Лекция 3 Применение языка С для встраиваемых систем

План курса «Встраиваемые микропроцессорные системы»:

Лекция 1: Введение. Язык программирования С

Лекция 2: Язык программирования С. Стандартная библиотека языка С

Лекция 3: Применение языка С для встраиваемых систем

Лекция 4: Микроконтроллер

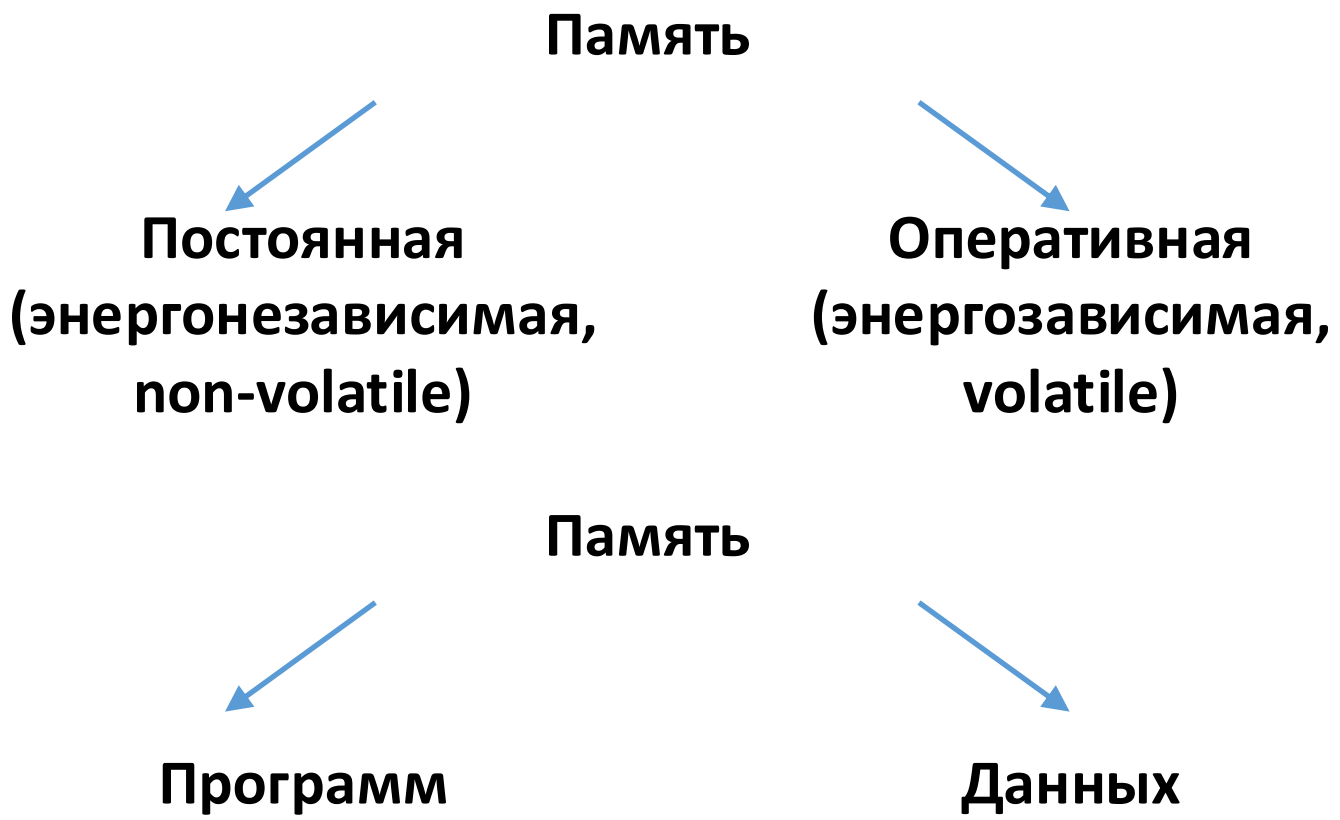
Лекция 5: Этапы разработки встраиваемых систем

Лекция 6: Разработка и отладка программ для встраиваемых систем

Лекция 7: Архитектура программ для встраиваемых систем

Лекция 8: Периферийные модули: USB, Ethernet

Классификация памяти



Постоянная память (ПЗУ) преимущественно используется как память программ, но также для данных – константы.
Оперативная память (ОЗУ) преимущественно используется как память данных, но в высокопроизводительных системах и как память программ.

Модель памяти в языке С

Память данных

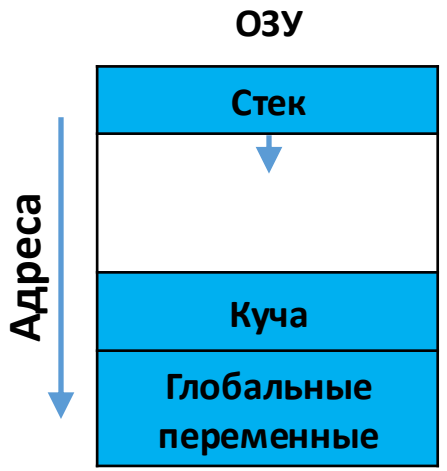
Статическая

выделение при компоновке,
глобальные переменные,
время жизни – время работы программы

```
int a;
int b = 42;
void main()
{
    static int j;
    …
}
```

Динамическая

выделение в процессе работы программы



Автоматическая

стековые переменные,
время жизни – внутри фигурных скобок

```
void main()
{
    int d;
    char e[100];
    …
}
```

Ручная

переменные из кучи,
время жизни – до вызова функции free

```
void main()
{
    char *p = malloc(100);
    …
    free(p);
}
```

Динамическая и статическая память

Статическая память

Выделение памяти при компоновке (link time).

\+ Легко управлять. Быстрая инициализация.

\+ Детерминированное поведение. Атомарность.

\- Фиксированный размер.

Оптимально когда нужно использовать ресурсы одновременно.

```
#define ARR_SIZE 32
char x[ARR_SIZE]; // создание
char a[ARR_SIZE];
…
{
 x = ;        // инициализация
 a = ;
 …
 filter(x, a);    // исполнение
}
…
```

Динамическая ручная память

Выделение памяти при исполнении (run time).

\+ Совместное использование общих ресурсов.

\+ Память может быть освобождена.

\- Сложнее управлять.

\- Недетерминированное поведение. Неатомарность.

Оптимально когда неизвестен необходимый объем памяти или несколько задач использует одни и те же ресурсы.

```
#define ARR_SIZE 32
char* x = malloc(ARR_SIZE); // создание
char* a = malloc(ARR_SIZE);
…
x = ;              // инициализация
a = ;
…
filter(x, a);          // исполнение
…
free(x);               // освобождение
free(a);
```

Секции памяти

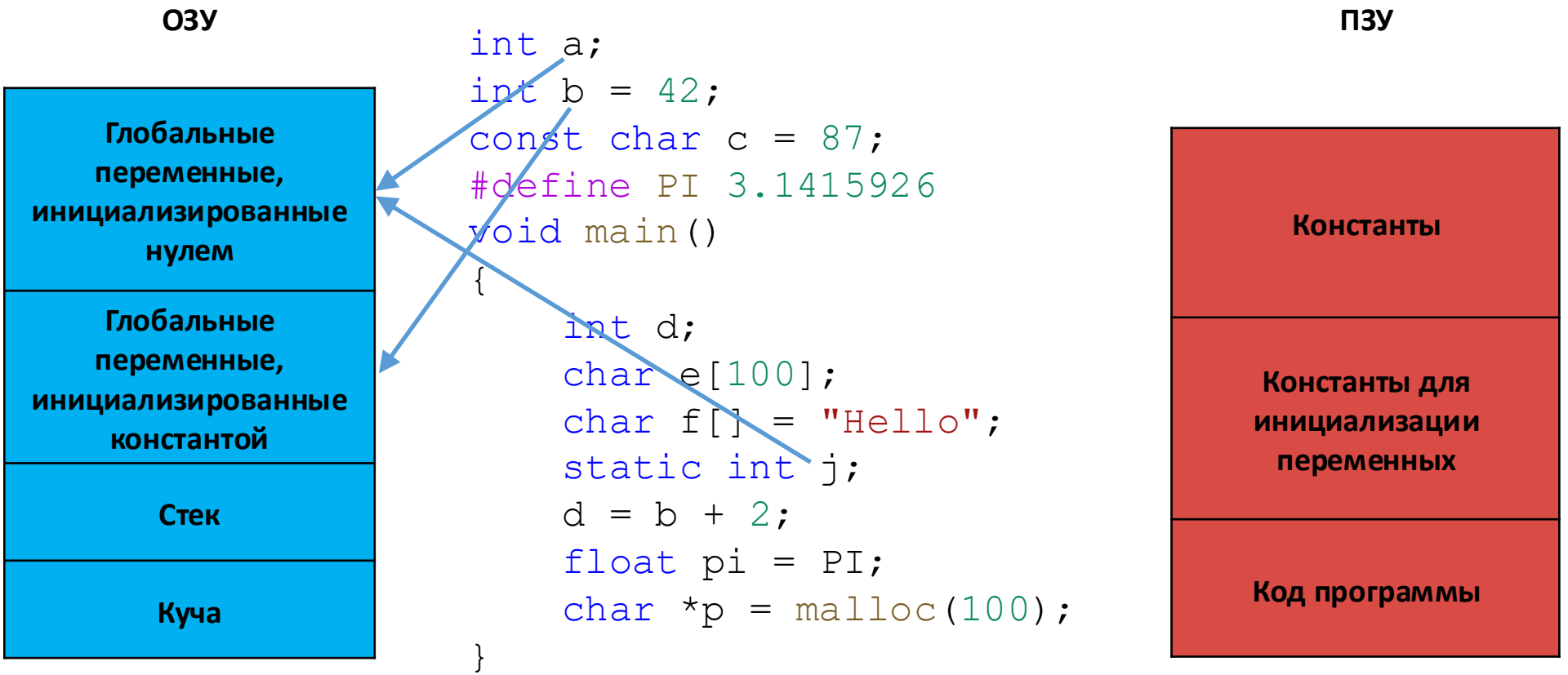
Наименование	Название компоновщика gcc	Название компоновщика Keil	Описание
Код программы	.text	ER_CODE, .text	Содержит исполняемый код.
Глобальные переменные, инициализированные константой	.data	RW_DATA, .data	Содержит переменные, которые инициализированы до старта программы.
Глобальные переменные, инициализированные нулем	.bss	RW_IRAM, .bss	Содержит переменные, которые инициализированы нулями.
Константы	.rodata	ER_RODATA, .constdata	Секция для констант и неизменяемых данных.
Стек	Не определена по умолчанию (задается в скрипте линкера)	Не определена по умолчанию (задается в скрипте линкера)	Хранит данные во время выполнения функций (локальные переменные, адреса возврата).
Куча	Не определена по умолчанию (задается в скрипте линкера)	Не определена по умолчанию (задается в скрипте линкера)	Используется для динамического выделения памяти во время выполнения программы.
Таблица векторов прерываний	Обычно находится в начале .text или отдельной секции .vectors (.isr_vectors)	ER_VECTORS, .intvec	Хранит адреса обработчиков прерываний.

Распределение памяти на секции производит компоновщик (linker) по сценарию компоновщика (linker script).

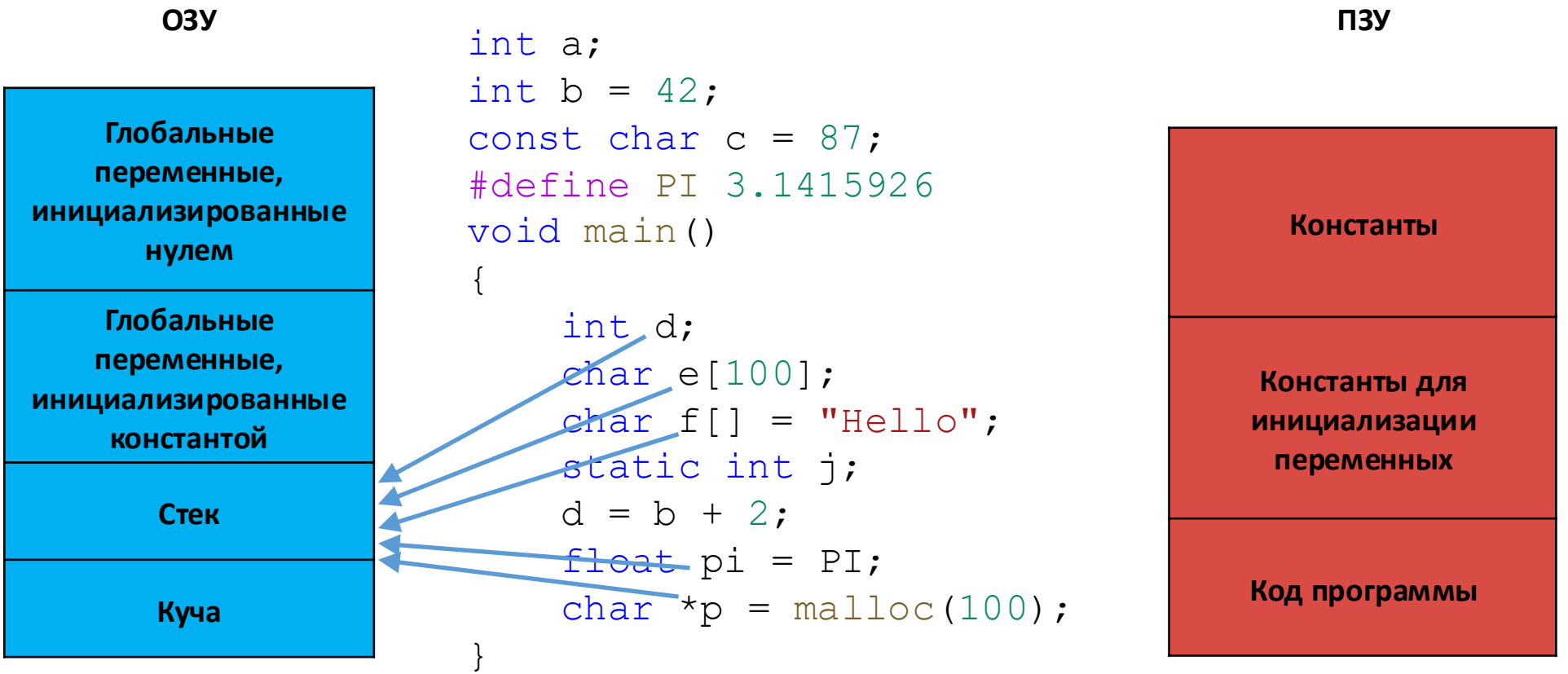
Модель памяти в языке С


```c
int a;
int b = 42;
const char c = 87;
#define PI 3.1415926
void main()
{
    int d;
    char e[100];
    char f[] = "Hello";
    static int j;
    d = b + 2;
    float pi = PI;
    char *p = malloc(100);
}
```

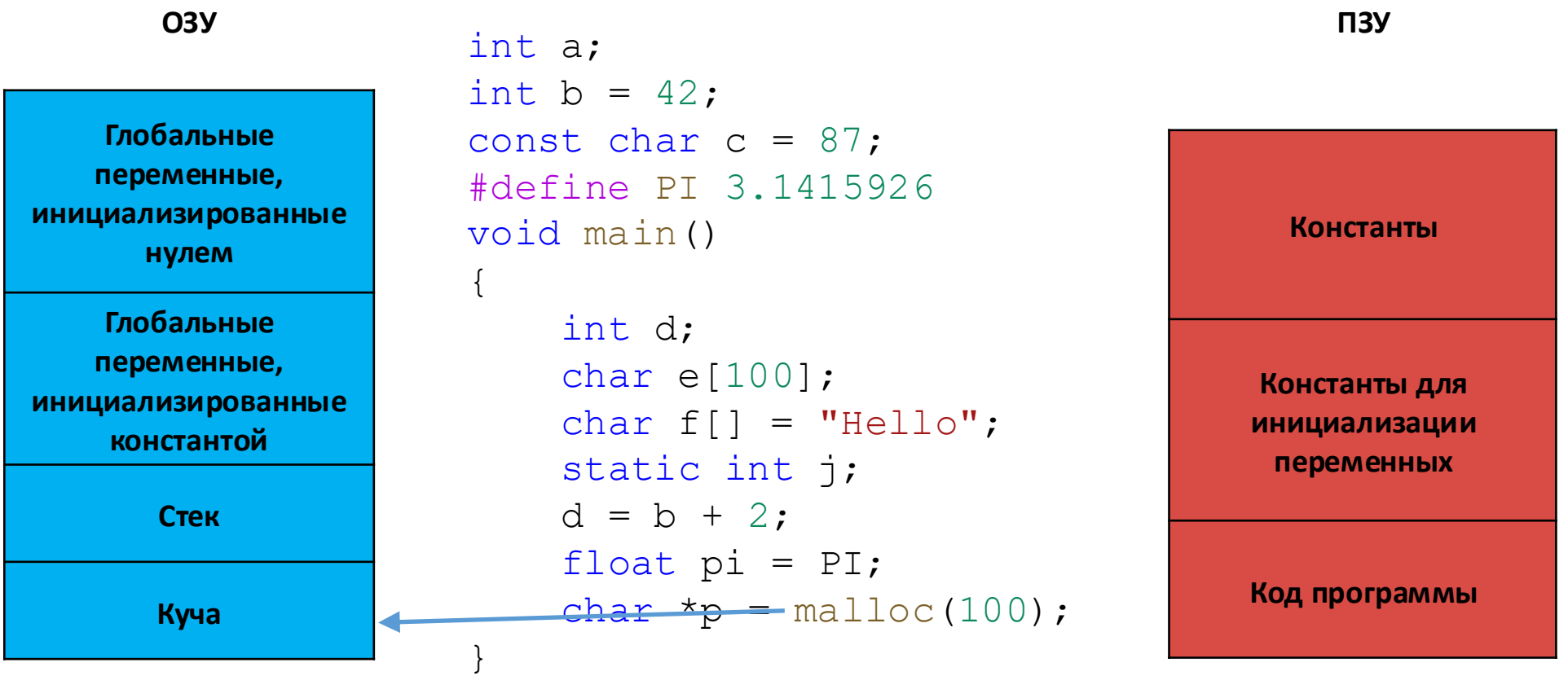
Модель памяти в языке С: глобальные переменные



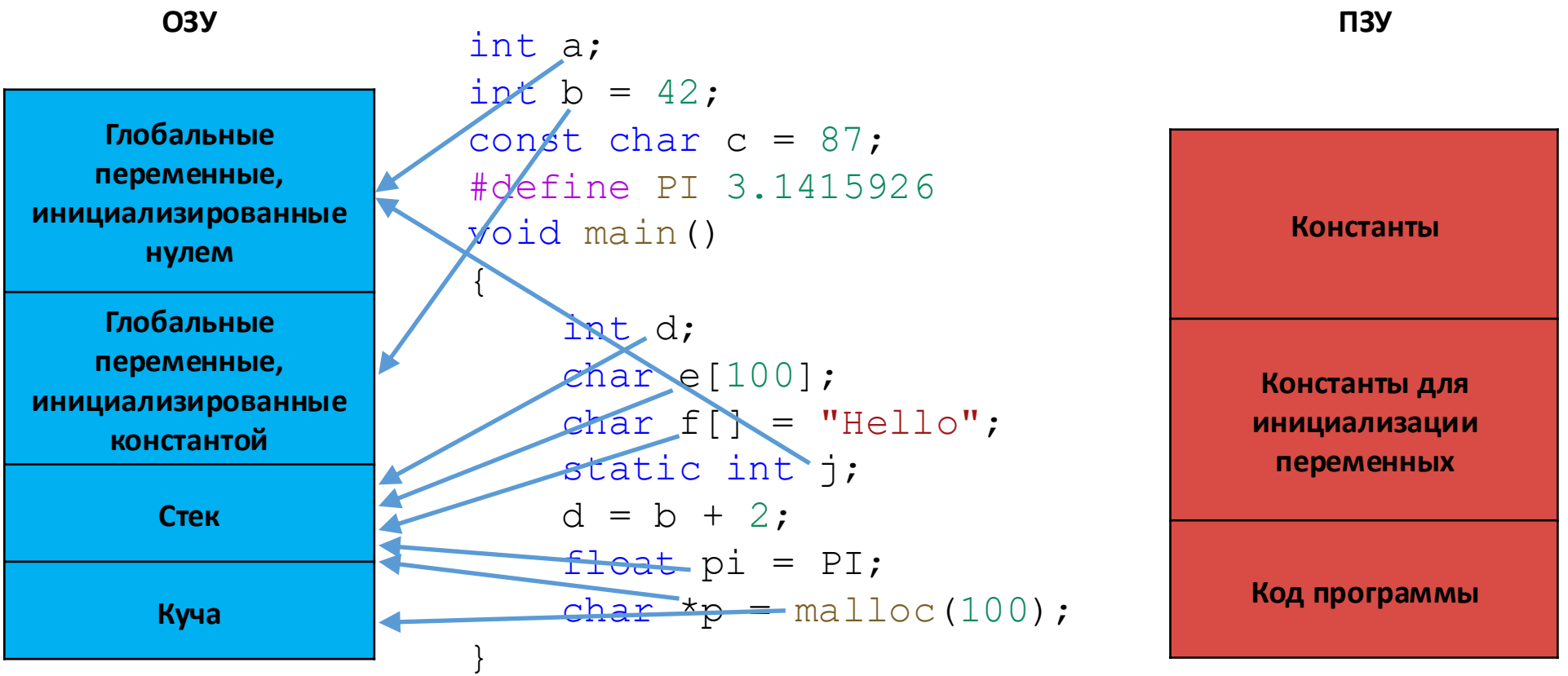
Модель памяти в языке С: локальные переменные



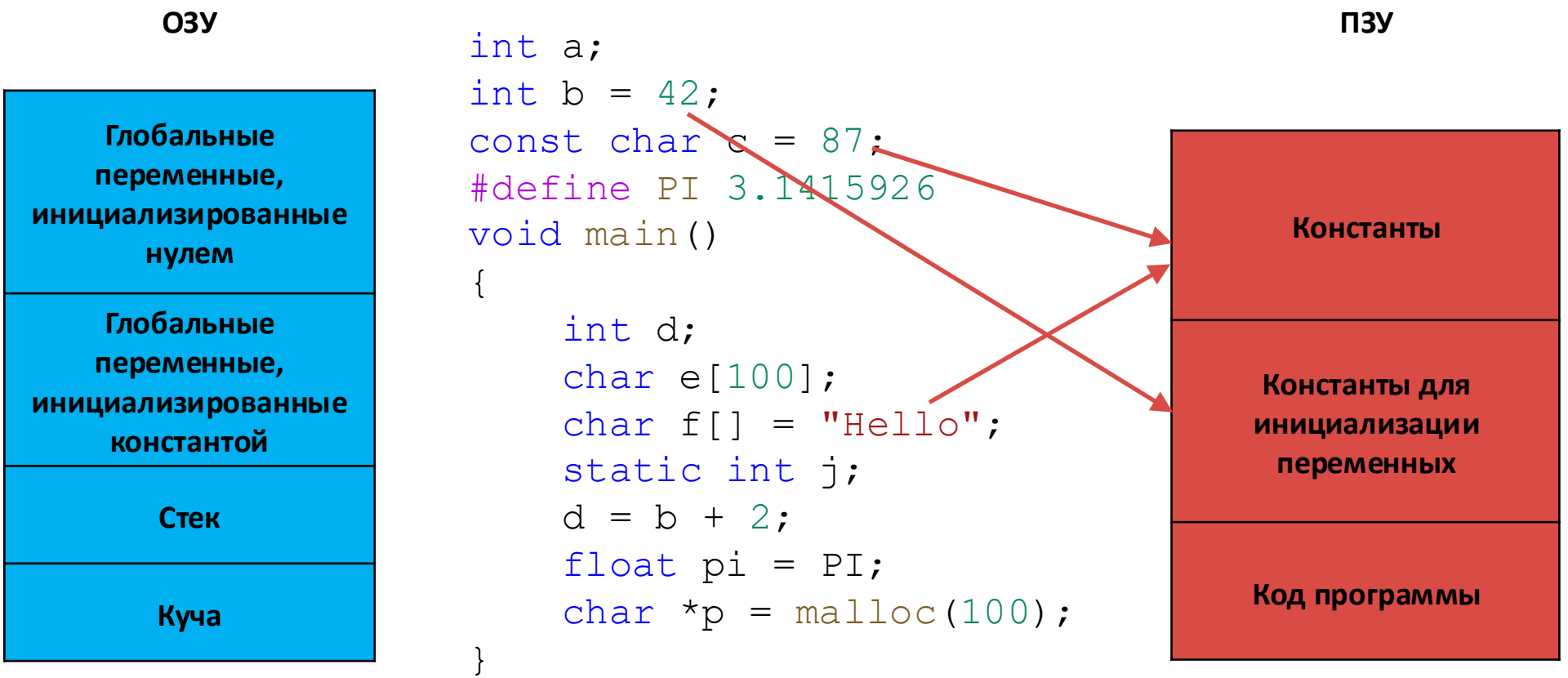
Модель памяти в языке С: куча



Модель памяти в языке С



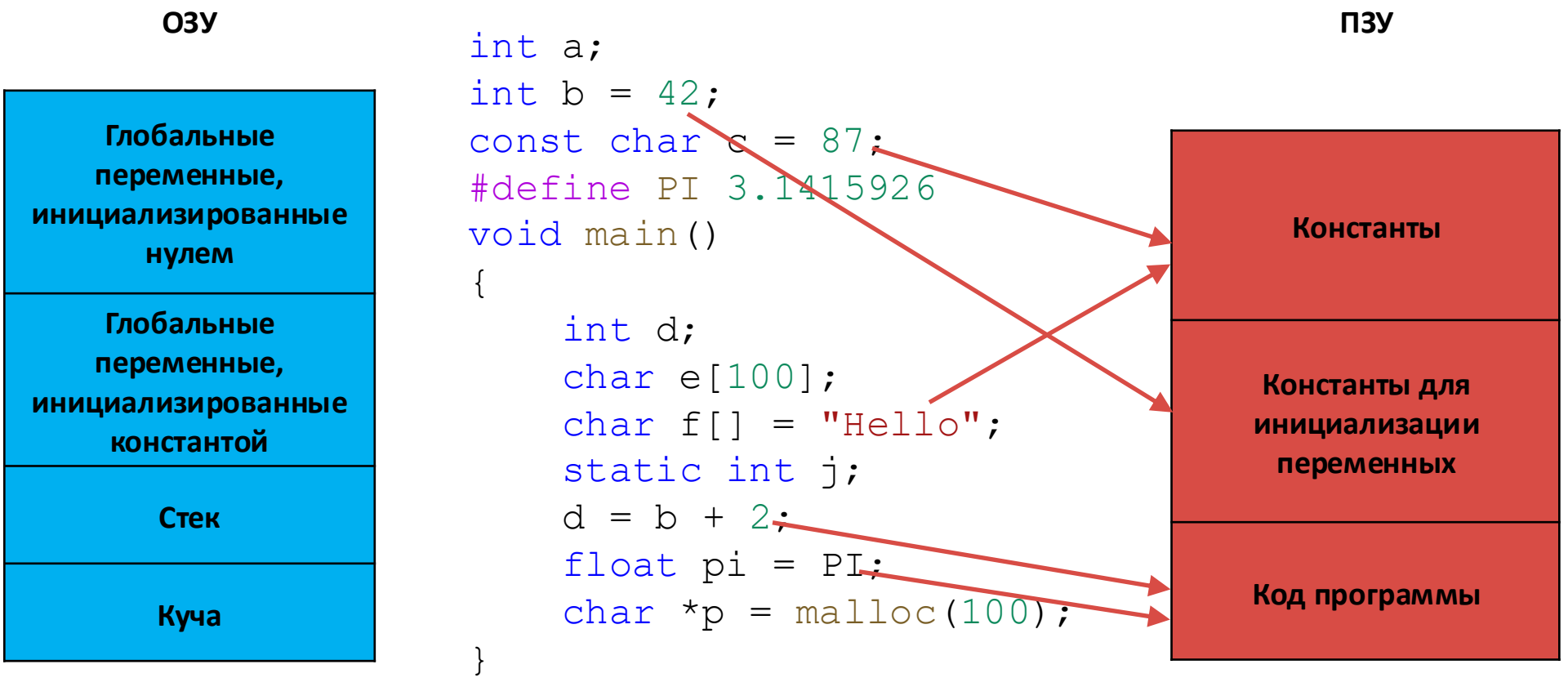
Модель памяти в языке С: константы



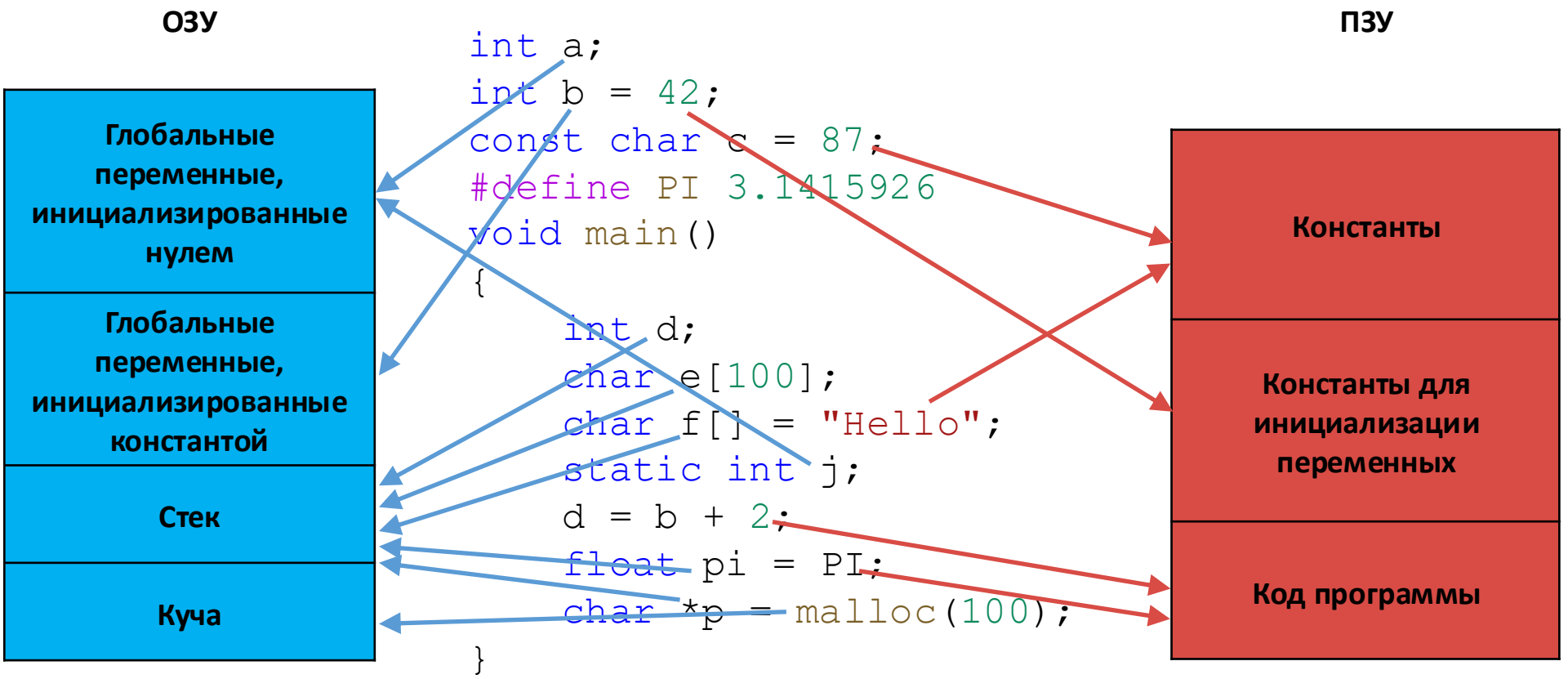
Модель памяти в языке С: литералы


```
int a;
int b = 42;
const char c = 87;
#define PI 3.1415926
void main()
{
    int d;
    char e[100];
    char f[] = "Hello";
    static int j;
    d = b + 2;
    float pi = PI;
    char *p = malloc(100);
}
```

Модель памяти в языке С



Модель памяти в языке С



Применение языка С для встраиваемых систем

Программирование встраиваемых систем – это низкоуровневое программирование. Поэтому язык должен иметь доступ к аппаратному обеспечению на регистровом уровне.

- Обращение к регистрам специальных функций периферийных модулей.
- Стартовый код.
- Обработка прерываний.
- Ассемблерные вставки.
- Стандартная библиотека и системные вызовы.

Уровни абстракции:
Обращение к регистрам специальных функций

1. Язык С + документация

```
#define PORTC *((volatile unsigned int *)(0x400B8000))
...
PORTC = PORTC | 1; /* Установить 1 в PC0 */
```

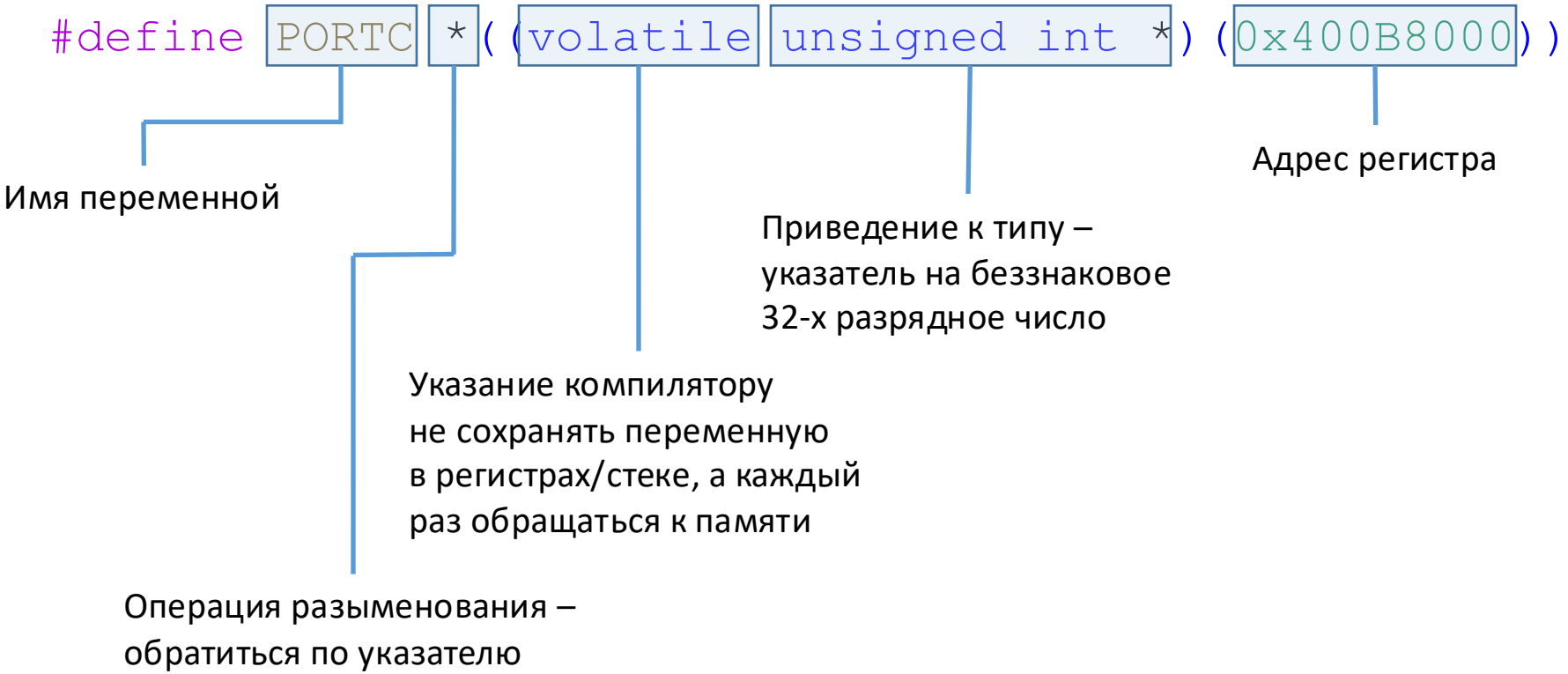
2. Язык С + заголовочные файлы + документация

```
#include "MDR32Fx.h"
...
MDR_PORTC->RXTX = MDR_PORTC->RXTX | 1; /* Установить 1 в PC0 */
```

3. Язык С + библиотека (например, Standard Peripheral Library для К1986ВЕ92QI)

```
#include <MDR32F9Qx_port.h>
...
PORT_SetBits(MDR_PORTC, PORT_Pin_0); /* Установить 1 в PC0 */
```

Уровни абстракции:
Обращение к регистрам специальных функций



Первая программа на С для микроконтроллера. Версия 1

```
#define RST_CLK_PER   (*((volatile unsigned int *)(0x4002001C)))
#define PORTC_RXTX    (*((volatile unsigned int *)(0x400B8000)))
#define PORTC_OE      (*((volatile unsigned int *)(0x400B8004)))
#define PORTC_ANALOG  (*((volatile unsigned int *)(0x400B800C)))
#define PORTC_PWR     (*((volatile unsigned int *)(0x400B8018)))

/* Функция main. Точка входа в программу */
int main(void)
{
    RST_CLK_PER = RST_CLK_PER | (1 << 23); /* Включаем тактирование порта C */

    PORTC_OE = PORTC_OE | 0x01;            /* Настраиваем ножку 0 порта C на вывод */
    PORTC_ANALOG = PORTC_ANALOG | 0x01;    /* Включаем цифровой режим работы ножки 0 */
    PORTC_PWR = PORTC_PWR | 0x02;          /* Настраиваем мощность выходного буфера ножки 0 */

    for (;;)
    {
        PORTC_RXTX = PORTC_RXTX ^ 0x01;   /* Инвертирование бита в регистре PORTC */
        for (volatile int i = 0; i < 100000; i++); /* Программная задержка */
    }
}
```

Первая программа на С для микроконтроллера. Версия 2

```c
#include <MDR32Fx.h>

/* Функция main. Точка входа в программу */
int main(void)
{
    /* Включаем тактирование порта C */
    MDR_RST_CLK->PER_CLOCK = MDR_RST_CLK->PER_CLOCK | (1 << 23);

    MDR_PORTC->OE = MDR_PORTC->OE | 0x01;          /* Настраиваем ножку 0 порта C на вывод */
    MDR_PORTC->ANALOG = MDR_PORTC->ANALOG | 0x01; /* Включаем цифровой режим работы ножки 0 */
    MDR_PORTC->PWR = MDR_PORTC->PWR | 0x02; /* Настраиваем мощность выходного буфера ножки 0 */

    for (;;)
    {
        MDR_PORTC->RXTX = MDR_PORTC->RXTX ^ 0x01; /* Инвертирование бита в регистре PORTC */
        for (volatile int i = 0; i < 100000; i++); /* Программная задержка */
    }
}
```

Первая программа на С для микроконтроллера. Версия 3

```
#include <MDR32Fx.h>
#include <MDR32F9Qx_config.h>
#include <MDR32F9Qx_rst_clk.h>
#include <MDR32F9Qx_port.h>
int main() {
    RST_CLK_PCLKcmd(RST_CLK_PCLK_PORTC, ENABLE);
    PORT_InitTypeDef Port_InitStructure;
    PORT_StructInit(&Port_InitStructure);
    Port_InitStructure.PORT_Pin = PORT_Pin_0;
    Port_InitStructure.PORT_OE = PORT_OE_OUT;
    Port_InitStructure.PORT_FUNC = PORT_FUNC_PORT;
    Port_InitStructure.PORT_SPEED = PORT_SPEED_FAST;
    Port_InitStructure.PORT_MODE = PORT_MODE_DIGITAL;
    PORT_Init(MDR_PORTC, &Port_InitStructure);
    for(;;) {
        for (volatile int i = 0; i < 100000; i++);
        PORT_SetBits(MDR_PORTC, PORT_Pin_0);
        for (volatile int i = 0; i < 100000; i++);
        PORT_ResetBits(MDR_PORTC, PORT_Pin_0);
    }
}
```

Стартовый код

Стартовый код (startup) обеспечивает корректный запуск программы. Без инициализации ресурсов система может быть неправильно настроена.

Основные функции стартового кода:

1. Инициализация аппаратных ресурсов: настройка тактовых генераторов (как правило, только частоты работы процессора), настройка стека (Stack Pointer), инициализация памяти (копирование данных из секции .data в ОЗУ и очистка секции .bss (секции для неинициализированных данных)).
2. Установка обработчиков прерываний.
3. Переход на точку входа в программу: передача управления функции `main()`.
4. Обработка аппаратных исключений: в случае возникновения аппаратных исключений (например, попытка доступа к неинициализированной памяти), startup код может содержать базовый обработчик для их отлова и корректной обработки.
5. Поддержка отладочных функций: включение/отключение отладочных интерфейсов и функций.

В операционных системах общего назначения (Linux, Windows, macOS) эти действия производит сама операционная система перед запуском любой программы.

Обработка прерываний в Cortex-M3

Файл startup_MDR32F9Qx.s:

```
AREA RESET, DATA, READONLY
__Vectors DCD __initial_sp ; Top of Stack
DCD Reset_Handler ; Reset Handler
DCD NMI_Handler ; NMI Handler
...
DCD Timer1_IRQHandler ; IRQ14
DCD ADC_IRQHandler ; IRQ17
...
```

Директива DCD (Define Constant Data) в ассемблере ARM используется для определения константных данных и размещения их в памяти.

Файлы разработчика, например, main.c:

```
/* Обработка прерывания по Timer1 */
void Timer1_IRQHandler(void) {
...
}
/* Обработка прерывания по ADC */
void ADC_IRQHandler(void) {
...
}
```

Ассемблерные вставки

Использование ассемблера:

- Для оптимизации по скорости выполнения и размеру программы.
- Для прямого манипулирования регистрами.
- Для использования старого ассемблерного кода в новых проектах.
- Для специальных инструкций (`WFI`, `BKP`, `SVC`).
- Для учебных целей.

Ассемблерные вставки

```
/* Для IDE Keil uVision */
__asm void add(int x1, int x2, int x3)
{
    ADDS R0, R0, R1
    ADDS R0, R0, R2
    BX LR
}

int swap32(int i)
{
    int res;
    __asm {
        REV res, i
    }
    return res;
}

__asm("WFI"); /* Выполнение одной команды */
```

Стандартная библиотека и системные вызовы

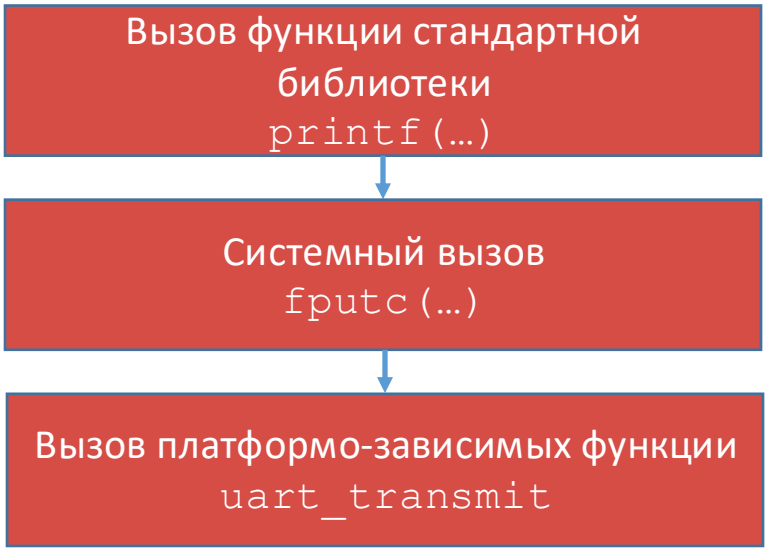
Для встраиваемых систем существует несколько реализаций стандартной библиотеки C, которые разработаны с учетом ограниченных ресурсов этих систем. Например, Newlib и mbed C library (для RTOS и bare-metal), uClibc-ng и musl (для Embedded Linux).

Для библиотеки newlibc (Newlib), системные вызовы (sys calls) представляют собой интерфейс между библиотекой и целевой системой. Это низкоуровневые функции для выполнения операций, которые требуют взаимодействия с операционной системы, файловой системой, драйверами периферии и так далее. Это работа с файлами, ввод/вывод, управление процессами и т. д.

Системные вызовы в newlibc могут быть реализованы по-разному в зависимости от целевой платформы.

Основные системные вызовы включают в себя такие операции, как:

- `read` и `write` — для чтения и записи данных в файловые дескрипторы;
- `open` и `close` — для открытия и закрытия файлов;
- и так далее.



Перенаправление потоков ввода/вывода

Стандартная библиотека содержит функции ввода/вывода:
`printf`, `scanf`, `fopen`, `fprintf`, `fwrite`, `fread`, `fclose` и другие.

Функции `printf`, `scanf` используют дескрипторы по умолчанию:
`stdin` – стандартный ввод, `stdout` – стандартный вывод, `stderr` – поток ошибок.

На устройствах без ОС если только требуется печать и чтение из последовательного интерфейса, то достаточно переопределить функции `fputc` и `fgetc` для библиотеки `Newlibc`.

```
int fputc(int c, FILE * stream)
{
  uart_write(c);
  return c;
}


int fgetc(FILE * stream)
{
  char c = uart_read();
  return c;
}
```

Для разных компиляторов и стандартных библиотек порядок перенаправления может отличаться.

Заключение

- Понимание модели памяти языка С позволяет писать оптимальный код.
- Язык С позволяет напрямую обращаться к памяти через указатели, но не позволяет обращаться к регистрам процессора.
- Применение для встраиваемых систем:
  - Обращение к регистрам специальных функций через указатели.
  - Стартовый код.
  - Обработка прерываний.
  - Ассемблерные вставки.
  - Системные вызовы.