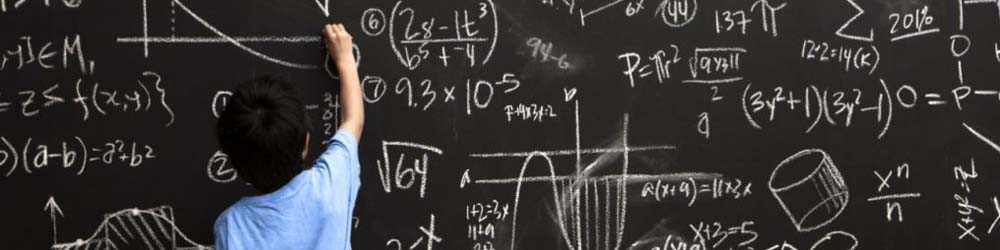
**Математические операции**



**Математика**

Одной из основных функций микроконтроллера является выполнение вычислений, как с числами напрямую, так и со значениями переменных. Начнём погружение в мир математики с самых простых действий:

* =

 присваивание

* +

 сложение

* -

 вычитание

* \*

 умножение

* /

 деление

* %

 остаток от деления

Рассмотрим простой пример:

int a = 10;

int b = 20;

int c = a + b; // c = 30

int d = a \* b; // d = 200

// так тоже можно

d = d / a; // d = 20

c = c \* d; // c = 600

По поводу последних двух строчек из примера, когда переменная участвует в расчёте своего собственного значения: существуют также **составные** операторы, укорачивающие запись:

* +=

 составное сложение:

a += 10

 равносильно

a = a + 10

* -=

 составное вычитание:

a -= 10

 равносильно

a = a - 10

* \*=

 составное умножение:

a \*= 10

 равносильно

a = a \* 10

* /=

 составное деление:

a /= 10

 равносильно

a = a / 10

* %=

 остаток от деления:

a %= 10

 равносильно

a = a % 10

С их использованием можно сократить запись последних двух строчек из предыдущего примера:

d /= a;

c \*= d;

Очень часто в программировании используется прибавление или вычитание единицы, для чего тоже есть короткая запись:

* ++

 (плюс плюс) инкремент:

a++

 равносильно

a = a + 1

* --

 (минус минус) декремент:

a--

 равносильно

a = a - 1

Порядок записи инкремента играет очень большую роль: пост-инкремент

var++

 возвращает значение переменной**до выполнения инкремента**. Операция пре-инкремента

++var

 возвращает значение **уже изменённой переменной**. Пример:

byte a, b;

a = 10;

b = a++;

// a получит значение 11

// b получит значение 10

a = 10;

b = ++a;

// a получит значение 11

// b получит значение 11

Как говорилось [в предыдущем уроке](https://alexgyver.ru/lessons/variables-types/) - локальные переменные нужно инициализировать, иначе в математических операциях получится непредсказуемый результат.

{

byte a; // просто объявляем

byte b = 0; // инициализируем 0

a++; // результат непредсказуем

b++; // результат 1

}

**Порядок вычислений**

Порядок вычисления выражений подчиняется обычным математическим правилам: сначала выполняются действия в скобках, затем умножение и деление, и в конце - сложение и вычитание.

**Скорость вычислений**

Математические вычисления выполняются процессором некоторое время, оно зависит от типа данных и типа операции. Вот время выполнения (в микросекундах) **не оптимизированных** компилятором вычислений **для Arduino Nano 16 МГц**:

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип данных** | **Время выполнения, мкс** | | |
| **Сложение и вычитание** | **Умножение** | **Деление, остаток** |
| int8\_t | 0.44 | 0.625 | 14.25 |
| uint8\_t | 0.44 | 0.625 | 5.38 |
|  |  |  |  |
| int16\_t | 0.89 | 1.375 | 14.25 |
| uint16\_t | 0.89 | 1.375 | 13.12 |
| int32\_t | 1.75 | 6.06 | 38.3 |
| uint32\_t | 1.75 | 6.06 | 37.5 |
|  |  |  |  |
| float | 8.125 | 10 | 31.5 |

* Нужно понимать, что не все во всех случаях математические операции занимают ровно столько времени, так как компилятор их **оптимизирует**. Можно помочь ему в этом, подробнее читайте в [уроке по оптимизации кода](https://alexgyver.ru/lessons/code-optimisation/).
* Операции с float выполняются гораздо дольше целочисленных, потому что в AVR нет аппаратной поддержки чисел с плавающей точкой и она реализована программно как сложная библиотека. В некоторых микроконтроллерах есть FPU - специальный аппаратный блок для вычислений с float.
* Операции целочисленного деления на AVR выполняются дольше по той же причине - они реализованы программно, а вот умножение и сложение с вычитанием МК делает аппаратно и очень быстро.

**Целочисленное деление**

При целочисленном делении результат **не округляется**по "математическим" правилам, дробная часть просто **отсекается**, фактически это округление вниз: и 9/10 и 1/10 дадут 0. При использовании float само собой получится 0.9 и 0.1. Если нужно **целочисленное деление с округлением вверх**, его можно реализовать так: вместо x / y записать (x + y - 1) / y. Рассмотренные выше примеры деления на 10 дадут результат 1.

Для округления по обычным математическим правилам можно использовать функцию round(), но она довольно тяжёлая, так работает с float.

**Переполнение переменной**

Вспомним предыдущий [урок о типах данных](https://alexgyver.ru/lessons/variables-types/): что будет с переменной, если её значение **выйдет из допустимого диапазона**? Тут всё весьма просто: при переполнении в бОльшую сторону из нового значения вычитается максимальное значение переменной, и у неё остаётся только остаток. Для сравнения представим переменную как ведро. Будем считать, что при наливании воды и заполнении ведра мы скажем стоп, выльем из него всю воду, а затем дольём остаток. Вот так и с переменной, что останется - то останется. Если переполнение будет несколько раз - несколько раз опорожним наше "ведро" и всё равно оставим остаток. Ещё один хороший пример - [кружка Пифагора](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D1%83%D0%B6%D0%BA%D0%B0_%D0%9F%D0%B8%D1%84%D0%B0%D0%B3%D0%BE%D1%80%D0%B0).

При переполнении в обратную сторону (выливаем воду из ведра), будем считать, что ведро полностью заполнилось. Да, именно так =) Посмотрим пример:

// тип данных byte (0.. 255)

byte val = 255;

// тут val станет равным 0

val++;

// а тут из нуля станет 246

val -= 10;

// переполним! Останется 13

val = 525;

// и обратно: val равна 236

val = -20;

**Особенность больших вычислений**

Для **сложения и вычитания** по умолчанию используется ячейка 4 байта (long), но для **умножения и деления** - 2 байта (int). Если при умножении или делении в текущем действии результат превысит 32768 - ячейка переполнится и мы получим некорректный результат. Для исправления ситуации нужно привести тип переменной к long перед вычислением, что заставит МК выделить дополнительную память. Например

a = (long)b \* c;

Для **цифр** существуют модификаторы, делающие то же самое:

* U или u - перевод в uint16\_t (от 0 до 65'535). Пример: 36000u
* L или l - перевод в int32\_t (-2 147 483 648… 2 147 483 647). Пример: 325646L
* UL или ul - перевод в uint32\_t (от 0 до 4 294 967 295). Пример: 361341ul

Посмотрим, как это работает на практике:

long val;

val = 2000000000 + 6000000; // посчитает корректно (т.к. сложение)

val = 25 \* 1000; // посчитает корректно (умножение, меньше 32'768)

val = 35 \* 1000; // посчитает НЕКОРРЕКТНО! (умножение, больше 32'768)

val = (long)35 \* 1000; // посчитает корректно (выделяем память (long) )

val = 35 \* 1000L; // посчитает корректно (модификатор L)

val = 35 \* 1000u; // посчитает корректно (модификатор u)

val = 70 \* 1000u; // посчитает НЕКОРРЕКТНО (модификатор u, результат > 65535)

val = 1000 + 35 \* 10 \* 100; // посчитает НЕКОРРЕКТНО! (в умножении больше 32'768)

val = 1000 + 35 \* 10 \* 100L; // посчитает корректно! (модификатор L)

val = (long)35 \* 1000 + 35 \* 1000; // посчитает НЕКОРРЕКТНО! Второе умножение всё портит

val = (long)35 \* 1000 + (long)35 \* 1000; // посчитает корректно (выделяем память (long) )

val = 35 \* 1000L + 35 \* 1000L; // посчитает корректно (модификатор L)

**Особенности float**

Помимо медленных вычислений, поддержка работы с float занимает много памяти, т.к. реализована в виде "библиотеки". Использование математических операций с float однократно добавляет примерно 1.5 кБ в память программы. С вычислениями есть такая особенность: если в выражении нет float чисел, то вычисления будут иметь целый результат (дробная часть **отсекается**). Для получения правильного результата нужно писать преобразование (float) перед действием, использовать float числа или float переменные. Также есть модификатор f, который можно применять только к цифрам float. *Смысла в нём нет, но такую запись можно встретить*. Смотрим:

float val; // далее будем присваивать 100/3, ожидаем результат 33.3333

val = 100 / 3; // посчитает НЕПРАВИЛЬНО (результат 33.0)

int val1 = 100; // целочисленная переменная

val = val1 / 3; // посчитает НЕПРАВИЛЬНО (результат 33.0)

float val2 = 100; // float переменная

val = val2 / 3; // посчитает правильно (есть переменная float)

val = (float)100 / 3; // посчитает правильно (указываем (float) )

val = 100.0 / 3; // посчитает правильно (есть число float)

val = 100 / 3.0f; // посчитает правильно (есть число float и модификатор)

При присваивании float числа целочисленному типу данных **дробная часть отсекается**. Если хотите математическое округление - его нужно использовать **отдельно**:

int val;

val = 3.25; // val станет 3

val = 3.92; // val станет 3

val = round(3.25); // val станет 3

val = round(3.92); // val станет 4

Следующий важный момент: из за особенности самой модели "чисел с плавающей точкой" - вычисления **иногда** производятся с небольшой погрешностью. Смотрите (значения выведены через порт):

float val2 = 1.1 - 1.0;

// результат 0.100000023 !!!

float val4 = 1.5 - 1.0;

// результат 0.500000000

Казалось бы, val2 должна стать ровно 0.1 после вычитания, но в 8-ом знаке вылезла погрешность! **Будьте очень внимательны**[при сравнении](https://alexgyver.ru/lessons/conditions/) float чисел, особенно со строгими операциями <=: результат может быть некорректным и нелогичным.