Углубляемся в С

Хохлов С.А.

МГТУ им. Н.Э. Баумана

23 сентября 2025 г.

Вызов функций в С

Вызов функции в С

- Далее рассмотрим типичный порядок вызова функции в С на обычной архитектуре (например ARM Cortex-M с соглашением ARM EABI / AAPCS).
- Детали могут отличаться на x86, RISC-V и т.п., но общая идея одинакова.

Алгоритм вызова функций в C

1. (caller) Подготовить аргументы функции
2. (caller) Сохранить регистры (опционально), вызвать функцию и сохранить адреса возврата
3. (callee) Пролог: сохранить регистры, выделить кадр стека для локальных переменных, разместить локальные переменные
4. (callee) Выполнить тело функции
5. (callee) Вернуть значения: через регистры и/или по указателю
6. (callee) Эпилог функции: освободить стек, восстановить значения регистров, перейти по адресу возврата
7. (caller) Восстановить регистры (опционально), продолжить выполнение

1. Подготовка аргументов функции

- Первые 4 аргумента функции (32-битные) передаются через регистры r0–r3.
- Остальные — в стек (вызывающий кладёт их сверху стека).
- Если нужны выравнивания, компилятор добавляет «пустые» слова.

2. Вызов

- Вызывающий сохраняет необходимые свои регистры (caller-saved, например r0–r3, r12, lr) если хочет их использовать после вызова.
- Инструкция BL func (branch-and-link) вызывает func и записывает в LR (link register) адрес возврата.

3. Пролог вызываемой функции

- Сохраняются регистры, которые обязана сохранить вызываемая сторона (обычно r4–r11, lr если будет использоваться).
- Уменьшается указатель стека (SP) — выделяется «кадр стека» для локальных переменных.
- Локальные переменные размещаются внутри этого кадра.

4. Тело функции

- Работа с параметрами: те, что не поместились в регистры, читаются из стека.
- Локальные переменные находятся по смещению от текущего SP или FP (frame pointer, если используется).

5. Возврат значения

- Возвращаемое значение:
  - скаляры до 32 бит — в r0 (для 64-бит — в r0/r1),
  - большие структуры — по указателю, который передал вызывающий.

6. Эпилог функции

- Освобождение кадра стека: SP возвращается на прежнее значение.
- Восстановление сохранённых регистров (включая LR).
- Инструкция BX lr или POP {..., pc} — переход обратно.

7. Возобновление вызывающей функции

- Вызывающий восстанавливает свои регистры (если сохранял) и продолжает выполнение.

Важные моменты:

- Стек растёт вниз (адрес уменьшается).
- Всё выравнивается обычно на 4- или 8-байтные границы.
- На x86 общая схема та же, но все параметры обычно кладутся в стек (в System V ABI — часть может идти в регистры).

Алгоритм вызова функции (еще раз)

1. (caller) Подготовить аргументы функции
2. (caller) Сохранить регистры (опционально), вызвать функцию и сохранить адреса возврата
3. (callee) Пролог: сохранить регистры, выделить кадр стека для локальных переменных, разместить локальные переменные
4. (callee) Выполнить тело функции
5. (callee) Вернуть значения: через регистры и/или по указателю
6. (callee) Эпилог функции: освободить стек, восстановить значения регистров, перейти по адресу возврата
7. (caller) Восстановить регистры (опционально), продолжить выполнение

Обработка прерываний

Обработка прерывания на ARM Cortex-M

Основное отличие прерывания от обычного вызова — автоматическое сохранение контекста ядром. Для Cortex-M (ARMv7-M/ARMv6-M) кратко:

Алгоритм обработки прерывания

1. (core) Пролог: сохранить регистры в стек, в том числе адрес возврата
2. (core) Загрузить адрес обработчика в PC (запустить обработчик)
3. (ISR) Пролог: сохранить используемые далее регистры, выделить стек для локальных переменных
4. (ISR) Выполнить тело обработчика
5. (ISR) Эпилог: восстановить сохраненные регистры
6. (core) Перейти по адресу возврата, восстановить регистры

1. Автоматический пролог (аппаратно)

- Процессор сам делает PUSH в текущий стек (MSP или PSP): xPSR, PC, LR, R12, R3..R0.
- Переключается на стек MSP, если вход был с PSP.

2. Вход в обработчик

- Ядро загружает адрес обработчика из векторной таблицы в PC.
- Включает маску прерываний нужного уровня.

3. Пролог обработчика (по желанию компилятора)

- Компилятор сохраняет регистры R4–R11, если они используются.
- Может выделить кадр стека для локальных переменных.

4. Работа обработчика

- Код ISR.

5. Эпилог обработчика

- Восстановление сохранённых регистров R4–R11.
- Инструкция BX lr или POP {...,PC} не нужна напрямую.

6. Автоматический возврат (аппаратно)

- При выходе из ISR процессор выполняет инструкцию BX LR с особым значением LR (EXC_RETURN),
- сам POP-ит ранее сохранённые R0–R3,R12,LR,PC,xPSR,
- возвращает управление прерванному коду.

Различия с вызовом функции

- Контекст (R0–R3, R12, LR, PC, xPSR) сохраняется/восстанавливается автоматически.
- Стек — всегда системный (обычно MSP).
- Векторная таблица определяет точку входа, нет инструкции BL.

Алгоритм обработки прерывания (еще раз)

1. (core) Пролог: сохранить регистры в стек, в том числе адрес возврата
2. (core) Загрузить адрес обработчика в PC (запустить обработчик)
3. (ISR) Пролог: сохранить используемые далее регистры, выделить стек для локальных переменных
4. (ISR) Выполнить тело обработчика
5. (ISR) Эпилог: восстановить сохраненные регистры
6. (core) Перейти по адресу возврата, восстановить регистры

Препроцессор в С

Препроцессор в С

Все директивы препроцессора обрабатываются до компиляции, т.е. на этапе препроцессинга, когда формируется окончательный исходный код для компилятора.

1. Условная компиляция

Директива	Назначение
#define NAME [value]	Определение макроса
#undef NAME	Удаление макроса
#ifdef NAME	Проверка, определён ли макрос
#ifndef NAME	Проверка, не определён ли макрос
#if <expr>	Компиляция блока, если выражение истинно
#elif <expr>	Иначе если (аналог else if)
#else	Альтернатива при ложном условии
#endif	Конец условного блока

2. Подключение файлов

Директива	Назначение
#include <file>	Подключение системного заголовочного файла
#include "file"	Подключение своего файла (по пути проекта)

3. Макросы и замена текста

Директива	Назначение
#define NAME [value]	Простое определение константы или фрагмента кода
#define FUNC(x) (x)*(x))	Макрос-функция с аргументами
#undef NAME	Удаление определения макроса

4. Прочие

Директива	Назначение
#error "text"	Генерация ошибки компиляции с сообщением
#warning "text"	Предупреждение (GCC/Clang)
#pragma ...	Специфичные для компилятора инструкции (например #pragma once)
#line <num> ["file"]	Изменяет номер строки и имя файла для сообщений компилятора/отладки

Структуры в С

Определение I

- Структура — это тип данных, включающий в себя набор значений различных типов.
- Порядок размещения значений в памяти задаётся при определении типа и сохраняется на протяжении времени жизни объектов. таким образом есть возможность доступа к полям, например, через указатели.

Объявление типа I

[1]

Общая форма объявления структуры

```
struct ИмяСтруктуры
{
 тип ИмяЭлемента1;
 тип ИмяЭлемента2;
 . . .
 тип ИмяЭлементаn;
};
```

Объявление типа II

Пример объявления структуры

```
1 struct date
2 {
3   int day;      // 4 байта
4   char *month; // 4 байта
5   int year;     // 4 байта
6 };
```

Объявление типа III

При объявлении структур, их разрешается вкладывать одну в другую:

```
1 struct person
2 {
3   char lastname[20];   // фамилия
4   char firstname[20]; // имя
5   struct date bd;      // дата рождения
6 };
```

Определение и инициализация полей структуры I

Определение (definition) экземпляра структуры

```
struct date bd;
```

Определение и инициализация полей структуры II

Инициализация

```
1  struct ИмяСтруктуры ИмяПеременной=
2          {
3              ЗначениеЭлемента1,
4              ЗначениеЭлемента2,
5              . . . ,
6              ЗначениеЭлементаn
7          };
```

- Пример: struct date bd={8,"июня", 1978};

Определение и инициализация полей структуры III

- Имя элемента структуры является составным. Для обращения к элементу структуры нужно указать имя структуры и имя самого элемента. Они разделяются точкой:

Обращение к элементу структуры

```
ИмяПеременной.ИмяЭлементаСтруктуры
printf("%d %s %d",bd.day, bd.month, bd.year);
```

Определение и инициализация полей структуры IV

- Имя структурной переменной может быть указано при объявлении структуры. В этом случае оно размещается после закрывающей фигурной скобки }. Область видимости такой структурной переменной будет определяться местом описания структуры.

Объявление переменной при объявлении структуры

```
1  struct complex_type  // имя структуры
2  {
3    double real;
4    double imag;
5  } number;     // имя структурной переменной
```

Битовые поля I

- Используя структуры, можно упаковать целочисленные компоненты еще более плотно, чем это было сделано с использованием массива.
- Набор разрядов целого числа можно разбить на битовые поля, каждое из которых выделяется для определенной переменной. При работе с битовыми полями количество битов, выделяемое для хранения каждого поля отделяется от имени двоеточием
- тип имя: КоличествоБит
- При работе с битовыми полями нужно внимательно следить за тем, чтобы значение переменной не потребовало памяти больше, чем под неё выделено.

Битовые поля II

Структура с битовыми полями

```
1 struct date
2 {
3   unsigned short day : 5;
4   unsigned short month : 4;
5   unsigned short year : 7;
6 };
```

Битовые поля III

```
1  #include <stdio.h>
2  struct {
3      unsigned int age : 3;
4  } Age;
5
6  int main() {
7
8      Age.age = 4;
9      printf("Sizeof(Age): %d\n", sizeof(Age));
10     printf("Age.age: %d\n", Age.age);
11
12     Age.age = 7;
13     printf("Age.age : %d\n", Age.age);
```

Битовые поля IV

```
14
15      Age.age = 8;
16      printf("Age.age : %d\n", Age.age);
17
18      return 0;
19  }
```

Output

```
Sizeof(Age): 4
Age.age: 4
Age.age: 7
Age.age: 0
```

Указатели на структуры I

- Доступ к элементам структуры или объединения можно осуществить с помощью указателей.
- Для этого необходимо инициализировать указатель адресом структуры или объединения.
- Для организации работы с массивом можно использовать указатель. При этом обращение к полям структуры через указатель будет выглядеть как:
- указатель->поле или
- (*указатель).поле
  - указатель — указатель на структуру или объединение;
  - поле — поле структуры или объединения;

Битовые операции

Полезные приёмы работы с битовыми операциями в C I

- список битовых операций в C
- как проверить, установлен ли бит в переменной или в регистре на позиции X
- как очистить бит в позиции X переменной или регистра
- как установить бит в позиции X переменной или регистра
- как переключить бит в позиции X переменной или регистра
- как умножить и делить на 2^N

Битовые операции в C I

Оператор	Описание
&	побитовое И
|	побитовое ИЛИ
^	побитовое ИСКЛЮЧ. ИЛИ
~	побитовая инверсия
>>	сдвиг вправо
<<	сдвиг влево

Битовые операции в С II

X	Y	X & Y	X | Y	X ^ Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Битовые операции в C III

Проверить, установлен ли бит в переменной или в регистре на позиции X

(Variable & (1<< X))

```
if(var & (1 << 2))
    printf("Bit is set");
else
    printf("Bit is not set");
```

Битовые операции в C IV

Очистить бит в позиции X переменной или регистра

```
Register &= ~ (1 << X)
```

```
GPIOA &= ~(1 << 2);      // очистить второй бит GPIO PORTA
```

Битовые операции в C V

Установить бит в позиции X переменной или регистра

Register |= (1 << X)

```
GPIOA |= (1 << 5);      // Установить 5й бит GPIO PORTA
```

Битовые операции в C VI

Переключить бит в позиции X переменной или регистра

```
Register ^= (1 << X)
```

```
GPIOA ^= (1<< 3);        // Переключить 3й бит PORTA
```

Битовые операции в C VII

Умножение и деление на 2^N

- [Число >> N] // 2^N = делитель
- [Число << N] // 2^N = множитель

[1] Елена Вставская. Сложные типы данных в Си : структуры, объединения, битовые поля. URL: https://prog-cpp.ru/c-struct/ (дата обр. 30.09.2024).