**Microcontroller\_Answers**

**MC\_01\_p.26**

1. 0x0804DA 16 = 0000.1000.0000.0100.1101.1010 2
2. 0000 -> every digit is 1 bit -> 4\*1=4 -> we have 6 four-digit numbers -> 6\*4=24 bits
3. 2^24 \* 1 Byte = 16 777 216 Bytes : 1024 = 16384 KB : 1024 = 16 MB

**MC\_01\_p.31(30)**

**Первоначальный вариант с Андреем:**

1. **ROM (ПЗУ)** – read-only memory (постоянно-запоминающее устройство)

0х00000000 = 0 - min

0x1FFFFFFF = 536870911 - max номер адреса

1\*16^7 + 15\*16^6 + 15\*16^5 + 15\*16^4 + 15\*16^3 + 15\*16^2 + 15\*16^1 + 15\*16^0 =

= 536870911

max-min+1=536870912 количество адресов

Это подобно ссылкам. У тебя есть 536870912 ссылок. А по каждой ссылке можно получить некоторую область памяти.

Для того, чтобы узнать сколько памяти. Нужно знать размер объекта по одной ссылке.

536870912/8 = 67108864 Bytes :1024 = 65536 KB : 1024 = 64 MB

Мы рассмотрели для 1 битной шины)))

**RAM (ЗУПД)** – random-access memory (запоминающее устройство с производным доступом)

0х20000000 = 2\*16^7 = 536870912

0x3FFFFFFF = 1073741823

max-min+1= 536870912 количество адрессов

**Peripheral Registers**

0x40000000 = 4\*16^7 = 1073741824

0xFFFFFFFF = 4294967295

max-min+1= 3221225472количество адрессов

64+64+384 = 512 MB

**Вариант из лекции:**

1. “Old” PC Systems: 32-bit address bus -> 2^2 \* 2^10 \* 2^10 \* 2^10 = 4 GB

2^32 \* 1 Byte = 2^2 \* 2^10 \* 2^10 \* 2^10 = 4 GB

1. 0x200…00 – 0x00…00 = 2\*16^7 = 2\* ((2^4)^7) = 2\* 2^28 = 2^8 \* 2^10 \* 2^10 = 512 MB

0x400…00 – 0x20…00 = 0x20…00 = 512 MB

1. 4 GB – 1 GB = 3 GB
2. 0x1 0000 0000 – 0x4000 0000 = 0xC000 0000 = ….. = 3 GB
3. 0x4…. 0x10 = 0xC -> 0xC/0x10 = 12/16 = 3/4 \* 4 GB = 3 GB

**MC\_01\_p.37(33)**

1. **µC** – controls and processes

**µProcessor** – just processes (and it’s included into µC)

**CPU** – central processing unit

1. storage shelf = data processor

worker = general purpose registers

boss = control unit

loading bay = bus interface

1. What are the steps of a machine instruction execution?
2. The fetch of a machine instruction from memory involves all 3 busses. What is

their task?

address bus: indentify address

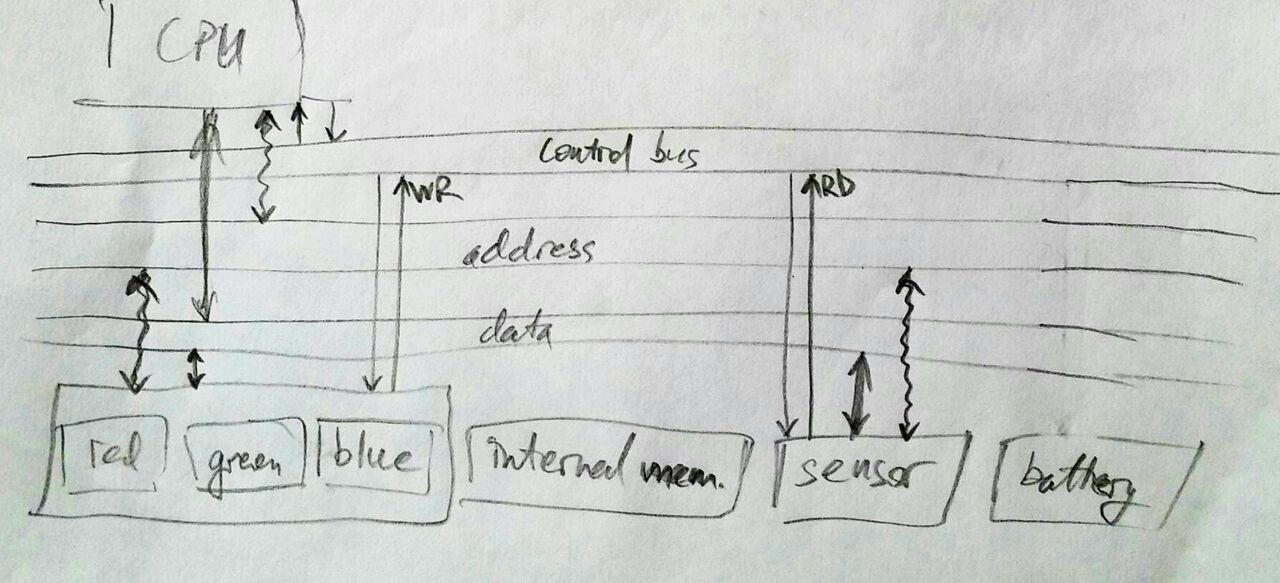
control bus: controls direction, commands to write/read, pause/continue, stop/interrupt, whatever

data bus: transferring data (basically, a channel)

1. Given : A microcontroller has a 16-bit address bus and an 8-bit data bus. What is the maximum addressable memory if the smallest unit is 1 byte?

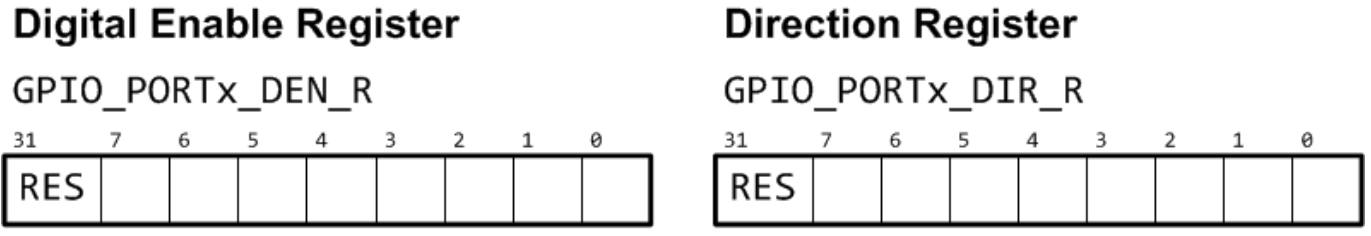
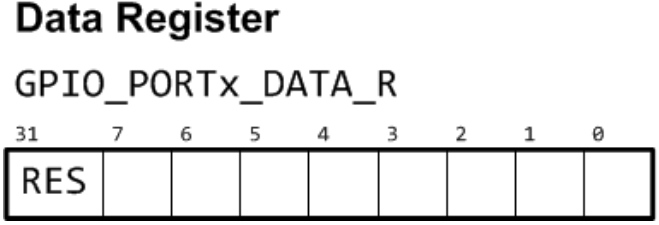
2^16= 65536 Bytes = 64 KB

1. You are designing a battery powered, microcontroller controlled digital temperature sensor module which reads a temperature sensor and lights an RGB-LED with red (too cold), green (convenient), blue (too cold). If you have to choose a microcontroller for this task, which performance parameters would you regard as important?



**MC\_02\_p.35(34)**

PJ(3:0) shall be configured as outputs, PJ(7,5) as inputs. The other pins shall be three-stated. Program the basic GPIO registers.

Digital Enable Register

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| RES | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |

<- given just 7,5,3-1 and they are switched on  
Direction Register

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| RES | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

<- input = 0 (->), output = 1(<-)  
Data Register

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| RES | ? | ? | ? | ? | ? | ? | ? | ? |

? – no given data

**receive clock = (20 MHz / 65) / 16 = 19230.77 Hz (=19231 kbit/s)**

**MC\_02\_p.48(44)**

1. Give the data directions (CPU -> PM or PM -> CPU) of the following data types:

Status: RDY

Control: RUN

Config: CH2…Ch0

Data: CD7…CD0

1. Pins(ships) are inserted into ports (harbours).

Pins:

* metal leads (pads) that connect µC electrically with environment (= external

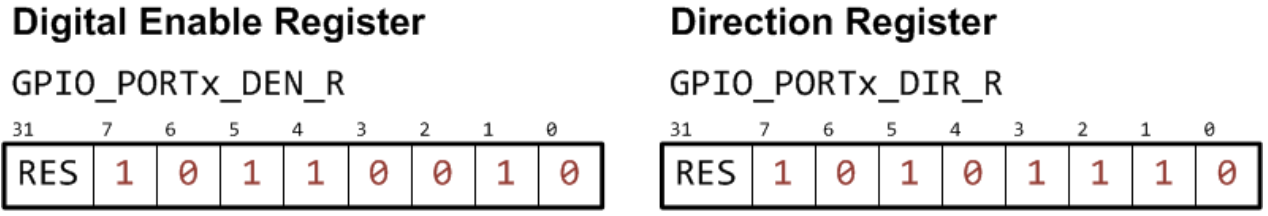
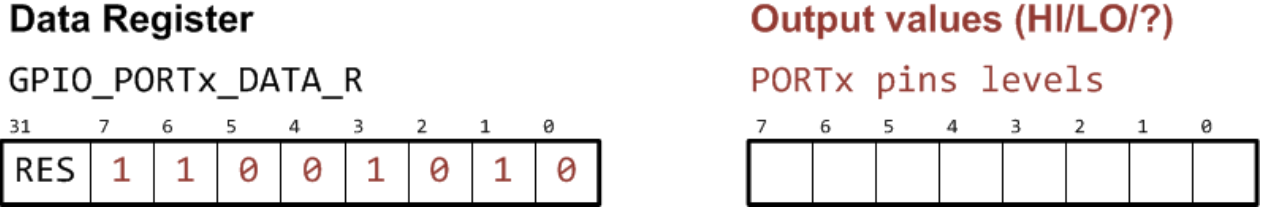
circuitry)

* input/output signals of µC run over pins
* dedicated pins for voltage supply, ground, crystal/oscillator, reset

Ports:

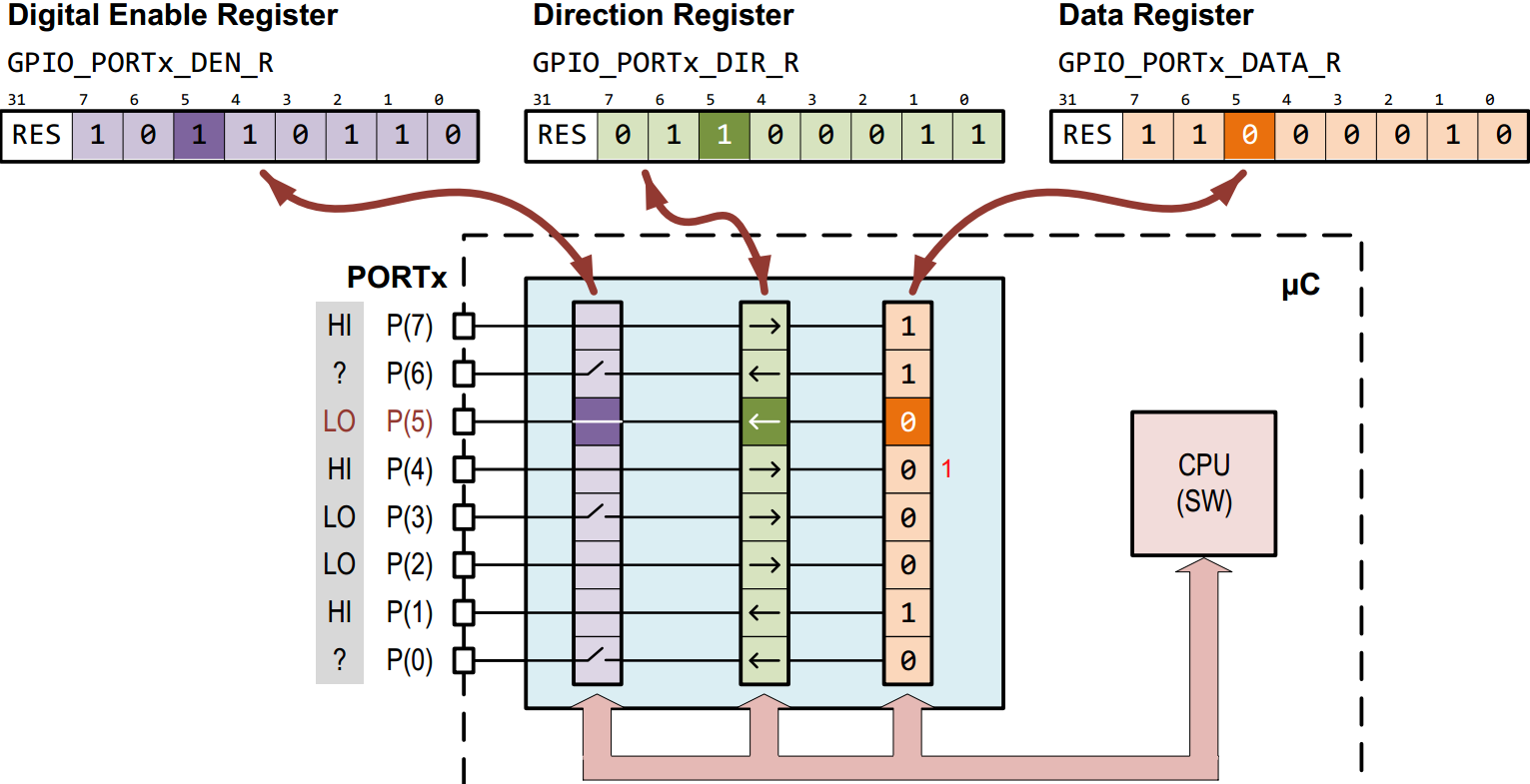
* General Purpose Input/Output ports (=“harbour”, “Hafen”) convert between external digital voltage levels (high, low) and internal binary values (1,0)
* GPIO ports can change direction, be three-stated or adjusted w.r.t. load/drive behaviour (pull-up/pull-down resistor, variable drive)
* a port typically aggregates 4, 8 or more pins for efficient parallel access by SW

1. Given the following register values, what are the output values (HI,LO,?) of PORTx (assumption: port pins are unconnected)?

Output Values

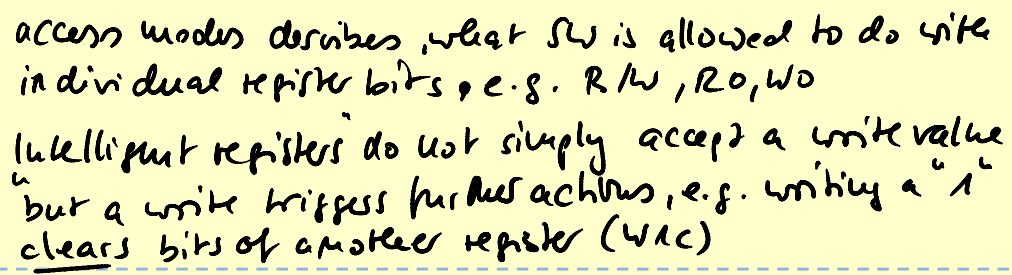
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| HI | HI | LO | LO | ? | ? | HI | LO |



1. **What are register access modes? What is an “intelligent register”?**

Register access modes: (modes=types?)

Intelligent registers: R/WC, R/W1C



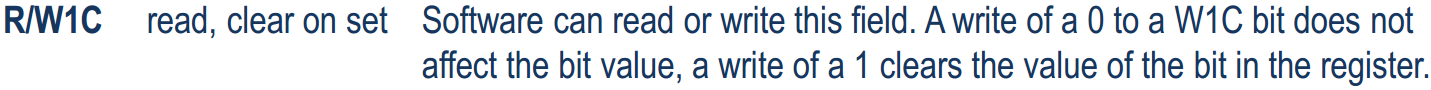
1. A register contains 01001110 and all bits are of type R/W1C. What is the contents of the register after writing 01101001 into abovementioned 01001110?

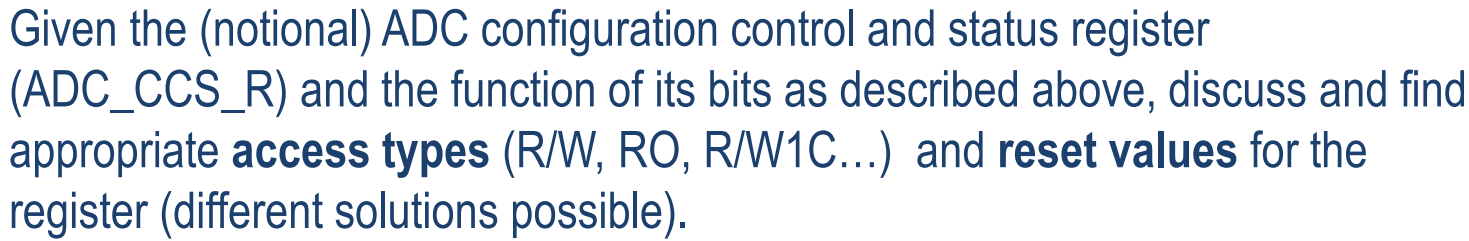
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

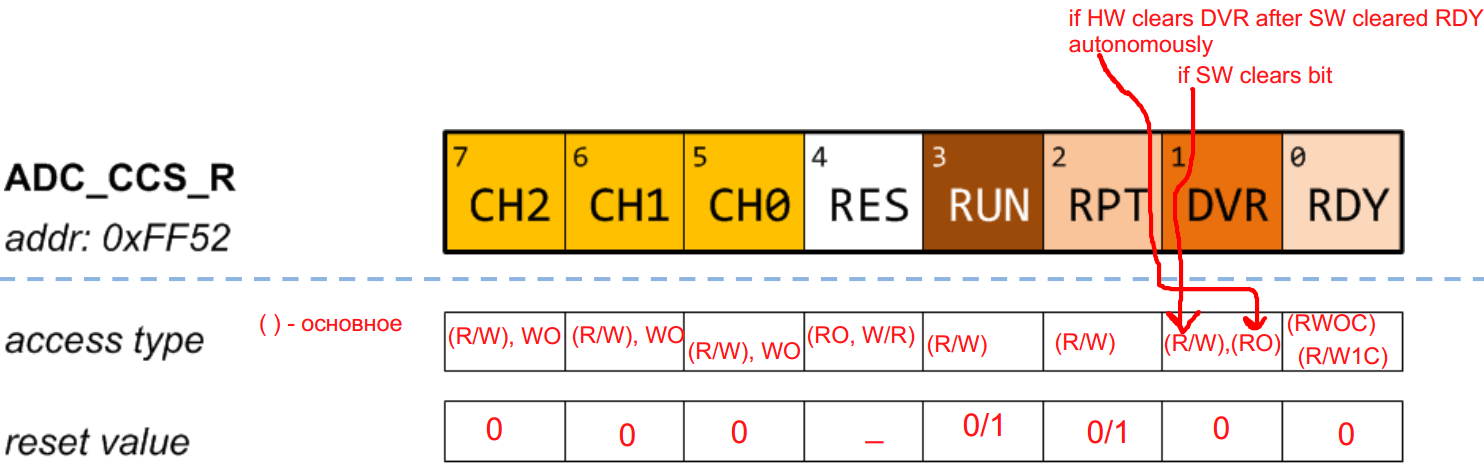
given

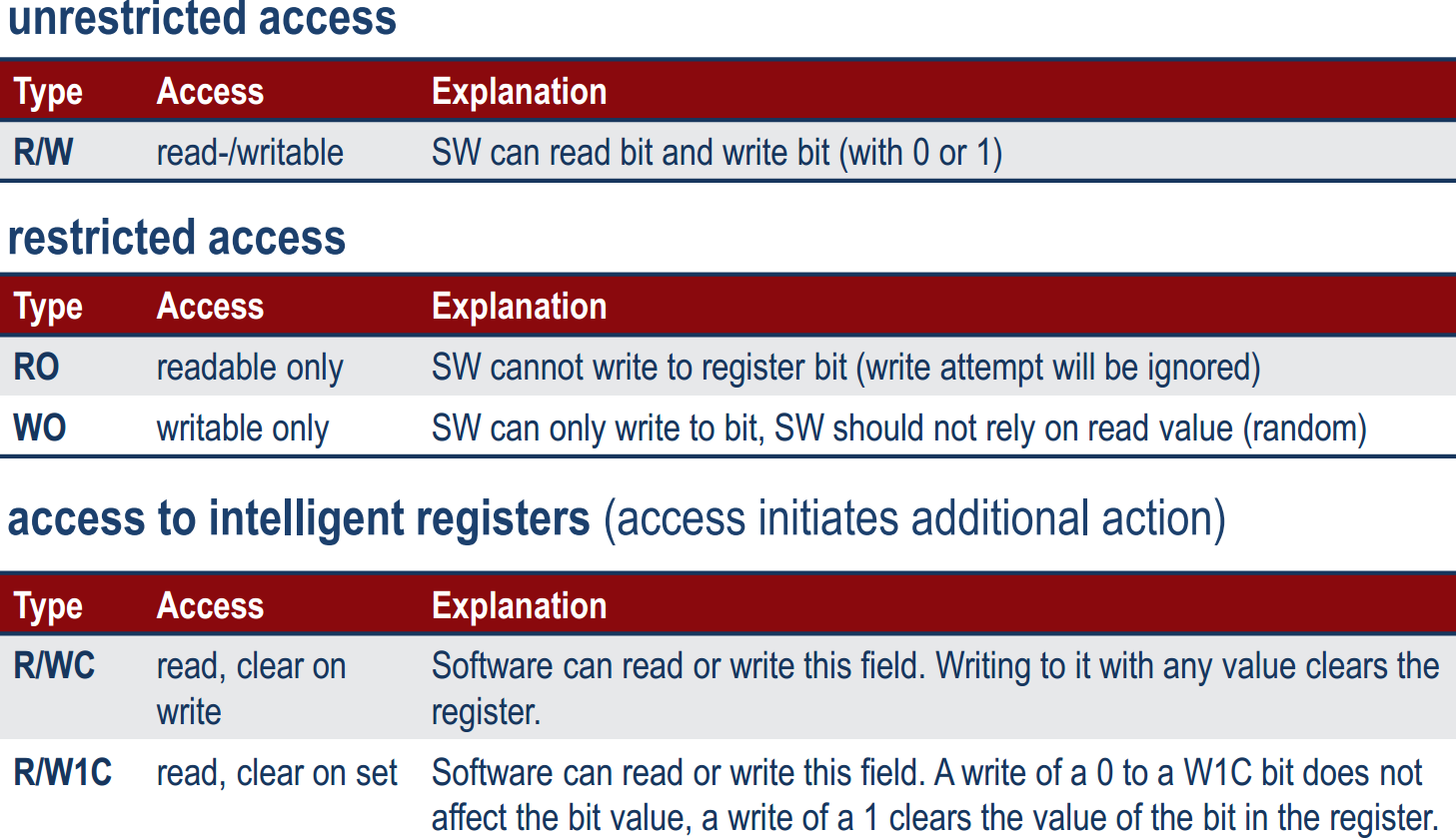
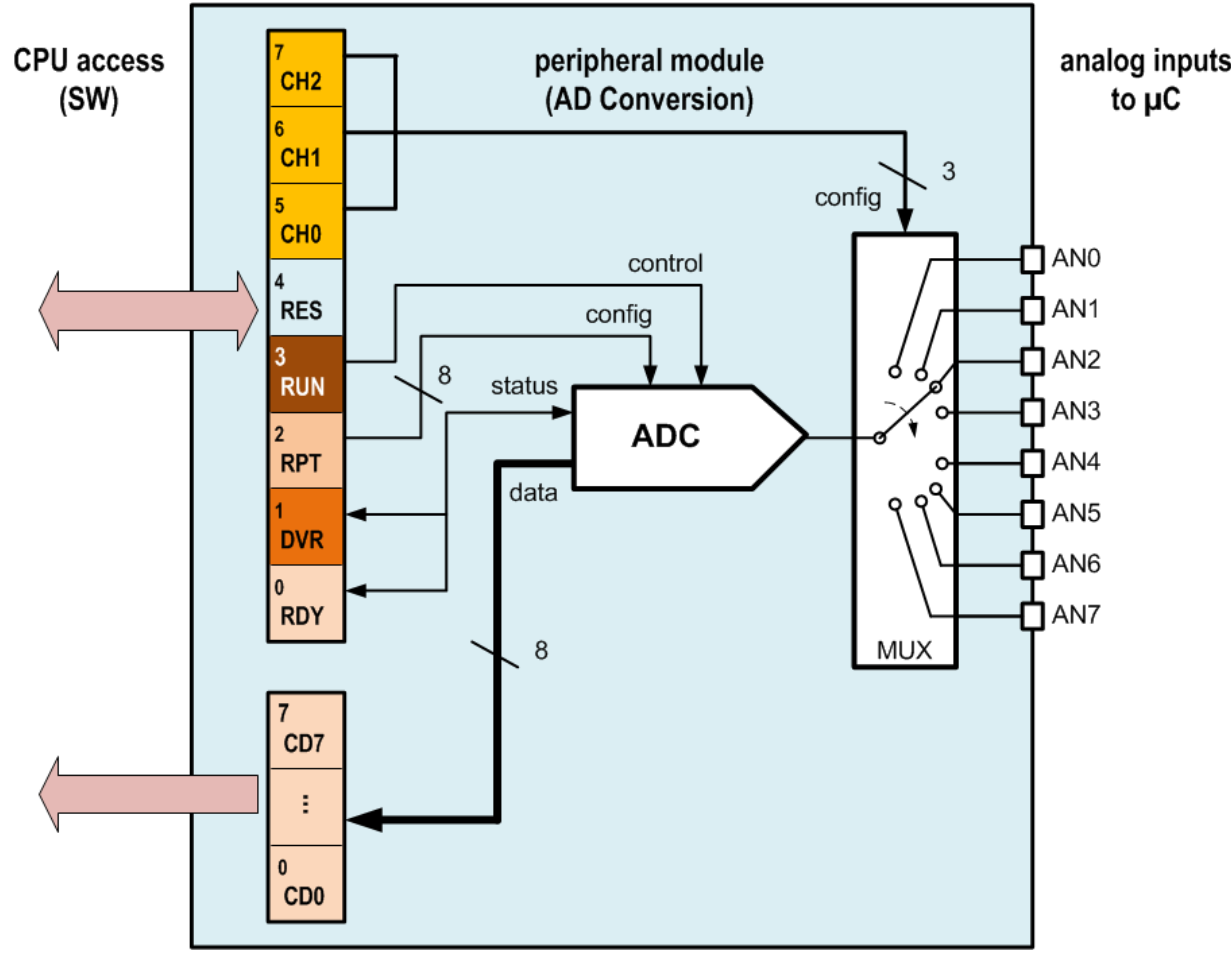
write

result (check the R/W1C explanation below)

**MC\_02 p.13(13)**







Internship:

1. Any cases of internship that resulted into “not finished”? What was the matter?
2. Are there any general rules of when choosing an internship, meaning “acceptable or not acceptable kind of jobs from the viewpoint of HAW”?
3. Are there any nuances with getting an internship for hearing impaired?

**MC\_03 p.13(13)**

You want to build an customized garden shed (you don't like prefabricated self-assembly kits). How do you proceed?

Answer:

* think: how big, design, budget, what for, location (requirements)
* blue print: construction (elements), design, exact dimensions, material ()
* resources: tools, raw materials (wood, sheet, metal, )
* schedule
* purchase the material
* plan work step and realize

**MC\_03 p.22(22)**

int val = 0;

for(int i= 0; i < 100; ++i)

{

if (val % 3==0)

printf(“Fizz”);

else if (val % 5 == 0)

printf(“Buzz”);

else if ( (val % 3 == 0) && (val % 5 == 0))

printf(“FizzBuzz”);

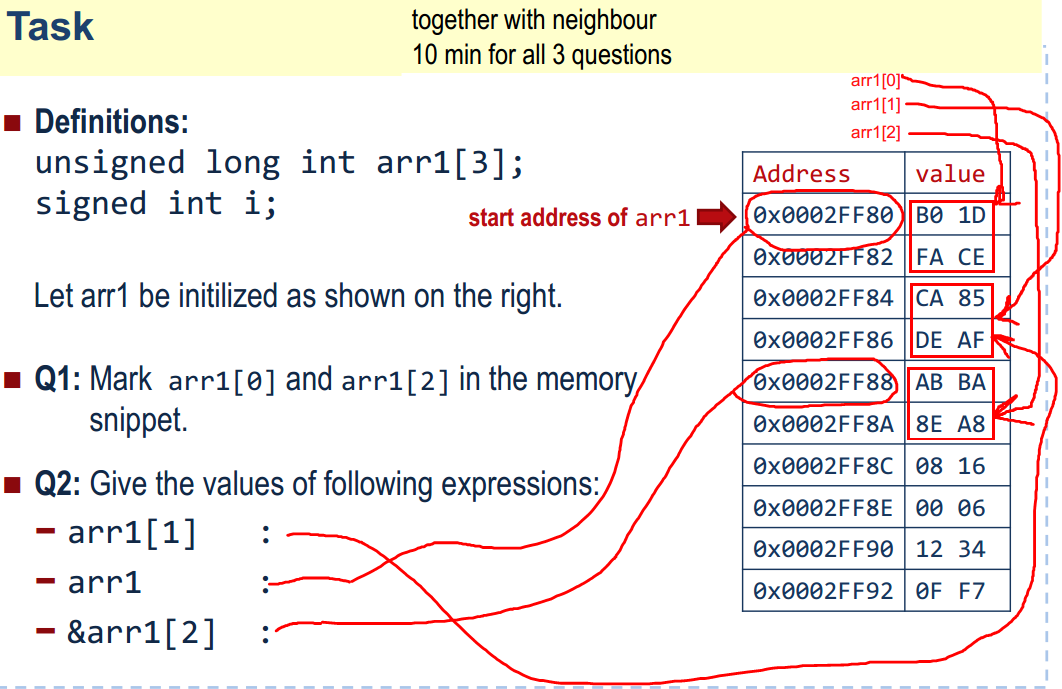
else

printf(“%d”, val);

++val;

}

**MC\_03 p.31(30)**



**По индианам**

0x02FF80

0x02FF84

0x02FF88

Это адреса ячеек памяти. Это просто отображение кусочка оперативной памяти.

Так как между ними 4 байта, то в каждой строке по 4 ячейки. Каждая ячейка это 1 байт.

Запомни, что когда используются 16битные числа (с буквами) то два символа = 1 байт

short int a = 20000; // 0x4E20 - это 16ричное представление числа 20000, чтобы понимать какой байт в какую ячейку попадает

long int b = 300000; // 0x000493E0 - это 16 ричное представление числа 300000

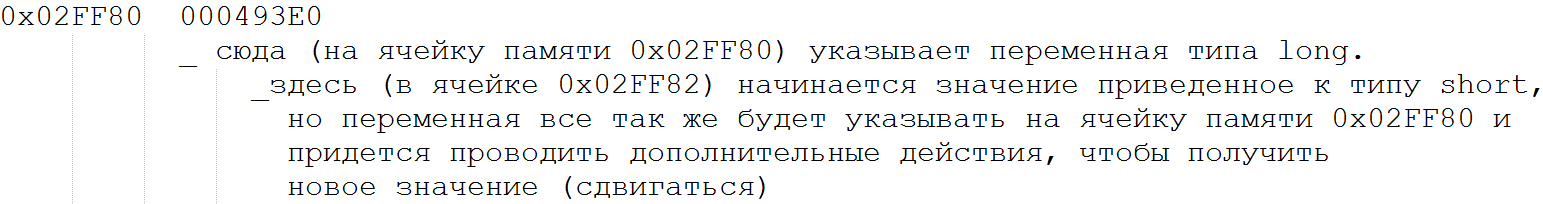
тип short int занимает 2 байта (ячейки).

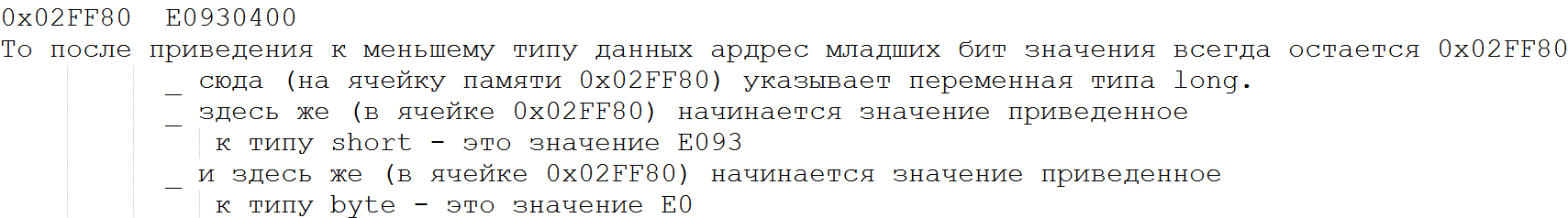
тип long int (в С++ можно сокращать до long, но не знаю, можно ли в Си) занимает 4 байта (ячейки).

**Little endian:** Advantage: variable type (size) can be changed without changing start address of variable)

Здесь о том, что если первыми в памяти идут старшие биты числа, как нам представлять проще, то при приведении числа к меньшему типу данных (long => short) происходит обрезание старших бит, чтобы отобразить такое значение

Было 000493E0, остается 93E0. Сама переменная ссылается на тот же самый адрес памяти.

Если же используется little endian



Intel x86 - это собирательное имя для всех процессоров, которые используются в персональных компьютерах

Intel 64bit обратно совместим с Intel x86. Следовательно на них так же little endian

**Кратенько по массивам**

Когда ты создаешь массив, ты обязан указать сколько элементов в нем будет хранится. Таким образом программа понимает сколько байт памяти нужно выделить под массив.

Имя массива всегда указывает на самое начало блока памяти выделенного под массив. В языке Си - это по сути указатель.

Мы позднее увидим, что все что умеют массивы, можно совершить при помощи указателей. Если же мы указываем после имени массива квадратные скобки - в них говорится, на сколько элементов от начала массива нужно сместится вправо(вниз). На самом деле направление зависит от того, как нарисована память, но обычно ее рисуют сверху вниз и слева направо (таблица).

Address value

0x0002FF80 B0 1D

0x0002FF82 FA CE

0x0002FF84 CA 85

0x0002FF86 DE AF

0x0002FF88 AB BA

0x0002FF8A 8E A8

В этом представлении нужно смотреть на адреса, разница между ними говорит сколько байт в строке 0x0002FF82 - 0x0002FF80 = 2 байта

B0 - первый байт 1D - второй байт

У нас число unsigned long int, следовательно ему нужно 4 байта или 2 строки в таком представлении.

Эту же таблицу можно написать так:

Address value

0x0002FF80 B0 1D FA CE

0x0002FF84 CA 85 DE AF

0x0002FF88 AB BA 8E A8

Не знаю почему препод сразу так не нарисовал. С другой стороны, ты должен быть готов видеть, сколько именно байт в каждой строке и научиться смотреть на адреса начала каждой строки

Теперь по поводу адресов и значений. Если есть такое описание и массив был выделен начиная с адреса памяти 0x0002FF80

unsigned long int arr1[3];

То запись

arr1

означает "спросить по какому адресу памяти начинается массив". Ответ: 0x0002FF80

этот адрес у массива не меняется никогда, после того, как он был раз присвоен!

Используя оператор & можно получить адрес и остальных элементов массива

&arr1[0] даст тот де ответ, что и arr1. Не путать с arr1[0] - это "спросить значение по этому адресу"!

а & - "спросить адрес переменной"!

Give the values of following expressions:

Обозначает ответить на вопрос, что вернет так или иная команда.

В случае с массивами это может быть два типа значений либо значение элемента (когда просто квадратные скобки arr1[1]), либо адрес этого элемента, если используется & либо имя массива без ничего! Адрес это тоже значение!

Просто от наличия [] или & компилятор понимает какое из двух значений нам нужно.

Это классический случай двойственности указателей. Их прямым значением всегда является адрес.

Но когда используется звездочка, то мы так говорим "дай значение записанное по этому адресу памяти"

Еще раз, рассмотрим наш пример и варианты ответов

arr1[0] вернет B0 1D FA CE

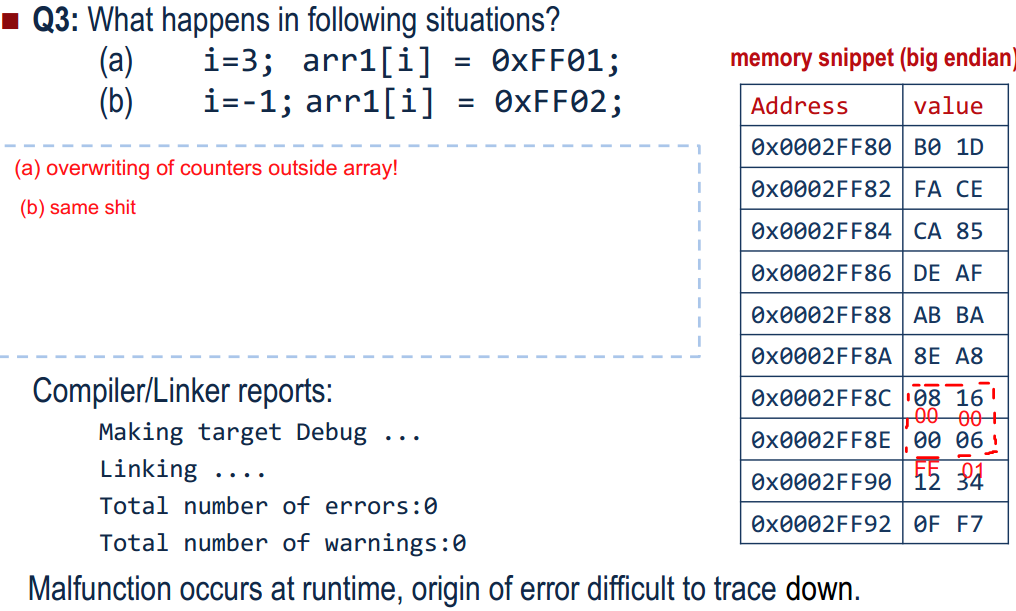
arr1[1] вернет CA 85 DE AF

arr1[2] вернет AB BA 8E A8

arr1 равно как и &arr1[0] вернет 0x0002FF80

&arr1[1] вернет 0x0002FF84

&arr1[2]. Что вернет? 0x0002FF88



arr[3] - это не от 0 до 3, а от 0 до 2 <=========!!!!!!!!!!!!!!!!!!!

Следовательно, написать arr1[3] физически можно, но фактически мы полезем в область памяти, которая под массив не была выделена

И прочитаем/изменим другую переменную, что рано или поздно приведет к ошибкам работы программы

В языке Си и С++ в чистых массивах нету контроля вхождения индекса массива в допустимый диапазон. И такие ошибки сложно вылавливать.

Считается, что на нем пишут профи, они таких ошибок не делают. Все в угоду скорости.

arr[-1]. Как ты думаешь, что может это обозначать?

С массива нужно понимать главное, в квадратных скобках указывают на сколько элементов от начала массива нужно сдвинуться.

Если число больше 0 - то сдвиг происходит вниз, в сторону увеличения адреса, а если меньше - то вверх, адрес уменьшается.

Если указать число большее чем к-во элементов в массиве - мы получим каки - то блок памяти за пределами массива.

А если меньше 0 - тоже блок за пределами массива но перед массивом.

Получается, что отрицательный индекс, если говорим о массиве, всегда за пределами массива, а положительный только тогда, когда он больше или равен длине массива.

Здесь уточняю "если говорим о массиве" потому, что при работе с указателями отрицательное смещение нормальная практика.

Указатель ведь может быть на любой элемент массива, например на последний. Тогда -1 даст предпоследний элемент.

-3 даст элемент который за 3 позиции до конца. При работе с указателями точно так же можно выйти за пределы выделенной памяти

и отследить это еще сложнее, ну да будем как-то разбираться.

Последнее, что здесь следует сказать. Очень много ошибок, которые позволяет увидеть компилятореще до запуска программы. В таком случае жить проще, так как такие ошибки легко увидеть. Но вот ошибки неверного индекса массива компилятор никак отследить не может, а в случае языка Си и принципиально не хочет, считая, что это снизит скорость работы программы. Такие ошибки проявляются не сразу иногда программа будет иметь такую ошибку, но она выскочит не в момент перезаписи памяти, а намного позднее, например, когда она уже закрывается. Так что здесь может быть только один совет - писать обращение по индексу очень внимательно.

**УКАЗАТЕЛИ**

===============================

Главное, что нужно понимать, указателю нельзя присвоить ничего другого кроме как адрес памяти.

1) Адрес памяти можно получить, используя оператор &. этот оператор можно применять вообще к чему угодно. Он возвращает адрес

блока памяти с которого начинаются данные, которые хранятся в этой переменной.

Когда мы создаем любую переменную любого типа, она не явно для нас, выделяет область памяти, который начинается с какого-то адреса

Нас мало интересует конкретное число адреса. Нас может интересовать только что конкретно по этому адресу хранится.

И для того, чтобы получить возможность изменить данные, которые хранятся в переменной мы можем либо использовать саму переменную, либо присвоить ее адрес указателю и далее через указатель, косвенно, делать все тоже самое с этой переменной.

2) Адрес памяти можно присвоив указателю другой указатель, который ранее получил через оператор & адрес какой-то переменной

При присваивании указателя указателю оператор & НЕ ИСПОЛЬЗУЮТ! Дело в том, что указатель - это просто специальная переменная, под которую тоже выделяется память (обычно 4 байта, не зависимо от того, на какой тип данных она указывает), у которой тоже есть адрес, но этот адрес нужен только для указателей ни указатели (именно так реализованы двухмерные массивы) и сейчас нам это не нужно.

Так вот, еще раз, значением указателя, является адрес другой переменной. В самом начале указатель хранит какой-то случайный адрес и его использовать нельзя. Поэтому, после создания указателя, прежде чем его использовать, ему нужно присвоить адрес

какой-то переменной! Именно поэтому если нужно одному указателю присвоить значение другого, просто пишут p1 = p2; без &!!!

Properties -> Advanced -> Compile as C <---- !!!!!!!!!!!!!!!!!!

int a = 100;

int \*p;

p = &a;

После этих команд, не важно, что меняется, значение переменной a или через указатель \*p = someValue, обе переменные будут возвращать

одно и тоже значение. Здесь прикол именно в том, что мы используем \*p - значение по адресу, на который указывает указатель,

в данном случае это переменная a.

Для того, чтобы отвязать указатель, ему можно присвоить адрес другой переменной, тогда ситуация повторится, но уже для той переменной

**По поводу ++i и i++**

int i = 0;

for( ; i < 4; i++)

int i = 0;

for( ; i < 4; ++i)

Для цикла for нету разницы как написать в последней секции ++i или i++. Так же нету связи между

int i = 0;

for( ; i < 4; ++i)

int i;

for(i = 0 ; i < 4; ++i)

int i = 0;

for( ; i < 4; i++)

int i;

for(i = 0 ; i < 4; i++)

Все циклы равны, скорее всего абсолютно, зависит от компилятора.

Прикол в том, что ++i если не оптимизировать, немного быстрее, именно поэтому я предпочитаю его в циклах.

Что-бы понять почему он быстрее, напиши метод, который изменяет значение переменной на 1 и возвращает измененное значение

и еще один метод, который изменяет значение переменной на 1, но возвращает предыдущее значение

Но так как это очень частая операция, то компилятор ее выполняет, скорее всего через регистры процессора, где все одинаково быстро

Так что, если коротко, разницы между 4мя циклами нету никакой!!!!

Разница есть только в выражениях, где ты и изменяешь i и хочешь получить значение этой переменной i

Например:

int i = 0; //i = 0, в этот момент переменной a еще не существует

int a = ++i; //переменной i присваивается значение ни 1 больше чем было, не важно это команда ++i или i++

//здесь переменная a создается и ей тут же присваивается значение 1, так как использован оператор ++i.

//Этот оператор сначала изменяет значение переменной i и потом уже возвращает это значение тому,

//Следовательно, после этой команды a = 1 i = 1

int b = i++; // оператор i++ сначала возвращает это значение переменной b, а затем изменяет значение переменной i

// => после этой команды b = 1 i = 2

int i = 10;

int a = i++;

int b = ++i;

i=12

a=10

b=12

Проверим, что ты запомнил ;)

byte\* p2; //это тоже создание указателя, просто ему пока не присвоен адрес,

его можно (нужно!) присвоить позднее, до любого разыменовывания!

byte\* p1 = 100; //главное нужно понять, что p1 - это указатель, указатель сначала должен получить адрес. Следовательно в самом начале указателю нельзя присвоить значение, только адрес либо переменной либо ячейки памяти!

Никогда при создании указателя не получится присвоить неясно какой ячейке значение!

p1 = 200; //тогда что делает эта команда, если ранее p1 был объявлен как указатель? даётся другой адрес

а если

\*p1 = 300; // даётся значение на адрес 200

все три записи ниже равнозначны, но расскажу почему иногда пишут byte \*p1 = 100;

byte \*p1 = 100;

byte \* p1 = 100;

byte\* p1 = 100;

прикол еще в одном идиотизме Си

как ты думаешь, что создается такой командой?

int\* a, b, c; //int \*a, \*b,\*c; - это было бы логично, но в Си много чего нелогично :(

На самом деле это равносильно:

int\* a;

int b;

int c;

Именно тот прикол, что звездочка относится к одной переменной заставляет людей придвигать ее к переменной, чтобы показать, что она все же принадлежит переменной. Но тогда возникает непонимание, когда создаешь переменную и разыменовываешь указатель.

Так как запись, когда в одну строку создаются переменные лучше не использовать - многострочную проще коментировать и она читабельнее, то ситуаций с непонятками при int\* a, b, c; автоматически избегаем и тогда новичку понятнее запись

int\* a;

Предлагаю проверить еще пару примеров, так как времени осталось мало, а еще углубиться мне тебя жалко ;)

short int a = 100;

short int b = 200;

short int \*p3 = &a;

short int \*p4 = &b;

short int \*p5 = p3;//эта команда связывает p5 с a, используя p3, как посредник и все остальное время она связана с a

p3 = 100;

p3 = &b;

b = 150;

p3 = &a;

a = 50;

a = 50 , b = 150, \*p3 = 50, \*p4 = 150, \*p5 = 50

\*p3 = 70;

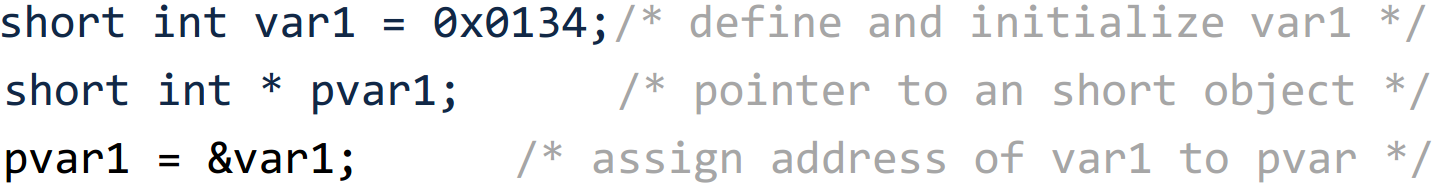
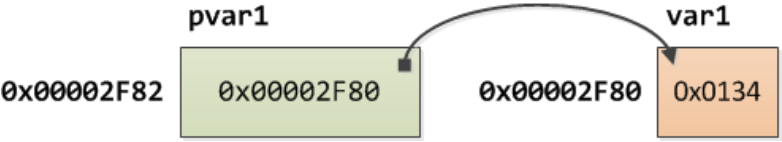
b = a;

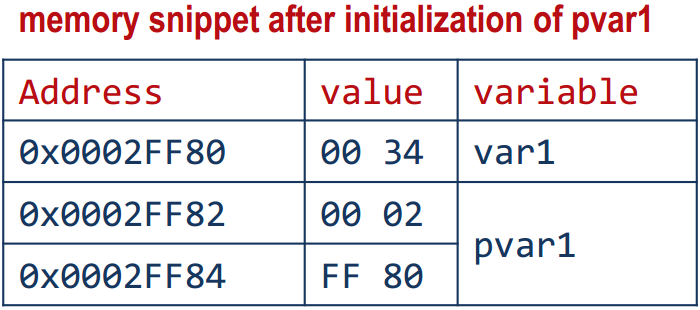
a = 70 , b = 70, \*p3 = 70, \*p4 = 70, \*p5 = 70

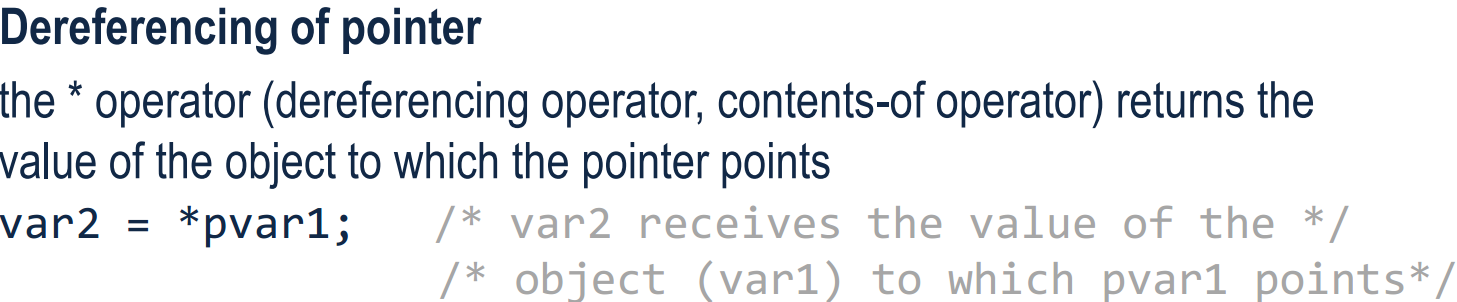
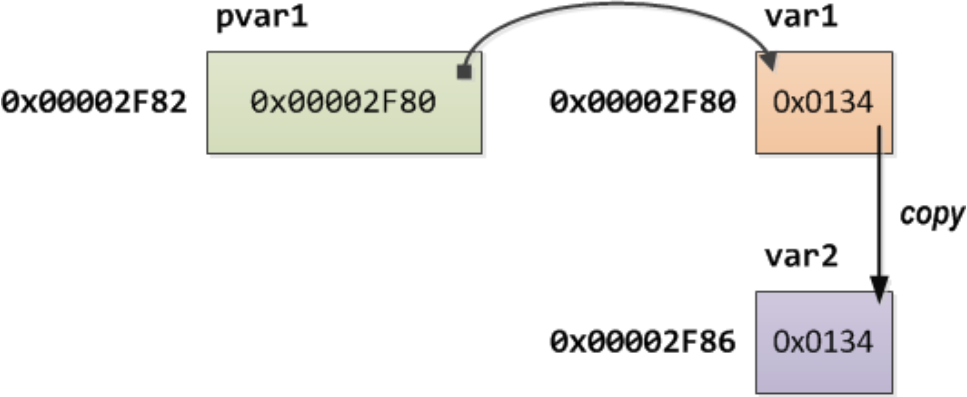
**MC\_03 p.44(41)**

**Who assigns a memory location (address) to a variable? How can we find out the address?**

Ampersand ‘&’ is the address operator, returns address (memory location) of var1 in the example below



**Who assigns a memory location (address) to a variable?**

При сборке программы есть несколько этапов. Препроцессинг, Компиляция, линковка (именно в этом порядке).

Компиляция - переводит текст на языке программирования в машинный или близкий к нему код (зависит от языка).

Линковка объединяет файлы в один и определяет относительные адреса каждого метода.

Потом, при выполнении, программа загружается в память начиная со свободного места в оперативной памяти, которое выделит

операционная система, и методы попадаю уже в конкретные адреса памяти, путем прибавления смещения к стартовому адресу памяти

который был выделен при старте программы.

Переменные бывают двух типов.

Локальные и те, которые выделяются в куче - глобальной памяти.

Локальные - это те переменные, которые ты создаешь в каждом методе.

Под все эти переменные память резервируется заранее и она имеет фиксированный размер. Сколько конкретно памяти выделить, определяет программист или IDE в момент компиляции и линковки программы (для тех, кто работает в Visual Studio, это неразрывные процессы, но олдскульные программисты, запускали раньше последовательно для этого два различных приложения)

Так как память для локальных переменных имеет фиксированный размер, то теоретически можно получить ситуацию, что ее не хватило.

Именно поэтому были придуманы средства для работы с динамическими переменными. В языке Си - через указатели.

Но для начала закончим с локальными

Локальные переменные создаются по мере обращения к ним (того, как операционная система выполняет программу).

Вот вызвал ты какой-то метод, в нем есть переменные - в области памяти локальных переменных будет найдено свободное место и каждая переменная получит свой уникальный адрес. Даже запуск два раза одной и той же программы не гарантирует, что одни и те же переменные получат те же адреса памяти. На самом деле это и не важно, нам не важно знать конкретные адреса.Нам достаточно быть уверенными, что памяти под эти переменные хватило и какой у них тип.

Тип определяет, сколько байт в памяти выделяется под каждую конкретную переменную.

Память под такие переменные, как правило, выделяется подряд.

Представим такую запись и что указатель свободной памяти указывает на адрес 0x100

int a; //4 bytes

byte b; //1 byte

short c;//2 bytes

float d;//4 bytes

double e;//8 bytes

Переменная a получит адрес 0x100 (при другом запуске программы это может быть другой адрес - не важно, рассматриваем какой-то конкретный случай). Так как типу int нужно 4 байта, то следующая переменна b получит адрес на 4 байта больше,

чем переменная a, а именно 0x100+4 = b:0x104

переменная c получит адрес на единицу дальше, чем переменная b, так как та имеет тип byte, следовательно c: 0x105

a: 0x100

b: 0x104

c: 0x105

d: 0x107

e: 0x10B //я специально придумал такие числа, чтобы здесь показать нюанс. Адреса договорились подавать в 16й системе

об этом говорит 0x перед числом

Таим образом, на вопрос, кто выделяет память переменным - общий ответ операционная система или среда выполнения (например в Java - JRE, в C# - CLR. Не путать с IDE! В этих языках программы собираются в промежуточный код, который уже при работе программы на лету компилируется в машинные коды средой выполнения)

**How can we find out the address?**

Сам адрес мы получаем при помощи оператора &. Указатели призваны дать возможность обращаться к этой памяти - читать/изменять ее.

Это же можно делать и при помощи переменных, но только простых.

Например, массивы - это физически указатели, просто скрытые от программиста.

Еще немного о памяти и переменных.

Выше мы рассмотрели, как выделяется память под переменные, а что с массивами?

массив в языке Си - это переменная-указатель, которая всегда указывает на самый первый элемент массива, но используя специальный синтаксис дает возможность обратится и к другим элементам.

Например

int arr[5] = {3, 5, 4, 2, -1};

По сути мы выделили память под идущие подряд 5 переменных типа int и переменная-указатель arr указывает на самый первый

Пусть этот массив начинается с области памяти 0x105, чтобы жизнь малиной не казалась ;)

arr: 0x105

Тогда запись arr[i] говорит, сместись относительно начала массива на i\*sizeof(int) элементов (int, так как в примере массив int)

Конкретнее: arr[1] говорит элемент, который находится по адресу 0x105+1\*4 = 0x109

arr[2] говорит элемент, который находится по адресу 0x105+2\*4 = 0x10D

arr[3] говорит элемент, который находится по адресу 0x105+3\*4 = 0x111 (0x10D + 4 = 0x111 => 0x10D, 0x10E, 0x10F, 0x110, 0x111)

arr[4] говорит элемент, который находится по адресу 0x105+4\*4 = 0x115 (0x111 + 4 = 0x115)

Другими словами, по можно записать какие значения хранятся по каким адресам:

arr[0] = 3: 0x105

arr[1] = 5: 0x109

arr[2] = 4: 0x10D

arr[3] = 2: 0x111

arr[4] = -1: 0x115

Теперь вопрос, а что произойдет, если написать arr[5] или arr[-1], что мы уже видели в лекции

Ответ: применяем туже самую формулу

arr[5] 0x105+5\*4 = 0x119

arr[-1] 0x105+(-1)\*4 = 0x101

Получается, с точки зрения программы мы можем обратится к области памяти, которая за пределами массива, что перед ним (отрицательное смещение), что после него (индекс равный длинне массива и больше) отрицательное смещение обычно указывает на область, которая была выделена ранее, следовательно попробовав изменить значение

через отрицательный индекс мы изменим какую-то переменную, которая была определена ранее

Например:

int a = 100;

int arr[5] = {3, 5, 4, 2, -1};

int b = 200;

arr[-1] = 50; // эта команда запишет в переменную "a" значение 50!!! Так как она была объявлена прямо за 4 байта раньше массива и ее адрес 0x101

arr[5] = 30; // эта команда запишет 30 в переменную b, которая получит адрес сразу за границей массива!

Массив не может изменить свой размер после того, как был объявлен его размер. Так как под него память выделяется только раз!

В Джава массивы сделаны хитрее, они выделяются в куче и им можно потом перевыделить память и сослаться на новую, но не отвлекаемся.

Все может быть еще "веселее", например была такая запись (для 0x00 00 03 E8)

byte a = 100; 00 E8

byte b = 150; 00 03

byte c = 200; 03 00

byte d = 150; E8 00

int arr[5] = {3, 5, 4, 2, -1};

команда arr[-1] укажет на область памяти на 4 байта раньше начала массива и это переменная a!!! так как все переменные выше занимают по одному байту

еще больше сюрприз будет если записать

arr[-1] = 1000;

эта команда изменит все 4 переменные (a, b, c, d)

смотрим в калькуляторе как именно. В 16ричной системе 1000 (10) =

в b = 00 или 03 (0xE8 03 00 00), в c = 03 или 00 и в d = E8 или 00.

Вот такой неожиданный прикол. И главное в Си никто нас не предупредит, что произошел такой казус. А вот Java/Python/C# сразу скажут так как они контролируют (за счет небольших издержек, чтобы все индексы попадали в границы массива)

Рассказываю как по значениею в 16чной системе понять сколько байт

каждые два символа - это один байт. Следовательно 1-2 символа - 1 байт, 3-4 - 2 байта, 5-6 - 3 байта и тд.

Сколько нужно байт, чтобы сохранить значение 0xA1B3C5D4E7F1 ?

Ответ: 6 байт

Если символов нечетное к-во - спокойно можно добавить самым первым 0, например 0xA12 = 0x0A12

Ну и тогда, сколько байт нужно для числа 0x45?

Для того, чтобы записать что-то по конкретному адресу - одно единственное решение, использовать указатель

byte\* p = 0x2000; // так мы говорим, что указатель указывает именно по тому адресу, который просят

\*p = 0x45; // записываем значение по адресу, на который указывает указатель

пожелание, указателям давать "p" в имени, тогда будет проще видеть, где указатель, а где обычные переменные

Теперь еще немножко о указателях и арифметических операциях с ними

Над указателями можно проводить несколько операций

Прибавить к адресу значение (тоже самое было с массивами arr[5] например, будет та же самая формула)

Вычесть из адреса значение (тоже самое было с массивами arr[-1] например, будет та же самая формула)

Инкременты/декременты - тоже самое, просто изменяем только на 1, но нужно смотреть они стоят перед или после указателям

Например \*(p++), говорит, дай значение по адресу, куда сейчас указывает указатель, а его самого сдвинь вперёд на следующий элемент

++p говорит дай адрес следующего элемента за адресом, куда указывает указатель и сам указатель передвинь на этот же адрес

Другими словами, нужно смотреть, есть ли звездочка, чтобы понять, что вернет команда: адрес(когда нет звездочки) или значение (когда звездочка есть) и это будет по текущему адресу (++-- после указателя) или по измененному (++-- перед указателем)

Ну и чтобы решить это задание, нужно вспомнить битовые операции &(and) и |(or).

Так вот везде где стоят эти операторы, мы работаем не с адресом, а со значением

Нужно значение по полученному адресу перевести в двоичную систему (калькулятором) и выполнить действие над тем, что стоит за

бинарной операцией

Например:

byte a[3] = {100, 200, 250};

byte\* p = a; //еще можно записать как &a[0], но это тоже самое

команда \*(p++) & 30 говорит

возьми значение по текущему адресу (это начало массива, следовательно число 100) и

выполни с ним операцию & над другим числом (30)

эта же команда говорит: "указатель переведи после этого на следующие элемент (a[1])"

100 в двоичной системе

30 в двоичной системе

01100100‬

&

=

00000100

Теперь смотрим, что в десятиричной системе это 8.

если после этого будет команда \*(--p) | 20;

это значит, что нужно взять значение из ячейки перед текущей (это снова ячейка a[0], так как текущая была a[1]) = 8

и выполнить операцию

00000100

|

=

00010100

0 & 0 = 0

1 & 0 = 0

0 & 1 = 0

1 & 1 = 1

0 | 0 = 0

1 | 0 = 1

0 | 1 = 1

1 | 1 = 1

~0 = 1

~1 = 0

;-)

Но не нужно инкремент путать с командами вида (p +- n)

Эта команда говорит "сразу прибавь/вычти из адреса". Здесь прячется маленький подвох!

Эта команда \*(p+2) дает значение по новому адресу, но не изменяет сам указатель!

Другими словами, после нее указатель продолжит указывать на первый элемент массива!

Препод сделал задание с большим числом подвохов

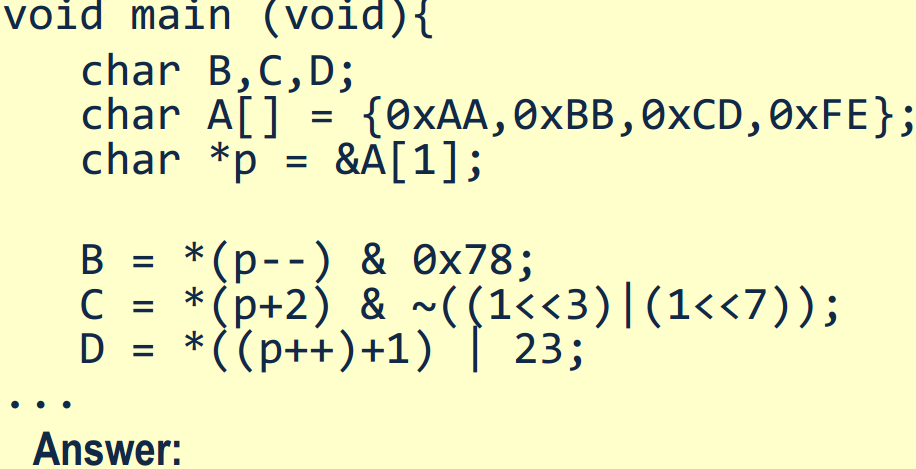
Команды вида 1<<n 1>>n - это очень просто, просто двигаем единичку на указанное число бит

**You want to write 0x45 to a register at address 0x2000. Give the C expression.**

byte\* var = 0x2000;

\* var = 0x45;

**Determine the values of B,C,D after execution of the following program lines:**



//0xBB = 10111011

// |

//23 = 00010111

// =>10111111 = 0xBF

D=0xBF B=0x38 C=0x45

самая сложная и хитрая команда

\*((p++)+1) | 23

она берет текущий адрес и к этому адресу прибавляет 1 - дает адрес следующего за текущим элементов

после этого берет значение, по получившемуся адресу (у нас это элемент &A[1])

влияние ++ имеет только на то, что после того, как закончится весь-весь расчет, она передвинет указатель на следующий элемент

но для вычисления числа она никакого влияния не имеет. А вот если бы было ++p, то мы бы получили в выражениее &A[2]

на самом деле (p++)+1 эквивалентно ++p и так, как написано в задании, никто не пишет, но препод хочет научить вас думать

текущее p++ (см. внизу про р++ и ++р) => 0xAA

+1 => 0xBB

итого, после расчёта \*((p++)+1) | 23 => 0xBB

Пойми, команда p+1 не изменяет сам указатель, она дает ячейку за ним

а вот команда ++p изменяет его сразу

а вот команда p++ не изменяет его для вычисления выражения, но изменяет после того, как выражение будет обработано!

**Determine the values of B,C,D after execution of the following program lines:**

**void main (void)**

{

char B,C,D;

char A[] = {0xAA,0xBB,0xCD,0xFE};

char \*p = &A[1]; //0xBB

B = \*(p--) & 0x78; //0xBB & 0x78 => 0x38; а после этого "p" становится &A[0]!

C = \*(p+2) & ~((1<<3)|(1<<7)); //0xAA => 0xCD, затем 0xCD & ~((1<<3)|(1<<7)

//(1<<3)=> 0001<<3=1000 (1<<7)=> 0001<<7= 10000000

// 00001000

// |

// 10000000

// = 10001000

// ~10001000 => 01110111

//

//

// &

// 01110111

// = 01000101 => 0x45

//после всего этого расчета p не изменится и останется указывать на &A[0]

D = \*((p++)+1) | 23; // 0xAA => 0xBB => 0xBB | 23 ......

//0xBB =

// |

//23 =

// =>10111111 = 0xBF

...

}

Answer:

0xBB =

&

0x78 =

=

00111000 => 0x38

Главный вывод, что над указателями можно использовать арифметические операции, но не каждая из них изменяет сам указатель

Когда мы используем оператор вида \*(p + someValue) - это означает возьми значение, которое смещено на someValue элементов после элемента, на который указывает указатель

Когда мы используем оператор вида \*(p - someValue) - это означает возьми значение, которое смещено на someValue элементов перед элементом, на который указывает указатель

Эти команды возвращают значение из другой ячейки, но после их выполнения указатель не изменяет своего положения! Он указывают на ту самую ячейку, куда указывал до этих команда

Можно еще увидеть запись вида p2 = p1 + someValue; или p2 = p1 - someValue;

Здесь p2 получит не новое значение (ибо нету значка \*), а адрес новой ячейки, которая будет вычислена по тому-же самому правилу!

При этом переменная p1 не изменится ;)

Другим видом арифметических команд есть операторы инкремента/декремента.

Они мало того, что дают адрес, еще и изменяют сам указатель! Разница в том, до указателя они стоят или после

Если до - то в выражении будет использован новый адрес больше(++)/меньше(--) текущего на один элемент.

Сам указатель тоже сместится сразу на этот же новый адрес.

Если же ++/-- стоит ПОСЛЕ указателя - то в выражении будет использован текущий адрес указателя, и сам указатель изменит адрес уже после того, как вся команда, в которой он выполняется, будет обработана.

char A[] = {0xAA,0xBB,0xCD,0xFE};

char \*p = &A[1]; //0xBB - это не значение p, это значение по адресу, на которой он указывает!

//какой конкретно адрес не важно, нужно главное понимать, что он указывает на элемент с индексом 1 в массиве A

char\* p2 = p--; //чему будет равняться p и p2 после выполнения этой команды?

//p=0xAA p2=0xBB правильно писать \*p=0xAA \*p2=0xBB, а если говорить об адресах, то можно записать так

//p=&A[0] p2=&A[1]

p2 = ((++p)+2); //p2=&A[3] p = &A[1]

//(++p)+2 = p+3, но p = p + 1 произойдет сразу, так как ++ стоит до указателя

//(p++)+2 = p+2, но p = p + 1 => именно так изменится переменная p после того, как вся эта команда выполнится

p = p2 - 2; //p=&A[1] p2=&A[1] => ошибка, т.к. p2=&A[3]

**MC\_03 p.52-53 (46-47)**

#define GRAV\_CONST 6.674e-11 /\* gravitational constant \*/

#define ASIZE 120 /\* array size \*/

#define DEBUG\_OUT\_L1 /\* level1 debug output \*/

В языке Си для констант используется не совсем современный способ определения констант через #define

Уже в языке C++ очень рекомендуют не использовать #define для констант, так как они не имеют типа и компилятор не может работать с ними так же, как с настоящими константами, но в языке Си выбора нет, это расплата за то, что языку очень много лет.

В чем прикол дефайнов?

Представь себе, что ты придумываешь себе текстовые метки, которые равняются чему-то осознанному

например метка ASIZE равняется размеру массива, который ты будешь использовать позднее в программе

Ты создаешь дефайн #define ASIZE 120 и далее везде в программе вместо того, чтобы писать 120, везде пишешь более читабельное ASIZE. Ну либо даешь более читабельное значение, главное, чтобы тебе самому было понятно, что оно обозначает.

Когда в коде просто написано 120 - это называется использовать magic number. В момент написания программы еще понятно, что это за 120, но пройдет время и ты забудешь.

Так вот, после того, как запускается процесс компиляции, препроцессор до начала компиляции проходится по всей программе и везде, где встречает метку из дефайна, подставляет то, что стоит справа от него (для ASIZE 120 - это 120)

Это не число 120, это замена в коде программы ASIZE на 120.

После того, как все замены произведены - начинается процесс компиляции, которому уже на страшно, что нету имен переменных

Из этого правила выпадает только #define DEBUG\_OUT\_L1. Он преследует другие цели. Это инструмент условной компиляции

У него справа нет никакого значения, здесь самим значением является имя константы

Она потом используется в коде в виде конструкций

#ifdef DEBUG\_OUT\_L1

printf(“GRAV\_CONST=%f\n“,cArray[3]);

#endif

Что говорит эта конструкция?

Если выше ты определил константу DEBUG\_OUT\_L1, то строчка printf(“GRAV\_CONST=%f\n“,cArray[3]); будет добавлена в программу

Если же не определил, то не будет и это сообщение не будет выводится в собранной программе.

Когда такое нужно?

Например у тебя есть программа для пользователей, которая должна работать максимально быстро. Но тебе иногда нужно отлаживать эту программу

В коде ты пишешь в тех местах, где нужно вывести значение некоторых переменных на экран конструкцию вида

#ifdef DEBUG\_OUT\_L1

printf(“Some debug output %f\n“,someVariable);

#endif

После того, как ты нашел все ошибки ты не удлаяешь строчку #define DEBUG\_OUT\_L1, а просто комментируешь ее

//#define DEBUG\_OUT\_L1

Автоматически, когда программа собирается с закоментированной //#define DEBUG\_OUT\_L1 в нее не попадет отладочный вывод

но у тебя появляется возможность быстро снова его включить, просто раскоментировав в одном месте #define DEBUG\_OUT\_L1

По такому же принципу работает еще сборка программ под разные ОС. Например под MAC у тебя работает одна команда доступа

к файлам, а под Windows другая. Ты пишешь

#ifdef WINDOWS

//some windows related code

#elseif MAC

//soem MAC related code

#else

//other OS code

#endif

**Макросы**

Позволяют упростить и одновременно усложнить жизнь программистам :).

С одной стороны, он позволяют делать какие-то шаблоны, которые позволяют сократить к-во кода, которое ты пишешь

Например, тебе нужно часто писать variable\*variable;

Ну ты придумываешь макрос

#define SQUARE(X) X\*X

Он говорит, что везде в программе, где ты напишешь

int a = 10;

int b = SQUARE(a);

препроцессор потом напишет

int b = a\*a;

пока упрощение сомнительное, но если имена переменных будут длиннее

int c = SQUARE(someLongEnougName);

заменится на

int c = someLongEnougName\*someLongEnougName;

все классно?

это быстрее чем функция, для вызова которой нужно использовать стэк, перемещаться на адрес функции, а потом возвращаться назад

в место вызова.

Но проблема в том, что здесь не интеллектуальная подстановка

Допустим ты захотел написать int c = (a + b)^2; пишем int c = SQUARE(a + b); Так?

На самом деле препроцессор не столь интеллектуален и выполнит этот код буквально

int c = a + b\*a + b; Это тоже самое, что мы хотели?

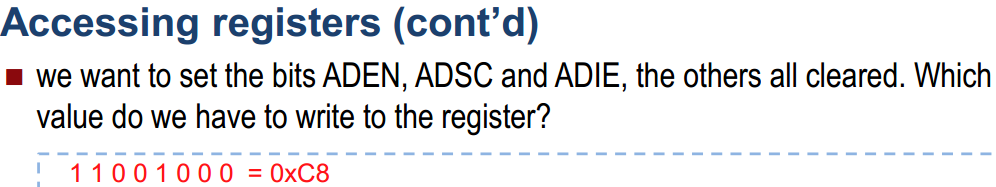
Именно по-этому приходится макрос писать в виде

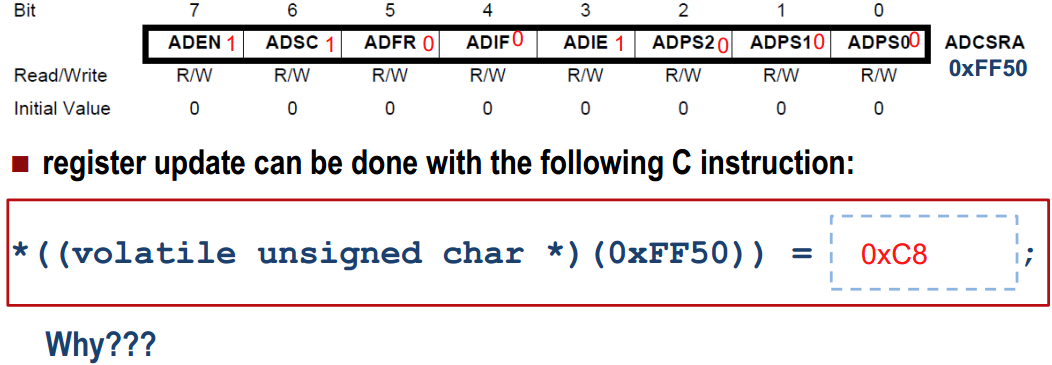
#define SQUARE(X) ((X)\*(X))

Из-за того, что с использованием макрософ вылезает много приколов и ньюансов чаще всего, если тебе нужно поведение функции – лучше создать функцию. Так будет обслуживать программу проще и потом ошибки находить тоже намного проще.

Макросы используются при написании программ, которые пишут программы, так они могут оказаться очень полезными, но это нужен приличный опыт работы с ними, так что предлагаю на этом пока закончить их обсуждение ;)

**MC\_03 p.55(50)**





**Работа с регистрами через указатели**

\*((volatile unsigned char \*)(0xFF50)) = someValue; //someValue - значение, которое занесли в регистр

0xFF50 - адрес памяти, по которому находится регистр

(volatile unsigned char \*) - так как не используется переменная-указатель, то прежде чем обратится к адресу, нам нужно сказать какого типа значение там хранится

volatile используем, так как это регистр и при работе с регистрами привыкнешь его писать :)

unsigned char - говорит, что это 1 байт без знака

\* - что мы не просто написали число 0xFF50, а обращаемся по адресу 0xFF50 к значению, которое является unsigned char

Так если бы там хранилось число int, мы бы написали (volatile int \*)

будь у нас указатель, было бы привычнее

volatile unsigned char\* p = 0xFF50; //создали уже привычный нам указатель, который указывает на регистр по адресу 0xFF50

\*p = someValue; // записали по этому адресу значение someValue

мне приятнее использовать указатель - он понятнее, хоть и немного длиннее

по сути запись (volatile unsigned char \*)(0xFF50) - это команда создай временный указатель по адресу 0xFF50 на беззнаковый байт

\*() вокруг (volatile unsigned char \*)(0xFF50) - просто обертка, которая говорит - дай значение по этому адресу.

Когда явно используется указатель, то это упрощается до \*p

если бы хотели прочитать значние из регистра, написали бы

volatile unsigned char someVariable = \*((volatile unsigned char \*)(0xFF50));

Но проще делать так

volatile unsigned char\* someRegister = 0xFF50; //здесь в указатель один раз записали адрес и больше о нем не думаем

unsigned char a = \*someRegister; //прочитать значение из регистра и записать в переменную a

\*someRegister = 0x100;//записали значение 0х100 в регистр

В языке Си, который используется для работы с микроконтроллерами уже будет некоторое к-во заголовочных файлов вида

tm4c1294ncpdt.h, в которых будут прописаны константы вида

#define ADCSRA (\*((volatile unsigned char \*)(0xFF50)))

использыемые для работы с часто затребованными регистрами

Тогда нету нужды помнить все эти адреса, а достаточно подключить в свою программу нужный заголовочный файл

#include "tm4c1294ncpdt.h"

и дальше везде, где нужен доступ к регистру ADCSRA писать просто

unsigned char someVariable = ADCSRA;

**Манипуляция с битами, кратко повторим**

Чтобы установить какой-то бит, нужно использовать битовую операцию | (OR)

Например, у нас есть какая-то информация в регистре и нам нужно оставить ее неизменной, кроме бита с конкретным номером

1) Сначала нам нужно получить маску для этого бита. Это сделать очень просто, выполнить операцию 1 << bitNumber

Так если нам нужен 3й бит, пишем 1<<3 => 00001000.

2) Теперь накладываем эту маску на существующее значение, каким бы оно не было, такая маска в паре с |(OR) не тронет ни один бит

кроме того, который есть в маске

Например, у нас в регистре 01100001 и нам нужно установить 3й бит, т.е. маска 00001000. Выполняем

01100001

|

00001000

=

01101001

Как видим, изменился только 3й бит

Если нужно изменить больше одного бита, то эту операцию можно повторить несколько раз, каждый раз для очередного бита получив

новую маску или поучить комбинированную маску.

Например, нам нужно установить бит 7, 5 и 3

маска для бита 7 = 1<<7, для бита 5 = 1<<5, для бита 3 = 1<<3. Объединенная маска (1<<7)|(1<<5)|(1<<3) = 10101000

После этого, все так же складываем эту маску с нашим регистром и получим гарантированную установку только 7, 5 и 3 го бита.

Например, в регистре 00000010

00000010

|

10101000

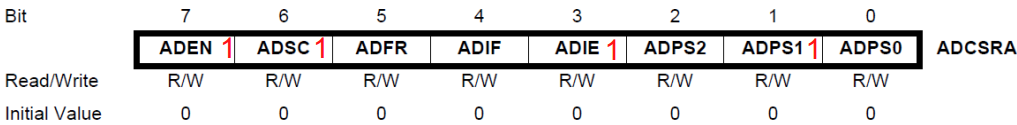
=

10101010

Главную идею, как работать с регистрами, и зачем все же нужны указатели ты понял ;)?

**MC\_03 p.58(52)**

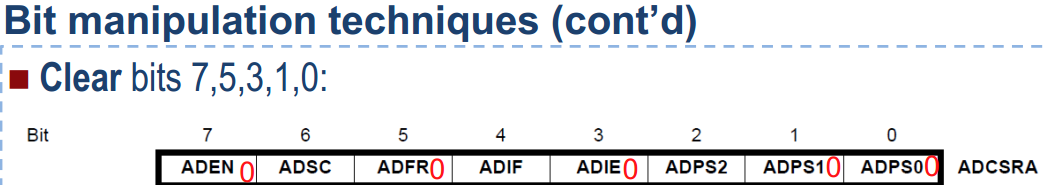


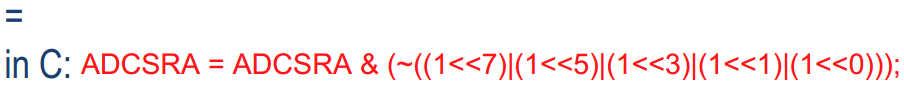


Answer:



**MC\_03 p.63 (56)**





**Clear bits 7,5,3,1,0:**

Это значит, нужно выставить ноль в перечисленных битах и не тронуть остальные

Как и в прошлой задаче, мы не знаем, какие значения были во ВСЕХ битах до операции, это и не важно, битовые операции позволяют изменить только те биты, которые нам нужны, не тронув остальные. Для того, чтобы выставить какой-то бит, мы на прошлом уроке научились создавать маску и используя битовую операцию | устанавливать только необходимые биты.

Для очистки битов сначала тоже нужно сделать маску, только немного не так.

Нам нужно получить маску, которая будет иметь 0 против всех битов, которые мы хотим обунлить и 1 против всех остальных.

Так как мы знаем, что у нас всего 8 бит, то задачу можно немного упростить.

Создадим маску, как в прошлый раз, только с единичками против тех бит, которые мы не хотим менять.

И наложим эту маску на текущее значение, используя побитовую операцию &(AND)

11111111 (здесь будет текущее состояние регистра, может быть каким угодно, в худшем для очистки битов случае 11111111)

& но может быть любым, для нас это не будет иметь значения

01010100

=

01010100

В случае, когда были все единицы, мы действительно получим такое же значение, как и маска, но попробуй наложить эту же маску

на число 00011010

00011010

&

01010100

=

00010000 - как видишь, сейчас мы получили другой результат. Биты в которых был 0 и которые мы не собирались трогать - остались неизмененными

Теперь вопрос, как получить, используя методику прошлого урока, маску 01010100?

Как используя операцию сдвига и | получить число 01010100

(1<<6)|(1<<4)|(1<<2)=01010100

Я предложил этот способ как максимально похожий на то, что мы делали на прошлом уроке.

Для создания маски, используем не те биты, которые нужно очистить, а те, которые нужно оставить нетронутыми.

Есть и другой способ получения того же, но чуть хитрее. Точно так же, как и раньше создадим маску на биты, которые нужно очистить

(1<<7)|(1<<5)|(1<<3)|(1<<1)|(1<<0)= 10101011

А теперь над этой маской проведем операцию инверсии ~. Она меняет 1на 0, а 0 на 1

~((1<<7)|(1<<5)|(1<<3)|(1<<1)|(1<<0)) = 01010100

Этот второй способ работает для всех случаев, а первый, который мы рассмотрели вначале, будет сложным, когда у нас будет не 1 байт, а 2 и больше

Так например, если нам нужно очистить все те же биты 7, 5, 3 и 2й, но уже в регистре из 2х байт, маска должна выглядеть 1111111101010100 и создать ее первым способом уже долго

(1<<15)|(1<<14)|(1<<13)|(1<<12)|(1<<11)|(1<<10)|(1<<9)|(1<<8)|(1<<6)|(1<<4)|(1<<2)

А второй способ левый байт заполнит битами автоматически и останется таким же ~((1<<7)|(1<<5)|(1<<3)|(1<<1)|(1<<0))

Еще раз, я показал первый способ для очистки бит потому, что он был очень похож на способ установки бит, но универсальным есть инверсия

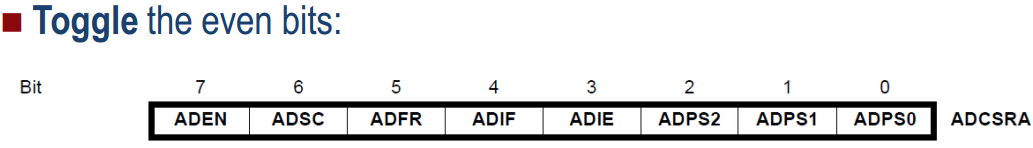
ADCSRA = ADCSRA & (~((1<<7)|(1<<5)|(1<<3)|(1<<1)|(1<<0)));

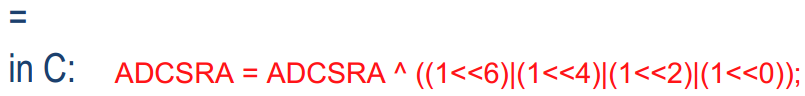
Здесь так много скобок потому, что инверсию нужно сделать не на (1<<7), а на все выражение (1<<7)|(1<<5)|(1<<3)|(1<<1)|(1<<0),

а третья пара потому, что & имеет свой приоритет и может выполнится раньше |. Надежнее все, в чем сомневаешься, оборачивать в скобки

(1<<7)|(1<<5) - Здесь скобки ставим потому, что операция << имеет самый низкий приоритет и 1<<7|1<<5 компилятор поймет как (1<<(7|1))<<5, что не одно и тоже

**MC\_03 p.63 (56)**





**Toggle the even bits**

В данном случае просят изменить значение четных бит на противоположные (переключить их в противоположное значение). Там где был 0 - выставить 1, там где был 1 - выставить 0. 1<<0 можно записать коротко как 1, но можно и так:

(1<<6)|(1<<4)|(1<<2)|(1<<0) = 01010101

11111111

^

01010101

= 10101010

00000000

^

01010101

=

01010101

00001111

^

01010101

=

01011010

Как видим, как и обещано, переключаются только те биты, против которых в маске мы поставили 1

10010100

|

=

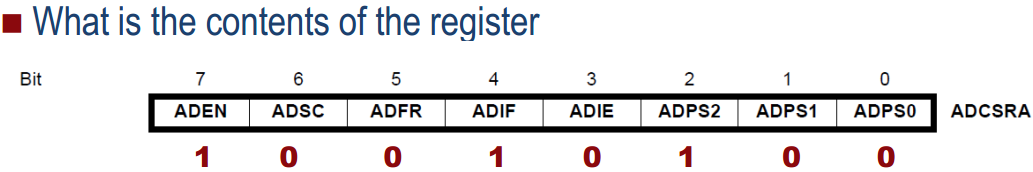
11010101

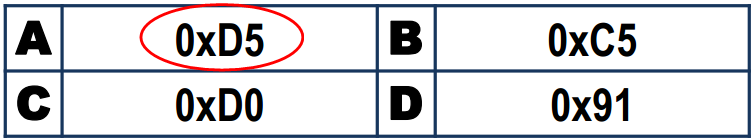
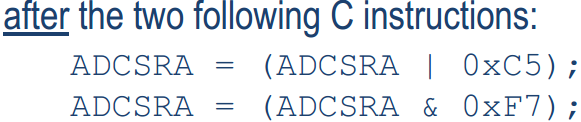
11010101

&

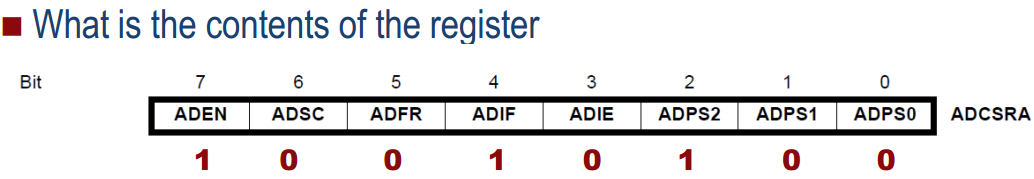
= 11010101

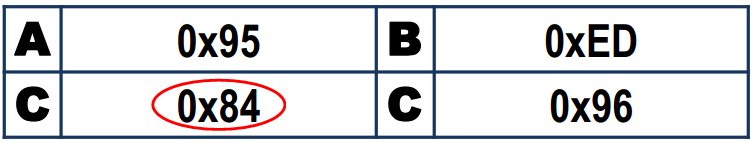
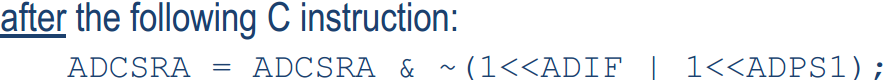
**MC\_03 p.65(57)**





**MC\_03 p.69(60)**





10010100 & ~(1<<ADIF | 1<<ADPS1)

(1<<ADIF | 1<<ADPS1) = (00010000 | 00000010)

00010000

|

00000010

=

00010010

~00010010= 11101101

10010100

&

11101101

=

10000100

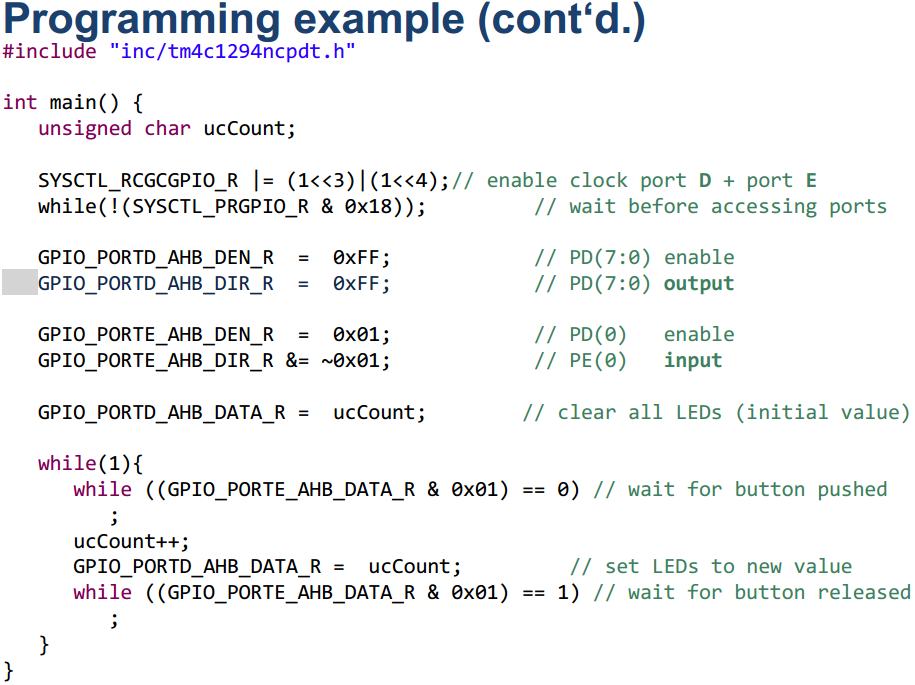
Ты делал долго расчет, но с опытом приходит понимание, что делает эта последовательность без них

~(1<<ADIF | 1<<ADPS1) - делает маску для очистки битов, следовательно вместе с & мы просто очищаем биты с номерами 4 и 1

смотрим на 10010100 и после очистки этих бит получаем 10000100 - тоже самое, что после длинных расчетов ;)

Очистка бит - выставить в конкретных битах 0

**MC\_03 p.74(64)**



**Программирование микроконтроллеров**

SYSCTL\_RCGCGPIO\_R |= (1<<3)|(1<<4);

SYSCTL\_RCGCGPIO\_R = SYSCTL\_RCGCGPIO\_R | (1<<3)|(1<<4);

while ((GPIO\_PORTE\_AHB\_DATA\_R & 0x01) == 0); // wait for button pushed

Обычно цикл while выглядит так

while(someCase)

{

}

здесь же while(someCase);

эта команда обозначает - не делай ничего, пока выполняется условие. Признак “не делай ничего” - это точка с запятой

смотрим внимательнее на условие (GPIO\_PORTE\_AHB\_DATA\_R & 0x01) == 0

оно состоит из двух частей

(GPIO\_PORTE\_AHB\_DATA\_R & 0x01) - первая, что она делает?

(GPIO\_PORTE\_AHB\_DATA\_R & 0x01)

эта команда очищает все биты кроме нулевого!

В результате, у нее есть два возможных результата, либо 1 в нулевом биты (так как все остальные делаем 0) либо 0

Первый результат возможен, если в нулевом биты была единица, второй - если 0

Следовательно проверка (GPIO\_PORTE\_AHB\_DATA\_R & 0x01) == 0 - говорит, "проверь, а 0 ли в нулевом бите ;)"

Судя по всему, когда в нулевом бите 0 - это признак того, что кнопка НЕ нажата и мы в таком случае ничего не делаем

Действительно while ((GPIO\_PORTE\_AHB\_DATA\_R & 0x01) == 0); можно переписать как

while ((GPIO\_PORTE\_AHB\_DATA\_R & 0x01) != 1); и это может быть понятнее

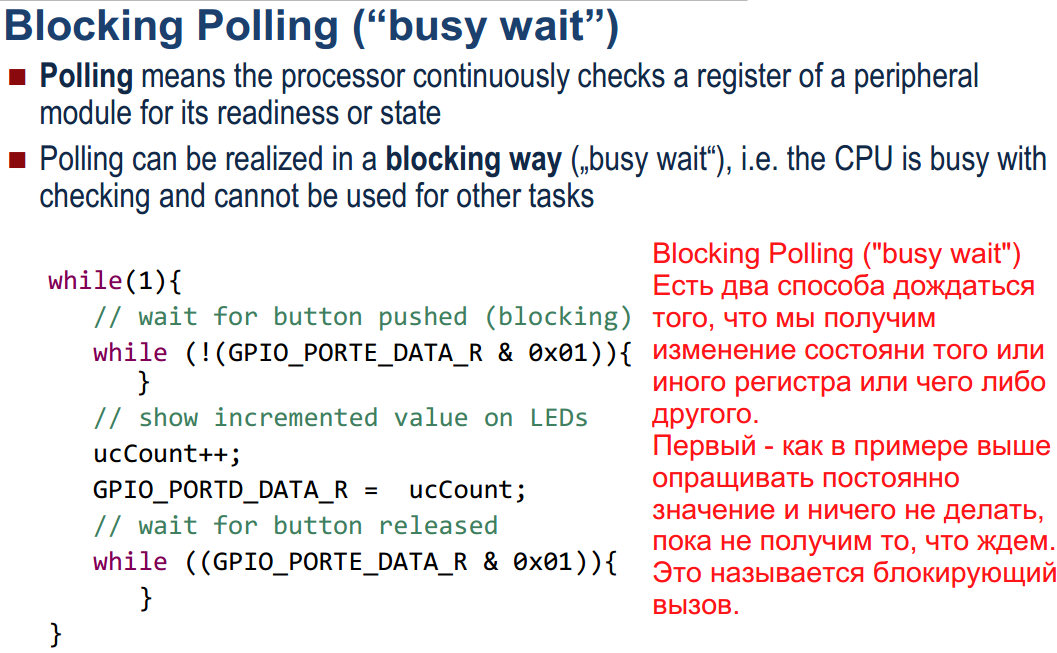
0x01 = 0000 0001

while ((GPIO\_PORTE\_AHB\_DATA\_R & 0x01) == 1);

Тогда этот цикл говорит, пока в нулевом бите 1 - не делай ничего, другими словами ждем, пока кнопка будет отпущена, после чего в нулевом бите будет 0 и цикл закончится

т.е. "== 1" намекает на "проверь, а 1 ли в нулевом бите"

**MC\_03 p.77(67)**



**Interrupt**

Это второй способ, так называемых прерываний. Для меня это давно забытое слово, так как оно низкоуровневое и я последний раз его

использовал на Assembler.

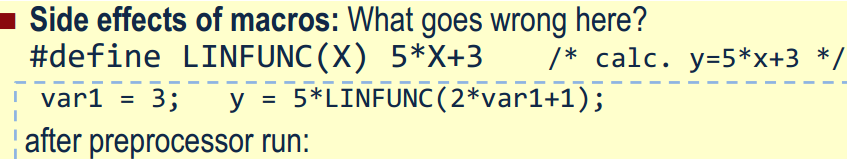
Этот способ намного более оптимальный, его прелесть в том, что мы получаем вызов кода только тогда, когда действительно произошло какое-то событие.

Например, была нажата клавиша - будет вызвано прерывание отвечающее за обработку клавиша

Изменилось время (оно меняется не раз в секунду, а раз в некоторое к-во милисекунд) - будет вызвано прерывание, отвечающее за обработку событий таймера и тп

Сейчас они просто в общих чертах об этом способе говорят, без конкретики, которая будет, я так понимаю, позднее ;)!!!!!!

**MC\_03 p.82(71)**



var1 = 3; y = 5\*LINFUNC(2\*var1+1);



#define LINFUNC(X) 5\*(X)+3 (добавили скобки для Х)

Пояснение:

y = 5\*LINFUNC(2\*3+1);

//y = 5\*LINFUNC(7); ожидаемо, что будет так, но на самом деле, и в этом печаль, что нет

y = 5\*LINFUNC(2\*var1+1);

будет заменено препроцессором на

y = 5\*2\*var1+1+3; а мы ведь ожидали y = 5\*(2\*var1+1)+3;

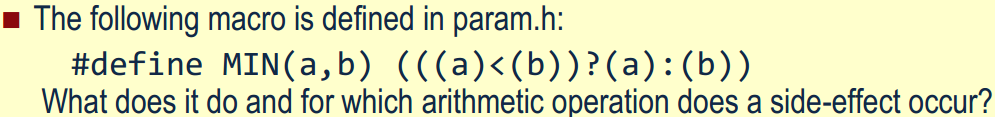
именно поэтому в макросах приходится ставить скобки там, где в другом случае они не нужны

исправить нужно макрос: #define LINFUNC(X) 5\*(X)+3 (добавили скобки для Х)

y = 5\*(2\*var1+1)+3;

y = 5\*(7)+3;

y = 38



int c = MIN(5, 10);

int a = 5;

int b = 10;

ind d = MIN(a+b, a-b); //MIN(15,-5) - так ожидаем, но макросы работают не так. Именно в этом сайд эффект!

Задание здесь было в том, чтобы понять, что макрос всего лишь возвращает минимальное из двух значений

Теоретически, можно было написать без скобок

#define MIN(a,b) a<b?a:b

Но из-за того, что макросы обрабатываются как текст, при подстановке в макрос выражение получим не то, что ожидаем (side effect), и именно поэтому приходится писать макросы используя идиотские скобки где только можно и именно поэтому я рекомендую их (таких макросов) избегать вообще

#define MIN(a,b) a<b?a:b

ind d = MIN(a+b, a-b);

будет обработан как

a+b<a-b?a+b:a-b

здесь не сравнится результат сложения и вычитания, а сложится a и b<a(которое будет равно либо 0 (когда b<a) либо 1 (иначе)) и из этой суммы вычтется b

Другими словами, компилятор поймет это как

(a+(b<a))-b?a+b:a-b

(5+(10<5)-10 = 5+0- 10 = -5

-5?5+10:5-10

После того, как вычислили слева от ? это число сравнивается с 0, так как оно не равно нулю, возвращается то, что стоит до знака “:”

Здесь двоеточие не "разделить"!!! это часть тернарного оператора. Разделить в языке Си /

Следовательно ответ этого выражения то, что стоит до “:” это 5+10 = 15

Получается, что в ответ на сравнение 15 и -5 мы получили ответ, что 15 меньше -5. Это бред.

Именно в этом побочный эффект этого макроса! Именно потому, чтобы не было такой ошибки ПРИХОДИТСЯ ставить скобки и писать

#define MIN(a,b) (((a)<(b))?(a):(b))

этот макрос будет работать без ошибок

допустим, что в ответ на сравнение 15 и -5 мы получили ответ, что 15 больше -5

нет, функция ведь называлась MIN, следовательно макрос безо всех скобок работает неправильно, именно потому их там так много

Это макрос не пишет ничего, он вернет одно из двух чисел. Когда он написан правильно, он возвращает всегда минимальное значение.

Задача была показать, что без скобок макрос может работать неправильно.

Почитай и попробуй использовать тернарный оператор в простой программе.



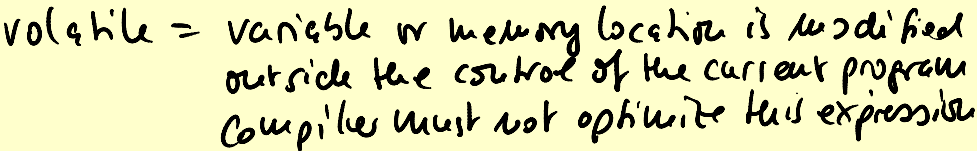
**Polling**: simpler to test and debug; data consistency (последовательность/согласованность) not an issue

**Interrupt**: more efficient since SW only active when “smth has to be done”



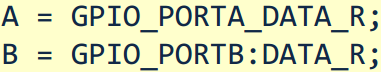
volatile используем, т.к. это регистр и при работе с регистрами привыкнешь его писать :)

volatile indicates that this register can be changed by HW (e.g. ADIF flag set when ADC completed) and instructs the compiler not to optimize (i.e remove the line)



**MC\_03 p.86(73)**







~((1<<7)|(1<<2)|(1<<0)) = ~(10000101) = 01111010

C = A & ~((1<<7)|(1<<2)|(1<<0));//можно написать так можно написать, так четко видно, какие биты очистили

C = A & ~(0x85) = A & 0x7A; //так тоже можно написать и будет правильно, но по коду не будет понятно, что мы сделали и придется смотреть в калькуляторе



D = B & ~((1<<4)|(1<<3)|(1<<2)|(1<<1)|(1<<0)) = B& ~(0x1E);



это значит, что нужно установить биты в конкретные значения

(!!!!! Через страницу показано правильное решение – мы чуток перепутали !!!!!)

в данном случае, нужно 3му биту переменной А присвоить тоже самое значение, что у 7го бита переменной B, 2му биту - 6й бит, 1му - 5й и 0му - 4й

E = A | B =

здесь все намного сложнее, чем до этого было. Проблема в том, что для бит, которые нужно выставить в 0 - нужно делать очистку, а для

тех, которые придется выставить в 1 - нужно делать установку.

Такую операцию можно сделать только для каждого бита написав проверку, если тот, который копируем - равен 1, то делаем для него маску

копирования, а для того, который равен 0 - маску очистки. Так как копируется 4 бита - нужно выполнить это действие 4 раза!

Думаю лучше всего привести сначала примере

Предположим мы копируем биты 7:4 переменной B в биты 3:0 переменной А.

B = 10101111

A = 00000000 любое значение, не важно какое, но напишем конкретное, чтобы было видно результат

Берем 7й бит переменной B, это 1, следовательно в 3й бит переменной A нужно произвести установку бита

Берем 6й бит переменной B, это 0, следовательно в 2й бите переменной A нужно произвести очистку бита

Берем 5й бит переменной B, это 1, следовательно в 1й бит переменной A нужно произвести установку бита

Берем 4й бит переменной B, это 0, следовательно в 0й бите переменной A нужно произвести очистку бита

другими словами, после выполнения этих операций переменная B не изменится, а в переменной A будет значение 00001010. Понятно почему?

Для того чтобы написать программу, нам нужно научиться выполнять еще одну битовую операцию - "тест бита"

Тестирование бита похоже на установку, с небольшим добавлением

Маска создается, как для установки, но потом над этой маской проводят операцию & вместо | и сравнивают результат с 0. Если результат

равен 0 - тестируемый бит 0, если неравен - 1.

Например

Нужно проверить бит 5 у переменной value со значения 1101111

Маска будет 1<<5, так как нас интересует 5й бит

и условие в программе пишем так

if (value & (1<<5) != 0)

printf("5th bit is 1");

else

printf("5th bit is 0");

как это работает?

1101111

&

0010000

= 0000000 - результат равен 0 следовательно этот бит равен 0

1111111

&

0010000

=

0010000 - результат НЕ равен 0 следовательно этот бит равен 1

Следовательно, задание сводится к тому, что нужно будет написать 4 if, по одному для каждого копируемого бита. Покажу как быть с первым

в задании, дальше сможешь дописать сам

if ((B & (1<<7)) != 0)

E |= (1<<3);

else

E &= ~(1<<3);

теперь допиши для остальных бит ;)

if ((B & (1<<6)) != 0)

E |= (1<<2);

else

E &= ~(1<<2);

if ((B & (1<<5)) != 0)

E |= (1<<1);

else

E &= ~(1<<1);

if ((B & (1<<4)) != 0)

E |= (1<<0);

else

E &= ~(1<<0);



Я понимаю это задание так

Нас интересует то, что было в переменной A, но часть ее битов нужно взять от переменной B и заменить ими биты в A и присвоить результат

переменной E, но при этом саму переменную A изменять не нужно!

Тогда мы просто копируем в переменную E, то, что было в переменной A, а потом все манипуляции проводить над переменой E

Так как биты и там и там идут по порядку, то можно использовать цикл

E = A;

int i;

for(i = 0; i < 4; ++i)//i =< 4 здесь две ошибки! правильно писать <= и если ты хочешь выполнить что-то n раз, то пишут i = 0; i < n; ++i

{

//в этом цикле осталось только немножко подправить и все будет ок. По сути для чисел из нуля нужно получить 7 в условии и 3 в манипуляции

//когда будем работать с 6м битом, нужно получить 6 в условии и 2 в манипуляции

//когда будем работать с 5м битом, нужно получить 5 в условии и 1 в манипуляции

//когда будем работать с 4м битом, нужно получить 4 в условии и 0 в манипуляции

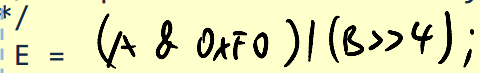
if ((B & (1<<(7-i))) != 0)

E |= (1<<(3-i));

else

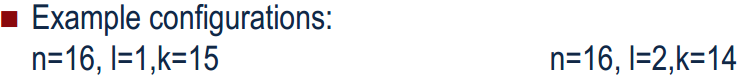
E &= ~(1<<(3-i));

}



**MC\_04 p.35(35)**

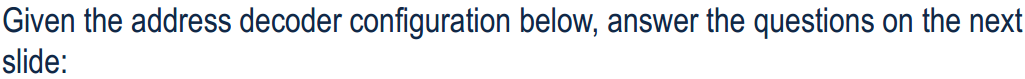


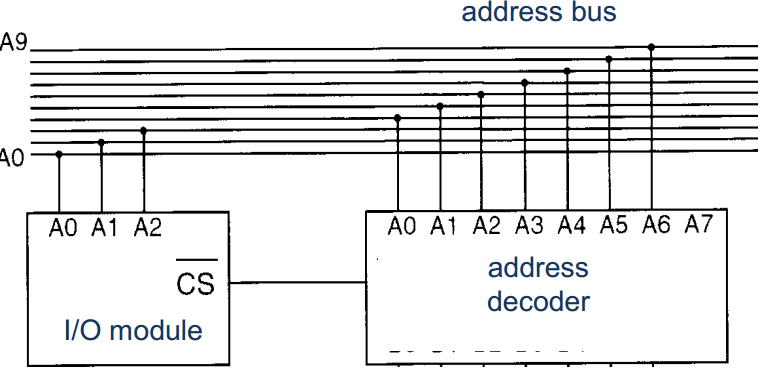


**MC\_04 p.36(36)**

**MC\_04 p.37(37)**





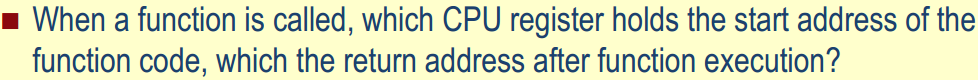


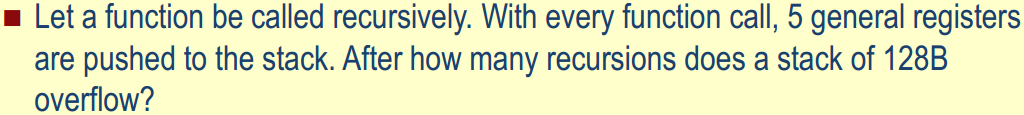
**[a] How large is the totally addressable memory space (if the smallest addressable unit is 1B)? How many registers has the I/O module?**

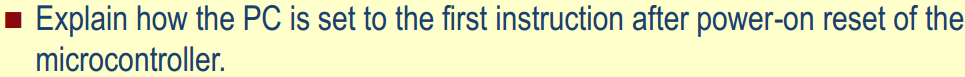
**[b] At how many possible base addresses can the I/O module be placed in the system? Can the module have the base address 0x2E4?**

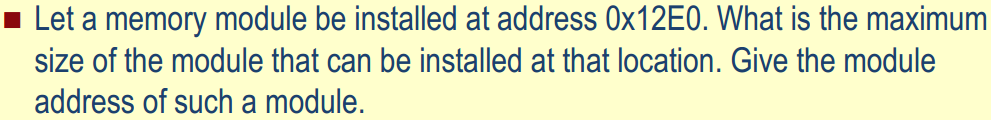
**[c] The I/O is to be placed at address 0x1F8. Find a Boolean equation for the address decoder that generates the CS signal. Draw the schematic.**

**MC\_04 p.41(40)**









MC\_05 p.9(9)