. Para poder inserir e remover de forma eficiente os elementos na fila, o índice do **último** elemento da fila é também armazendo na estrutura da fila. Duas operações são associadas à manipulação dos dados de uma fila: **enfileirar** (*enqueue*) e **desenfileirar** (*dequeue*).

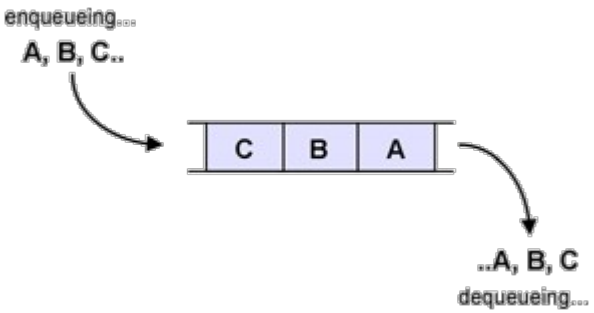

Figura 2: Fila (Fonte: [3])

O conceito de fila é aplicado em vários estágios de processamento no nosso microcontrolador. Vimos, por exemplo, no Capítulo 6 que mais de uma condição de interrupção pode acontecer num periférico. E todas estas condições convergem para uma única linha de interrupção IRQ do NVIC. Na forma como o NVIC processa as solicitações de interrupção, podemos considerar que há uma fila de solicitações com 2 lugares em cada linha: um lugar para solicitação pendente e um lugar para solicitação ativa. Isso significa que, quando ocorrem simultaneamente diferentes condições de interrupção, somente uma é reconhecida como pendente. Se esta exceção pendente deslocar para ativa, uma nova solicitação ocupará o lugar de pendência. Todas as outras condições serão "perdidas" se elas não mantiverem a sua solicitação até que consigam entrar no lugar de pendência.

A implementação em *software* do conceito de fila, em que a vacância de um lugar faz com que todos os elementos sucessores desloquem automaticmente de uma casa para frente, é usualmente complexa em termos da quantidade de acessos à memória. O seguinte trecho de código em C implementa uma fila de "num" elementos com uso de *struct* e arranjo de tamanho fixo alocado dinamicamente.

```
#include <stdlib.h>
#include <string.h>

struct fila {
        unsigned int ultimo;
        unsigned int num;
        int *elementos;
};

void cria_fila (struct fila *f, unsigned int n)
{
        f->ultimo = 0;
        f->num = n;
        if (n == 0)
                f->elementos = NULL;
        else
                f->elementos = (int *)malloc(sizeof(int)*n);
}

uint8_t enfileira (struct fila *f, int v)
{
        if (f->ultimo >= f->num) return 0;

   f->elementos[f->ultimo++] = v;
   return 1;
}
```

```
int desenfileira (struct fila *f)
{
        int v;

        if (f->ultimo > 0) {
           v = f->elementos[0];
           memcpy (&(f->elementos[0]), &(f->elementos[1]), (f->ultimo - 1)*sizeof(int));
           f->ultimo--;
           return v;
        }
        return 0xffffffff;     ///< invalid value
}

int primeiro (struct fila *f)
{
   return(f->elementos[0]);
}

int ultimo (struct fila *f)
{
        if (f->ultimo)
                return(f->elementos[f->ultimo-1]);
        else
                return 0xffffffff;   ///< invalid value
}

uint8_t fila_cheia (struct fila *f)
{
        if (f->ultimo == f->num) return 1;
        return 0;
}

uint8_t fila_vazia (struct fila *f)
{
        if (f->ultimo == 0) return 1;
        return 0;
}

int tamanho (struct fila *f) {
        return (f->num);
}

void deleta_fila (struct fila *f)
{
    if (f->elementos)
         free((void *)f->elementos);
    f->ultimo = f->num = 0;
    f->elementos = NULL;
}
```

7.4 *Buffer* Circular

Uma forma de contornar o custo de deslocamentos de todos os elementos quando se remove um elemento de uma fila de tamanho fixo *n* é utilizar *buffers* circulares. Um ***buffer* circular**

(*circular buffer*) é uma fila de tamanho fixo que não tem fim nem início, como mostra Figura 3. Esta estrutura dispõe de dois indexadores, ou ponteiros, para manipular os dados armazenados no *buffer*. Eles são chamados ***head*** e ***tail***, e são ilustrados como duas setas na Figura 3. O produtor (de dados) **coloca** (*put*) um dado no *buffer*, incrementando o indexador de acesso *head*, e o consumidor (de dados) **consome** (*remove*) um dado do *buffer*, incrementando o indexador de acesso *tail*.

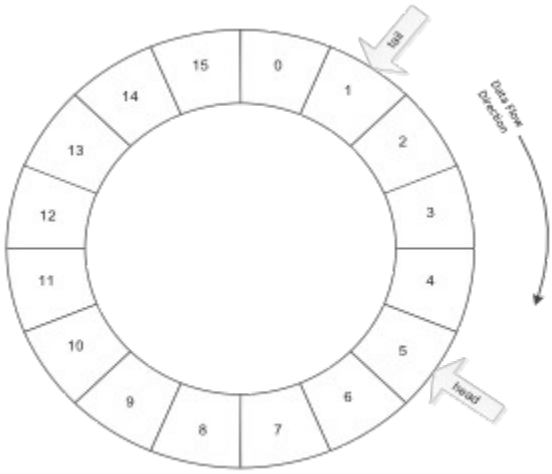

Figura 3: *Buffer* circular (Fonte: [4])

Podemos implementar esta estrutura com um arranjo de tamanho fixo, operando os seus índices com aritmética modular (%), de forma que o índice de acesso volta automaticamente para 0 quando se chega ao último elemento do arranjo. Uma implementação é mostrada no seguinte trecho de código em C:

```c
#include <stdlib.h>

struct bufCircular {
        unsigned int head;
        unsigned int tail;
        unsigned int num;
        int *elementos;
};

void cria_bufCircular (struct bufCircular *c, unsigned int n)
{
        c->head = 0;
        c->tail = 0;
        c->num = n;
        if (n == 0)
                c->elementos = NULL;
        else
                c->elementos = (int *)malloc(sizeof(int)*n);
        return;
}
```

```
uint8_t puxa (struct bufCircular *c, int v)
{
        unsigned int ind = (c->head+1)%c->num;
        if (ind == c->tail) return 0;
        c->elementos[c->head] = v;
        c->head = ind;
        return 1;
}

int remove (struct bufCircular *c)
{
        int v;
        if (c->head != c->tail) {
                v = c->elementos[c->tail];
                c->tail = (c->tail+1)%c->num;
                if (c->head == c->tail) c->head = c->tail = 0;
                return v;
        } else {
                return 0xffffffff;          ///< invalid value
        }
}

int tail (struct bufCircular *c)
{
        if (c->head != c->tail) {
                return(c->elementos[c->tail]);
        } else {
                return 0xffffffff;                  ///< invalid value
        }
}

int head (struct bufCircular *c)
{
        if (c->head != c->tail) {
                if (c->head == 0) return (c->elementos[c->num-1]);
                return(c->elementos[c->head-1]);
        } else {
                return 0xffffffff;                  ///< invalid value
        }
}

uint8_t bufCircular_cheia (struct bufCircular *c)
{
        if ((c->head+1)%c->num == c->tail) return 1;
        return 0;
}

uint8_t bufCircular_vazia (struct bufCircular *c)
{
        if (c->head == c->tail) return 1;
        return 0;
}

int tamanho (struct bufCircular *c)
{
    return (c->num);
}

void deleta_bufCircular (struct bufCircular *c)
```

```
{
    if (c->elementos)
        free((void *)c->elementos);
    c->head = c->tail = c->num =0;
    c->elementos = NULL;
}
```

Observe que a simples troca do movimento de elementos de um *buffer* pelo movimento de ponteiros consegue reduzir drasticamente a quantidade de acessos à memória, principalmente quando *n* for muito grande. Esta vantagem tem levado à substituição da estrutura fila pela *buffer* circular nos programas de sistemas embarcados, mesmo que a implementação da aritmética modular para manipular os índices seja um pouco mais trabalhosa. Hoje em dia, é uma estrutura muito utilizada em sistemas embarcados para transferência de dados entre processos de diferentes velocidades de processamento.

7.5 Listas Ligadas

Diferentemente das estruturas anteriores, a estrutura de lista ligada é uma estrutura dinâmica no sentido de que a quantidade dos seus elementos pode crescer ou diminuir conforme a demanda da aplicação. Uma **lista ligada** (*linked list*) é uma coleção de elementos alocados dinamicamente e conectados pelos ponteiros, como mostra Figura 4. Cada elemento contém um campo de endereço para o próximo elemento que não necessariamente precisa ser adjacente ao seu predecessor ou ao seu sucessor em termos de ocupação da memória.

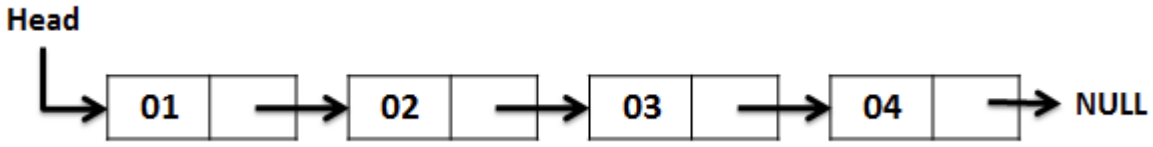

Figura 4: Lista ligada simples (Fonte: [5])

A lista ligada é usualmente recomendada em situações em que a quantidade total de elementos não é conhecida a priori. Através do recurso de alocação dinâmica oferecida pela linguagem C podemos inserir ou remover elementos da lista no tempo de execução como mostra o seguinte trecho de código em C:

```
#include <stdlib.h>

struct listaLigada {
        int elemento;
        struct listaLigada *prox;
};

void inicializa_listaLigada (struct listaLigada **h)
{
        *h = NULL;
        return;
}

void insere_item (struct listaLigada **h, int v)
{
        struct listaLigada *ptr = (struct listaLigada *)malloc(sizeof(struct listaLigada));
```

```
        ptr->prox = *h;
        ptr->elemento = v;
        *h = ptr;
}

uint8_t remove_item (struct listaLigada **h, int v)
{
        struct listaLigada *cur, *pred;

        for (pred=NULL, cur=*h; cur; pred=cur, cur=cur->prox) {
                if (cur->elemento == v) {
                        if (pred) {
                                pred->prox = cur->prox;
                        } else {
                                *h = cur->prox;
                        }
                        free((void *)cur);
                        return 1;
                }
        }
        return 0;
}

int tamanho (struct listaLigada *h)
{
        unsigned int i;
        struct listaLigada *ptr;

        for (i=0, ptr=h; ptr; ptr=ptr->prox) {
                i++;
        }
        return i;
}

void lista_2_arranjo (struct listaLigada *h, int **array)
{
        struct listaLigada *ptr;
        unsigned int i;

        *array = (int *)malloc(sizeof(int)*tamanho(h));

        for (i=0, ptr=h; ptr; ptr=ptr->prox) {
                (*array)[i++] = ptr->elemento;
        }
}

void deleta_listaLigada (struct listaLigada **h)
{
        struct listaLigada *ptr;

        while (*h) {
                ptr = *h;
                *h = ptr->prox;
                free((void *) ptr);
        }
   *h = NULL;
}
```

Observe que a função “lista_2_arranjo” converte uma lista ligada num arranjo cujo tamanho é computado dinamicamente, em tempo de execução. Em termos de memória e de facilidades em acesso aos elementos, o arranjo certamente supera em desempenho quando comparado com a lista ligada.

7.6 Estruturas de Dados em Sistemas Embarcados

É difícil conceber um algoritmo eficiente sem ter por trás uma boa estrutura de dados. Todos nós já utilizamos, por exemplo, o motor de busca (*search engine*) do *Google*. Você conseguiria imaginar um eficiente algoritmo de busca num amontoado de dados sem que estes estejam minimamente organizados [6]? Você teria alguma ideia de como o *Google map* consegue traçar diferentes rotas entre dois pontos quaisquer em tempo interativo, sem ter todas as possíveis rotas previamente estruturadas na memória [7]?

Nos projetos de programas para sistemas embarcados, onde se busca algoritmos eficientes tanto em termos de memória quanto em termos de tempo de execução, dados estruturados podem ser decisivos no desempenho destes sistemas. Vimos alguns exemplos neste capítulo. A tabela de desvio é uma estrutura que facilita o algoritmos de busca pela rotina de serviço que atende uma certa exceção. A pilha é uma estrutura que facilita a restauração dos dados no algoritmo de chaveamento de contextos aninhados no algoritmo de gerenciamento das exceções. A fila ou o *buffer* circular é uma estrutura que facilita a orgnaização de dados num algoritmo de transferência de dados entre o processador e os periféricos. A única estrutura questionável seria a lista ligada. Mostramos que para cada lista ligada há um arranjo equivalente. E um arranjo, além de ocupar menos espaço de memória, oferece a vantagem de podermos acessar aleatoriamente os seus elementos diretamente pelos índices. Portanto, em sistemas embarcados o uso de lista ligada é usualmente reservado para ligar blocos de memóra fragmentados (*memory pool*).

Enfim, as estruturas de dados são muito mais relacionadas com os algoritmos que elas suportam do que com o ambiente em que os algoritmos são executados. Se um algoritmo pode se beneficiar com os dados orgnizados numa certa forma, é interessante ter esta organização em qualquer ambiente em que ele é implementado, seja ele um microcontrolador ou um computador pessoal. Portanto, a escolha de um algoritmo e a seleção de uma estrutura de dados como a solução de um problema em sistemas embarcados devem ser conjunta. Devemos sempre ponderar as vantagens de aumentar a eficiência dos algoritmos e as desvantagens de *overhead* de memória e tempo de processamento

Referências

[1] New York University. EL6483. Some very useful data structures commonly used in embedded systems programming (Jump tables, Circular buffers, Linked lists, Stacks and queues, Memory pools) http://crrl.poly.edu/EL6483/embedded_systems_data_structures.pdf

[2] Sippey Fun Labs. Implement Re-entrant interrupt handler for ARM Cortex-M3/4

https://sites.google.com/site/sippeyfunlabs/embedded-system/how-to-run-re-entrant-task-scheduler-on-arm-cortex-m4
[3] RMIT University. Basic Queues
http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/04/queue-basic.html
[4] Ken Wada. Ring Buffer Basics
http://www.embedded.com/electronics-blogs/embedded-round-table/4419407/The-ring-buffer
[5] Codingfreak. Implementation of Singly Linked List
http://codingfreak.blogspot.com/2009/08/implementation-of-singly-linked-list-in.html
[6] How Google works?
http://computer.howstuffworks.com/internet/basics/google1.htm
[7] The Simple, Elegant Algorithm That Makes Google Maps Possible
http://motherboard.vice.com/read/the-simple-elegant-algorithm-that-makes-google-maps-possible

Setembro de 2016