MODERON

2023

Руководство по языку программирования «Moderon Logic»

Руководство по языку программирования «Moderon Logic»

Содержание:

1. Введение

Среда разработки «Moderon Logic» представляет собой расширение редактора Visual Studio Code для поддержки текстового языка программирования **EESPL** (Easy Embedded Systems Programming Language).

Расширение включает в себя:

- набор инструментов для разработки (компилятор, отладчик, загрузчик, редактор ресурсов);
- анализатор кода (подсветка синтаксиса; проверка ошибок; поиск определений; определения типов данных; поиск ссылок, символов; переименование символов; навигация по коду; авто дополнение кода; подсказки для типов и имен переменных);
- базовые библиотеки функций для работы с ПЛК, шаблоны.

Синтаксис языка EESPL является C и Swift подобным. Язык адаптирован под программирование ПЛК. Имеет следующие возможности:

- работа с переменными, массивами, константами, перечислениями, ссылками;
- математические операторы;
- работа с циклами, итерации по массивам;
- создание своих типов, структур, классов (без наследования);
- создание и перегрузка функций;
- лямбда выражения;
- стандартная библиотека (математические функции, работа с памятью, прочее утилиты);
- встроенная в язык работа с ПЛК:
- работа с периферией (реле, дискретные/аналоговые входа, аналоговые выхода, дисплей, кнопки, RS-485, Ethernet);
- работа с Modbus регистрами, как с обычными переменными;
- готовые виджеты для создания пользовательских страниц (работа с экраном с возможностью динамического изменения страниц);
- работа с ОС ПЛК (таймеры, задачи, часы реального времени, WatchDog таймер).

1.1. Язык

1.2. Начало работы

2. Основы языка и синтаксис

2.1. Переменные и константы. Типы данных.

Любая разрабатываемая программа обладает двумя качествами: она может хранить некоторые данные и может выполнять действия – логическую последовательность. Любая программа для ПЛК «Moderon» обязательно должна содержать две базовые функции: функция **init()** – выполняется один раз после первоначальной подачи питания на контроллер, и функция **loop()** – циклическая функция, которая непрерывно выполняется до тех пор, пока ПЛК работает. Для хранения данных в EESPL, как и в других языках программирования, используются переменные. Переменная представляет именованный участок памяти ПЛК, в котором хранится некоторое значение.

Переменные имеют имя и значение. Для определения глобальной переменной используется ключевое слово **var**. Например:

```
1 var name = "Tom"
```

В данном случае переменная имеет имя "name" и в качестве значения хранит строку "Tom". После определения переменной мы можем использовать ее, например, передать ее функцию printf, чтобы вывести ее значение в консоль разработчика:

```
1 var name = "Tom"
2 print(name)
```

Отличительной особенностью глобальных переменных является то, что мы можем изменять их значение многократно во время работы программы. Например:

```
1 var name = "Tom" // значение Tom
2 name = "Bob" // значение Bob
```

Кроме переменных для хранения данных в программе могут использоваться константы. Константы подобны переменным, они также хранят некоторое значение, за тем исключением, что определяются с помощью ключевого слова **let,** и мы не можем после их инициализации изменить их значение:

```
1 let name = "Tom" // значение Tom
2 // name = "Bob"  - так сделать нельзя, так как name - константа
```

Таким образом, если значение некоторой переменной в течении программы меняться не будет, то вместо этой переменной лучше использовать константу. Хорошим тоном в программировании является рекомендация использовать константы, когда нет необходимости в дальнейшем изменении данных.

Мы можем определить сразу несколько переменных и констант на одной строке. В этом случае они должны разделяться запятой:

```
1 var name = "Tom", surname = "Smith"
```

Правила именования

Переменные и константы должны иметь уникальные имена. Нельзя использовать в программе несколько переменных и (или) констант с одними и теми же именами.

Причем хорошей практикой является использование названий в так называемом "верблюжем регистре" или CamelCase. То есть названия начинаются с маленькой буквы. Если название состоит из нескольких слов, то только первое из них начинается с маленькой буквы. Например:

```
1 var personName = "Tom"
2 var personStreatAddress = "St. Patrick avenue, 20"
```

Типы данных

Каждая переменная или константа хранит в себе значение определенного типа. Например, выше использованная переменная name хранит строку:

```
1 var name = "Tom"
```

В языке EESPL используются следующие типы данных:

- **Int8**: представляет целые числа со знаком размером не более 8 бит (от -128 до 127)
- **UInt8**: представляет целые положительные числа размером не более 8 бит (от 0 до 255)
- **Int16**: представляет целые числа со знаком размером не более 16 бит (от -32768 до 32767)
- **UInt16**: представляет целые положительные числа размером не более 16 бит (от 0 до 65535)
- **Int32**: представляет целые числа со знаком размером не более 32 бита (от -2147483648 до 2147483647)
- **UInt32**: представляет целые положительные числа размером не более 32 бита (от 0 до 4294967295)
- **Float**: 32-битное число с плавающей точкой, содержит до 6 чисел в дробной части
- **Double**: 64-битное число с плавающей точкой, содержит до 15 чисел в дробной части
- **Bool**: представляет логическое значение true или false
- **String**: представляет строку
- **Char**: представляет отдельный символ

Тип переменных и констант может определен явно или неявно. Выше тип определялся неявно. Но мы также можем явным образом определить тип:

```
1 var age: Int = 32
2 var name: String = "Tom"
```

Определение переменной с типом происходит по шаблону:

```
1 var название_переменной: тип_переменной = значение_переменной
```

Также мы можем сначала определить переменную, а потом присвоить ей значение:

```
1  var name: String
2  name = "Tom"
```

Также можно определить сразу набор однотипных переменных:

```
1 var height, weight: Double
```

Неявная типизация

Если мы явным образом не указываем тип переменных и констант, то он автоматически выводится системой на основании хранящегося значения. Например:

```
1 var name = "Tom"
```

Здесь явным образом не указан тип переменной name, однако поскольку она хранит строку, то система будет рассматривать эту переменную как объект типа String. Или, например:

```
1 var age = 32
```

Здесь также явно не указан тип переменной, поэтому система будет рассматривать эту переменную как объект Int16, то есть целое число.

Однако при таком подходе следует учитывать, что EESPL не всегда выводит те типы, которые нам могут быть нужны. Например, все целые числа EESPL воспринимает как объекты типа Int, а дробные числа - как объекты типа Double. Это надо учитывать, чтобы не попасть в некорректные ситуации. Например:

```
1    var d = 3.4            // тип Double
2    var f : Float = 1.2
3    d = f                    // ! Ошибка - разные типы
```

В данном случае на основании присвоенного значения переменная d будет представлять тип Double, а переменная f представляет тип Float, поэтому при присвоении переменной d значения f мы получим ошибку.

EESPL является типобезопасным языком со строгой типизацией, поэтому после того, как для переменной будет установлен тип, мы его уже изменить не сможем. Так, в следующем случае мы столкнемся с ошибкой:

```
1    var age: Int
2    age = "Tom"
```

Ошибка возникает, так как переменная age ожидает число, а строка "Том" не является числом и не соответствует переменной age по типу.

Мы можем присваивать значение переменной или константы другой переменной:

```
1    var age: Int16 = 22
2    var years = age
```

2.2. Числовые типы. Операции с числами

Числовые данные представлены следующими типами:

- **Int8**: целое число со знаком размером не более 8 бит (от -128 до 127)
- **UInt8**: целое положительное число размером не более 8 бит (от 0 до 255)
- **Int16**: целое число со знаком размером не более 16 бит (от -32768 до 32767)
- **UInt16**: целое положительное число размером не более 16 бит (от 0 до 65535)
- **Int32**: целое число со знаком размером не более 32 бита (от -2147483648 до 2147483647)
- **UInt32**: целое положительное число размером не более 32 бита (от 0 до 4294967295)
- **Float**: 32-битное число с плавающей точкой, содержит до 6 чисел в дробной части
- **Double**: 64-битное число с плавающей точкой, содержит до 15 чисел в дробной части

Для работы с числами в EESPL можно использовать целочисленные литералы (типа 2, 3, 78) и литералы чисел с плавающей точкой, в которых разделителем между целой и дробной частью является точка,

например, 1.2, 3.14, 0.025. При этом все целочисленные литералы рассматриваются как значения типа Int, а все дробные литералы - как значения типа Double.

```
1   let a = 2   // Int
2   let b = 2.0 // Double
```

Если мы хотим присвоить числовой литерал переменным или константам типов, отличных от Int и Double, то компилятор может автоматически выполнять преобразование:

```
1   let n : Int16 = 10  // неявное преобразование от Int к Int16
2   let x : Float = 3.14 // неявное преобразование от Double к Float
```

Для числовых типов большое значение имеет размерность, то есть количество бит, которое данный тип содержит. Например, тип UInt8 не может хранить число больше чем 255. Поэтому у нас не получится присвоить переменной этого типа, например, число 1000:

```
1   var age: UInt8 = 1000 //здесь ошибка
```

Минимальное и максимальное значение для определенного числового типа можно получить с помощью констант min и max:

```
1 var minInt16 = Int16.min    // -32768
2 var maxInt16 = Int16.max    // 32767
3 var minUInt16 = UInt16.min  // 0
4 var maxUInt16 = UInt16.max  // 65535
```

Форматы записи числовых данных

По умолчанию EESPL работает с десятичной системой исчисления. Однако, как и многие другие языки программирования, он может работать и с другими системами:

- **десятичная**: числа используются так, как они есть, без каких-либо префиксов
- **двоичная**: перед числом используется префикс 0b
- **восьмеричная**: перед числом используется префикс 0o
- **шестнадцатеричная**: перед числом используется префикс 0x

Например, запишем число 10 во всех системах исчисления:

```
let decimalInt = 10
let binaryInt = 0b1010      // 10 в двоичной системе
let octalInt = 0o12         // 10 в восьмеричной системе
let hexInt = 0xA            // 10 в шестнадцатеричной системе
```

Для чисел с плавающей точкой возможна запись в двух системах: десятичной и шестнадцатеричной. Для упрощения записи длинных чисел в десятичной системе мы можем использовать символ e (экспонента).

Например:

```
var a = 1.25e2      // 125
var b = 1.25e-2     // 0.0125
```

Для записи чисел с плавающей точкой в шестнадцатеричной системе используется префикс p:

```
var a = 0xFp2   //15 * 2 в степени 2 или 60.0
var b = 0xFp-2  //15 * 2 в степени -2  или 3.75
```

EESPL обладает полноценным набором арифметических операций. Арифметические операции производятся над числами:

- +

Сложение двух чисел:

```
var a = 6
var b = 4
var c = a + b // 10
```

- -

Вычитание двух чисел:

```
var a = 6
var b = 4
var c = a - b // 2
```

- -

Унарный минус. Возвращает число, умноженное на -1:

```
1   var a = -6
2   var b = -a // 6
3   var c = -b // -6
```

- *

Умножение:

```
1   var a = 6
2   var b = 4
3   var c = a * b // 24
```

- /

Деление:

```
1   var a = 8
2   var b = 4
3   var c = a / b // 2
```

При делении стоит учитывать, какие данные участвуют в делении и какой результат мы хотим получить. Например, в следующем случае выполняется деление дробных чисел:

```
1   let n : Double = 10
2   let d : Double = 4
3   let x : Double = n / d // 2.5
```

Результатом операции чисел Double является значение типа Double, которое равно 2.5. Но если мы возьмем значения типа Int, то результат будет иным:

```
1   let n : Int = 10
2   let d : Int = 4
3   let x : Int = n / d // 2
```

Оба операнда операции представляют тип Int, поэтому результатом операции является значение типа Int. Оно не может быть дробным, поэтому дробная часть отбрасывается, и мы получаем не 2.5, а число 2.

- %

Возвращает остаток от деления:

```
var a = 10
var b = 4
var c = a % b // 2
```

При арифметических операциях надо учитывать, что они производятся только между переменными одного типа. Например, в следующем примере мы получим ошибку:

```
var a: Int8 = 10
var b: Int32 = 10
var c = a + b
```

a и b должны в данном случае представлять один и тот же тип.

И также арифметические операции возвращают объект того же типа, к которому принадлежат операнды операции. Например, в следующем примере мы получим ошибку:

```
var a: Int8 = 10
var b: Int8 = 10
var c: Int32 = a + b
```

В данном случае переменная c, как и a и b, также должна представлять тип Int8.

Ряд операций сочетают арифметические операции с присваиванием

- +=

Присвоение со сложением, прибавляет к текущей переменной некоторое значение:

```
var a = 6
a += 10
printf(a)    // 16
// эквивалентен
// a = a + 10
```

- -=

Присвоение с вычитанием, вычитает из текущей переменной некоторое значение:

```
var a = 10
a -= 6
printf(a)    // 4
// эквивалентно
// a = a - 6
```

- *=

Присвоение с умножением, умножает текущую переменную на некоторое значение, присваивая ей результат умножения:

```
var a = 10
```

```
a *= 6
printf(a)    // 60
// эквивалентно
// a = a * 6
```

- /=

Присвоение с делением, делит значение текущей переменной на другое значение, присваивая результат деления:

```
var a = 10
a /= 2
printf(a)    // 5
// эквивалентно
// a = a / 2
```

- %=

Присвоение с остатком от делением, делит значение текущей переменной на другое значение, присваивая переменной остаток от деления:

```
var a = 10
a %= 4
printf(a)    // 2
// эквивалентно
// a = a % 4
```

2.3. Преобразование числовых данных

В арифметических операциях все операнды должны представлять один и тот же тип данных. Результатом операции является значение того же типа, что и тип операндов:

```
var a: Int8 = 10
var b: Int8 = 8
var c: Int8 = a + b
printf(c)        // 18
```

Однако в силу различных причин не всегда операнды представляют один и тот же тип. И также не всегда тип переменной или константы, которой присваивается результат операции, совпадает с типом операндов. И в подобных случаях преобразование будет произведено неявным образом.

Если операнды операций представляют числовые литералы, которые относятся к разным типам, то EESPL автоматически выполняет преобразование:

```
let x = 10/3.0
printf(x) // 3.33333333333333
```

В данном случае числовой литерал 10 представляет тип Int, а 3.0 - тип Double. В этом случае первый литерал преобразуется к типу Double, и выполняется деление.

Однако если операнды представляют константы или переменные или являются результатами каких-то других операций или выражений, то в этом случае нам нужно явно выполнять преобразование типов. Для

этого применяются специальная функция-инициализатор типов данных : **CastTo**(*$**mun**$* ***значение***). Первый аргумент функции совпадает с названиями типов данных: Int8, UInt8, Float, Double и т.д. Так, аргумент **$double$** преобразует значение к типу Double, а вторым аргументом передается само значение. Например:

```
let d = 8         // представляет тип Int
let g = CastTo($double$ d) // преобразуем в тип Double
printf(g)  // 8.0
```

При этом мы не можем напрямую передать значение типа int переменной типа Double, несмотря на то, что вроде бы обе переменных после преобразования все равно содержат фактически число 8:

```
let d = 8         // представляет тип Int
let g: Double = d // ! Ошибка - разные типы
```

То есть в данном случае нам обязательно надо использовать явное преобразование типов: let g = Double(d)

Преобразование типов может помочь, если операнды представляют разные типы. Например, в следующем случае мы столкнемся с ошибкой:

```
let n = 10
let d = 3.0
let x = n / d // ошибка компиляции
```

Чтобы код заработал, используем явное приведение типов:

```
let n = 10
let d = 3.0
let d = 3.0
```

В данном случае значение константы n преобразуется в Double, и так как константа d также представляет тип Double, то ошибки не возникнет, а константа x также будет представлять тип Double.

Либо мы могли бы преобразовать второй операнд к типу Int:

```
let n = 10
let d = 3.0
let x = n / Int(d)  // x = 3
```

Так как оба операнда теперь представляют тип Int, то и константа x будет представлять Int и будет равна 3.

Другой пример: сумма значений Int8 присваивается переменной типа Int32, то есть отличается тип операндов от типа результата:

```
var a: Int8 = 10
var b: Int8 = 10
var c: Int32 = a + b    // Error
```

Здесь результатом операции является значение типа Int8, так как оба операнда представляют этот тип. И несмотря на то, что это целое число, мы не можем его просто присвоить переменной типа Int32. Необходимо выполнить преобразование типов:

```
var a: Int8 = 10
var b: Int8 = 10
var c: Int32 = Int32(a) + Int32(b)
```

В этом случае переменный a и b преобразуются к типу Int32.

2.4. Поразрядные операции с числами

Поразрядные или побитовые операции выполняются над отдельными разрядами целых чисел. Каждое число имеет определенное представление в памяти. Например, число 5 представлено в двоичном виде следующим образом: 101. Число 7 - 111.

В EESPL есть следующие поразрядные операции:

- & (операция логического умножения или операция И)

сравнивает два соответствующих разряда двух чисел и возвращает 1, если соответствующие разряды обоих чисел равны 1. Иначе возвращает 0.

```
let a = 6        // 110
let b = 5        // 101
let c = a & b    // 100 - 4
printf(c)         // 4
```

В данном случае число a равно 6, то есть 110 в двоичном формате, а число b равно 5 или 101 в двоичном формате. В итоге при выполнении операции получится число 4 или 100.

- | (операция логического сложения или операция ИЛИ)

возвращает 1, если хотя бы один из соответствующих разрядов обоих чисел равен 1. Иначе возвращает 0.

```
let a = 6        // 110
let b = 5        // 101
let c = a | b    // 111 - 7
printf(c)         // 7
```

- ^ (операция исключающее ИЛИ)

возвращает 1, если соответствующие разряды обоих чисел не равны.

```
let a = 6        // 110
let b = 5        // 101
let c = a ^ b    // 011 - 3
printf(c)         // 3
```

- ~ (операция инверсии или операция НЕ)

принимает один операнд и инвертирует все его биты. Если разряд равен 1, то он получает значение 0 и наоборот, если разряд равен 0, то он становится равным 1.

```
let a = 6        // 0000000000000110
let b = ~a       // 1111111111111001
printf(b)         // -7
```

- << (поразрядный сдвиг влево)

Сдвигает биты первого операнда влево на количество разрядов, заданное вторым операндом. То есть первый операнд справа дополняется нулями, количество которых равно второму операнду.

```
let a = 6        // 110
let b = 2
let c = a << b   // 110 << 2 = 11000
printf(c)         // 24
```

- >> (поразрядный сдвиг вправо)

Сдвигает биты первого операнда вправо на количество разрядов, заданное вторым операндом.

```
let a = 13       // 1101
let b = 2
let c = a >> b   // 1101 >> 2 = 11
printf(c)         // 3
```

И также есть ряд операций, которые сочетают поразрядные операции с присваиванием:

- &=: присвоение после поразрядной операции И.

Присваивает левому операнду результат поразрядной конъюнкции его битового представления с битовым представлением правого операнда: A &= B эквивалентно A = A & B

- |=: присвоение после поразрядной операции ИЛИ.

Присваивает левому операнду результат поразрядной дизъюнкции его битового представления с битовым представлением правого операнда: A |= B эквивалентно A = A | B

- ^=: присваивание после операции исключающего ИЛИ.

Присваивает левому операнду результат операции исключающего ИЛИ его битового представления с битовым представлением правого операнда: A ^= B эквивалентно A = A ^ B.

- <<=: присваивание после сдвига разрядов влево.

Присваивает левому операнду результат сдвига его битового представления влево на определенное количество разрядов, равное значению правого операнда: A <<= B эквивалентно A = A << B

- >>=: присваивание после сдвига разрядов вправо.

Присваивает левому операнду результат сдвига его битового представления вправо на определенное количество разрядов, равное значению правого операнда: A >>= B эквивалентно A = A >> B.

Применение операций:

```
var a = 13       // 1101
a &= 5       // 1101 & 0101 = 0101
printf(a)         // 5
a |= 6           // 0101 | 0110 = 0111
printf(a)         // 7
a ^= 4           // 111 ^ 100 = 011
printf(a)         // 3
a <<= 3    // 11 << 3 = 11000
printf(a)         // 24
a >>= 1    // 11000 >> 1 = 1100
printf(a)         // 12
```

2.5. Строки. Типы Char и String

Для работы с текстом применяются два типа данных: Char и String. Char представляет отдельный символ, а String - строку из нескольких символов. При этом надо отметить, что String - это не просто набор объектов Char, это отдельный тип, который по функциональности отличается.

Самый простой способ определения строки и символов представляет использование строковых литералов - значений в двойных кавычках:

```
var a: Char = "a"
var hello: String = "hello"
```

В отличие от строки в переменную типа Char мы не можем засунуть больше одного символа, то есть в следующем случае мы столкнемся с ошибкой:

```
var a: Char = "abc"
```

Строки могут быть пустыми, то есть по сути не содержать ничего, но при этом быть инициализированными:

```
var str1: String = ""
var str2: String = String()
```

Пустая строка создается при помощи присвоения "" или при помощи инициализатора String()

Управляющие последовательности

Строка может содержать специальные символы, которые называются управляющими последовательностями или эскейп-последовательностями. Основные из них:

- \n: перевод на новую строку
- \t: табуляция
- \": кавычка
- \\: обратный слеш

Например, мы хотим определить многострочный текст, в котором будут использоваться кавычки и обратные слеши:

```
let text = "ООО \"Рога и копыта 777\" \nГен. директор: Иванов"
printf(text)
```

Moderon Logic интерпретирует данную строку следующим образом:

ООО "Рога и копыта 777"
Ген. директор: Иванов

Но, для создания многострочных строк можно заключать строку в тройные кавычки """:

```
let text = """
ООО "Рога и копыта 777"
Ген. директор: Иванов
"""
```

```
5    printf(text)
```

Результат будет тот же, что и в примере выше. Кроме того, внутри многострочных строк кавычки не надо экранировать с помощью обратного слеша.

Преобразование в строку

Для преобразования значений других типов данных к строке надо передать преобразуемое значение в инициализатор:

```
1    var number: Int = 22
2    var str: String = String(number)     // "22"
```

Конкатенация строк

Для конкатенации (объединения) строк используется оператор + (плюс):

```
1    var str: String = ""     // ""
2    str += "hello"           // "hello"
3    str = str + " world"     // "hello word"
```

Также можно объединить строки и числовые значения, например, для вывода на экран статических показаний:

```
1    var age: Int = 22
2    var str: String = "Age: " + age      // "Age: 22"
3
4    var weight: Double = 70.58
5    str = "Age: " + age + " and weight: " + weight       // "Age: 22 and
weight: 70.58"
```

2.6. Тип Bool. Условные выражения

Тип Bool представляет логическое значение true (истина) или false (ложь). То есть объект Bool может находиться в двух состояниях:

```
1    var isEnabled: Bool = true
2    isEnabled = false
```

Объекты типа Bool нередко являются результатом условных выражений, то есть таких выражений, которые представляют некоторое условие, и в зависимости от истинности условия возвращают true или false: true - если условие истинно и false - если условие ложно.

Операции сравнения

Операции сравнения сравнивают два значения и в зависимости от результата сравнения возвращают объект типа Bool: true или false.

- ==

Операция равенства. Сравнивает два значения, и если они равны, возвращает true:

```
var a = 10
var b = 10
var c = a == b
printf(c)    // true, так как a равно b
var d = 8
c = a==d
printf(c) // false, так как a не равно d
```

- **!=**

Операция неравенства. Сравнивает два значения, и если они неравны, возвращает true:

```
var a = 10
var b = 10
var c = a != b
printf(c)    // false, так как a равно b
var d = 8
c = a!=d
printf(c) // true, так как a не равно d
```

- >

Сравнивает два значения и возвращает true, если первое значение больше второго:

```
var a = 15
var b = 10
var c = a > b
printf(c)    // true, так как a больше чем b
var d = 8
c = d > a
printf(c) // false, так как d меньше чем a
```

- <

Сравнивает два значения и возвращает true, если первое значение меньше второго:

```
var a = 15
var b = 10
var c = a < b
printf(c)    // false, так как a больше чем b
var d = 8
c = d < a
printf(c) // true, так как d меньше чем a
```

- >=

Сравнивает два значения и возвращает true, если первое значение больше или равно второму:

```
var a = 10
var b = 10
var c = a >= b
printf(c)    // true, так как a равно b
var d = 8
c = d >= a
printf(c) // false, так как d меньше чем a
```

- <=

Сравнивает два значения и возвращает true, если первое значение меньше или равно второму:

```
var a = 15
var b = 10
var c = a <= b
printf(c)    // false, так как a больше чем b
var d = 9
c = d <= a
printf(c) // true, так как d меньше a
```

Логические операции

Логические операции выполняются над объектами типа Bool и в качестве результата также возвращают объект Bool.

- !

Логическое "НЕ" или операция отрицания. Она инвертирует значение объекта: если он был равен true, то операция возвращает false, и наоборот:

```
var isEnabled: Bool = true
var result = !isEnabled // false
```

- &&

Логическое "И" или операция логического умножения. Она возвращает true, если оба операнда операции имеют значение true:

```
let isEnabled: Bool = true
let isAlive = true
let result = isEnabled && isAlive   // true - так как оба операнда равны true

let a: Bool = true
let b: Bool = false
let c: Bool = true
let d = a && b && c // false, так как b = false
```

- ||

Логическое "ИЛИ" или операция логического сложения. Она возвращает true, если хотя бы один из операндов операции равен true:

```
var isEnabled: Bool = true
var isAlive = false
isEnabled || isAlive    // true, так как isEnabled равен true

var a: Bool = true
var b: Bool = false
var c: Bool = false
a || b || c // true, так как a = true
```

Нередко логические операции объединяют несколько операций сравнения:

```
let a = 10
let b = 12
let c = a > 8 && b < 10
let d = a > 8 || b < 10
printf(c)    // false
```

```
6   printf(d)     // true
```

2.7. Структуры

Структуры представляют набор значений разного типа, которые рассматриваются как один объект. Для создания структуры используются фигурные скобки, внутри которых записываются все элементы структуры:

```
1 struct Timer {
2    xIn:bool
3    uiPt:uint32
4    xQ:bool
5    uiEt:uint32
6    lastTime:uint32
7 }
```

В данном примере мы создаем пользовательскую структуру данных для удобного использования в дальнейшем коде программы. Обращение к переменных внутри структуры происходит с использованием точки.

```
1 var userTimer : Timer
2 userTimer.xIn = true
3 Print(userTimer.xIn) // true
```

2.8. Условная конструкция If. Тернарный оператор

При разработке программы часто бывает необходимо произвести некоторые действия в зависимости от выполнения тех или иных условий. Если определенные условия соблюдаются, то выполнить одни действия, если не соблюдаются, то выполнить другие. И для этого в языках программирования применяются условные конструкции. Рассмотрим, какие условные конструкции есть в языке EESPL.

Конструкция if/else

Конструкция if проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код:

```
1 if условие {
2   // набор действий
3 }
```

Например:

```
1  let num1 = 22
2  let num2 = 15
3  if num1 > num2{
4
5print("num1 больше чем num2")
6}
```

Здесь если первое число больше второго, то сработает весь код в блоке if, который располагается между открывающей и закрывающей фигурными скобками. Если же первое число окажется меньше второго, тогда действия в конструкции if работать не будут,

Важно, что после слова if в этой конструкции должно идти значение типа Bool. Поскольку результат операции сравнения как раз возвращает логическое значение, то в данном случае ошибок не возникнет.

Однако если мы укажем после if просто число или строку, то программа завершится ошибкой, как в следующем случае:

```
let num1 = 22

let num2 = 15

if num1{
print("num1 больше чем num2")
}
```

Если при проверке условия нам надо выполнить какие-либо альтернативные действия, то мы можем использовать блок else:

```
let num1 = 22
let num2 = 15
if num1 > num2{

    print("num1 больше чем num2")
}
else{
    print("num1 меньше чем num2")
}
```

Но при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. Используя конструкцию else if, мы можем обрабатывать дополнительные условия:

```
let num1 = 22
let num2 = 15
if num1 > num2{

    print("num1 больше чем num2")
}
else if (num1 < num2){

    print("num1 меньше чем num2")
}
else{

     print("num1 и num2 равны")
}
```

Тернарный оператор

Тернарный оператор аналогичен простой конструкции if и имеет следующий синтаксис:

```
[первый операнд - условие] ? [второй операнд] : [третий операнд]
```

В зависимости от условия тернарный оператор возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Например:

```
var num1 = 10
var num2 = 6
var num3 = num1 > num2 ? num1 - num2 : num1 + num2
```

В данном случае num3 будет равно 4, так как num1 больше num2, поэтому будет выполняться второй операнд: num1 - num2.

2.9. Циклы

Цикл for-in

С помощью цикла for-in мы можем перебрать элементы коллекции (массивы, множества, словари) или последовательности. Он имеет следующую форму:

```
for объект_последовательности in последовательность {

    // действия
}
```

Например, переберем элементы массива:

```
for item in 1...5 {

    print(item)
}
```

Выражение 1 to 5 образует последовательность из пяти чисел от 1 до 5. И цикл проходит по всем элементам последовательности. При каждом проходе он извлекает одно число и передает его в переменную item. Таким образом, цикл сработает пять раз.

С помощью оператора where можно задавать условия выборки из последовательности элементов:

```
for i in 0...10 where i % 2 == 0 {

    print(i) // 0, 2, 4, 6, 8, 10
}
```

Здесь из последовательности 0...10 извлекаются только те элементы, которые соответствуют условию после оператора where - i % 2 == 0, то есть четные числа.

Цикл while

Оператор while проверяет некоторое условие, и если оно возвращает true, то выполняет блок кода. Этот цикл имеет следующую форму:

```
while условие {

    // действия
}
```

Например:

```
var i = 10
while i > 0 {

    print(i)
    i-=1
}
```

При этом надо внимательно подходить к условию. Если оно всегда будет возвращать true, то мы

Цикл repeat-while

Цикл repeat-while сначала выполняет один раз цикл, и если некоторое условие возвращает true, то продолжает выполнение цикла. Он имеет следующую форму:

```
repeat {

    // действия

} while условие
```

Например, перепишем предыдущий цикл while:

```
var i = 10

repeat {

    print(i)
    i-=1
} while i > 0
```

Здесь цикл также выполнится 10 раз, пока значение переменной i не станет равно 0.

Но рассмотрим другую ситуацию:

```
var i = -1

repeat {

    print(i)
    i-=1
} while i > 0
```

Несмотря на то, что переменная i меньше 0, но цикл выполнится один раз.

Операторы continue и break

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором break.

```
for i in 0...10 {
    if i == 5{
        break
    }
    print(i) // 0, 1, 2, 3, 4
}
```

Поскольку в цикле идет проверка, равно ли значение переменной i числу 5, то когда перебор дойдет до числа 5, сработает оператор break, и цикл завершится.

Теперь поставим себе другую задачу. А что если мы хотим, чтобы при проверке цикл не завершался, а просто переходил к следующему элементу. Для этого мы можем воспользоваться оператором continue:

```
1 for i in 0...10 {
2     if i == 5{
3         continue
4     }
5     print(i) // 0, 1, 2, 3, 4, 6, 7, 8,, 9, 10
6 }
```

В этом случае цикл, когда дойдет до числа 5, которое не удовлетворяет условию проверки, просто пропустит это число и перейдет к следующему элементу последовательности.

2.10. Функции

Функция представляет набор инструкций, который имеет имя (имя функции) и может использоваться повторно в различных местах программы. Функция имеет следующее формальное определение:

```
1 func имя_функции (параметры) -> тип_возвращаемого_значения {
2
3     // набор инструкций
4 }
```

Сначала идет ключевое слово func, после которого идет имя функции. Для именования функции применяются в принципе те же правила, что и при именовании переменных. Для имени функции используется режим camel case.

После имени функции в скобках указываются параметры. Если параметров нет, то просто идут пустые скобки.

Если функция возвращает какое-либо значение, то после параметров в скобках идет стрелка и тип возвращаемого значения. И в конце в фигурных скобках собственно идет блок кода, который и представляет функцию.

Определим простейшую функцию:

```
1 func printName(){
2
3     print( "Меня зовут Том")
4 }
```

Здесь функция называется printfName. Эта функция ничего не возвращает, поэтому тут после скобок сразу идут фигурные скобки с набором операторов. Данная функция просто выводит строку «Меня зовут Том».

Далее по имени функции мы можем ее вызвать многократно. Для вызова функции указывается ее имя, после которого в скобках перечисляются значения для ее параметров:

```
1 func printName(){
2
3     print("Меня зовут Том")
4 }
5
6 printName()
7 printName()
8 printName()
```

В частности, здесь функция вызывается три раза.

Теперь используем параметры:

```
1 func printInfo(name: String, age: Int){
2
3    print("Имя: \(name) ; возраст: \(age)")
4 }
5
6 printInfo(name: "Tom", age: 18)    // Имя: Tom ; возраст: 18
7 printInfo(name: "Bob", age: 35)    // Имя: Bob ; возраст: 35
```

Количество параметров может быть произвольным. В данном случае мы используем два параметра: для передачи имени и возраста. Для каждого параметра определено имя и тип. Например, первый параметр называется name и имеет тип String.

При вызове функции нам надо учитывать имя и тип параметров. При вызове функции ей необходимо передать значения для всех ее параметров по имени. То есть указывается имя параметра и через двоеточие его значение: name: "Tom", причем передаваемое значение должно соответствовать параметру по типу:

```
1 printfInfo(name:"Tom", age: 18)
```

Хотя мы можем указать, что при вызове функции не надо указывать имена параметров при ее вызове. Для этого перед именем параметра нужно поставить подчеркивание. Между подчеркиванием и названием параметра должен идти пробел:

```
1 func printInfo(_ name: String, _ age: Int){
2
3    print("Имя: \(name) ; возраст: \(age)")
4 }
5
6 printInfo("Tom", 18)     // Имя: Tom ; возраст: 18
```

При вызове подобной функции значения параметрам передаются по позиции. При этом опять же передаваемое значение должно соответствовать параметру по типу.

Значения параметров по умолчанию

Если мы определили два параметра в функции, то соответственно при ее вызове мы также должны передать ей два значения для параметров. Однако мы также можем установить для параметров значения по умолчанию:

```
1 func printInfo(name: String = "Tom", age: Int = 22){
2
3    print("Имя: \(name) ; возраст: \(age)")
4 }
5
6 printInfo(name: "Bob", age: 18)    // Имя: Bob ; возраст: 18
7 printInfo(name: "Alice")  // Имя: Alice ; возраст: 22
8 printInfo() // Имя: Tom ; возраст: 22
```

И если при вызове функции мы не передадим для какого-то параметра значение, то этот параметр будет использовать значение по умолчанию.

2.11. Возвращение функцией значения

Функция в EESPL может возвращать некоторое значение или результат. В прошлой теме были определены функции, которые ничего не возвращают:

```
1   func printfHello(){ printf("Hello world")}
```

Такая функция ничего не возвращает, возвращаемый тип не указан. Хотя фактически она будет эквивалентна следующей функции:

```
1   func printfHello() -> Void { printf("Hello world")}
2   func printfHello() -> () { printf("Hello world")}
```

Тип Void указывает, что функция фактически ничего не возвращает.

Теперь напишем функцию, которая бы возвращала какое-либо значение:

```
1   func sum (_ x: Int, _ y: Int) -> Int{
2   return x + y
3   }
4
5   printf(sum(4,5))      // 9
6   printf(sum(5,6))      // 11
```

Итак, здесь определена функция sum, которая возвращает сумму двух чисел. В качестве возвращаемого типа указан тип Int.

Если функция возвращает какое-либо значение, отличное от Void, то в теле функции нам надо использовать оператор return. После этого оператора ставится возвращаемое значение.

В данном случае мы возвращаем значение Int, значит, после return обязательно должно располагаться значение типа Int или выражение, которое возвращает объект Int. Важно также понимать, что после вызова оператора return работа функции завершается, поэтому после выражения return нет смысла ставить какие-либо еще инструкции.

Поскольку функция возвращает некоторое значение, то мы можем присвоить это значение какой-либо переменной / константе и затем использовать в программе. Единственное, надо учитывать, что тип возвращаемого значения функции должен совпадать с типом переменной / константы:

```
1   func sum (_ x: Int, _ y: Int) -> Int{
2   return x + y
3   }
4   let a = sum(4,5)
5   let b = sum(10, 23)
```

2.12. Вложенные и рекурсивные функции

Вложенные функции

Одни функции могут содержать другие функции. Вложенные функции еще называются локальными. Локальная функция доступна только в рамках той функции, внутри которой она определена. Как правило, локальная функция оформляет блок действий, которые могут многократно использоваться в рамках внешней функций, но вне этой внешней функции больше нигде не используются.

Например, определим функцию, которая вычисляет разницу площадей двух окружностей:

```
func compare(_ r1: Double, _ r2: Double){

    func square(_ r: Double) -> Double{ return r * r * 3.14}

    let s1 = square(r1)
let s2 = square(r2)

printf("разница площадей:", (s1 - s2))
}

compare(16.0, 15.0)
```

И чтобы не писать повторно один и тот же код для каждой окружности, можно определить одну функцию и затем многократно ее вызывать. И если не планируется использовать эту функцию в других частях программы вне функции compare, то ее можно определить как локальную.

Рекурсивные функции

Рекурсивные функции представляют особые функции, которые могут вызывать сами себя. Традиционным примером подобных функций служат функции вычисления факториала и чисел Фибоначчи. Например, функция факториала:

```
func factorial(_ n: Int) -> Int{

    if n == 0{

return 1
    }

    return n * factorial(n-1)
    }

var x = factorial(6)     // 720
```

Здесь функция факториала возвращает 1, если переданное в функцию число равно 0. Иначе функция вызывает саму себя.

Другой пример - функция вычисления чисел Фибоначчи:

```
func fibbonachi(_ n: Int) -> Int{

    if n == 0{

return 0
    }
else if n == 1{

return 1
    }
    return fibbonachi(n-1) + fibbonachi(n-2)
}
var z = fibbonachi(6)    // 8
```

2.13. Перегрузка функций

В EESPL нам доступен механизм перегрузки функций, то есть мы можем определять функции с одним и тем же именем, но разным количеством или типом параметров:

```
1   func sum(_ x: Int, _ y : Int){
2       printf(x+y)
3   }
4   func sum(_ x: Int, _ y: Int, _ z: Int ){
5   printf(x+y)
6   }
7
8   func sum(_ x: Double, _ y: Double){
9   printf(x + y + z)
10  }
11
12  sum(1, 2)            // 3
13  sum(1.2, 2.3)        // 3.5
14  sum(2, 3, 4)         // 9
```

В данном случае все три функции называются sum, но так как они отличаются либо по количеству параметров, либо по их типу. При вызове данной функции EESPL по типу и количеству параметров сможет распознать, какую именно версию функции sum надо использовать.

Также перегруженные версии одной функции могут отличаться по типу возвращаемого значения:

```
1   func sum(_ x: Int, _ y : Int) -> Int{
2   return x + y
3   }
4   func sum(_ x: Int, _ y : Int) -> Double{
5   return 2 * Double(x + y)    // преобразует результат в Double
6   }
7
8   let a : Int = sum(1, 2)      // 3
9   let b : Double = sum(1, 2)   // 6.0
10
11  printf(a)     // 3
12  printf(b)     // 6.0
```

В данном случае определены две версии функции sum, которые совпадают по типу и количеству параметров, но различаются по типу возвращаемого значения.

Константа a представляет тип Int, поэтому в выражении let a : Int = sum(1, 2) компилятор увидит, что необходима версия, которая возвращает значение Int, и будет использовать эту версию. Аналогично в выражении let b : Double = sum(1, 2) константа b представляет тип Double, поэтому здесь будет применяться версия функции sum, которая возвращает значение Double. То есть в данном случае у нас ошибок не возникнет.

Однако теперь рассмотрим другую ситуацию:

```
1   func sum(_ x: Int, _ y : Int) -> Int{
2   return x + y
3   }
4   func sum(_ x: Int, _ y : Int) -> Double{
5   return 2 * Double(x + y)    // преобразует результат в Double
6   }
7
8   let a = sum(1, 2)    // Ошибка
9   let b = sum(1, 2)    // Ошибка
```

Здесь явным образом не указано, какой тип представляют константы a и b, поэтому их тип будет выводиться изрезультата вызова sum(1, 2). Но теперь компилятор не знает, какую именно версию функции sum использовать, так как тип констант неизвестен. В итоге в данном случае мы столкнемся с ошибкой.

2.14. Перечисления

Перечисление (enumeration) определяет общий тип для группы связанных значений. Причем сами объединенные в перечисление значения могут представлять любой тип - число, строку и так далее.

Для создания перечисления используется ключевое слово **enum**:

```
enum {
BUTTON_LEFT = 0,
BUTTON_RIGHT,
BUTTON_DOWN,
BUTTON_UP,
BUTTON_ESC,
BUTTON_ENTER
}
```

В данном случае перечисление называется Season и представляет времена года и имеет четыре значения. То есть фактически Season представляет новый тип данных.

Также допустима сокращенная форма перечисления значений:

```
enum Season{
case Winter, Spring, Summer, Autumn
}
```

2.15. Массивы

Массив представляет упорядоченную коллекцию элементов одного типа, к которым можно обратиться по индексу - номеру элемента в массиве. Индекс представляет число - объект типа Int и начинается с нуля. По сути массив представляет собой обычную переменную или константу, которая хранит несколько объектов наподобие кортежа.

Объявление массива

Объявление массива имеет следующие формы записи:

```
// Полная форма
var имяМассива: Array<Тип>
// Краткая форма
var имяМассива: [Тип]
```

Например:

```
var numbers: [Int]
```

Здесь объявлен массив numbers, который хранит объекты типа Int.

Но просто объявить массив недостаточно. Его, как и любую другую переменную, надо инициализировать перед использованием. То есть определить для него начальное значение. Все значения, которые входят в массив, перечисляются в квадратных скобках: [элемент1, элемент2, элемент3, ...]. Например:

```
var numbers: [Int] = [1, 2, 3, 4, 5]
var numbers2: Array<Int> = [1, 2, 3, 4, 5]
print(numbers)
```

Здесь массивы numbers и numbers2 содержит по 5 элементов. Мы также можем при объявлении не указывать тип массива, в этом случае система сама выведет тип исходя из входящих в него элементов:

```
var numbers = [1, 2, 3, 4, 5]
```

Мы также можем определить пустой массив:

```
var numbers = [Int]()
// или так
var numbers2 : [Int] = []
print("В массиве numbers \(numbers.count) элементов")    // В массиве
numbers 0 элементов
```

В таком массиве будет 0 элементов. С помощью свойства count можно получить количество элементов в массиве.

Доступ к элементам массива

Каждый элемент в массиве имеет определенный индекс, по которому мы можем получить или изменить элемент:

```
var numbers = [11, 12, 13, 14, 15]
print(numbers[0])    // 11
numbers[0] = 21
print(numbers[0])    // 21
```

Для обращения к элементу массива после названия массива в квадратных скобках используется индекс элемента: numbers[0]

В данном случае у нас пять элементов в массиве, индексация в массивах начинается с нуля, поэтому первый элемент всегда имеет индекс 0, а последний элемент в данном случае будет иметь индекс 4. Если же мы попытаемся обратиться к элементу с большим индексом, например:

```
print(numbers[5])    // ошибка
```

То мы получим ошибку.

Если нам надо изменить несколько элементов подряд, то мы можем использовать операцию последовательности для записи индексов:

```
var numbers = [5, 6, 7, 8, 3]
numbers[1...3] = [105, 106, 103]
print(numbers)  // 5, 105, 106, 103, 3
```

В данном случае выражение 1...3 указывает на набор индексов от 1 до 3. И таким образом, элементам с этими индексами мы можем присвоить значения.

Размер массива

С помощью свойства count можно получить число элементов массива:

```
var numbers: [Int] = [1, 2, 3, 4, 5, 6, 7, 8]
print("В массиве numbers \(numbers.count) элементов")    // В массиве numbers 8 элементов
```

В дополнение к нему свойство **isEmpty** позволяет узнать, пуст ли массив. Если он пуст, то возвращается значение true:

```
var numbers: [Int] = [1, 4, 8]
if numbers.isEmpty {
    print("массив пуст")
} else {
     print("в массиве есть элементы")
}
```

Перебор массива

С помощью цикла for можно перебрать элементы массива:

```
var numbers: [Int] = [1, 2, 3, 4, 5, 6, 7, 8]

for i in numbers {
    print(i) // 1, 2, 3, 4, 5, 6, 7, 8,
}
```

Перебор массива через индексы:

```
var numbers: [Int] = [1, 2, 3, 4, 5, 6, 7, 8]

for i in 0 ..< numbers.count {
    print(numbers[i]) // 1, 2, 3, 4, 5, 6, 7, 8,
}
```

Вместо применения цикла также можно использовать метод **forEach(),** которые перебирает все элементы. В качестве параметра этот метод принимает функцию, которая производит действия над текущим перебираемым элементом:

```
var numbers: [Int] = [1, 2, 3, 4, 5, 6, 7, 8]
numbers.forEach({print($0)})
```

В данном случае передается анонимная функция, которая также с помощью функции print выводит значение элемента.

С помощью метода **enumerated()** можно одновременно получить индекс и значение элемента:

```
var names: [String] = ["Tom", "Alice", "Kate"]

names.enumerated().forEach({print("\($0) - \($1)")})

for (index, value) in names.enumerated() {
    print("\(index) - \(value)")
}
```

Создание массива из последовательности

Специальная форма инициализатора в качестве параметра принимает последовательность, из которой создается массив:

```
var numbers = Array (1...5)      // [1, 2, 3, 4, 5]
var numbers2 = [Int] (3..<7)     // [3, 4, 5, 6]

print(numbers)    // [1, 2, 3, 4, 5]
print(numbers2)    // [3, 4, 5, 6]
```

Еще одна форма инициализатора позволяет инициализировать массив определенным числом элементов одного значения:

```
var numbers = [Int] (repeating: 5, count: 3)
// или так
var numbers2 = Array (repeating: 5, count: 3)
// эквивалентно массиву var numbers: [Int] = [5, 5, 5]
print(numbers)    // [5, 5, 5]
```

Однако стоит учитывать, что если подобным образом создается массив из объектов классов - ссылочных типов, то все элементы массива будут хранить ссылку на один и тот же элемент в памяти:

```
class Person{
    var name: String
    init(name: String){
        self.name = name
    }
}
let tom = Person(name: "Tom")

var people = Array (repeating: tom, count: 3)

people[0].name = "Bob"

for person in people{
    print(person.name)
}

// Bob
// Bob
// Bob
```

Сравнение массивов

Два массива считаются равными, если они содержат одинаковые элементы на соответствующих позициях:

```
var numbers: [Int] = [1, 2, 3, 4, 5]
let nums = [1, 2, 3, 4, 5]

if numbers == nums{
    print("массивы равны")
}
else {
    print("массивы не равны")
}
```

В данном случае массивы numbers и nums имеют одинаковое количество элементов, и все элементы на соответствующих позициях равны, поэтому оба массива равны.

3. Системные функции

3.1. Встроенные системные функции

CastTo($type$ val) -> type

Данная функция предназначена для явного приведения типа данных переменной в случае необходимости. Возвращает преобразованную переменную нужного типа.

Имя аргумента	Тип данных	Описание

$type$	type	Тип данных, к которому приводим исходную переменную. Обязательно обрамление знаками $
val	VAR	Имя переменной, которая приводится к желаемому типу

Возвращает преобразованную переменную нужного типа.

Пример применения:

```
1 var Heater: bool = true
2 var IntHeater: uint16
3 IntHeater  = CastTo($uint16$ Heater)  // 1
```

SizeOf(val or $type$) -> uint32

Данная функция предназначена для вычисления размера типа или выражения в байтах.

Имя аргумента	Тип данных	Описание
val	type	Переменная или стандартный тип данных, для которой производится вычисление размера

Данная функция возвращает значение типа uint32

Пример:

```
1 Var myValue : uint16 = 30000
2 Var TestSize = SizeOf(myValue)  // 2 байта
3 TestSize = SizeOf($bool$)  // 1  байт
```

Length (arr) -> uint32
Данная функция используется для вычисления количества элементов массива, обычно применяется при переборе элементов массива циклом for

Имя аргумента	Тип данных	Описание
arr	Массив любого типа	Массив данных, количество элементов которого необходимо вычислить

Данная функция возвращает значение типа uint32

Пример:

```
1 var myArray = [1,2,3,4,5]
2 let arrSize = Length(myArray) // 5
```

CreateIMG(name, path) -> void

Данная функция добавляет изображение в проект программы, для дальнейшего вывода на экране.

Имя аргумента	Тип данных	Описание
name	VAR	имя переменной для изображения (png/jpg/bmp and etc)
path	String	путь файла

Данная функция ничего не возвращает

Пример:

```
1 CreateIMG(img_ctrl, "./../../../../res/control.png")
```

Далее, для вывода на экран, используем специальную функцию вывода **image(**img_ctrl**, true)** $$. Описание этой функции находится в разделе 5 данного руководства.

CreateFont(name, path, size)-> void

Данная функция добавляет текстовый шрифт в проект программы, для дальнейшего использования при отрисовке пользовательских страниц.

Имя аргумента	Тип данных	Описание
name	VAR	имя переменной для шрифта (myFont)
path	string	путь до шрифта (.ttf) (“C:\Kinco\font.ttf”)
size	Uint32	высота шрифта в пикселях (12,18,22 и т.д.)

Данная функция не возвращает значения

Пример:

```
1 CreateFont(fnt1, "./../../../../res/russia.ttf", 18)  // создаем шрифт fnt1
```

Применение шрифтов подробно описано в разделе 5 данного руководства.

MODBUS_ADD_LOCAL_H_REG(name, addr, min, max, default, USE_EEPROM)

Функция служит для создания локальной сетевой переменной в раздел памяти Holding Modbus Registers.

Имя аргумента	Тип данных	Описание
name	VAR	Имя регистра, по нему можно обращаться в программе для чтения или записи регистра
addr	Uint16	Адрес регистра (0… 0xFFFF). Если указать auto, то адрес высчитыватся автоматически (+1 к адресу предыдущего созданного регистра)
min	Uint16	Минимально допустимое значение
max	Uint16	Максимально допустимое значение
default	Uint16	Значение регистра по умолчанию
USE_EEPROM	Uint16	Указатель, является ли данный регистр энергонезависимым (0 = нет, 1 = да)

Данная функция не возвращает значения

Пример:

```
MODBUS_ADD_LOCAL_H_REG(var1, auto, 11, 0x100-41, 2, 0)
MODBUS_ADD_LOCAL_H_REG(auto, 2, 12, 0x100-42, 3, 1)
MODBUS_ADD_LOCAL_H_REG(auto, 3, 13, 0x100-43, 4, 1)
MODBUS_ADD_LOCAL_H_REG(var4, 4, 14, 0x100-44, 5, 0)
MODBUS_ADD_LOCAL_H_REG(var5, 5, 15, 0x100-45, 6, 1)
MODBUS_ADD_LOCAL_H_REG(auto, 10, 16, 0x100-46, 7, 1)
MODBUS_ADD_LOCAL_H_REG(auto, auto, 17, 0x100-47, 8, 0)
```

MODBUS_ADD_LOCAL_I_REG(name, addr, default)

Данная функция служит для создания локального регистра в таблицу Inputs Modbus Registers. Данный тип регистра доступен извне только для чтения.

Имя аргумента	Тип данных	Описание
name	VAR	Имя регистра, по нему можно обращаться в программе для чтения или записи регистра
addr	Uint16	Адрес регистра (0… 0xFFFF). Если указать auto, то адрес высчитыватся автоматически (+1 к адресу предыдущего созданного регистра)
default	Uint16	Значение регистра по умолчанию

Данная функция не возвращает значения

Пример:

```
MODBUS_ADD_LOCAL_I_REG(variable1, 0 , 2)  // адрес 0, значение по умолчанию 2
MODBUS_ADD_LOCAL_I_REG(variable2, auto, 3)  // адрес 1
MODBUS_ADD_LOCAL_I_REG(variable3, auto, 4)
```

```
MODBUS_ADD_LOCAL_I_REG(variable4, auto, 5)
MODBUS_ADD_LOCAL_I_REG(variable5, auto, 6)
MODBUS_ADD_LOCAL_I_REG(input6, 13, 7)  // 13 адрес
MODBUS_ADD_LOCAL_I_REG(input7, auto, 8)
```

MB_READ_LOCAL_REG(regType, addr) -> uint16

Данная функция возвращает текущее значение ранее созданного локального регистра Modbus.

Имя аргумента	Тип данных	Описание
regType	uint32	Тип регистра (**MB_L_HOLDING** или **MB_L_INPUT**)
Addr	uint16	Адрес регистра (0… 0xFFFF). Можно использовать имя переменной

Данная функция возвращает значение Uint16

Пример :

```
let var5_val: uint16 = MB_READ_LOCAL_REG(MB_L_HOLDING, var5)
let TestVal = var5 // то же самое, что и в предыдущей строке
let var1_val: uint16  = MB_READ_LOCAL_REG(MB_L_HOLDING, 0x01)
let vari6_val: uint16  = MB_READ_LOCAL_REG(MB_L_INPUT, input6)
```

MB_WRITE_LOCAL_REG(regType, regAddr, val)

Данная функция присваивает значение ранее созданному локальному регистру Modbus.

Имя аргумента	Тип данных	Описание
regType	uint32	Тип регистра (**MB_L_HOLDING** или **MB_L_INPUT**)
regAddr	uint16	Адрес регистра (0...0xFFFF). Можно использовать имя переменной
val	uint16	Новое значение регистра

Данная функция не возвращает значения

Пример:

```
MB_WRITE_LOCAL_REG(MB_L_HOLDING, var5, 0xAA55)  // записать в регистр var5 значение 0xAA55
MB_WRITE_LOCAL_REG(MB_L_INPUT, input6, 0xAA55)
input6 = 35 // еще один альтернативный вариант изменения регистра в программе
```

MB_GET_ERROR(UART) -> Uint32

Данная функция предназначена для получения кода ошибки, когда порт **RS485** используется как **MASTER**

Имя аргумента	Тип данных	Описание
UART	uint32	Номер порта RS485 (0 или 1)

Данная функция возвращает значение типа uint32

UART_CFG (PORT_NAME, baudRate, parity, stopBits)

Данная функция предназначена для параметризации и настройки сетевых параметров порта **RS485.** Вызов данной функции производится при инициализации ПЛК.

Имя аргумента	Тип данных	Описание
PORT_NAME	uint32	Имя порта (UART_COM_1 или UART_COM_2)
baudRate	uint32	Скорость соединения (4800, 9600, 19200, 38400 и т.д.)
parity	uint32	Четность порта (0 = NONE, 1 = EVEN, 2 = ODD)
stopBits	Uint32	Количество стоп бит (1 , 2)

Данная функция ничего не возвращает

Пример:

```
UART_CFG(UART_COM_1, 115200, UART_PARITY_EVEN, UART_TWO_STOPBIT)
```

MODBUS_CFG (PORT_NAME, isMaster, timeout, devAddr) -> void

Данная функция настраивает указанный порт RS-485 Modbus

Имя аргумента	Тип данных	Описание
PORT_NAME	uint32	Настраиваемый порт (UART_COM_1/UART_COM_2)
isMaster	uint32	Использовать порт в режиме Master
timeout	uint32	Ожидание опроса для режима Slave. Если поставить 0, то таймаута не будет
devAddr	Uint8	Адрес устройства (для режима Slave)

Данная функция не возвращает значения

Пример:

```
MODBUS_CFG(UART_COM_1, MB_SLAVE_ID, 1000, 0x02) // настроили порт как Slave
// указали таймаут ожидания 1000 мс и адрес устройства - 2
```

RELAY_SET(pinNum, level) -> void

Данная функция необходима для управления физическим дискретным выходом ПЛК

Имя аргумента	Тип данных	Описание
pinNum	uint32	Номер выходного реле (нумерация идет с 0, например, DO2 = 1, DO5 = 4 и т.д.)
level	Uint8	1 - включить выход, 0 – выключить выход

Данная функция не возвращает значения

RELAY_TOGGLE(pinNum) -> void

Данная функция переключает указанный дискретный выход на значение, противоположное текущему

Имя аргумента	Тип данных	Описание
pinNum	Uint32	Номер выходного реле (нумерация идет с 0, например, DO2 = 1, DO5 = 4 и т.д.)

Данная функция не возвращает значения

RELAYS_TOGGLE(pinMask) -> void

Данная функция переключает сразу все указанные дискретные выхода в битовой маске

Имя аргумента	Тип данных	Описание
pinMask	Uint32	Маска реле (1 << RELAY_1) | (1 << RELAY_3)

Данная функция не возвращает значения

RELAYS_SET(pinMask, level) -> void

Данная функция активирует либо деактивирует все указанные дискретные выходы в битовой маске

Имя аргумента	Тип данных	Описание
pinMask	Uint32	Маска реле (1 << RELAY_1) | (1 << RELAY_3)
level	Uint8	1 - включить выход, 0 – выключить выход

Данная функция не возвращает значения

AO_OUTPUT(pinNum, level) -> void

Данная фукнция задает значение указанному аналоговому выходу 0-10 В

Имя аргумента	Тип данных	Описание
pinNum	Uint32	Номер выхода (AO_1, ...)
level	Uint32	Выходное напряжение в мВ (0-10000)

Данная функция не возвращает значения

UI_READ_DI (pinNum) -> Uint32

Данная функция предназначена для считывания универсального входа, настроенного как дискретный (DI)

Имя аргумента	Тип данных	Описание
pinNum	Uint32	Номер входа (UI_1, ...)

Данная функция возвращает значение Uint32

UI_READ_AI (pinNum) -> Uint32

Данная функция предназначена для считывания универсального входа, настроенного как аналоговый (токовый или температурный датчик)

Имя аргумента	Тип данных	Описание
pinNum	Uint32	Номер входа (UI_1, ...)

Данная функция возвращает значение Uint32

UI_READ_ALL_DI (pinMask) -> Uint32

Функция позволяет считать сразу все нужные универсальные входы, которые настроены как дискретные, используя битовую маску

Имя аргумента	Тип данных	Описание
pinMask	Uint32	Маска входов (1 << UI_1) | (1 << UI_3)

Данная функция возвращает значение Uint32

UI_CFG (pinNum, mode) -> void

Данная функция позволяет настроить тип универсального входа

Имя аргумента	Тип данных	Описание
pinNum	Uint32	Номер настраиваемого входа (UI_1, ...)
mode	Uint32	Режим входа (UI_MODE_DI/UI_MODE_AI/UI_MODE_TEMP)

Данная функция не возвращает значения

UI_CFG_SET_RANGE(pinNum, min, max, offset) -> void

Данная функция нужна для настройки диапазона значения входов, которые сконфигурированы как токовые (4…20 мА)

Имя аргумента	Тип данных	Описание
pinNum	Uint32	Номер настраиваемого входа (UI_1, ...)
min	Uint32	Нижняя граница диапазона, значение при входном токе в 4 мА
max	Uint32	Верхняя граница диапазона, значение при входном токе 20 мА
offset	Uint32	Поправочный коэффициент. Данное число прибавляется к вычисленному значению

Данная функция не возвращает значения

UI_CFG_SET_B(pinNum, beta) -> void

Данная функция необходима, если универсальный вход был сконфигурирован как термосопротивление. Термисторы характеризуются рядом параметров, такими, как максимальный допустимый ток, точность, сопротивление при определённой температуре (как правило, при 25°C). Одним из параметров, характеризующим степень изменения сопротивления в зависимости от температуры является коэффициент температурной чувствительности, обозначаемый B. Этот коэффициент рассчитывается на основе значений сопротивления при двух конкретных значениях температур. Во многих случаях этими температурами выбираются 25°C и 100°C. Обычно температуры, использованные при вычислении коэффициента, указываются после буквы, например B3950.

Имя аргумента	Тип данных	Описание
pinNum	uint32	Номер настраиваемого входа (UI_1, ...)
beta	Uint32	Бетта-коэффициент подключаемого ко входу термосопротивления

Данная функция не возвращает значения

Пример:

```
UI_CFG_SET_B(UI_11, 3950)  // настраиваем коэффициент равным 3950
```

4. Библиотека функциональных блоков.

4.1. RS (FB) – классический RS триггер

Функциональный блок RS.

Блок представляет стандартный RS триггер, применяется для сохранения дискретного значения, например – для сохранения события или аварии. Приоритет у входа RESET.

Пример применения:

```
Var RSinst : RS // создаем экземпляр объекта
RSinst.xS = DI_1 // кнопка пуск
RSinst.xR = DI_2 // кнопка стоп с приоритетом
RS_func(RSinst) // функция обработки объекта
DO_1 = RSinst.xQ // состояние выхода блока
```

Область применения	Имя	Тип	Комментарий
Вход	xS	BOOL	Установливает xQ в значение TRUE
	xR	BOOL	Установливает xQ в значение FALSE (приоритет)
Выход	xQ	BOOL	Выходное значение

4.2. SR (FB) – триггер с приоритетом SET

Функциональный блок SR.

Блок представляет стандартный SR триггер, применяется для сохранения дискретного значения, например – для сохранения события или аварии. Приоритет у входа SET.

Пример:

```
Var RSinst : RS // создаем экземпляр объекта
RSinst.xS = DI_1 // кнопка пуск с приоритетом
RSinst.xR = DI_2 // кнопка стоп
SR_func(RSinst) // функция обработки триггера
DO_1 = RSinst.xQ // состояние выхода блока
```

Вход:

Область применения	Имя	Тип	Коментарий
Вход	xS	BOOL	Установливает xQ в значение TRUE (приоритет)

	xR	BOOL	Установливает xQ в значение FALSE
Выход	xQ	BOOL	Выходное значение

4.3. CTD (FB) – Счетчик вниз

Функциональный блок CTD.

Счетчик, уменьшает значение выхода на единицу каждый импульс на входе.

Пример:

```
CTDInst : Counter
CTDInst.xCU = DI_1 // считаем импульсы на входе DI1
CTDInst.iPV = 5 // начнем отсчет с числа 5
CTDInst.xR = DI_2 // этот вход сбросит выход снова до 5
CTD_func(CTDInst) // вызов функции счетчика
let CurrentValue = CTDIns.iCV // вывод текущего значения
```

Вход:

Область применения	Имя	Тип	Коментарий
Вход	xCD	BOOL	Каждый импульс уменьшает выход iCV на единицу
	xR	BOOL	TRUE: Сбросить iCV до значения iPV
	iPV	INT16	Начальное значение отсчета
Выход	xQ	BOOL	TRUE если iCV = 0
	iCV	INT16	Текущее значение счётчика

4.4. CTU (FB) – Счетчик вверх

Функциональный блок CTU

Счетчик, увеличивает значение выхода на единицу каждый импульс на входе.

Пример:

```
CTUInst : Counter
CTUInst.xCU = DI_1 // считаем импульсы на входе DI1
CTUInst.iPV = 5 // при достижении данного значения выход xQ станет равным 1
CTUInst.xR = DI_2 // этот вход сбросит выход iCV в нулевое значение
CTU_func(CTUInst) // вызов функции счетчика
let CurrentValue = CTUIns.iCV // вывод текущего значения
```

Вход:

Область применения	Имя	Тип	Коментарий
Вход	iCU	BOOL	Увеличивает iCV на единицу по фронту
	xR	BOOL	TRUE: Сброс iCV на 0
	iPV	WORD	Верхний предел для выхода xQ
Выход	xQ	BOOL	TRUE если CV > = PV
	iCV	WORD	Текущее значение счетчика

4.5. TOF (FB) – Таймер на отключение

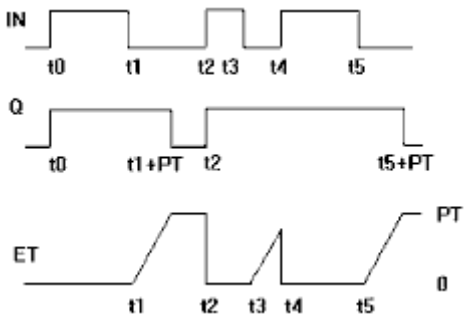
Функциональный блок TOF.

Представляет собой таймер с задержкой выключения.

Пример:

```
TOFInst : Timer
```

```
TOFInst.xIn = DI_1 // вход
TOFInst.uiPt = 5000 // время задержки в мс
TOF_func(TOFInst) // функция таймера
Let VarBOOL2 = TOFInst.xQ // присвоим выход таймера переменной
```



Вход:

Область применения	Имя	Тип	Коментарий
Вход	xIn	BOOL	Падение по фронту: запускает счетчик задержки Нарастающий фронт: сбрасывает счетчик задержки
	uiPt	UINT32	Время для таймера задержки [мс]
Выход	xQ	BOOL	TRUE, если xIn = TRUE FALSE если xIn = FALSE и время задержки uiPt истекло

	uiEt	UINT32	Сколько времени прошло с момента пропадания сигнала на входе xIn

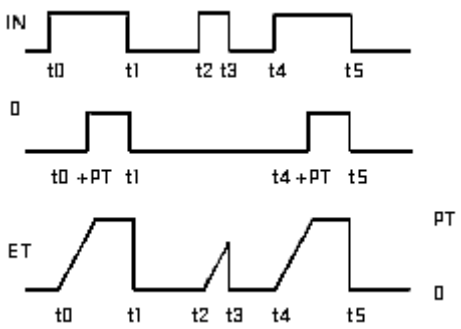
4.6. TON (FB) – Таймер на включение

Функциональный блок TON.

Представляет собой таймер с задержкой включения.

Пример:

```
TONInst : Timer
TONInst.xIn = DI_1 // вход
TONInst.uiPt = 5000 // время задержки в мс
TON_func(TONInst) // функция таймера
let VarBOOL2 = TONInst.xQ // присвоим выход таймера переменной
```



Вход:

Область применения	Имя	Тип	Коментарий
Вход	xIn	BOOL	Нарастающий фронт: запускает счетчик задержки Падение по фронту: сбрасывает счетчик задержки
	uiPt	Uint32	Время для счетчика задержки [мс]
Выход	xQ	BOOL	FALSE если xIn = FALSE TRUE через время uiPt после того, как xIn = TRUE
	uiEt	Uint32	Время, прошедшее с момента нарастания фронта на входе xIn

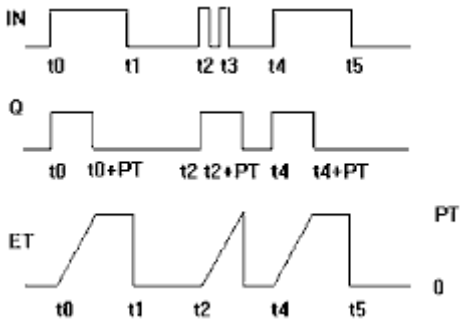
4.7. TP (FB) – Импульсный таймер

Функциональный блок TP.

Блок представляет собой импульсный таймер.

Пример:

```
TPInst : Timer
TPInst.xIn = DI_1 // вход
TPInst.uiPt = 5000 // время задержки в мс
TP_func(TPInst) // функция таймера
Let VarBOOL2 = TPInst.xQ // присвоим выход таймера переменной
```



Вход:

Область применения	Имя	Тип	Коментарий
Вход	xIn	BOOL	Восходящий фронт запускает импульсный таймер и устанавливает xQ в TRUE
	uiPt	Uint32	Длительность импульса (высокий сигнал на выходе xQ)
Выход	xQ	BOOL	При появлении нарастающего фронта на входе xIn данный выход будет в состоянии TRUE на время uiPt мс
	uiEt	Uint32	Время, прошедшее с момента запуска импульсного таймера.

4.8. F_TRIG (FB) – Триггер по нижнему фронту

Функциональный блок F_TRIG.

Выявляет спадающий фронт логического сигнала.

Пример:

```
FTRIGInst : EdgeTrig;
FTRIGInst.xCLK = DI_1
F_TRIG_func(FTRIGInst)
VarBOOL2 = FTRIGInst.xQ;
```

Вход:

Область применения	Имя	Тип	Коментарий
Вход	xCLK	BOOL	Логический сигнал, подлежащий проверке
Выход	xQ	BOOL	TRUE: Обнаружен падающий фронт

4.9. R_TRIG (FB) – Триггер по верхнему фронту

Функциональный блок R_TRIG.

Выявляет нарастающий фронт логического сигнала.

Пример:

```
RTRIGInst : EdgeTrig;
RTRIGInst.xCLK = DI_1
R_TRIG_func(RTRIGInst)
VarBOOL2 = RTRIGInst.xQ;
```

Вход:

Область применения	Имя	Тип	Коментарий
Вход	xCLK	BOOL	Логический сигнал, подлежащий проверке
Выход	xQ	BOOL	TRUE: Обнаружен восходящий фронт

5. Встроенные функции интерфейса

func DrawScreen_1(ctx: &EG_userContext_t) {

GUI_createContext() $$

Пример функции, в которой можно производить отрисовку

Пример:

```
func DrawScreen_1(ctx: &EG_userContext_t) {
```

GUI_setDraw(drawFunc: {(ctx: &EG_userContext_t)->void})

Обычная функция (без $$). Вызыватся для указания функции, которая будет отвечать за отрисовку.

Пример:

```
GUI_setDraw(DrawScreen_1)
```

GUI_buttonHandler(buttonPos: uint32, callback: {(isPressed: uint8)->uint8})

Обычная функция (без $$). Задает обработчик для конкретной кнопки контроллера внутри функции для отрисовки.

Имя аргумента	Тип данных	Описание
buttonPos	uint32	

callback	uint8

Данная функция возвращает значение uint32, uint8

Пример:

```
    if isPressed {
        Println("ESC -- pressed")
    } else {
        Println("ESC -- released")
    }
    return 0
})
```

label(text: string, alignment: uint32) $$

Функция обязательно вызывается в самом начале функции, которая отвечает за отрисовку.

Имя аргумента	Тип данных	Описание
text	string	
alignment	uint32	позиция текста относительно осей: GUI_ALIGN_CENTER, GUI_ALIGN_LEFT, GUI_ALIGN_RIGHT

Данная функция возвращает значение uint32, Uint8

Пример:

```
label("text", GUI_ALIGN_CENTER) $$ // выводит надпись text на экран
```

switcher(value: &uint32, bitPos: uint8) $$

Бинарный переключатель (управляет указаным битом в переменной)

Имя аргумента	Тип данных	Описание
value	Ссылка	Имя переменной, из которой берется бит данных
bitPos	uint8	Порядковый номер бита

Данная функция возвращает значение Uint8

Пример:

```
switcher(relays, i) $$
```

input_int(value: &int32, min: int32, max: int32, alignment: uint32, isEditable: uint8) $$

ввод/отображение целого числа из переменной. Если isEditable = false, то число только отбражается

Имя аргумента	Тип данных	Описание

value	ссылка
min	int32
max	int32
alignment	uint32
isEditable	uint8

Данная функция возвращает значение int32, uint32, uint8

Пример:

```
input_int(h_var, 0, 10000, GUI_ALIGN_CENTER, true) $$
```

input_float(value: &f32, min: f32, max: f32, decimalNumberSize: uint8, alignment: uint32, isEditable: uint8) $$

ввод/отображение дробного числа из переменной. Если isEditable = false, то число только отбражается

Имя аргумента	Тип данных	Описание
value	&f32	
min	f32	
max	f32	
decimalNumberSize	uint8	кол-во цифр после запятой
alignment	uint32	
isEditable	uint8	

Данная функция возвращает значение f32, uint32, uint8

Если value, min или max -- не float, то соответствующее значение переводится в float делением на 10.0 decimalNumberSize раз. Например, если decimalNumberSize = 1, а min = 325, то min переводится в 32.5

Пример:

```
input_float(ai, -60.0f, 80.f, 1, GUI_ALIGN_CENTER, false) $$
```

image(img: &IMG_dat_t, callback: {(uint8)->void}, isKeepAspectRatio: uint8)

Вывод изображения.

Имя аргумента	Тип данных	Описание
img	&IMG_dat_t	

callback	uint8	Функция, которая вызывается, если на изображение нажать. Аргумент callback можно опустить.
isKeepAspectRatio	uint8	Сохранять ли пропорции относительно вертикали.

Данная функция возвращает значение uint8, &IMG_dat_t,

Пример:

```
//image(img: &IMG_dat_t, isKeepAspectRatio: uint8)
//-- image(img_car, true)
/** image(img_car, $(e) {
  if e {
    Println("Pressed")
  } else {
    Println("Released")
  }
} true)
```

button(label: string, color: uint16, radius: uint8, callback: {(uint8)->void})

Вывод кнопки.

Имя аргумента	Тип данных	Описание
label	string	
color	Uint16	Цвет кнопки.
radius	uint8	Радиус закругления кнопки в процентах относительно высоты кнопки.
callback	uint8	Функция, которая вызывается, при нажати, отжатии кнопки.

Данная функция не возвращает значение.

Пример:

```
button("clickMe", GET_COLOR(255, 255, 255), 50, $(e) {
  if e {
    Println("Pressed")
  } else {
    Println("Released")
  }
} true)
```

fill(color: uint16)

Данная функция заливает доступное пространство цветом color.

Имя аргумента	Тип данных	Описание
color	uint16	

Данная функция возвращает значение uint16

Пример:

```
fill(GET_COLOR(255, 255, 255))
```

set_color(textColor: uint16, backgroundColor: uint16)

Данная функция задает цвет в текущем контексте для фона и шрифта.

Имя аргумента	Тип данных	Описание
textColor	uint16	
backgroundColor	uint16	

Данная функция возвращает значение uint16

Пример:

```
set_color(COLOR_BLACK, COLOR_WHITE) $$
```

set_font(font: &FNT_font_t)

Данная функция задает шрифт в текущем контексте.

Имя аргумента	Тип данных	Описание
font	&FNT_font_t	

Данная функция возвращает значение &FNT_font_t

Пример:

```
set_font(new_font12) $$
```

combo_box(var: &int32, color: uint16, list: string[], vals: int32[], uint8: isEditable)

Имя аргумента	Тип данных	Описание
var	&int32	
color	uint16	

list	string
vals	int32
isEditable	uint8

Данная функция возвращает значение uint16, &int32

Пример:

```
combo_box(CurrentSeason, COLOR_BLACK, enumList[21],2, false) $$
```

list(height: uint16, isDrawCursor: uint8)

Имя аргумента	Тип данных	Описание
height	uint16	
isDrawCursor	uint8	

Данная функция возвращает значение uint16, uint8.

Пример:

row(isSkip: uint8, colsCount: uint16)

Имя аргумента	Тип данных	Описание
isSkip	uint8	
colsCount	uint16	

Данная функция возвращает значение uint16, uint8

Пример:

```
row(!elemVisible, [70,30]) $
```

or row(isSkip: uint8, cols: uint16[])

Имя аргумента	Тип данных	Описание
isSkip	uint8	
cols	uint16	

Данная функция возвращает значение uint16, uint8.

layout(height: uint16, colsCount: uint16)

Имя аргумента	Тип данных	Описание
height	uint16	
colsCount:	uint16	

Данная функция возвращает значение uint16

Пример:

```
layout(12, [8,92]) $
```

or layout(height: uint16, colsCount: uint16[])

Имя аргумента	Тип данных	Описание
height	uint16	
colsCount	uint16	

Данная функция возвращает значение uint16