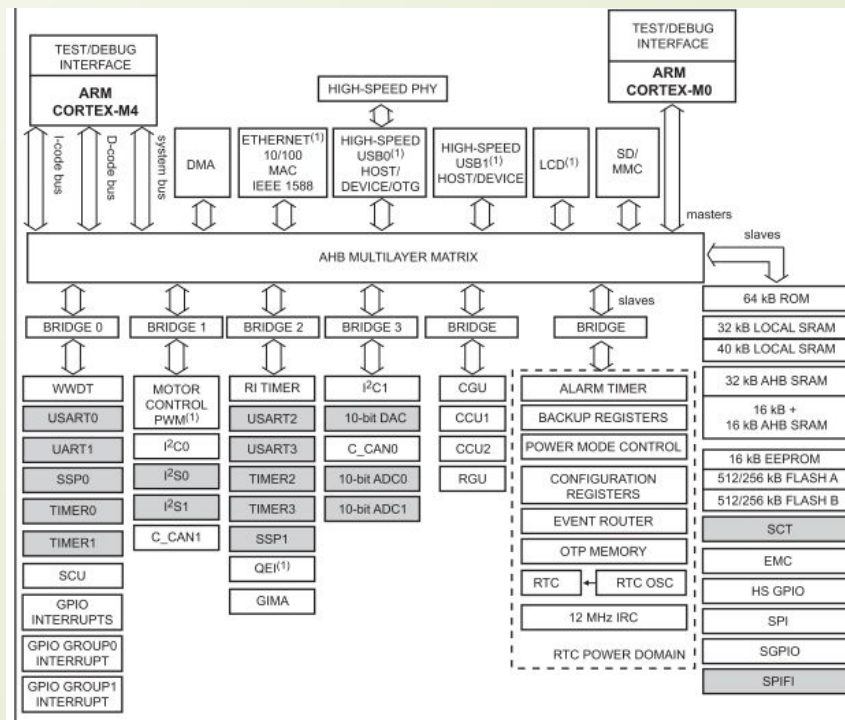



Встроенные Системы Управления



Микроконтроллер

- ❑ Микросхема, содержащая в себе функционально законченный компьютер – с ядром (процессором), памятью и периферийными устройствами.
- ❑ Программирование микроконтроллера – не только описание алгоритма, но и решение задачи управления входящими в состав микроконтроллера периферийными устройствами.





Для обработки информации и принятия решений, необходимо:

- Превратить входную информацию в данные, пригодные для хранения в памяти (то есть, превратить в **числа**);
- **Расположить** данные в памяти определенным образом;
- Выполнить определенные **действия над числами**, хранящимися в памяти;
- Куда-то выдать **результат**.



Программа

Алгоритм



Типы и структуры
данных



Интерфейс с
внешним миром



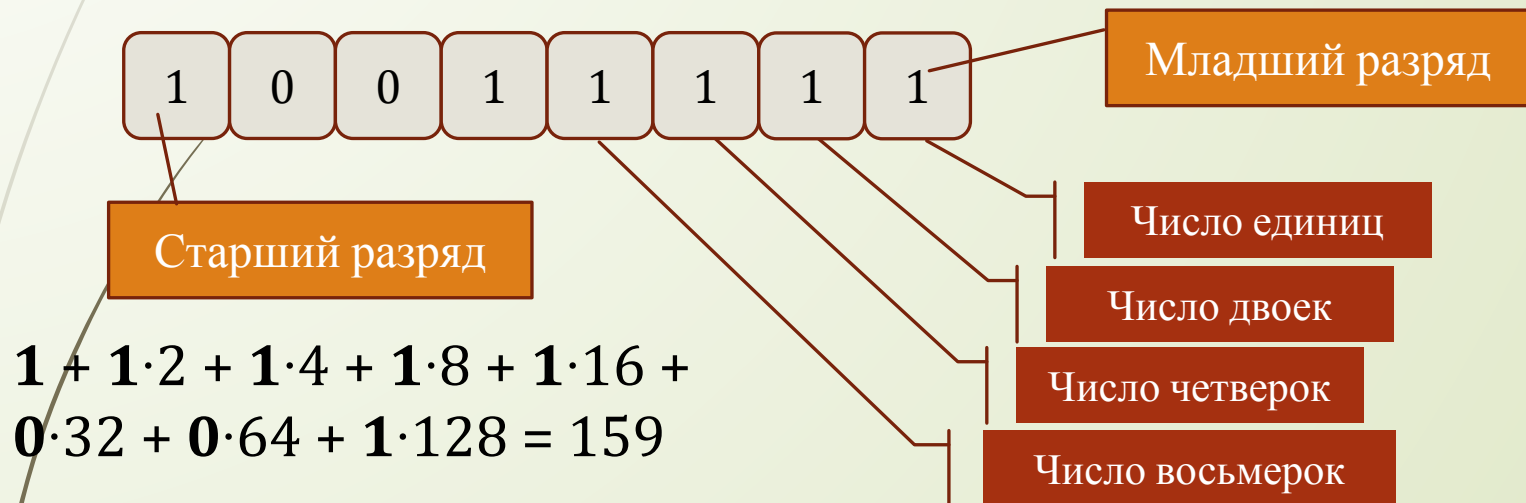
Элементы алгоритмов

- **Следование.** Последовательность действий, выполняющихся друг за другом.
- **Условие.** Точка принятия решения, после которой выполнение алгоритма пойдет по одному из нескольких возможных путей.
- **Цикл.** Ряд действий, выполняющихся неоднократно над разными или одними и теми же исходными данными.

Числа

$9 + 5 \cdot 10 + 1 \cdot 100 = 159$ – это десятичная запись числа

- Двоичная система использует всего два значка для записи чисел.
- Соответственно, младший разряд хранит число единиц, следующий за ним – число двоек, следующий – число четверок и т.д.
- Что хранит старший разряд – зависит от размера ячейки памяти



□ Двоичная система используется для хранения чисел в памяти компьютера, для их обработки в процессоре и, как правило, для передачи по линиям связи.

Числа

Размер ячеек памяти как правило кратен **восьми** двоичным разрядам (восьми битам):

Размер ячейки	Объем	Диапазон значений	
		В двоичном виде	В десятичном виде*
8 бит	Байт	0000 0000 – 1111 1111	0 – 255
16 бит	Слово (Короткое слово)	0000 0000 0000 0000 – 1111 1111 1111 1111	0 – 65 535
32 бита	Длинное слово	0000 0000 0000 0000 0000 0000 0000 0000 – 1111 1111 1111 1111 1111 1111 1111 1111	0 – 4 294 967 295
64 бита	64-битное слово	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 – 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111	0 – 18 446 744 073 709 551 615

Зачастую удобнее использовать
шестнадцатеричную запись чисел

Взаимодействие вычислительной системы с внешним миром

□ Устройства ввода:


- Кнопки, клавиши, переключатели, энкодеры;
- Датчики;
- Источники потоковых данных (аудио, видео).

□ Устройства вывода:

- Индикаторы (дискретные, знакосинтезирующие);
- Графические дисплеи;
- Цифро-аналоговые преобразователи;
- Исполнительные устройства (реле, электродвигатели и т.п.);
- Устройства вывода потоковых данных (аудио, видео).

□ Устройства передачи данных:

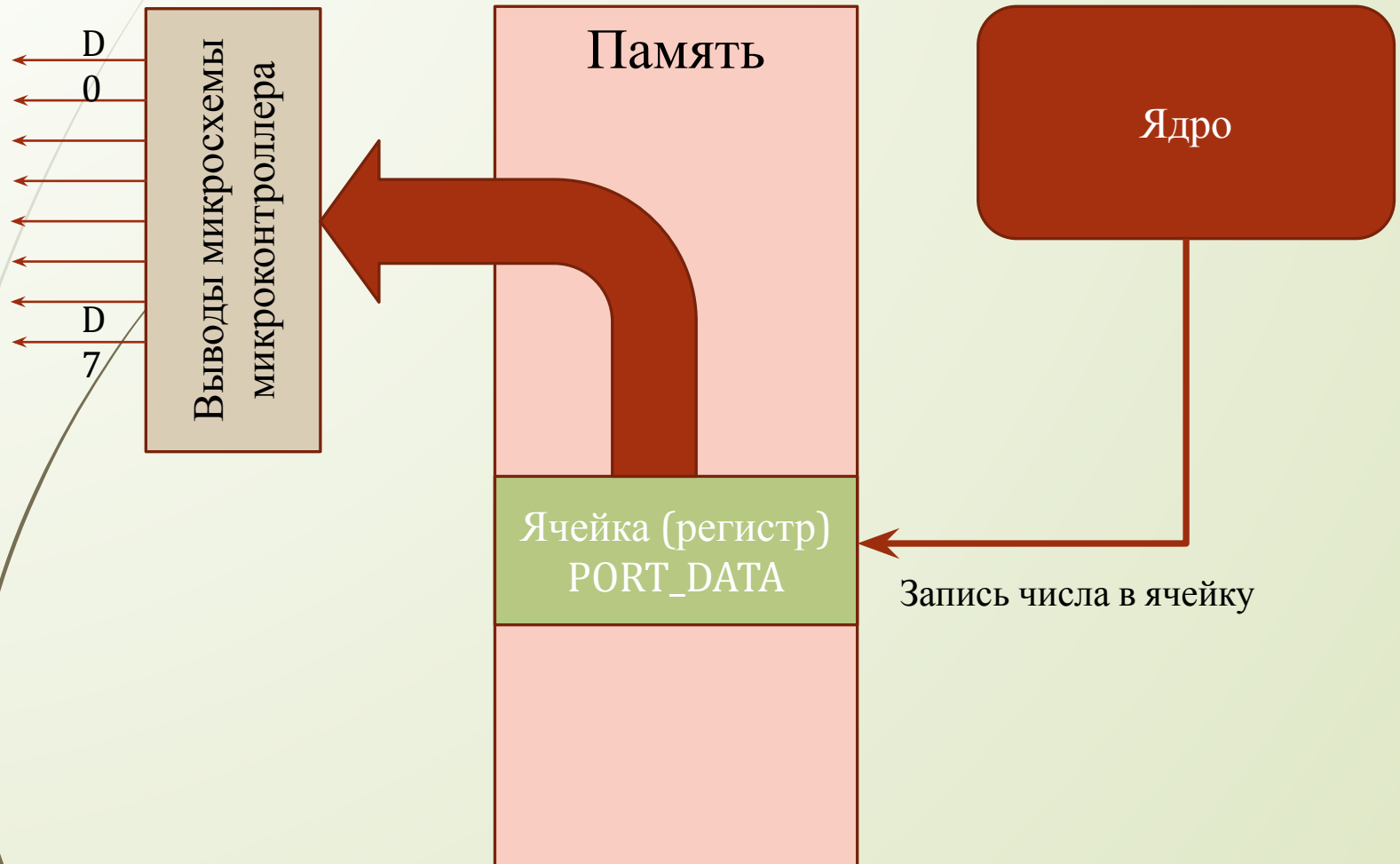
- Внутренние интерфейсы (шины I2C, SPI, UART и т.п.);
- Межблочные интерфейсы (шины RS232, RS422/485, CAN, ...);
- Внешние интерфейсы (Ethernet, USB, Bluetooth, WiFi, ...).



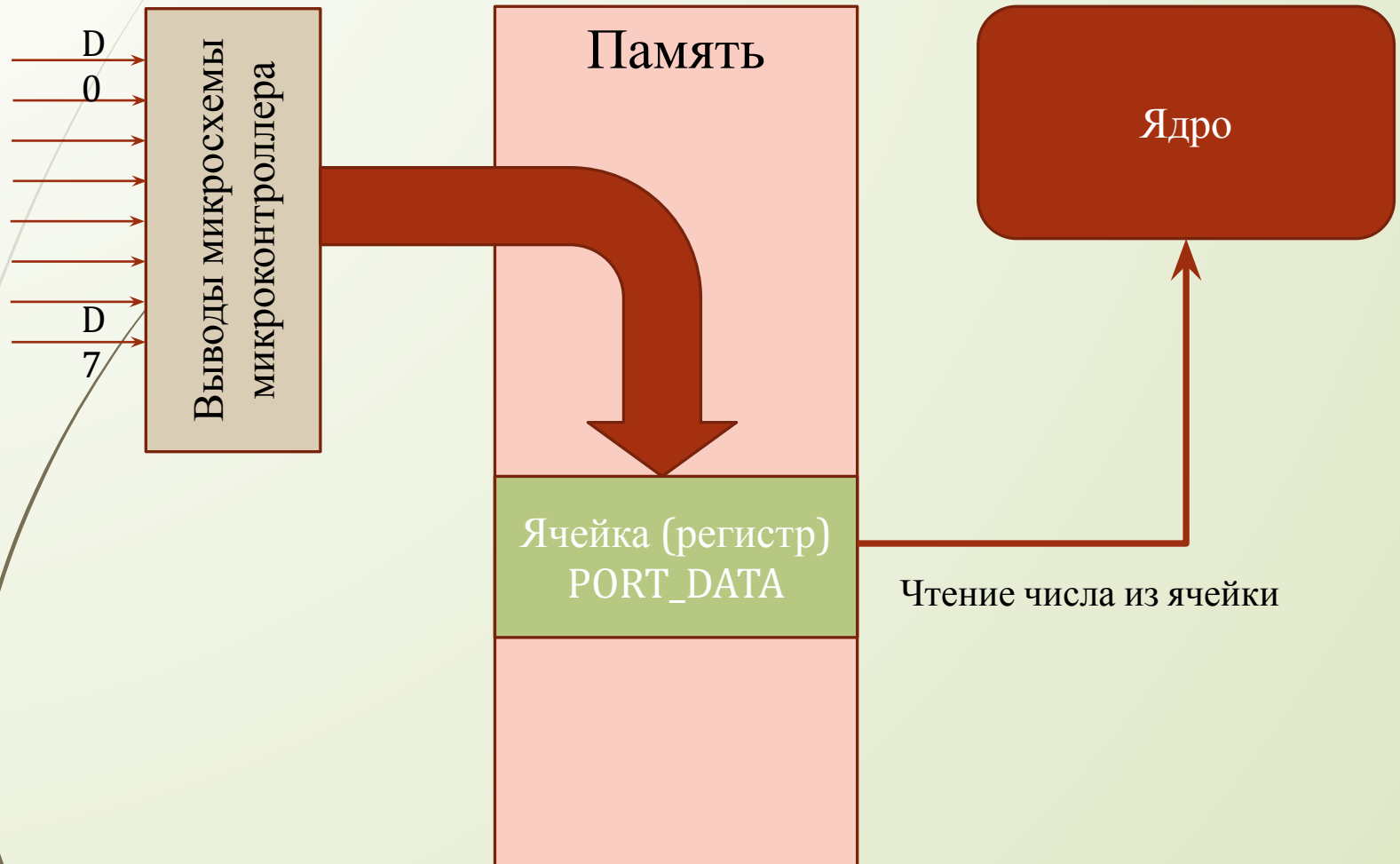
Взаимодействие вычислительной системы с внешним миром

- Все внешние устройства, как правило, представлены для ядра в виде выделенных областей памяти.
- Программа, **читая** содержимое выделенных областей, определяет состояние устройства и получает данные из внешнего мира.
- Программа, **изменяя** содержимое областей памяти, меняет поведение устройства и отправляет данные во внешний мир.

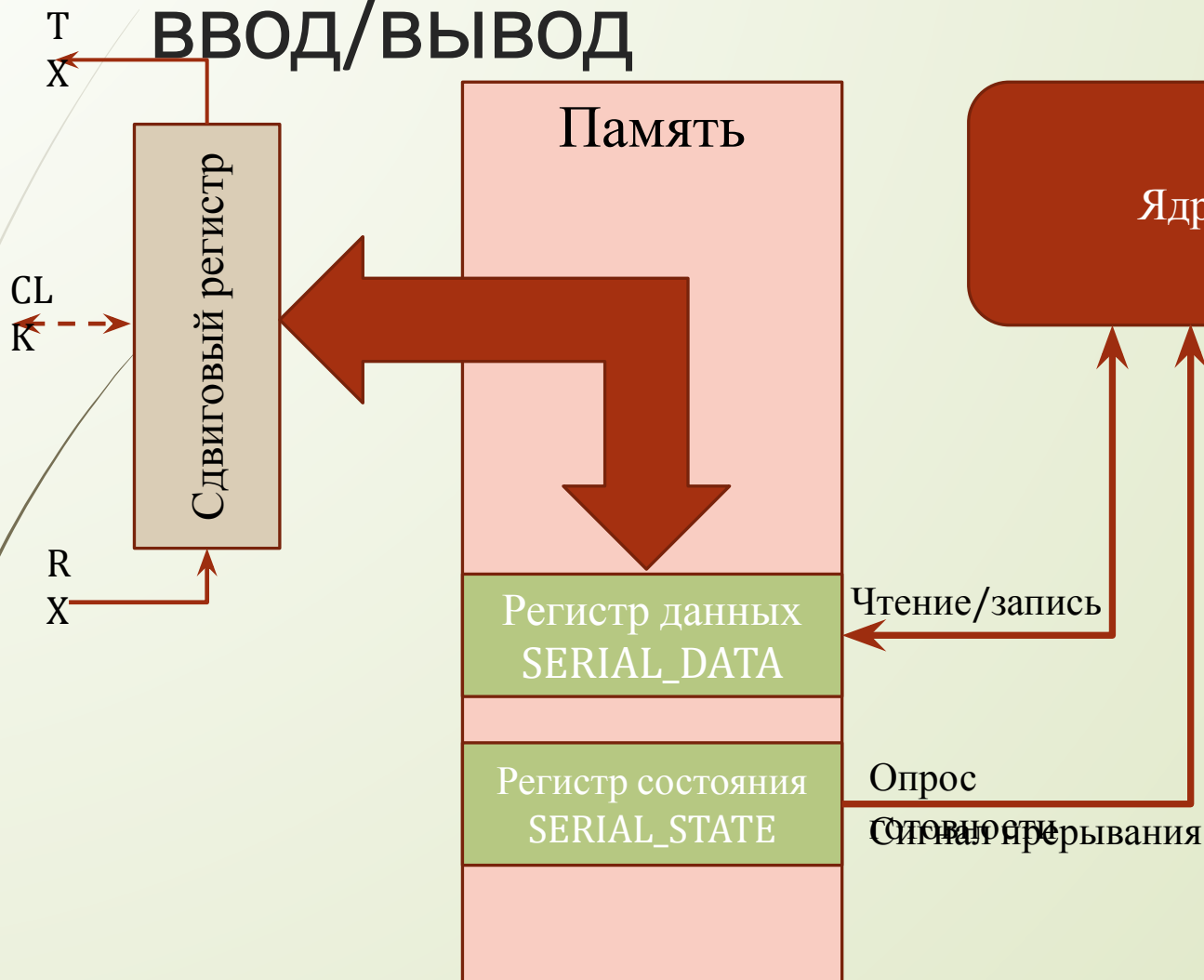
Параллельный вывод



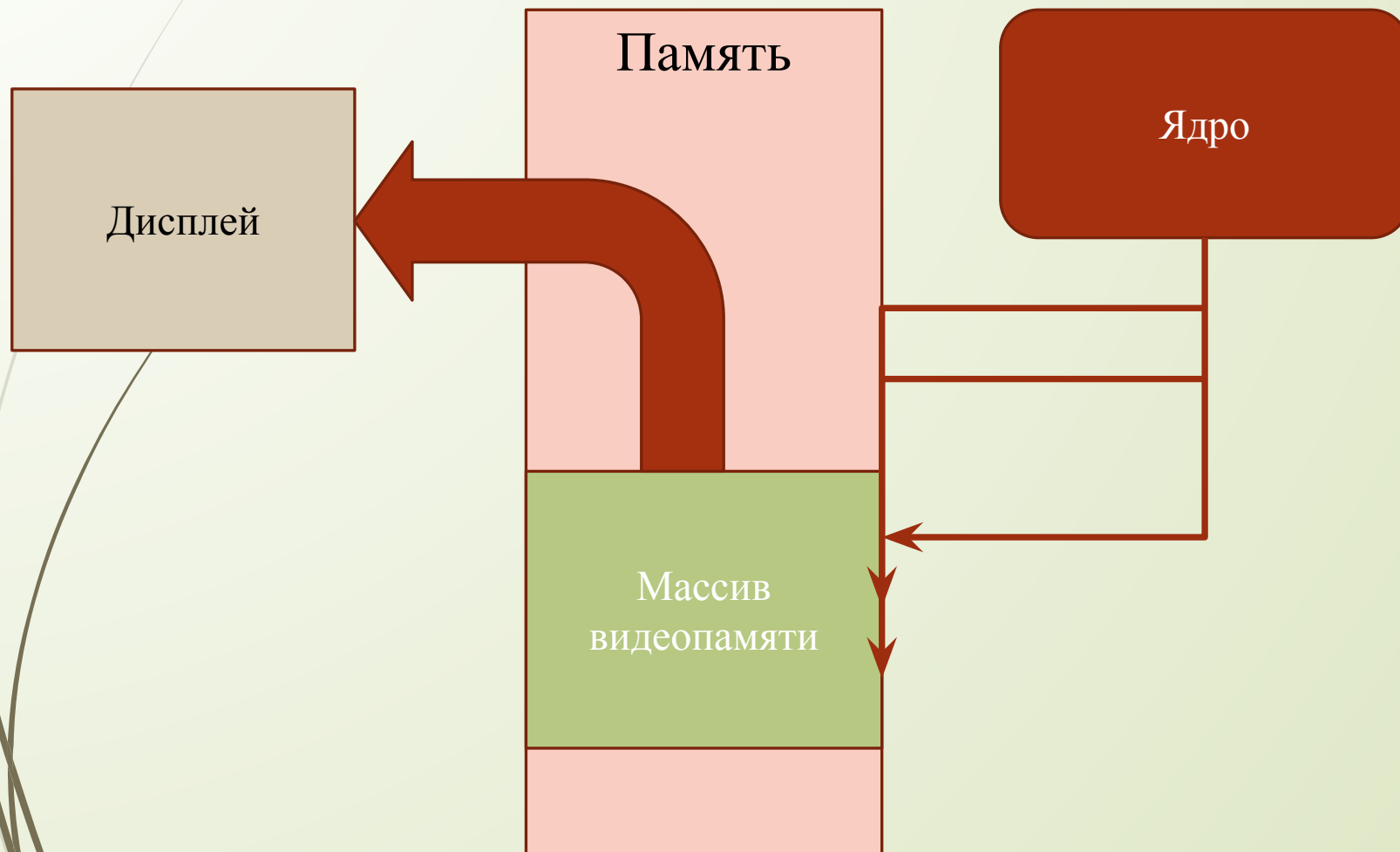
Параллельный ввод



Последовательный ВВОД/ВЫВОД



ПОТОКОВЫЙ ВЫВОД



Язык Си

- Программа на языке Си представляет собой набор текстовых файлов (в простейшем случае – один текстовый файл).
- Текст программы состоит из инструкций (*операторов*), разделенных знаком «;».
- Операторы можно условно поделить на три категории:
 - описание данных;
 - вычисление;
 - управление;

Переменные

Переменная – это область в памяти компьютера, которая имеет имя и хранит некоторое значение.

- Значение переменной может меняться во время выполнения программы.
- При записи в ячейку нового значения старое стирается.

Тип переменной – от него зависит размер области памяти и то, как именно в ней хранятся данные:

- **int** – целое число (В ОС Windows 32 - 4 байта)
- **float** – вещественное число, *floating point* (4 байта)
- **char** – символ, *character* (1 байт)
- ... и много других разных типов...

Объявление переменных

- Объявление переменной – это описание переменной, которая в дальнейшем будет использоваться в программе. Переменная объявляется для того, чтобы компилятор знал, где и как хранить данные в памяти.
- Объявление переменной представляет из себя запись, состоящую из названия типа переменной и имени переменной.
- В языке Си переменную можно объявить в любом месте файла с программой (но до её первого использования).

Объявление переменной

- Синтаксис оператора объявления:
имя_типа имя_переменной;
- Тип переменной определяет, какие данные в ней хранятся
- В конце оператора объявления переменной ставится символ «;»
- Имена переменных должны быть информативными, но не слишком длинными.
- Место в памяти для размещения переменной выделяется автоматически, в процессе компиляции программы, на этапе так называемой компоновки.

Типы переменных

Тип переменной (и вообще тип любых данных) – указание компилятору на то, сколько памяти выделить для хранения переменной и как с ней обращаться (как рассматривать ее содержимое).



Типы скалярных переменных

Название в языке C	Описание	Диапазон значений
<code>unsigned char</code>	Неотрицательное (беззнаковое) целое число, занимает одну ячейку памяти (8 бит)	0...255
<code>signed char</code>	Целое число со знаком, занимает 8 бит	-128...+127
<code>unsigned short</code>	Беззнаковое целое, 16 бит	0...65535
<code>signed short</code>	Целое со знаком, 16 бит	-32768...+32767
<code>unsigned long</code>	Беззнаковое целое, 32 бита	0...4 294 967 295
<code>signed long</code>	Целое со знаком, 32 бита	-2147483648...2147483647
<code>unsigned long long</code>	Беззнаковое целое, 64 бита	0...18446744073709551614
<code>signed long long</code>	Целое со знаком, 64 бита	-9223372036854775808... 9223372036854775807
<code>float</code>	Вещественное число одинарной точности, 32 бита	-3,4E+38 ... +3,4E+38
<code>double</code>	Вещественное число двойной точности, 64 бита (или больше)	-1,7E+308 ... +1,7E+308
<code>int</code>	Целое число, размер которого определяется платформой	???

- Слова «**unsigned**» / «**signed**» можно не указывать, однако в этом случае результирующий тип определяется компилятором

Функция получения размера переменной

- В языке Си есть специальная встроенная функция **sizeof()**, которая может посчитать размер переменной *в байтах*:

```
char x;  
int x_size = sizeof(x);    // x_size = 1  
long count;  
int cs = sizeof(count);    // cs = 4
```

- В качестве аргумента функции **sizeof** можно использовать просто имя типа:

```
int s = sizeof(short);     // s = 2
```

Константы

Константами называют числа, символы и строки, встречающиеся в тексте программы в явном виде. Например, в выражении $a = b * 3 + 5$ числа 3 и 5 – константы (тогда как a и b – это имена переменных).

Запись чисел (числовых констант) в Си:

255 // десятичное число

0xFF // шестнадцатеричное число

'x' // значение, соответствующее ASCII коду символа

3.14 // натуральное число с фиксированной точкой

1.1E+3 // натуральное число с плавающей точкой


Оператор присваивания

- Присваивание – это запись значения в переменную
имя_переменной = выражение;
- Оператор присваивания состоит из имени переменной, знака «=» и *выражения*, значение которого вычисляется в процессе выполнения оператора:

Result = 1;

Average = (a + b) / 2;

count = count + 1;



Переменные – регистры управления

- ❑ Все периферийные устройства микроконтроллера управляются при помощи набора регистров, находящихся в общем адресном пространстве.
- ❑ Для управления устройствами объявляют переменные, адреса размещения которых соответствует адресам регистров устройств.
- ❑ Имена таких переменных как правило, объявлены в заголовочном файле, поставляемом производителем микроконтроллера.

Переменные – регистры управления

- Например, в микроконтроллере LPC2368 состояние порта ввода-вывода управляется регистрами FIO_nPIN, где n – номер порта.
- В программе доступны переменные FIO0PIN, FIO1PIN и т.д.
- Например, установка лог. «1» на третьем выводе порта 1 соответствует установке третьего бита в переменной FIO1PIN:

`FIO1PIN |= (1 << 3);`

Table 245. Register overview: GPIO port (base address 0x400F 4000)
The highest pin number on each port depends on package size (see [Table 240](#)).

Name	Access	Address offset	Description	Reset value	Width	Reference
PIN4	R/W	0x2110	Port pin register port 4	ext[1]	word (32 bit)	Table 263
PIN5	R/W	0x2114	Port pin register port 5	ext[1]	word (32 bit)	Table 263
PIN6	R/W	0x2118	Port pin register port 6	ext[1]	word (32 bit)	Table 263
PIN7	R/W	0x211C	Port pin register port 7	ext[1]	word (32 bit)	Table 263
MPIN0	R/W	0x2180	Masked port register port 0	ext[1]	word (32 bit)	Table 264
MPIN1	R/W	0x2184	Masked port register port 1	ext[1]	word (32 bit)	Table 264
MPIN2	R/W	0x2188	Masked port register port 2	ext[1]	word (32 bit)	Table 264
MPIN3	R/W	0x218C	Masked port register port 3	ext[1]	word (32 bit)	Table 264
MPIN4	R/W	0x2190	Masked port register port 4	ext[1]	word (32 bit)	Table 264
MPIN5	R/W	0x2194	Masked port register port 5	ext[1]	word (32 bit)	Table 264
MPIN6	R/W	0x2198	Masked port register port 6	ext[1]	word (32 bit)	Table 264
MPIN7	R/W	0x219C	Masked port register port 7	ext[1]	word (32 bit)	Table 264
SET0	R/W	0x2200	Write: Set register for port 0 Read: output bits for port 0	0	word (32 bit)	Table 265

Выражения в языке C

- Везде, где согласно правилу написания того или иного оператора требуется присутствие того или иного числового значения, записывается *выражение* языка C.
- Любое выражение состоит из *операндов* (переменных или констант), соединенных *знаками операций*. Знак операции – это символ или группа символов, которые сообщают компилятору о необходимости выполнения определенных арифметических или логических действий над операндами.
- Операции в выражениях выполняются в строгой последовательности согласно их приоритету.
- Порядок выполнения операций может регулироваться с помощью круглых скобок.
- По количеству операндов различают *унарные* и *бинарные* операции. У унарных операций один операнд, а у бинарных их два.

Арифметические

Знак операции	Описание операции	Примеры использования
()	Вызов функции	a = get_data();
[]	Обращение к элементу массива	b = table[i];
.	Обращение к элементу структуры	c = settings.volume;
-	Изменение знака числа	x = -y;
++	Увеличение числа на единицу	i++;
--	Уменьшение числа на единицу	j--;
&	Взятие адреса переменной	address = &x;
(тип)	Преобразование типа	n = (short)b;
*	Умножение	a = b * c;
/	Деление	x = y / 25;
%	Остаток от деления	k = x % 10;
+	Сложение	n = m + n;
-	Вычитание	a = b - 10;
+=, -=, *=, /=, ...	Составное присваивание (изменение значения переменной)	n += 2; a /= 8;



Сложные типы данных

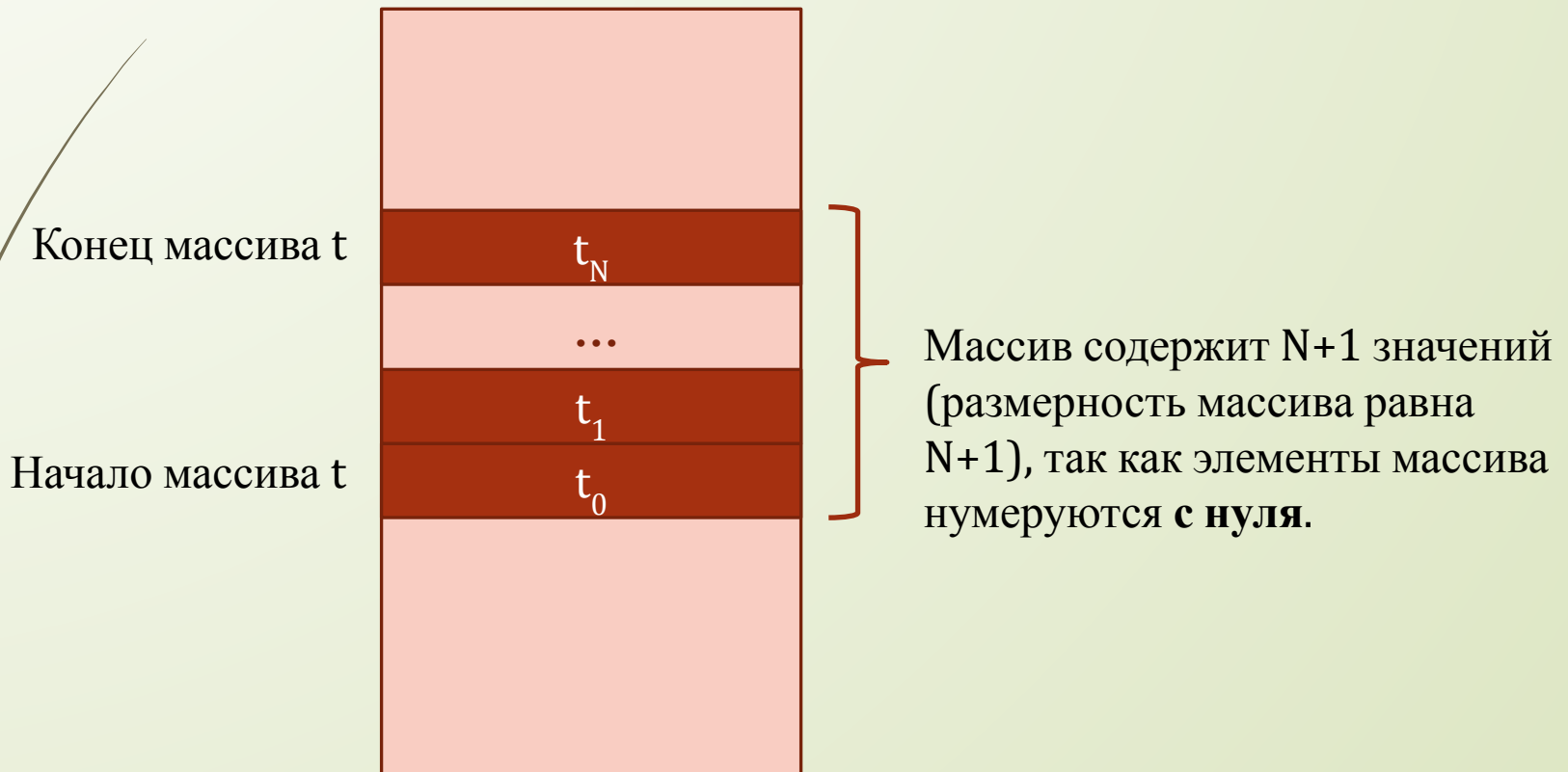
Сложные типы состоят из нескольких простых (скалярных) типов. Сложные типы в Си – это массивы, структуры и объединения.

Массивы

- Массив или вектор – переменная, хранящая ряд значений одного типа

$$t_0, t_1, t_2, \dots, t_N$$

- Значения хранятся в памяти подряд, друг за другом
- Объем занимаемой памяти определяется **типом** значений и их общим количеством:



Массивы

Массив объявляется так же, как и простая (скалярная) переменная, однако после имени в квадратных скобках указывается количество элементов:

```
unsigned char data[10]; // объявление массива из 10 байтов
```

```
signed short t[120]; // объявление массива из 120 чисел типа короткое  
// целое со знаком
```

- Можно сразу же при объявлении заполнить массив значениями. В этом случае после знака «=» ставятся фигурные скобки и перечисляются значения элементов массива друг за другом через запятую:

```
// массив, хранящий число дней в месяцах:
```

```
char DayInMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,  
31};
```

- В этом случае размерность можно вообще не указывать, так как она автоматически определяется по количеству перечисленных значений:

```
char DayInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Работа с элементами

массивов

Для доступа к элементу массива в квадратных скобках после имени указывают номер или *индекс* элемента:

```
// в переменную innerT записываем значение t5
```

```
signed short innerT = t[5];
```

- Элементы массивов нумеруются с **нуля**
- В качестве индекса элемента можно указывать не только числовые константы, но и вообще любые выражения:

```
// подсчет суммы дней в году
```

```
short sum = 0;
```

```
char i;
```

```
for (i = 0; i < 12; i++)
```

```
    sum += DayInMonth[i];
```

Работа с элементами

массивов

Для того, чтобы подсчитать количество элементов массива, можно воспользоваться функцией **sizeof()**, разделив размер массива в байтах на размер одного элемента:

```
signed short t[] = {10, 150, 2500, 64, -180};  
int tsz = sizeof(t) / sizeof(short); // tsz = 5
```

Если массив состоит из байтов (тип char), то sizeof() просто возвращает размер такого массива, так как размер одного элемента в нем – 1 байт:

```
// подсчет суммы дней в году  
short sum = 0;  
char i;  
for (i = 0; i < sizeof(DayInMonth); i++)  
    sum += DayInMonth[i];
```

Строки символов

Строка символов в языке Си считается массивом из 8-разрядных чисел (значений типа char). [На самом деле это не всегда так!]

- В конце любой строки компилятор всегда добавляет невидимый символ с кодом 0 ('\0') – признак конца строки

```
char string[25]; // строка длиной до 24 символов
```

- Для заполнения строки конкретным значением после знака «=» просто пишется строковая константа – символы в двойных кавычках. Признак конца строки (символ с нулевым кодом) добавится автоматически:

```
char prompt[] = "Hello"; // массив займет 6 байтов в памяти
```

- В языке C есть набор системных функций для работы со строками (вычисление длины, поиск фрагментов, копирование и т.п.). Заголовки этих функций находятся в подключаемом файле **string.h**.

Строковые функции

Функция	Описание	Примеры использования
<code>strlen(s)</code>	Вычисление длины строки	<code>int len = strlen(mes);</code>
<code>strcpy(s,d)</code>	Копирование строки d в строку s	<code>strcpy(buf, mes);</code>
<code>strncpy(s,d,n)</code>	Копирование n символов строки d в строку s	<code>strcpy(buf, mes, 5);</code>
<code>strcat(s,d)</code>	Добавление строки d в конец строки s	<code>strcat(mes, "вкл");</code>
<code>strncat(s,d,n)</code>	Добавление n символов строки d в конец строки s	<code>strcat(mes, buf, 8);</code>
<code>strchr(s,c)</code>	Поиск первого вхождения символа c в строку s	<code>char* pos = strchr(buf, ':');</code>
<code>strstr(s1,s2)</code>	Поиск первого вхождения строки s2 в строку s1	<code>char* pos = strstr(buf, "HTTP/");</code>
<code>strcmp(s1,s2)</code>	Сравнение строк	<code>if (!strcmp(mes,"OK")) {...}</code>
<code>strncmp(s1,s2,n)</code>	Сравнение первых n символов в строках	<code>if (!strncmp(ans,"Content-Length:",15)) {...}</code>

Строки символов

Так как в конце строки стоит признак конца строки (символ с кодом 0), то фактическая длина строки может быть *меньше* чем размер массива, предназначенного для её хранения.

```
char string[25]; // строка длиной до 24 символов
```

```
int res = 1000;
```

```
sprintf(string, "Result = %d", res);
```

R	e	s	u	l	t		=		1	0	0	0	\0	?	?	?	?	?	?	?	?	?	?	?
---	---	---	---	---	---	--	---	--	---	---	---	---	----	---	---	---	---	---	---	---	---	---	---	---

- То есть, можно выделить сразу много памяти для хранения разных строк! Однако, с другой стороны, всегда остается опасность переполнения массива, если действовать не аккуратно:

```
char message[] = "Hi! Hello from my SuPeR PuPeR Software!";
```

```
char buffer[16];
```

```
strcpy(buffer, message);
```

- А надо так:

```
strncpy(buffer, message, sizeof(buffer)-1);
```

Структуры

- Язык Си позволяет объединять переменные разных типов в структуры:

```
struct Point
{
    signed short x;
    signed short y;
    unsigned long color;
};
```

- Ключевое слово **struct** говорит, что это составной тип (структура). Далее в фигурных скобках перечисляются элементы, входящие в новый составной тип.
- В данном случае описана структура с именем Point, хранящая информацию о точке на экране – ее координаты (x и y) и цвет (color).

Структуры

- Для объявления переменной типа структуры пишут ключевое слово `struct`, затем имя структуры, а затем имя переменной:

```
struct Point pixel;
```

- Для доступа к элементу структуры используется символ точки, который разделяет имя переменной и имя элемента структуры:

```
pixel.x = 10;
```

```
pixel.y = 25;
```

Расположение структур в

памяти. Элементы структуры хранятся последовательно, друг за другом, но с учетом выравнивания по границам битов/байта/слова/длинного слова.

```
struct Point
{
    signed short x;
    signed short y;
    unsigned long color;
}__attribute__((__packed__));
```

Байт 0	x
Байт 1	
Байт 2	y
Байт 3	
Байт 4	color
Байт 5	
Байт 6	
Байт 7	

По границам байтов или слов
(__attribute__((__packed__)))

Байт 0	x
Байт 1	
Байт 2	-
Байт 3	-
Байт 4	y
Байт 5	
Байт 6	-
Байт 7	-
Байт 8	color
Байт 9	
Байт 10	
Байт 11	

По границам двойных слов

Структуры с битовыми

полями

- Для доступа к отдельным битам и просто для экономии памяти можно указать размеры элементов структуры с точностью до одного *бита*!
- Порядок следования битов – от младших к старшим.

```
struct /* Watchdog Timer Control Register */
{
    unsigned char WDTIN : 1;
    unsigned char WDTRS : 1;
    unsigned char WDTEN : 1;
    unsigned char : 1;
    unsigned char WDTPR : 1;
    unsigned char WINBEN : 1;
    unsigned char : 2;
} WDTCON_bit;
```

B7	B6	B5	B4	B3	B2	B1	B0
-	-	WINBEN	WDTPR	-	WDTEN	WDTRS	WDTIN

Объединения

- ❑ Объединения – это способ описать одну и ту же область памяти по разному.
- ❑ Объединение занимает в памяти столько байт, сколько занимает самой большой его элемент.

```
union LongNumber
{
    unsigned long Long32;
    unsigned char Byte[4];
};
```

```
union LongNumber num;
num.Long32 = 0x12345678;
printf("Bytes are: %x:%x:%x:%x", num.Byte[0],
        num.Byte[1], num.Byte[2], num.Byte[3]);
```

Bytes are: 78:56:34:12

Объединения

- Объединения удобно использовать для работы с пакетами данных.

```
struct Message_str
{
    unsigned char Header;
    unsigned char Command;
    unsigned char Parameters[4];
    unsigned char Csum;
}__attribute__((__packed__));

union Packet
{
    unsigned char Buffer[sizeof(struct Message_str)];
    struct Message_str Message;
};

union Packet incoming;

ReceivePacket(&incoming.Buffer, sizeof(union Packet));
if (incoming.Message.Command == START)
{
    doStart();
}
...
```

Header	1 байт
Command	1 байт
Params	4 байта
Csum	1 байт

Операции доступа к данным

Знак операции	Описание операции	Примеры использования
()	Вызов функции	<code>a = get_data();</code>
[]	Обращение к элементу массива	<code>b = table[i];</code>
.	Обращение к элементу структуры	<code>c = settings.volume;</code>

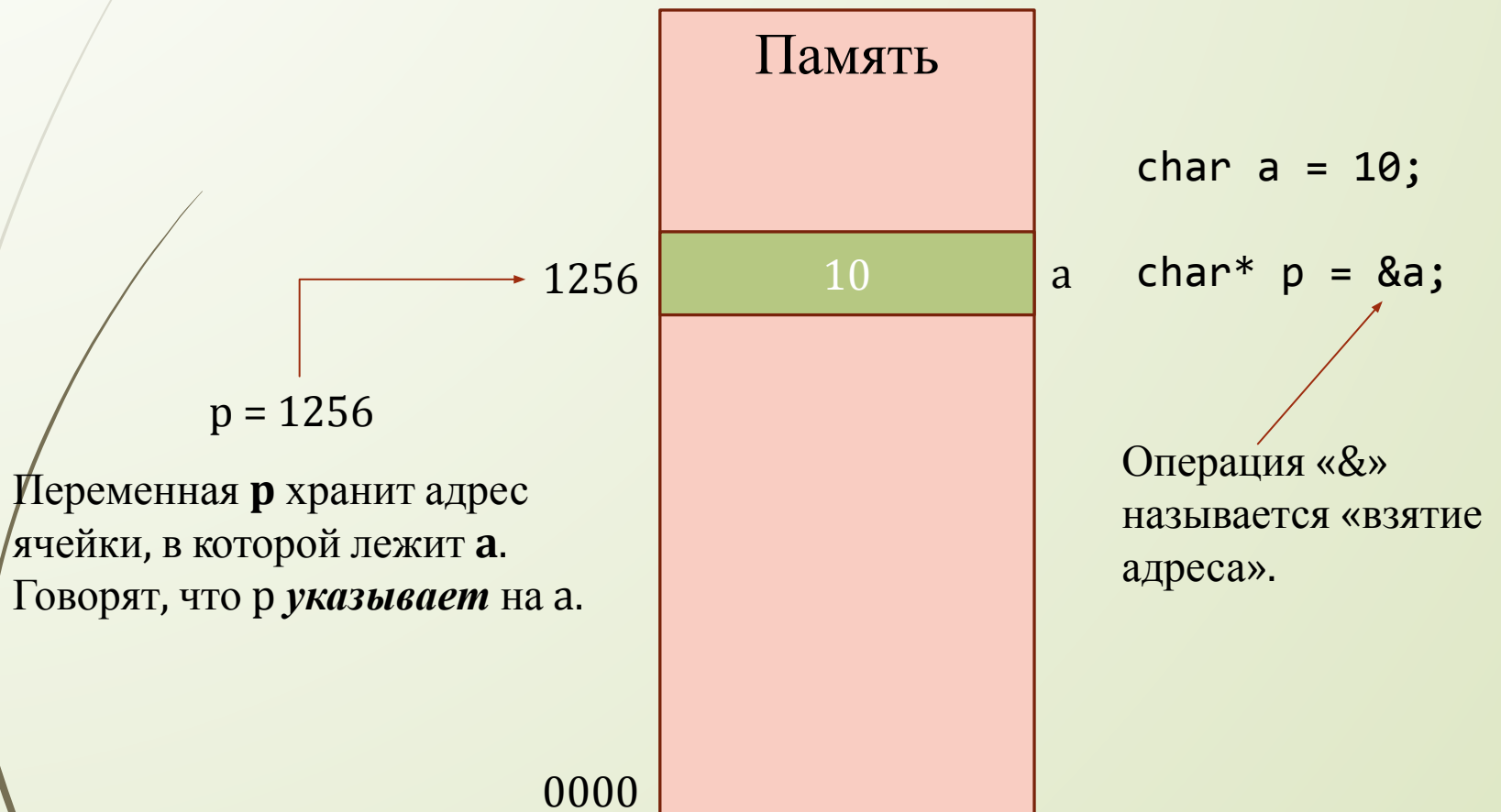


Указатели

Указатели – это переменные, хранящие адреса других переменных (то есть адреса ячеек в памяти).

Указатели

- Указатель – это переменная, которая хранит *адрес* ячейки памяти.



Объявление переменных-указателей

- При объявлении обычно задают тип данных, на которые будет указывать переменная:

`имя_типа* имя_переменной;`

Например:

`char* text_ptr;`

`unsigned short* current_temp;`

- Тип нужен для контроля за правильностью вызова функций с аргументами-указателями и для выполнения арифметических операций над указателями.
- На самом деле переменные-указатели – суть обычные числа с разрядностью, равной разрядности *шины адреса* системы. Они хранят числа – адреса ячеек.

Операции с указателями

- Для доступа к содержимому ячейки, на которую указывает указатель, перед именем переменной-указателя пишут значок звездочки «*»:

```
signed short t[25];
```

```
...
```

```
signed short* t_data = &t[0]; //t_data указывает на t0
```

```
signed short t0 = *t_data;
```

- К переменным-указателям можно прибавлять и вычитать константы. При этом указатель «сдвигается» на величину, кратную размеру типа:

```
// переходим к 5-му элементу массива t
```

```
t_data = t_data + 5;
```

```
signed short t5 = *t_data;
```

Операции с указателями

- Для доступа к полю структуры, на которую указывает указатель, между именем переменной-указателя именем поля структуры пишут стрелочку «->»:

```
struct Point
{
    signed short x;
    signed short y;
    unsigned long color;
};

double GetDistance(struct Point* p1, struct Point* p2)
{
    return sqrt((p1->x - p2->x) * (p1->x - p2->x) -
                (p1->y - p2->y) * (p1->y - p2->y));
}
```

Операции с указателями

- Указатели используют для работы с массивами и буферами.

// поиск максимального числа в массиве

```
long GetMax(long* data, int len)
{
    int i;
    // сначала максимальным считаем первое число
    long max = *data;
    for (i = 1; i < len; i++)
    {
        data++; // переходим к следующему числу
        if (*data > max) // если нашли новый максимум, ...
            max = *data; // ... то запоминаем новое
значение
    }
    return max;
}
```

Операции с указателями

- После имени переменной-указателя можно поставить квадратные скобки с индексом и работать с указателем как с обычным массивом.

// вычисление контрольной суммы массива

```
long CSum(long* data, int len)
{
    int i;
    long csum = 0;
    for (i = 0; i < len; i++)
        csum += data[i];
    return csum;
}
```


Указатель на физическую ячейку

- Можно объявить указатель, указывающий на ячейку с заданным адресом. Делается это при помощи операции приведения типов (ведь указатель – это просто число!)

```
#define GPIO_PORTB_DATA_R      (*((volatile unsigned long *)0x400053FC))
#define GPIO_PORTB_DIR_R      (*((volatile unsigned long *)0x40005400))
#define GPIO_PORTB_IS_R      (*((volatile unsigned long *)0x40005404))
```

указатель на ячейку

адрес ячейки

содержимое ячейки

- Вообще-то способов доступа к конкретному адресу – множество! Например, некоторые компиляторы позволяют объявлять переменные с явным указанием адреса:

```
__no_init volatile unsigned char TL0 @ 0x8A; /* Timer 0, Low Byte */
__no_init volatile unsigned char TL1 @ 0x8B; /* Timer 1, Low Byte */
```

Указатель на неопределённый тип

- В ряде случаев бывает нужна переменная, которая просто хранит адрес ячейки, без конкретного указания на то, что же в ней лежит. Такая переменная-указатель объявляется так:

```
void* mem_ptr;
```

- Арифметические операции с такой переменной сдвигают ее с шагом в один байт.
- Для доступа к данным, на которую указывает такая переменная, нужно применить преобразование типов:

```
short t = *((short*)mem_ptr);
```

Функции работы с памятью

- Библиотека `string.h`, помимо функций работы со строками, содержит средства для работы с кусками памяти. В такие функции передаются указатели типа `void*`.

Функция	Описание	Примеры использования
<code>memcpy(p1,p2,n)</code>	Копирование <code>n</code> байтов из области, на которую указывает <code>p2</code> , в область, на которую указывает <code>p1</code> . Области не должны перекрываться.	<code>memcpy((void*)buf1, (void*)in_msg, msg_len);</code>
<code>memmove(p1,p2,n)</code>	Копирование <code>n</code> байтов из области, на которую указывает <code>p2</code> , в область, на которую указывает <code>p1</code> . Области могут перекрываться (копирование происходит через промежуточный буфер).	<code>memmove((void*)buf1, (void*)buf2, 25);</code>
<code>memset(p,b,n)</code>	Заполнение байтом <code>b</code> области размером <code>n</code> , на начало которой указывает <code>p</code> .	<code>memset((void*)buf, 0, buf_len);</code>



Управление битами

При управлении внешними устройствами, входящими в состав микроконтроллеров, часто встает задача управления отдельными битами в переменных.

Битовые операции

Знак операции	Описание операции	Примеры использования
~	Поразрядная инверсия	<code>a = ~b;</code>
	Поразрядная операция ИЛИ	<code>x = a 0x07;</code>
&	Поразрядная операция И	<code>m = a & 0xF0;</code>
^	Поразрядная операция исключающее ИЛИ	<code>n = a ^ b;</code>
>>	Битовый сдвиг числа вправо	<code>a = b >> 4;</code>
<<	Битовый сдвиг числа влево	<code>a = b << 2;</code>
=, &=, ^=, >>=, <<=	Составные присваивания (битовые операции с одной переменной)	<code>n = 0x01;</code> <code>a <<= 8;</code>

Установка в «1»

Операция **ИЛИ** (|) с числом, в котором нужный бит равен **1**:

$$X | 0 = X$$

$$X | 1 = 1$$

Было		Стало
1100	<code>a =0x01;</code>	110 1

Было		Стало
1100	<code>a^=0x81;</code>	0 10 1

Инверсия

Операция **ИСКЛЮЧАЮЩЕЕ ИЛИ** (^) с числом, в котором нужный бит равен **1**:

$$X \wedge 0 = X$$

$$X \wedge 1 = \overline{X}$$

Сброс в «0»

Операция **И** (&) с числом, в котором нужный бит равен **0**:

$$X \& 0 = 0$$

$$X \& 1 = X$$

Было		Стало
0111	<code>a&=0x0E;</code> <code>a&=~0x01;</code>	011 0

Было		Стало
1100	(<code>a&0x01</code>)	000 0
	(<code>a&0x04</code>)	0 1 00

Проверка значения

Операция **И** (&) с числом, в котором нужный бит равен **1** (т.е. обнуление всех остальных битов):

$$X \& 0 = 0$$

$$X \& 1 = X$$

Управление
битами



Блоки операторов в Си

Это способ описания *действий*

Блок операторов

56

- ❑ Операторы могут быть сгруппированы в *блоки* или *составные операторы*:

{

(последовательность операторов)

}

- ❑ Блок органичен при помощи двух разделителей:
 - ❑ левая фигурная скобка «{» обозначает начало блока;
 - ❑ правая фигурная скобка «}» обозначает конец блока.
- ❑ Внутри блока могут быть объявлены переменные. Такие переменные называются *локальными переменными* блока; они возникают в памяти в начале исполнения блока, при выходе из блока эта память освобождается.

Функции

57

Функция – основная единица программы в языке Си. В функциях описываются действия, которые должна выполнять программа.

- По сути функция – это поименованный блок операторов, то есть *действия*, имеющие имя.
- Каждая функция имеет имя. Правила образования имен – такие же, как и правила образования имен переменных.
- Перед использованием функцию необходимо объявить или *определить*. В определении функции указывают ее имя, список параметров, тип возвращаемого значения и, наконец, *тело функции* – составной оператор, описывающий действия, выполняемые функцией.
- В любой программе на Си должна быть функция с именем **main** (главная функция). С функции **main**, в каком бы месте текста она не находилась, начинается выполнение программы.

Функция определяется следующим образом:

```
тип_функции имя_функции (список_параметров)
{
    тело_функции
}
```

- Имя типа, стоящее перед именем функции, задает тип возвращаемого функцией значения. Бывают функции, которые не возвращают никакого значения; в этом случае в качестве имени типа используется ключевое слово **void**.
- Список параметров – это список локальных переменных, автоматически получающих значения в момент вызова функции.
- Если функция не использует параметров, то в круглых скобках не пишут ничего или пишут слово **void** (на самом деле разница тут есть и весьма существенная).

Функция может получать параметры и возвращать значения:

```
double GetAverage(double a, double b)
{
    double result = (a + b) / 2;
    return result;
}
```

- Оператор **return** прерывает выполнение функции. Если функция возвращает число, после слова **return** необходимо указать выражение, значение которого передастся в качестве результата. **return** может использоваться в теле функции сколько угодно раз.

```
int GetMax(int a, int b)
{
    if (a > b)
        return a;
    return b;
}
```

Функции

60

- Если функция не возвращает ничего, вместо типа пишут «**void**»:

```
void BlinkLED (unsigned char period)
{
    LED_PORT |= 0x02;    // ВКЛЮЧИТЬ СВЕТОДИОД
    Pause(period);
    LED_PORT &= ~0x02;    // ВЫКЛЮЧИТЬ СВЕТОДИОД
    Pause(period);
}
```

Объявление переменных и область видимости

- Язык Си позволяет объявлять переменные в любом месте файла с исходным текстом, в том числе внутри любой функции.
- Переменные, объявленные «снаружи» функций, доступны в любой функции, описанной ниже этого объявления. Иными словами, **область видимости** таких переменных – **весь файл** (*на самом деле всё ещё сложнее!*).
- Переменные, объявленные внутри функции, доступны только в ней. Такие переменные называются *локальными*. Они создаются всякий раз при вызове функции и уничтожаются при выходе из нее.

```
short x, y;
short GetAverage(short a, short b)
{
    long res=((long)a + (long)b) / 2;
    return (short)res;
}
int main(void)
{
    short z;
    x = 10; y = 12;
    z = GetAverage(x, y);
    printf("z=%d", z);
}
```

Параметры функции также являются ее локальными переменными. Их изменение никак не влияет на объекты, находящиеся за её пределами.

Препроцессор

Непосредственно перед
компиляцией текст программы
обрабатывается специальным
текстовым процессором – т.н.
«препроцессором».

Препроцессор

- Непосредственно перед компиляцией текст программы обрабатывается специальным текстовым процессором – **«препроцессором»**.
- Любая строка, начинающееся с символа решетки (#), является командой для препроцессора.
- Команды препроцессора используются для замены фрагментов текста (макроподстановок), для условной компиляции текста и для включения в текст программы содержимого других файлов.

Препроцессор

- Типичный пример использования макросов (макроподстановок) – замена константы на символическое имя:

```
#define PI 3.14
```

- Если в тексте есть такая строка, то все встречающиеся слова «PI» препроцессор будет автоматически заменять на «3.14». Таким образом, программист может использовать имя PI вместо того, чтобы каждый раз писать значение 3.14.
- Правила хорошего тона рекомендуют вообще избегать любых числовых констант в тексте программы.

Препроцессор

65

Пример. В программе необходимо в разных местах мигать лампочкой 12 раз.

Тогда в начале текста программы можно записать:

```
#define N 12    // количество миганий
```

Далее в тексте программы можно использовать «N» вместо константы 12:

```
char A;  
for(A = 0; A < N; A++)  
{  
    switch_lamp_on();  
    delay(2);  
    switch_lamp_off();  
    delay(2);  
}
```

Препроцессор

66

- Команды **#include** “имя_файла” или **#include** <имя_файла> позволяют включить в текст программы содержимое другого файла.
- Эта простая возможность позволяет разделить программу на две части – собственно текст и т. н. *заголовочные файлы*.
- В заголовочном файле описывают все необходимые макросы и заголовки функций, которые затем будут доступны из всех файлов, в которые включен данный заголовочный файл.
- Традиционно заголовочные файлы имеют расширение **.h**.

```
#include <stdio.h>
#include "Project_Def.h"
int main()
{
    // текст программы
}
```

Заголовочные файлы

67

Файл prj_def.h

```
#define N 12
```

Файл 1.c

```
#include "prj_def.h"
void flash_lamp()
{
    int i;
    for (i = 0; i < N; i++)
    {
        switch_lamp_on();
        delay(2);
        switch_lamp_off();
        delay(2);
    }
}
```

Файл 2.c

```
#include "prj_def.h"
void beeps()
{
    int i;
    for (i = 0; i < N; i++)
    {
        beep_on();
        delay(10);
        beep_off();
        delay(10);
    }
}
```

Заголовочные файлы

68

Заголовочные файлы используют также для *расширения* области видимости функций и глобальных переменных.

Файл motor.h

```
void motor_control(int mode);  
int get_motor_mode(void);
```

В .h-файле объявлены
прототипы функций

Файл main.c

```
#include "motor.h"  
void door_mng(void)  
{  
    switch(door_status)  
    {  
        case OPEN:  
            motor_control(M_MOVE_CCW);  
            break;  
        case CLOSE:  
            motor_control(M_MOVE_CW);  
            break;  
        case IDLE:  
            motor_control(M_STOP);  
            break;  
    }  
}
```

Файл comm.c

```
#include "motor.h"  
void parse_cmd(char cmd, char* param)  
{  
    switch(cmd)  
    {  
        ...  
        case CMD_M_DIRECT_CTRL:  
            motor_control((int)param[0]);  
            break;  
        ...  
    }  
}
```

Заголовочные файлы

69

Глобальная переменная `system_state` хранит текущее состояние системы. Чтобы она была доступна в других файлах, повторим объявление этой переменной с атрибутом **extern** в .h-файле.

Файл `prj_def.h`

```
extern char system_state;
```

Файл `main.c`

```
#include "prj_def.h"

char system_state;

void main(void)
{
    system_state = SS_START;
    system_init();
    ...

    if (sensor_ready())
        system_state = SS_IDLE;

    ...
}
```

Файл `comm.c`

```
#include "prj_def.h"
void parse_cmd(char cmd, char* param)
{
    switch(cmd)
    {
        ...
        case CMD_GET_SYSTEM_STATUS:
            answer[0] = system_state;
            break;
        ...
    }
}
```

Взаимодействие между модулями (файлами) ПО

- *Обмен данными* между разными файлами проекта происходит при помощи:
 - Глобальных переменных
 - Функций-геттеров и функций-сеттеров.

```
char system_state;  
  
void main(void)  
{  
    ...  
}
```

```
extern char system_state;  
  
void parse_cmd(char cmd, char* param)  
{  
    switch(cmd)  
    {  
        ...  
        case CMD_GET_SYSTEM_STATUS:  
            answer[0] = system_state;  
            break;  
        ...  
    }  
}
```

Взаимодействие между модулями (файлами) ПО

```
static char system_state;


void main(void)
{
    ...
}

char get_system_state(void)
{
    return system_state;
}

void set_system_state(char new_state)
{
    system_state = new_state;
}
```

```
char get_system_state(void);
void set_system_state(char new_state);

void parse_cmd(char cmd, char* param)
{
    switch(cmd)
    {
        ...
        case CMD_GET_SYSTEM_STATUS:
            answer[0] = get_system_state();
            break;
        ...
    }
}
```



Перечисления (перечислимый тип)

Используется, если переменная может принимать строго определённый (перечислимый) набор значений.

Перечислени

- Язык **Я** позволяет ограничить набор возможных значений переменной и *назвать* каждое из них:

```
enum Motor_cmd
{
    M_STOP,
    M_MOVE_CW,
    M_MOVE_CCW
};
```

- В фигурных скобках перечисляются имена всех возможных значений
- Компилятор самостоятельно выбирает размер переменных перечисляемого типа и числовые значения для имён.
- Можно указать числовые значения «вручную»:

```
enum LED_State
{
    LED_OFF = 0,
    LED_ON = 0xFF
};
```

Перечисления

- Для объявления переменной перечисляемого типа пишут ключевое слово `enum`, затем имя типа и имя переменной:


```
enum LED_State led1_state, led2_state;  
void motor_control(enum Motor_cmd mode);
```

- При использовании переменной перечисляемого типа её значения пишут так, как было указано при объявлении типа:

```
led1_state = LED_OFF;  
if (led2_state == LED_ON) {...}
```


- Переменные перечисляемого типа можно преобразовывать в любой целочисленный тип и обратно:

```
answer[0] = (char)led1_state;  
motor_control((enum Motor_cmd)command[2]);
```



Приёмы программирования встроенных систем

При написании ПО для микроконтроллеров используют ряд стандартных решений



Приемы программирования встроенных систем

- ❑ Нисходящее и восходящее программирование: правильное проектирование набора функций
- ❑ Повсеместное использование машин состояний
- ❑ Использование переменных-таймеров для организации периодических событий
- ❑ Ожидание событий через последовательный опрос битов готовности устройств и глобальных переменных-флагов
- ❑ Синхронизация потоков через переменные-флаги и двойную буферизацию

Проектирование набора функций

Функции управления в программе могут встречаться в трех разных местах:

- ❑ Функции главного цикла (или функции главного потока);
- ❑ Функции периодических действий;
- ❑ Функции обработки событий

```
void main(void)
{
    system_init();
    while (1)
    {
        func_mng();
        motor_mng();
        comm_mng();
        ...
    }
}
```


```
char old_keys;
void key_scan(void)
{
    char new_keys = GetKeys();
    if (new_keys ^ old_keys)
    {
        pressed_keys = new_keys &
            (new_keys ^ old_keys);
    }
    old_keys = new_keys;
}
```

```
void UART_rx_int(void)
{
    char rx_data = UART0_DR;
    onRxChar(rx_data);
}
```

Функции главного цикла

- **Функции главного цикла** вызываются из главного бесконечного цикла программы. Частота вызова – максимально возможная (десятки или сотни кГц)*.
- Функции главного цикла обычно содержит главную машину состояний, реализует задачи опроса датчиков, ожидания событий, управления передачей данных по интерфейсам.
- В системах с низким потреблением (с батарейным питанием) функции главного цикла **отсутствуют** – вместо них процессор обычно спит.

* Особый случай – функции инициализации, вызываемые *до* главного цикла



Функции периодических действий

- **Функции периодических действий** вызываются либо из прерывания от таймера либо из главного цикла при возникновении флага переполнения таймера. Частота вызова – 1 кГц или 100 Гц.
- Обратите внимание! Эти функции могут вызываться либо в контексте прерывания либо в контексте главного потока!
- В системах с низким потреблением (с батарейным питанием) периодические функции – основа работы всего ПО, именно они содержат главную машину состояний.
- Функции периодических действий можно рассматривать как функции второго (третьего и т.д.) потока в системах с *вытесняющей многозадачностью*.

Функции обработки событий

- **Функция обработки событий** вызываются либо из прерывания от устройства либо из главного цикла при возникновении флага события.
- Эти функции тоже могут вызываться либо в контексте прерывания либо в контексте главного потока!
- Принципиальная особенность таких функций – их аperiodичность, асинхронность по отношению к главному потоку и потокам таймеров. Невозможно предсказать, когда они выполнятся снова. Кстати, вполне возможно, что вообще *никогда*.
- Синхронизация с функциями главного цикла может происходить при помощи двух механизмов:
 - Глобальные переменные - флаги
 - Callback-функции

Проектирование набора функций

- Кроме функции управления в программе должны присутствовать и иные функции – функции вычислений и функции нижнего уровня.
- При *восходящем* программировании – от средств к цели – сначала пишутся функции нижнего уровня, а затем уже – функции управления.
 - Для отладки пишут специальные тестовые функции управления – юнит-тесты или модульные тесты.
- При *нисходящем* программировании - от целей к средствам – сначала проектируются машины состояний и пишутся функции управления. Для функций нижнего уровня пишутся только прототипы.
 - Для отладки вместо настоящих функций нижнего уровня используют функции-имитаторы объектов управления.
 - Высший пилотаж – *кроссплатформенная симуляция*, когда ПО встроенной системы отлаживают на «большом» компьютере.


Задачи программирования встроенных систем

Можно выделить четыре типа задач, которые приходится решать при написании программ для встроенных систем:

- Управление устройством (объектом)
- Взаимодействие с пользователем
- Регулирование
- Обмен данными:
 - С внешними устройствами на плате
 - С другими вычислительными системами

Машина состояний

- В основе программирования **систем управления** лежит прием, называемый *программирование машины состояний*.
- Машина состояний – это метод написания программ, в котором каждому состоянию системы сопоставлено некое число («код»), определяющее ее поведение в данный конкретный момент.
- Описав полный набор всех возможных состояний системы, а также правила перехода между этими состояниями, можно написать простую и наглядную программу управления.



Пример задачи с машиной состояний

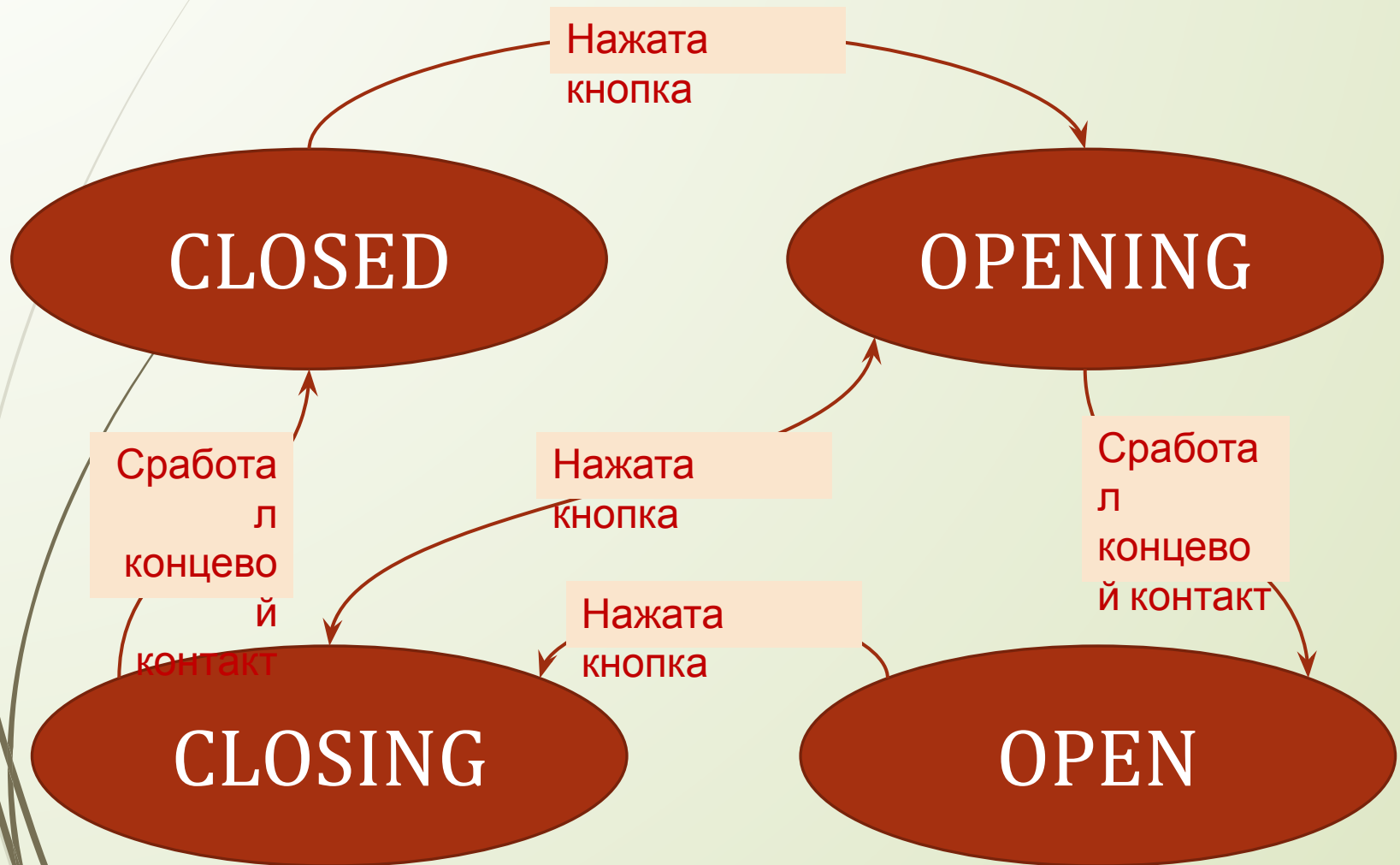
- Рассмотрим задачу управления автоматическими дверьми (например, воротами гаража и т.п.)
- Есть четыре состояния системы:
 - закрыта (обозначим его как CLOSED);
 - дверь открывается (OPENING);
 - дверь открыта (OPEN);
 - дверь закрывается (CLOSING).

Пример задачи с машиной состояний

- Управление осуществляется с помощью всего одной кнопки, которая открывает закрытую дверь, закрывает дверь открытую, а если дверь движется, то кнопка меняет направление движения (если дверь открывалась – она начнет закрываться и наоборот).
- Функция **GetKey(void)** возвращает 1, если была нажата кнопка.
- Функция **GetDoorSwitch(void)**, связанная с концевыми контактами на двери, возвращает 1, если дверь при движении уперлась в ограничитель, то есть полностью открылась или полностью закрылась.
- Управление двигателем двери осуществляется при помощи функции **ControlDoorMotor(char action)**, где параметр action может принимать три значения:
 - STOP (останавливает мотор);
 - RUN_CLOSE (запускает мотор в сторону закрытия);
 - RUN_OPEN (запускает мотор в сторону открытия).

Пример задачи с машиной состояний

Построим граф переходов для нашей задачи



Пример задачи с машиной

СОСТОЯНИЙ

```
while (1) // все происходит
{
    // в бесконечном цикле
    switch (state)
    {
        case CLOSED:
            ControlDoorMotor(STOP);
            if (GetKey())
                state = OPENING;
            break;

        case OPENING:
            ControlDoorMotor(RUN_OPEN);
            if (GetKey())
                state = CLOSING;
            if (GetDoorSwitch())
                state = OPEN;
            break;
```

```
        case OPEN:
            ControlDoorMotor(STOP);
            if (GetKey())
                state = CLOSING;
            break;

        case CLOSING:
            ControlDoorMotor(RUN_CLOSE);
            if (GetKey())
                state = OPENING;
            if (GetDoorSwitch())
                state = CLOSED;
            break;
    }
}
```

Именованние СОСТОЯНИЙ

Для того, чтобы наш пример с программой управления воротами скомпилировался, необходимо описать все символические имена с помощью директивы `#define`:

```
// СОСТОЯНИЯ СИСТЕМЫ
```

```
#define CLOSED      0
```

```
#define OPENING     1
```

```
#define OPEN        2
```

```
#define CLOSING     3
```

```
// КОМАНДЫ МОТОРА
```

```
#define STOP        0
```

```
#define RUN_OPEN    1
```

```
#define RUN_CLOSE   2
```


Именованье состояний

Описать символические имена можно и с помощью перечислимого типа:

```
// СОСТОЯНИЯ СИСТЕМЫ
```

```
enum Gate_State
```

```
{
```

```
    CLOSED, OPENING, OPEN, CLOSING
```

```
};
```

```
// команды мотора
```

```
enum Motor_Cmd
```

```
{
```

```
    STOP, RUN_OPEN, RUN_CLOSE
```

```
};
```

Сокращенное объявление СЛОЖНЫХ ТИПОВ В СИ

- Чтобы каждый раз не писать «struct» или «enum» при объявлении переменных сложных типов, можно использовать ключевое слово **typedef**.

typedef стандартный_тип имя_нового_типа

Например:

```
typedef struct {int x; int y;} point;
```

Объявление структуры

- typedef не создает «новый» тип, это просто способ символического обозначения существующего типа.
- Можно «переназывать» и скалярные типы:

```
typedef unsigned long uint32_t;
```

Сокращенное объявление СЛОЖНЫХ ТИПОВ В СИ

```
// СОСТОЯНИЯ СИСТЕМЫ
typedef enum
{
    CLOSED, OPENING, OPEN, CLOSING
} Gate_State;

// команды мотора
typedef enum
{
    STOP, RUN_OPEN, RUN_CLOSE
} Motor_Cmd;

...
Gate_State system_state = CLOSED;

...
void ControlDoorMotor(Motor_Cmd command);
```



Переменные-флаги

- В вырожденным случаем машины состояний является система с двумя состояниями. Переменная, которая описывает такую систему, называется переменной-флагом.
- Как правило, подобные бинарные машины состояний получаются, когда в программе необходимо ожидать некоего события. В этом случае возможны два состояния: событие не произошло или событие произошло.
- Как правило, в одном модуле (источнике события) флаг устанавливают, а в другом модуле – сбрасывают по окончании реакции на событие.

Переменные-флаги


- Пример: модуль интерфейса термостата. Устанавливает флаг «новая температура», когда пользователь заканчивает выбор нового значения температуры.

```
int new_temp_set = 0;
int ui_set_temp = 25;
void ui_mng(void)
{
    ...
    switch (get_key())
    {
        case KEY_UP:
            ui_set_temp++;
            break;
        case KEY_DOWN:
            ui_set_temp--;
            break;
        case KEY_SET:
            new_temp_set = 1;
            break;
    }
    ...
}
```

```
extern int new_temp_set;
extern int ui_set_temp;

int reg_set_temp;

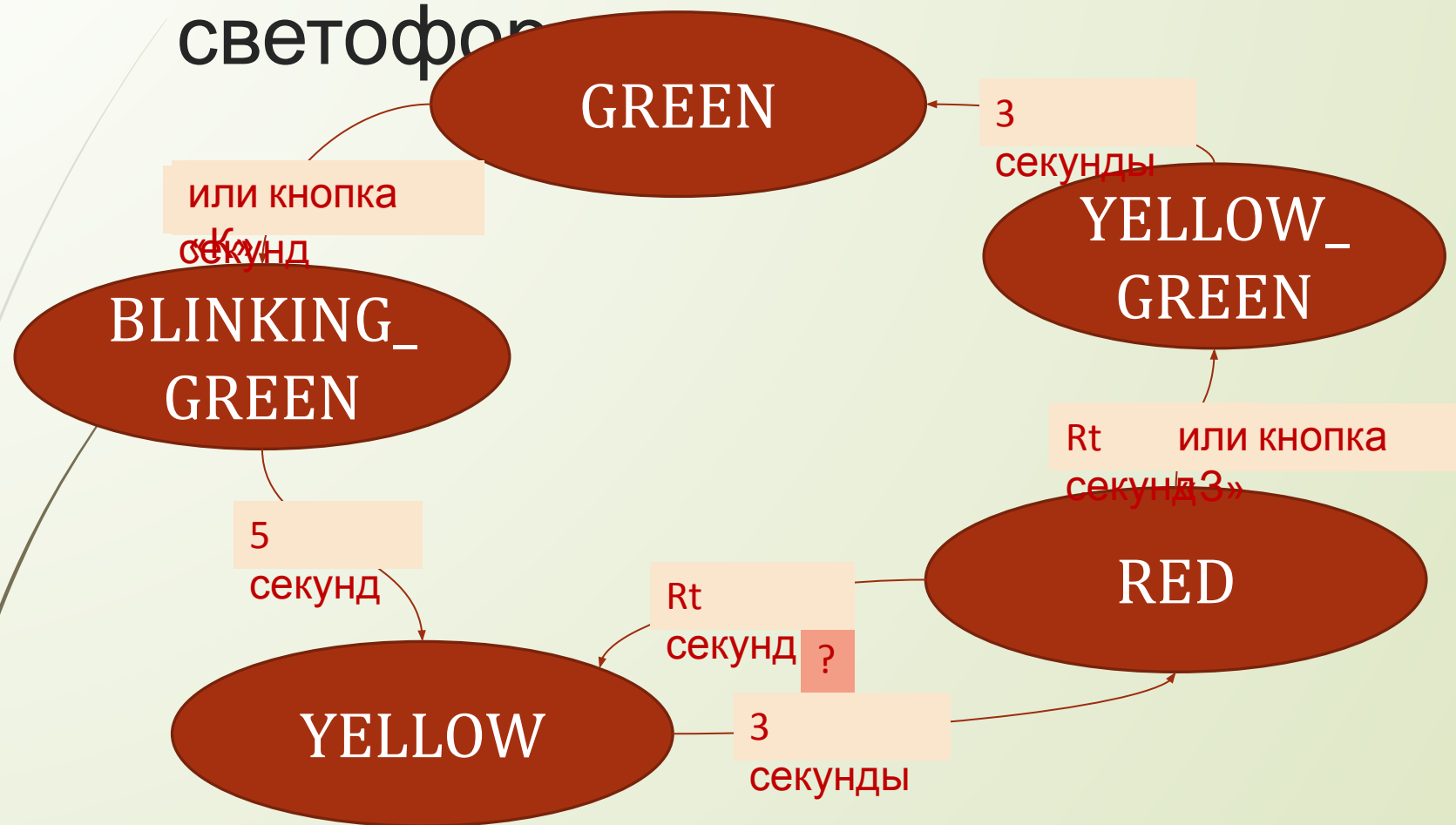
void func_mng(void)
{
    ...
    if (new_temp_set)
    {
        new_temp_set = 0; // сброс флага
        reg_set_temp = ui_set_temp;
    }
    ...
}
```



Пример с интерфейсом пользователя: управляемый светофор

- ❑ В программе, как правило, присутствуют несколько машин состояний, так как устройство взаимодействует с несколькими объектами.
- ❑ Рассмотрим пример управляемого светофора, который может работать как автоматически, так и по командам с пульта управления.
- ❑ Параметры работы светофора (длины фаз) будут заданы не константами, а переменными, которые впоследствии можно будет менять (по команде с пульта или по интерфейсу связи).
- ❑ Светофор моделируется на лабораторном стенде, при помощи трехцветного светодиода.

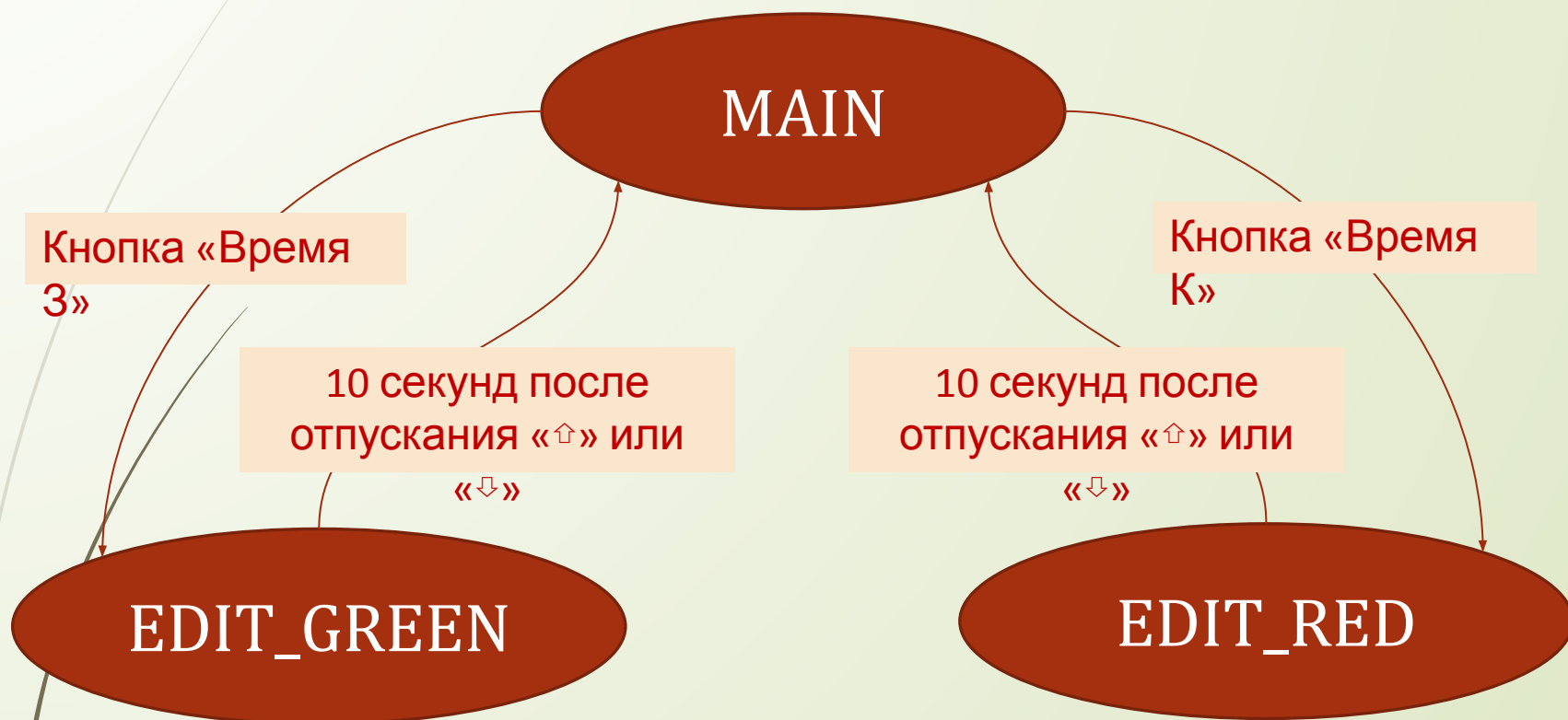
Машина состояний светофора



Машина состояний пульта

- Помимо основной машины состояний, в нашем устройстве будет еще одна машина, отвечающая за состояние интерфейса пользователя
- Интерфейс пользователя обеспечивает следующую функциональность:
 - Кнопки «К» и «З» для принудительного переключения фаз;
 - Отображение на индикаторах оставшегося времени текущей фазы или значения редактируемого времени фазы;
 - Кнопки «Время К» и «Время З», включающие режим редактирования времени фаз и кнопки «↑», «↓», изменяющие редактируемое значение времени.
 - Кнопки «Фикс. К» и «Фикс. З» для остановки смены фаз
- Таким образом, у интерфейса (пульта) есть три состояния: основное (MAIN), редактирование времени зеленой фазы (EDIT_GREEN) и редактирование времени красной фазы (EDIT_RED).

Машина состояний пульта



Использование переменных-таймеров

- Для нашего светофора нужно выдерживать (измерять) много разных временных интервалов.
- «Простой» подход – для каждого интервала своя отдельная переменная-счетчик, уменьшающаяся в функции-обработчике прерываний от таймера.


```
extern int Light_Time_Counter;
extern int UI_Time_Counter;
void Timer0AIntHandler(void)
{
    // сброс флага перезагрузки
    TIMER0_ICR_R = TIMER_ICR_TATOCINT;

    if (Light_Time_Counter > 0)
        Light_Time_Counter--;

    if (UI_Time_Counter > 0)
        UI_Time_Counter--;
}
```

```
int Light_Time_Counter = 0;
void func_mng(void)
{
    ...
    if (!Light_Time_Counter)
    {
        // переход к следующей фазе
        func_state = BLINKING_GREEN;
        Light_Time_Counter = 5000;
    }
    ...
}
```

```
int UI_Time_Counter = 0;
void ui_mng(void)
{
    ...
    if (!UI_Time_Counter)
    {
        // выход из редактирования
        ui_state = UI_MAIN;
    }
    ...
}
```



Использование переменных-таймеров

- У «простого» подхода два недостатка:
 - Плодятся глобальные переменные, причем модуль таймера должен «знать» о всех переменных-таймерах всех других модулей. Избавиться от глобальных переменных можно, заменив их на сеттеры или Callback-функции, но от многочисленных лишних связей «снизу вверх» это не избавит.
 - Набор действий в прерывании от таймера начинает зависеть от сложности системы.
- Выход: сделать **одну** глобальную переменную-счетчик или функцию, работающую с этим счетчиком.
- При этом модули, которые захотят считать время, включают в себя модуль таймера, а не наоборот: нет проблем с иерархией проекта.

Использование переменных-таймеров

```
unsigned int Clock_Counter = 0;
void Timer0AIntHandler(void)
{
    TIMER0_ICR_R = TIMER_ICR_TATOCINT;
    Clock_Counter++;
}

void Set_Timer(unsigned int* timer,
unsigned int period)
{
    *timer = Clock_Counter + period;
}

int Timer_Expired(unsigned int timer)
{
    if (Clock_Counter >= timer)
        return 1;
    return 0;
}
```

```
unsigned int Light_Timer;
void func_mng(void)
{
    ...
    if (Timer_Expired(Light_Timer))
    {
        // переход к следующей фазе
        func_state = BLINKING_GREEN;
        Set_Timer(&Light_Timer, 5000);
    }
    ...
}
```

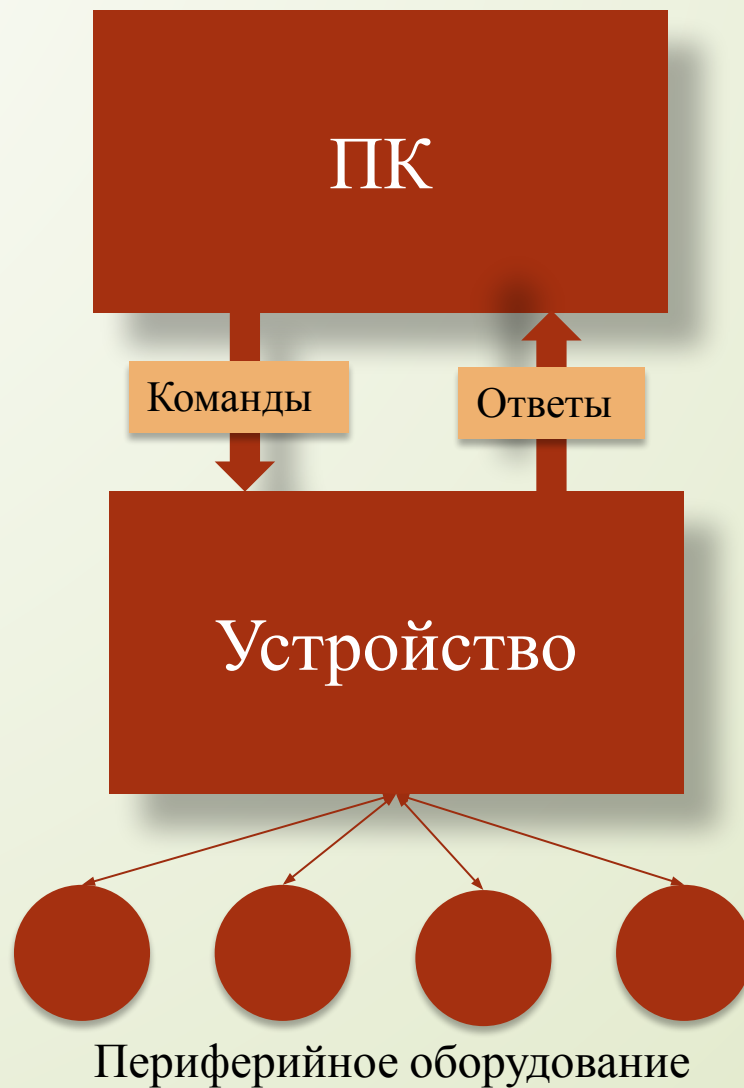
```
unsigned int UI_Timer;
void ui_mng(void)
{
    ...
    if (Timer_Expired(UI_Timer))
    {
        // выход из редактирования
        ui_state = UI_MAIN;
    }
    ...
}
```

Задачи взаимодействия устройств

Взаимодействие сводится к обмену данными между устройствами при помощи цифровых интерфейсов

- В результате коммуникации между устройствами (узлами) решаются две задачи:
 - Управление устройством
 - Получение состояния устройства (мониторинг)

Взаимодействие устройств

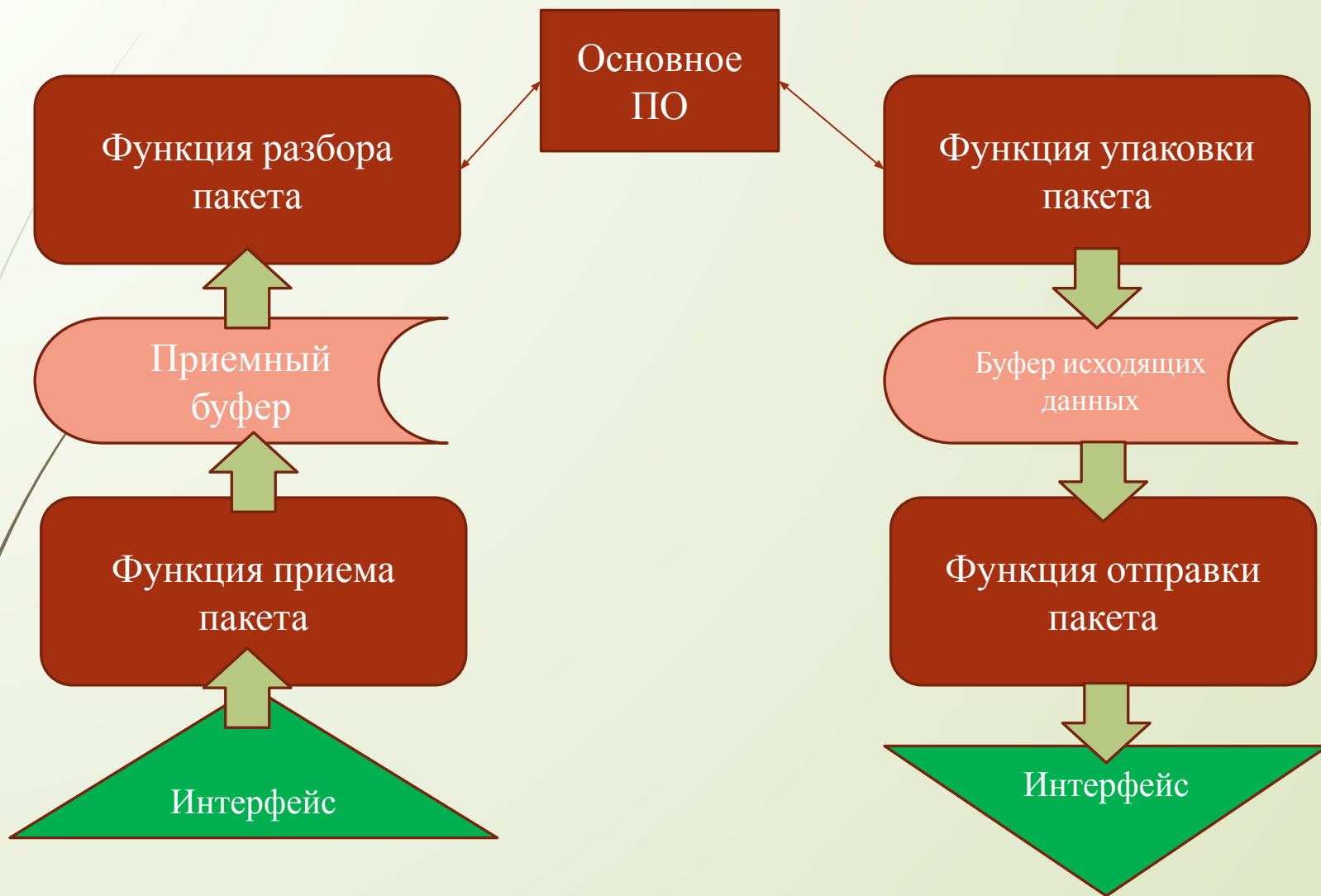




Функции коммуникационного ПО

- ❑ Функции приема и передачи данных через интерфейсные устройства
- ❑ Функции разбора принятых пакетов данных (функции-парсеры)
- ❑ Функции упаковки исходящих пакетов данных

Функции коммуникационного ПО



Прием пакета

- Основная задача – **синхронизация** между медленными интерфейсами, редкими, асинхронными событиями приёма и быстрым ядром.
- Возможны два способа реагирования на асинхронные события – через *опрос бита готовности* приемника и через *прерывания*.

Прием пакета

- **Опрос бита готовности:** в каждом устройстве (контроллеры SPI, UART, I2C,...) есть регистр *статуса*, в котором присутствуют бит (или биты), имеющие смысл «приёмник содержит входящий байт». Эти биты опрашиваются в *функции главного цикла*, которая и читает принятые устройством байты.
- **Достоинства:** минимальный расход процессорного времени, простой код, прием и разбор пакета происходит в одном и том же потоке.
- **Недостатки:** главный цикл должен гарантировано успевать прочитать входящий байт до того, как придет новый.

Прием пакета

- **Прерывание:** событие приёма байта вызывает прерывание ядра. Данные читаются из устройства в функции обработки прерывания.
 - Достоинства: быстрая реакция системы на пришедшие данные; меньше вероятность потери данных, больше свободы в написании функций главного цикла.
 - Недостатки: более сложный код, так как необходимо решать вопросы синхронизации потоков (прием и разбор пакетов происходит в разных потоках).

Прием пакета

```
#define PACKET_LEN 6
unsigned int rx_counter = PACKET_LEN;
unsigned char rx_buffer[PACKET_LEN];

void rx_mng(void)
{
    if (rx_counter < PACKET_LEN)
    {
        if (USART_SR & USART_SR_RXNE)
        {
            rx_buffer[rx_counter++] = USART_DR;
            // если пакет принят, то разбираем его
            if (rx_counter == PACKET_LEN)
                parse_packet(rx_buffer);
        }
    }
}

void start_rx(void)
{
    rx_counter = 0;
}
```

Опрос бита

```
#define PACKET_LEN 6
unsigned int rx_counter = PACKET_LEN;
unsigned char rx_buffer[PACKET_LEN];
int data_received_flag = 0;

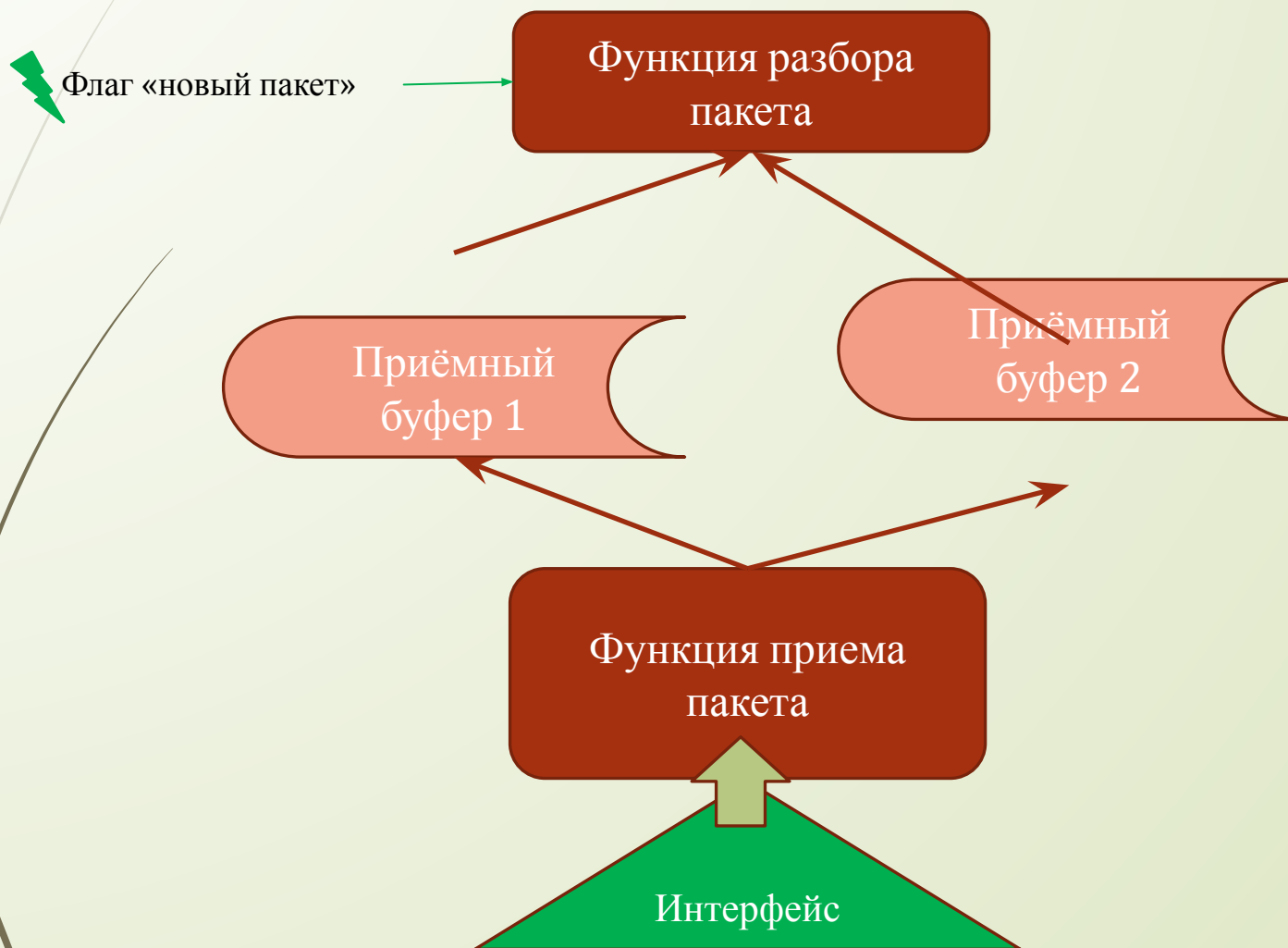
void USARTIntHandler(void)
{
    unsigned char rx_data;
    if (USART_SR & USART_SR_RXNE)
    {
        // всегда читаем байт, чтобы
        // сбросить флаг прерывания
        rx_data = USART_DR;

        if (rx_counter < PACKET_LEN)
        {
            rx_buffer[rx_counter++] = rx_data;
            // если пакет принят, то ставим флаг
            if (rx_counter == PACKET_LEN)
                data_received_flag = 1;
        }
    }
}
```

Прерывание

Двойная буферизация

- Позволяет начать прием нового пакета сразу же, не дожидаясь окончания разбора предыдущего пакета



Двойная буферизация

```
#define PACKET_LEN 6
unsigned int rx_counter = 0;
unsigned char rx_buffer1[PACKET_LEN];
unsigned char rx_buffer2[PACKET_LEN];
unsigned char* cur_rx_buffer = rx_buffer1;
unsigned char* parse_buffer = rx_buffer2;
int data_received_flag = 0;
void USARTIntHandler(void)
{
    unsigned char rx_data;
    if (USART_SR & USART_SR_RXNE)
    {
        rx_data = USART_DR;
        if (rx_counter < PACKET_LEN)
        {
            cur_rx_buffer[rx_counter++] = rx_data;
            if (rx_counter == PACKET_LEN)
            {
                parse_buffer = cur_rx_buffer;
                data_received_flag = 1;
                // переключаем буфер...
                if (cur_rx_buffer == rx_buffer1)
                    cur_rx_buffer = rx_buffer2;
                else
                    cur_rx_buffer = rx_buffer1;
                // ...и сразу готовы принимать дальше
                rx_counter = 0;
            }
        }
    }
}
```

```
extern unsigned char* parse_buffer;
extern int data_received_flag;

void rx_mng(void)
{
    if (data_received_flag)
    {
        parse_packet(parse_buffer);
        data_received_flag = 0;
    }
}
```



Отправка пакета

- Основная задача, которая здесь решается – синхронизация быстрого ядра и медленного интерфейса: данные отправляются в внешний мир гораздо медленнее, чем ядро способно их отдавать.
- Функции отправки аналогичны функциям приема: синхронизация возможна как при помощи опроса битов готовности (в этом случае проверяется бит «готовность передатчика»), так и при помощи прерываний.
- Недостатки, связанные с синхронизацией при помощи опроса бита готовности, здесь не столь существенны, так как при отправке данных невозможно «опоздать» (однако в некоторых случаях можно превысить максимально допустимое время между байтами – это уже зависит от протокола).

Отправка пакета

```
#define PACKET_LEN 6
unsigned int tx_counter = PACKET_LEN;
unsigned char tx_buffer[PACKET_LEN];

void tx_mng(void)
{
    if (tx_counter < PACKET_LEN)
    {
        if (USART_SR & USART_SR_TXE)
        {
            USART_DR = tx_buffer[tx_counter++];
        }
    }
}

void start_tx(void)
{
    tx_counter = 0;
}
```

Опрос бита

```
void USARTIntHandler(void)
{
    if (USART_SR & USART_SR_TXE)
    {
        if (tx_counter < PACKET_LEN)
        {
            USART_DR = tx_buffer[tx_counter++];
        }
    }
    else
    {
        // если нечего передавать,
        // то запрещаем прерывания
        UART_CR1 &= ~TXEIE;
    }
}

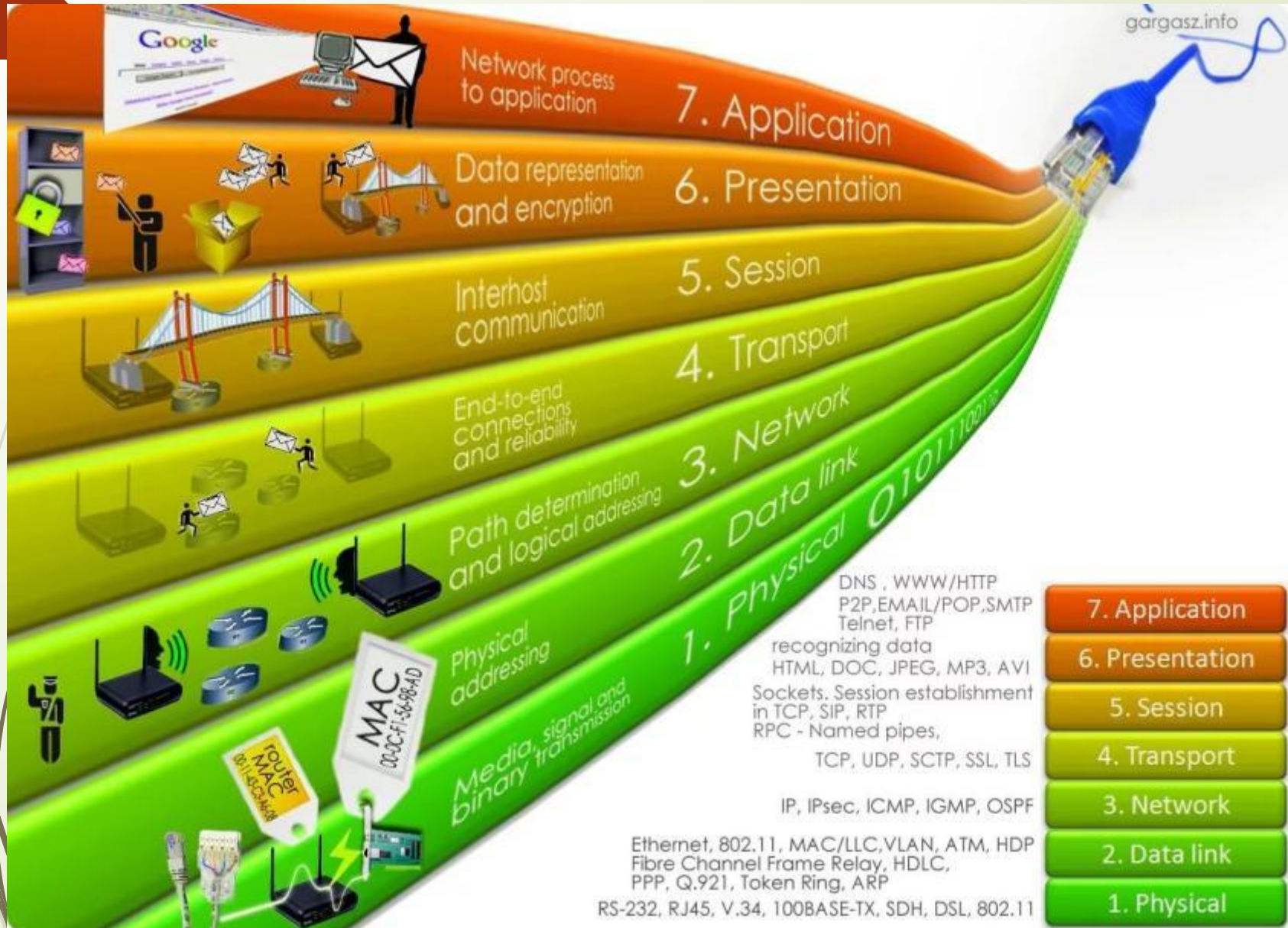
void start_tx(void)
{
    tx_counter = 0;
    UART_CR1 |= TXEIE; // разрешаем
    прерывание
}
```

Прерывание

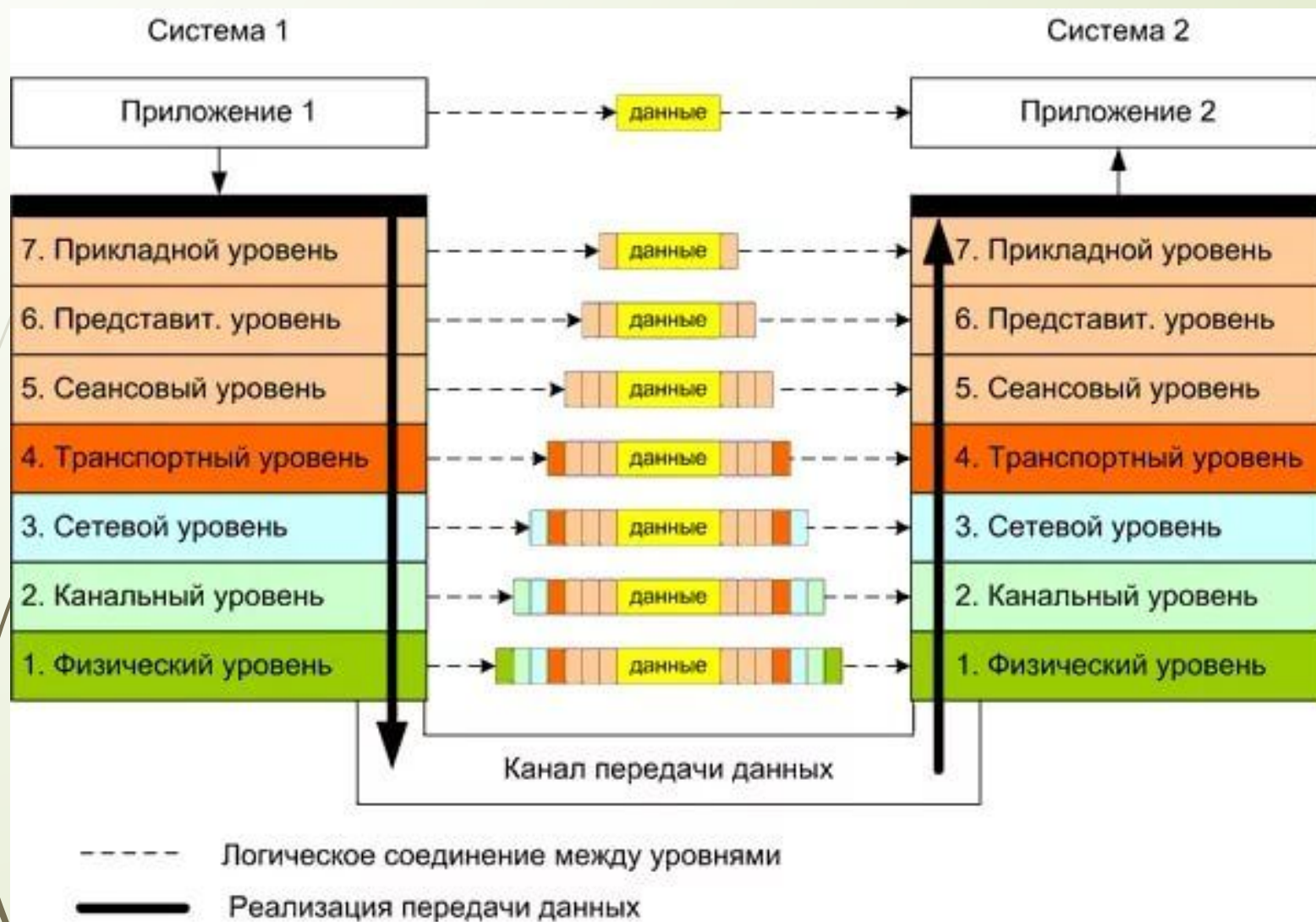
Взаимодействие систем



Взаимодействие систем



Взаимодействие систем



Упрощенный стек протоколов

- В простых системах часть уровней «сливаются» друг с другом.

Приложение	Прикладной уровень (функциональность системы)
Представление	
Сеанс	Сессионный уровень
Транспорт	Транспортный уровень
Сеть	
Канал	
Физика	Физический уровень

Упрощенный стек протоколов


□ Пример:

Формат транспортного пакета

Заголовок	Длина поля данных	Данные	Контрольная сумма (2 байта)	
0x01A0 + номер устройства в шине	0: Ping/Ack 1..255: Обычный пакет	От 0 до 255 байт	CRC ₁₆ L	CRC ₁₆ H

Формат пакета прикладного уровня:

Байт 0		Байт 1	Байты 2..6
биты 7..4 Категория	биты 3..0 Режим	Команда	Данные (параметры команды)



Задачи, решаемые на разных уровнях стека

□ Прикладной уровень:

- Поддержка основной функциональности устройства (команды, ответы на них, информирование об ошибках)
- Описание способов представления сложных объектов (длинные числа, строки и т.п.)

□ Уровень сессии:

- Обеспечение завершенности обмена (на каждый запрос должен быть получен ответ)

□ Транспортный уровень:

- Синхронизация источника и приемника
- Обеспечение надежности доставки
- В многоточечной сети: обеспечение адресации узлов

Задачи, решаемые на разных уровнях стека: прикладной уровень

Как правило, программно реализуется при помощи структур и объединений

```
typedef struct {  
    unsigned char command;  
    unsigned char params[6];  
} app_data_t;
```

// Команды

```
#define CMD_MOTOR_CONTROL      1  
#define CMD_MOTOR_CALIBR      2  
#define CMD_MOTOR_PARAMETERS  3
```

```
void execute_cmd(unsigned char* app_pkt)  
{  
    app_data_t* msg = (app_data_t*)app_pkt;  
    app_data_t answer;  
  
    switch (msg->command)  
    {  
        case CMD_MOTOR_CONTROL:  
            motor_ctrl(msg->params[0], msg->params[1]);  
            break;  
  
        case CMD_MOTOR_CALIBR:  
            motor_calibr(msg->params[0]);  
            break;  
  
        ...  
    }  
}
```

Задачи, решаемые на разных уровнях стека: прикладной уровень

Как правило, программно реализуется при помощи структур и объединений

```
// Поля пакета
#define COMMAND_IDX 0
#define PARAM_IDX 1

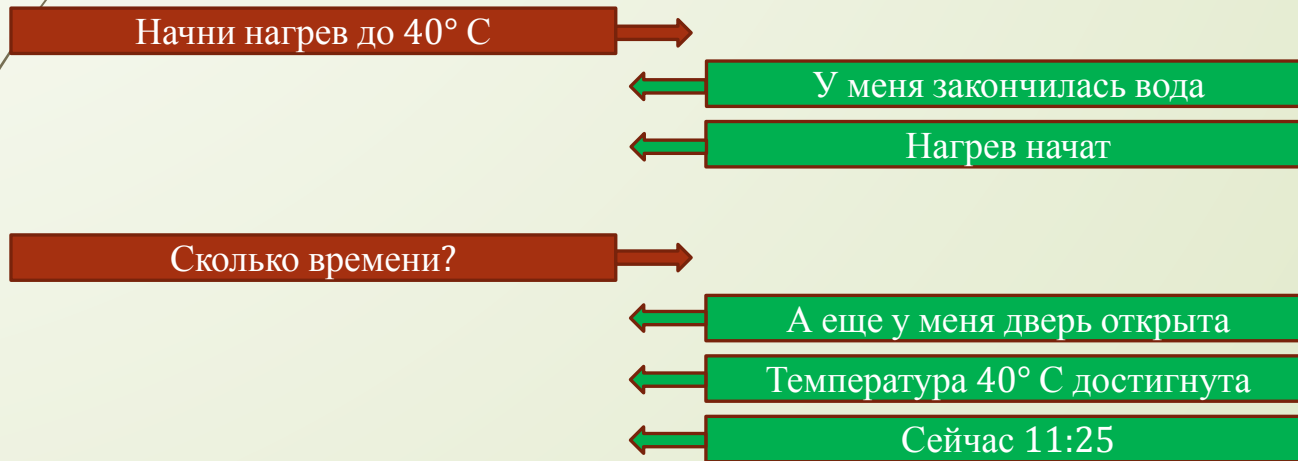
// Команды
#define CMD_MOTOR_CONTROL 1
#define CMD_MOTOR_CALIBR 2
#define CMD_MOTOR_PARAMETERS 3
```

```
void execute_cmd(unsigned char* app_pkt)
{
    switch (app_pkt[COMMAND_IDX])
    {
        case CMD_MOTOR_CONTROL:
            motor_ctrl(app_pkt[PARAM_IDX],
app_pkt[PARAM_IDX+1]);
            break;


        case CMD_MOTOR_CALIBR:
            motor_calibr(app_pkt[PARAM_IDX]);
            break;
        ...
    }
}
```


Задачи, решаемые на разных уровнях стека: уровень сессии

- Уровень сессии как самостоятельная сущность реализуется только если протокол допускает вложенность сессий:
 - Устройство может в любой момент начать говорить само, не дожидаясь команды от мастера.
 - Устройство может начать говорить даже сразу после команды мастера, отложив ответ на команду на потом.



Для управления такой сессией вводится понятие *идентификатора сессии*, который передается в каждом пакете.



Задачи, решаемые на разных уровнях стека: транспортный уровень

- Синхронизация источника и приемника:
 - В начало пакета добавляется уникальный заголовок
 - Спецификация временных интервалов между пакетами
 - В пакетах с переменной длиной в начало добавляется поле длины пакета
- Обеспечение надежности доставки:
 - В конец пакета добавляется контрольная сумма
- Адресация узлов в многоточечных сетях:
 - В начало пакета добавляется адрес устройства

Задачи, решаемые на разных уровнях стека: транспортный уровень

□ В итоге транспортный пакет может выглядеть так:

Заголовок	Адрес назначения	Длина пакета	Данные			Контрольная сумма
0xAA	Dst	LEN		...		CSUM

Или так:

Заголовок		Адреса		Длина пакета		Данные			Контрольная сумма	
0xDE	0xBE	Dst	Src	LEN _L	LEN _H		...		CRC16 _L	CRC16 _H

Реализация протокола

Прием пакета

- Машина состояний

Разбор пакета

- Разбор буфера (+двойная буферизация)

Выполнение команд

- Создание маркированной очереди команд
- Прямое исполнение вида «запрос-ответ»

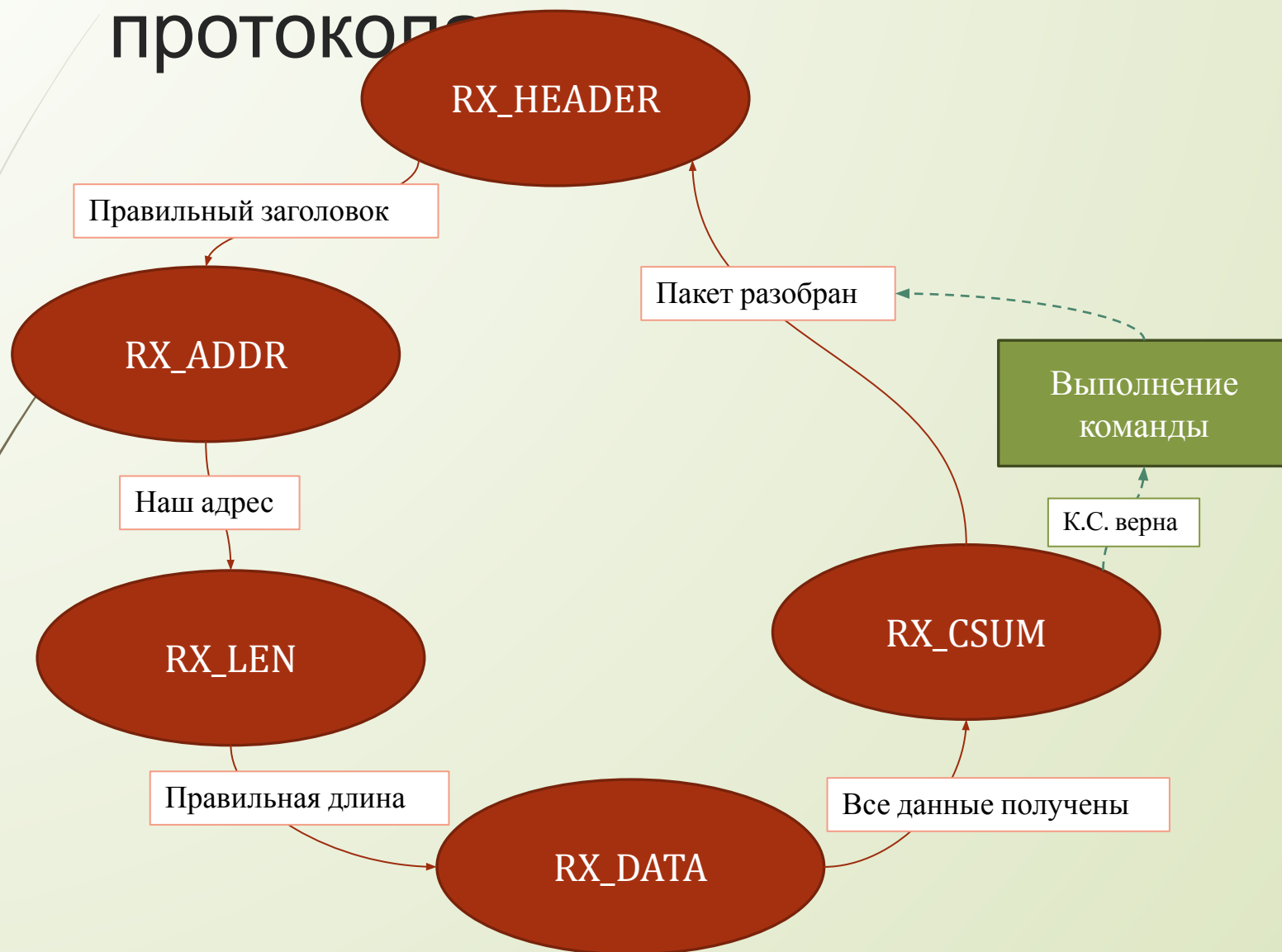
Подготовка ответа

- Заполнение буфера (+двойная буферизация)

Передача ответа

- Машина состояний или простой цикл

Реализация протокола





Реализация протокола



```
unsigned char packet_len;  
unsigned int rx_counter;  
unsigned char rx_buffer[MAX_PACKET_LEN];  
int data_received_flag = 0;  
  
enum  
{  
    RX_HEADER,  
    RX_ADDR,  
    RX_LEN,  
    RX_DATA,  
    RX_CSUM  
} rx_state = RX_HEADER;
```