ИНФОРМАТИКА

Вестник Сыктывкарского университета.
Серия 1: Математика. Механика. Информатика.
Выпуск 3 (28). 2018

УДК 519.683.8

ОЧЕРЕДИ НА МИКРОКОНТРОЛЛЕРАХ

В. А. Устюгов

В статье обоснована необходимость изучения алгоритмов и структур данных разработчикам программного обеспечения для встраиваемых систем. Рассмотрены преимущества, получаемые при рациональной организации программного кода. Описан вариант реализации простой структуры данных — очереди.
Ключевые слова: микроконтроллер, встраиваемая система, структуры данных, очередь.

На сегодняшний день изучение программирования встраиваемых систем является важным этапом образовательного трека инженера в области электроники. Это обусловлено широчайшей сферой применения микроконтроллеров и микропроцессоров в современных устройствах промышленной и домашней автоматики. Однако одна из проблем, которая часто сопровождает разработчика-студента или радиолюбителя, — это отсутствие в том же образовательном треке дисциплины, посвященной изучению эффективных алгоритмов программирования и структур данных. Это приводит, к сожалению, к тому, что часто программистские решения не продумываются досконально, и разработчик преследует цель лишь бы устройство работало. Усугубляется вышеобозначенная проблема тем, что приложения, построенные по этому принципу, действительно могут быть вполне работоспособны, и тогда вступает в силу негласный запрет на модификацию рабочих программ (Работает — не трогай!).

Разработчик, придерживающийся вышеописанной методики написания программ, очевидно, упускает из вида возможности масштабирования кода, качественной поддержки, эстетичности. При решении некоторых задач автоматизации применение сложных способов организации кода программ может показаться избыточным, однако их изучение необходимо для построения программ управления сложными си-

стемами, которые требуют, к примеру, эффективной реализации квазиодновременно выполняющихся процессов. Например, концепция применения конечных автоматов, предложенная А. А. Шалыто, позволяет построить легкорасширяемую программу с кооперативной многозадачностью, обеспечивающую адекватную реактивность, высокую структурированность, нетребовательность к ресурсам вычислительной системы для построения многозадачной среды.

В настоящей статье мы рассмотрим вариант реализации простой структуры данных — очереди на языке Си. Приведенный код библиотеки в силу портируемости языка может применяться не только во встраиваемых системах, но и в приложениях для настольных компьютеров.

1. Очереди во встраиваемых приложениях

При разработке сложных многозадачных приложений для микроконтроллеров возникают задачи взаимодействия с внешними источниками данных по различным интерфейсам. Чтобы исключить потерю информации, мы должны максимально быстро реагировать на поступившие извне сигналы, то есть наиболее рациональный способ обращения с такими сигналами — работа по прерыванию. Однако возникает вопрос, в какой момент времени мы должны обрабатывать полученные данные, если согласно общим принципам в обработчике прерывания программа должна находиться минимальное время? Задача усложняется, если источник этих данных активный, и пакеты с данными приходят от него чаще, чем мы можем позволить себе отвлекаться на их обработку.

Поставленную выше задачу можно решить, выделив некоторый буфер, в который будут помещаться пришедшие пакеты данных для дальнейшей обработки в ходе выполнения кода конечных автоматов приложения (или не обязательно их). Для удобства можно организовать такой интерфейс доступа к буферу, что автомату, для которого эти данные предназначены, не надо будет хранить состояния указателей на занятые позиции в этом буфере, тем более что пришедшие данные могут иметь различный размер, и в этом случае это число тоже требуется где-то хранить.

Один из возможных вариантов здесь — применение очереди. Очередь — это вид коллекции, работающий по принципу первым вошел, последним вышел, то есть поддерживаются две команды помещения в очередь и извлечения из очереди. Извлекать элементы в произвольном порядке нельзя, а напротив, извлечение происходит с одного конца, и на выход попадает элемент, наиболее долго находившийся к данному моменту в буфере.

Для того чтобы описать программную сущность очередь, разработаем специальный структурный тип. В нашем примере он будет называться `queueInstance`. Очередь на уровне памяти будет представлять собой буфер, каждый элемент которого хранит указатель на контейнер с данными, а именно другую программную сущность элемент очереди. Для работы с этим буфером заведем указатели на текущие позиции головы (`head`) и хвоста (`tail`) очереди. В данном случае это просто номера ячеек массива, хранящего указатели на элементы очереди. Напомним, что извлечение элементов очереди происходит с головы, а добавляются элементы в хвост.

Наконец, поскольку предполагается, что очередь могут использовать в своих целях различные автоматы, выделять память под контейнеры мы будем динамически. Следовательно, нам потребуется знать размер блока данных в контейнере и максимальную длину очереди в элементах.

Элемент очереди представляет собой указатель на структуру. В состав этой структуры входит числовой идентификатор, указатель на буфер, в котором хранится блок данных, и длина занятой части этого буфера в байтах. Это сделано для того, чтобы в очередь можно было помещать данные, приходящие от различных источников, соответственно, их размер может различаться. Для этих же целей введено поле структуры `id`, предназначенное для хранения идентификатора данных в буфере.

Отметим, что безопасная реализация функций для работы с очередью требует проверки размера блока данных, которые помещаются в очередь. В случае попытки поместить в буфер блок данных, размер которого превышает размер буфера, должен возвращаться некий код ошибки.

Наконец, в конце заголовочного файла приведены прототипы функций для работы с очередью: функция инициализации всех структур данных `queue_init()`, функция помещения элемента в очередь `queue_enqueue()` и извлечения элемента из очереди `queue_dequeue()`.

```
#ifndef QUEUE_H_
#define QUEUE_H_

#include <stdint.h>

typedef struct {
    uint8_t id;
```

```
    uint8_t size;
    uint8_t *buffer;
} queueElement;

typedef struct {
    uint8_t head;
    uint8_t tail;
    uint8_t length;
    uint8_t elementSize;
    queueElement *container;
} queueInstance;


enum {
    QUEUE_OK,
    QUEUE_FULL
};

extern void queue_init(queueInstance *queue,\\
        uint8_t length, uint8_t elementSize);
extern uint8_t queue_enqueue(queueInstance *queue,\\
        uint8_t id, uint8_t size, void *source);
extern queueElement *queue_dequeue(queueInstance *queue);

#endif /* QUEUE_H_ */
```

Рассмотрим теперь реализации вышеназванных функций.

Функция `queue_init()` принимает в качестве аргумента указатель на `экземпляр класса` очереди, то есть на конкретное воплощение абстрактной сущности, описанной с помощью структуры типа `queueInstance` и созданной для работы с каким-то конкретным потоком данных. Также среди аргументов длина очереди и размер блока данных одного элемента.

В теле функции инициализируются поля `tail` и `head`, причем нулями, поскольку в начальный момент времени очередь пуста. Далее в поле `length` копируется переданное значение длины очереди в элементах.

После этого динамически выделяется память для контейнера, содержащего элементы очереди с помощью функции `calloc()`, то есть выделенная память сразу инициализируется нулями. После этого в цикле происходит выделение памяти для всех элементов очереди. В реальных

приложениях всегда требуется проверять, что возвращают аллокаторы памяти, так как память может быть выделена не всегда, даже если имеющийся объем свободной памяти превышает требуемый.

Функция `queue_enqueue()` реализует помещение элемента в очередь. Для этой цели ей передается указатель на саму очередь и параметры для формирования элемента очереди: идентификатор, длина элемента в байтах и указатель на его расположение в памяти.

Перед тем как занять новую позицию в очереди, требуется проверка на наличие свободной позиции. Так, если в очереди `голова` находится сразу после `хвоста` (с учетом того, что буфер мы используем как циклический), это означает, что очередь полна, и запись нового элемента отвергается путем возврата кода `QUEUE_FULL`.

Если свободная позиция есть, производится непосредственная запись полей `id` и `size` структуры, на которую ссылается `хвост`, а также копируется содержимое буфера-источника. Далее производится сдвиг позиции `хвоста` с учетом того, что массив с указателями на элементы очереди мы используем как циклический буфер, то есть, добравшись до последнего элемента, мы переходим обратно к нулевому.

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

void queue_init(queueInstance *queue,\\
    uint8_t length, uint8_t elementSize) {

    uint8_t i = 0;

    queue->head = 0;
    queue->tail = 0;
    queue->length = length;

    queue->container =\\
        (queueElement *) calloc(length, sizeof(queueElement));

    for (i = 0; i<length; i++) {
        queue->container[i].id = 0;
        queue->container[i].size = 0;
        queue->container[i].buffer =\\
            (uint8_t *) calloc(elementSize, sizeof(uint8_t));
```

```
    }
}

uint8_t
queue_enqueue(queueInstance *queue, uint8_t id,\\
    uint8_t size, void *source) {

    if ( (queue->head == (queue->tail + 1)) || \\
            ((queue->head == 0) && \\
            (queue->tail == (queue->length - 1))))
        return QUEUE_FULL;

    (queue->container[queue->tail]).id = id;
    (queue->container[queue->tail]).size = size;
    memmove((queue->container[queue->tail]).buffer, source, size);

    if (queue->tail == (queue->length - 1))
        queue->tail = 0;
    else queue->tail += 1;

    return QUEUE_OK;

}

queueElement*
queue_dequeue(queueInstance *queue) {

    queueElement *res;

    if (queue->head == queue->tail)
        return (queueElement *)0;

    res = &(queue->container[queue->head]);

    if (queue->head == (queue->length - 1))
        queue->head = 0;
    else queue->head += 1;

    return res;
}
```

Наконец, функция `queue_dequeue()` возвращает указатель на головной элемент очереди и сдвигает позицию `головы` с учетом цикличности буфера. Предварительно производится проверка того, не является ли очередь пустой. Если это так, то возвращается нулевой указатель, то есть функция, в которой производится извлечение элемента из очереди, обязана выполнить проверку, не вернула ли очередь нулевой элемент.

Приведем также фрагмент кода, использующего очередь.

```
#include "queue.h"

#define SOME_QUEUE_LENGTH                (16)
#define SOME_QUEUE_ELEMENT_LENGTH        (8)
#define SOME_ID                          (1)
queueInstance someQueue;
...
int main() {

    queueElement *currentElement;

    queue_init(&someQueue,\\
    SOME_QUEUE_LENGTH,\\
    SOME_QUEUE_ELEMENT_LENGTH);
    ...
    // Очередь может пополняться во внешних функциях!
    queue_enqueue(&someQueue, SOME_ID, dataLength, dataBuffer);
    ...
    while (1) {
        ...
        currentElement = queue_dequeue(&someQueue);
        if (!currentElement) continue;
        // Действия с элементом очереди
    }
}
```

Поскольку доступ к очереди производится из различных мест программы, то есть очередь является разделяемым ресурсом, необходимо обеспечить атомарность операций с очередью. Например, извлечение из очереди и помещение элемента в очередь осуществлять, запрещая глобально обработку прерываний.

2. Заключение

Таким образом, применение простой структуры данных позволяет существенно улучшить качество кода, повысить устойчивость работы управляющей программы встраиваемой системы, обеспечить простую масштабируемость. Отметим, что приведенная реализация может быть легко модифицирована под требования конкретного приложения. Например, можно реализовать очередь поверх циклического списка, или изменить поведение очереди при попытке помещения элемента в заполненную структуру. Приведенный код библиотеки в силу портируемости языка Си может применяться не только во встраиваемых системах, но и в приложениях для настольных компьютеров.

Список литературы

1. **Поликарпова Н., Шалыто А.** Автоматное программирование. СПб.: Питер, 2011. 176 с.

2. **Мортон Дж.** Микроконтроллеры AVR. Вводный курс. М.: Додэка, 2010. 271 с.

3. **Шпак Ю.** Программирование на языке C для AVR и PIC микроконтроллеров. СПб.: КОРОНА-ВЕК, 2011. 546 с.

Summary

Ustyugov V. A. The queue on the microcontrollers

The article substantiates the need to study the algorithms and data structures for developers of the software for embedded systems. The advantages obtained by the rational organization of the program code are considered. An embodiment of a simple data structure — a queue is described.

Keywords: microcontroller, embedded system, data structure, queue.

References

1. **Polikarpova N.** *Avtomatnoye programmirovaniye* (Automata programming), SPb: Piter, 2011, 176 p.

2. **Morton J.** *Mikrokontrollery AVR. Vvodny kurs* (AVR microcontrollers. Introductory course), M.: Dodeka, 2010, 271 p.

3. **Shpak Yu.** *Programmirovaniye na yazyke C dlya AVR i PIC mikrokontrollerov* (C programming for AVR and PIC microcontrollers), SPb:Korona-Vek, 2011, 546 p.

Для цитирования: Устюгов В. А. Очереди на микроконтроллерах // *Вестник Сыктывкарского университета. Сер. 1: Математика. Механика. Информатика. 2018. Вып. 3 (28). С. 38–46.*

For citation: Ustyugov V. A. The queue on the microcontrollers, *Bulletin of Syktyvkar University. Series 1: Mathematics. Mechanics. Informatics*, 2018, 3 (28), pp. 38–46.

СГУ им. Питирима Сорокина *Поступила 24.11.2018*