

Руководство по написанию скриптов MyLogic

Оглавление

Оглавление	1
Введение.....	2
Поддержка работы со скриптами в конфигураторе	3
Вкладка «Скрипты».....	3
Редактор скриптов	5
Создание скрипта.....	12
Структура скрипта	12
Комментарии	14
Переменные и константы	14
Массивы	16
Строки	17
Структуры	17
Операторы	18
Функции	22
Событийная модель	25
Передача параметров на сервер	27
Отладка скрипта	29
Директивы препроцессора	31
Аргументы скрипта	34
Приложения А. Встроенные функции платформы	36
Таблица поддерживаемых функций	36
«tracker.inc» - основные функции платформы	39
«camera.inc» – функции работы с камерой (только для УМКа303)	50
«geofence.inc» – функции работы с геозонами	52
«serial.inc» – функции работы с последовательными портами	53
«can.inc» – функции работы с CAN. Доступно для УМКа302 и УМКа303	56
«ble.inc» - функции работы с BLE	60
«modem.inc» - функции работы с модемом	64
Функции работы с временем	66
Приложение Б. Список прилагаемых файлов с описанием языка	68
Документация	68
Примеры	68

Введение

Для написания скриптов MyLogic используется простой, не типизированный 32-битный скриптовый язык программирования Pawn с Си-подобным синтаксисом. Скрипт компилируется в байт-код, который запускается внутри виртуальной машины.

Назначение: построение нетиповой логики работы устройства, поддержка специфического или редко используемого оборудования.

В данном документе описаны возможности MyLogic для следующих устройств:

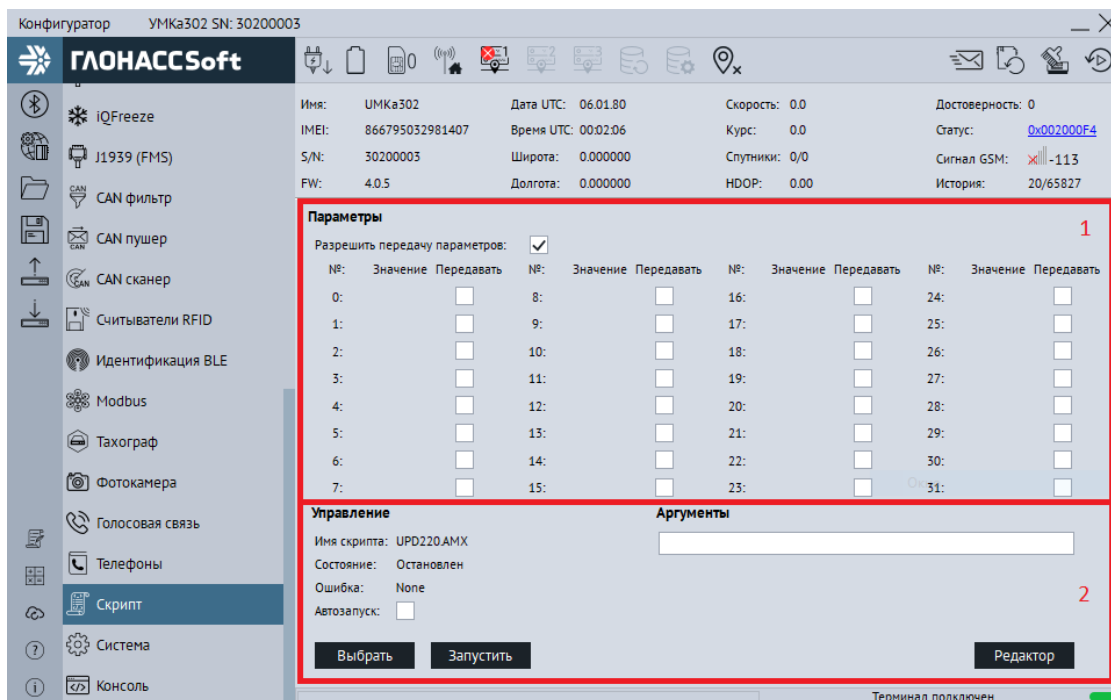
- УМКа302 версии ПО 4.3.2;
- УМКа303 версии ПО 0.5.6;
- УМКа310/УМКа310.В/УМКа311/УМКа312 версии ПО 1.6.1;
- УМКа314 версии ПО 0.11.0;
- УМКа315 версии ПО 2.3.6;

Таблица поддерживаемых функций MyLogic в зависимости от типа устройства и версии программного обеспечения приведена в приложении А.

Поддержка работы со скриптами в конфигураторе

Вкладка «Скрипты»

Для управления скриптами и передаваемыми параметрами используется вкладка «Скрипты»:



Условно страницу можно разделить на две части: управление передаваемыми параметрами (1) и управления скриптами (2).

Управление передаваемыми параметрами

Для передачи параметров на сервер необходимо: поставить галочки напротив «разрешить передачу параметров» (1) и напротив каждого передаваемого параметра (2). Не рекомендуется ставить галочки напротив номеров параметров использование и передача которых не планируется. Эта мера позволит увеличить максимальное количество точек в чёрном ящике устройства. Подробно о хранении и передаче параметров описано в разделе «Передача параметров на сервер».

Разрешить передачу параметров: ☒

№:	Значение	Передавать	№:	Значение	Передавать	№:	Значение	Передавать	№:	Значение	Передавать
0:	7	<input checked="" type="checkbox"/>	8:	0.0	<input checked="" type="checkbox"/>	16:		<input type="checkbox"/>	24:		<input type="checkbox"/>
1:	-74	<input checked="" type="checkbox"/>	9:	1.11	<input checked="" type="checkbox"/>	17:		<input type="checkbox"/>	25:		<input type="checkbox"/>
2:	968	<input checked="" type="checkbox"/>	10:		<input type="checkbox"/>	18:		<input type="checkbox"/>	26:		<input type="checkbox"/>
3:	0	<input checked="" type="checkbox"/>	11:		<input type="checkbox"/>	19:		<input type="checkbox"/>	27:		<input type="checkbox"/>
4:	45.063	<input checked="" type="checkbox"/>	12:		<input type="checkbox"/>	20:		<input type="checkbox"/>	28:		<input type="checkbox"/>
5:	38.995	<input checked="" type="checkbox"/>	13:		<input type="checkbox"/>	21:		<input type="checkbox"/>	29:		<input type="checkbox"/>
6:	31.900	<input checked="" type="checkbox"/>	14:		<input type="checkbox"/>	22:		<input type="checkbox"/>	30:		<input type="checkbox"/>
7:	170.9	<input checked="" type="checkbox"/>	15:		<input type="checkbox"/>	23:		<input type="checkbox"/>	31:		<input type="checkbox"/>

Если скрипт запущен и параметры достоверны, то напротив номера параметра будет отображаться его текущее значение (3).

Управление скриптами

В данной области отображаются имя текущего запущенного скрипта, его состояние и ошибка в случае её возникновения. Так же осуществляется управление автозапуском скрипта и задание аргументов (более подробно про аргументы скрипта описано в разделе «Аргументы скрипта»).

Управление скриптами осуществляется кнопками «Выбрать» и «Запустить/Остановить».

Управление	Аргументы
Имя скрипта: UPD220.AMX	3
Состояние: Выполняется	
Ошибка: None	
Автозапуск: <input type="checkbox"/>	
Выбрать	Редактор
Остановить	

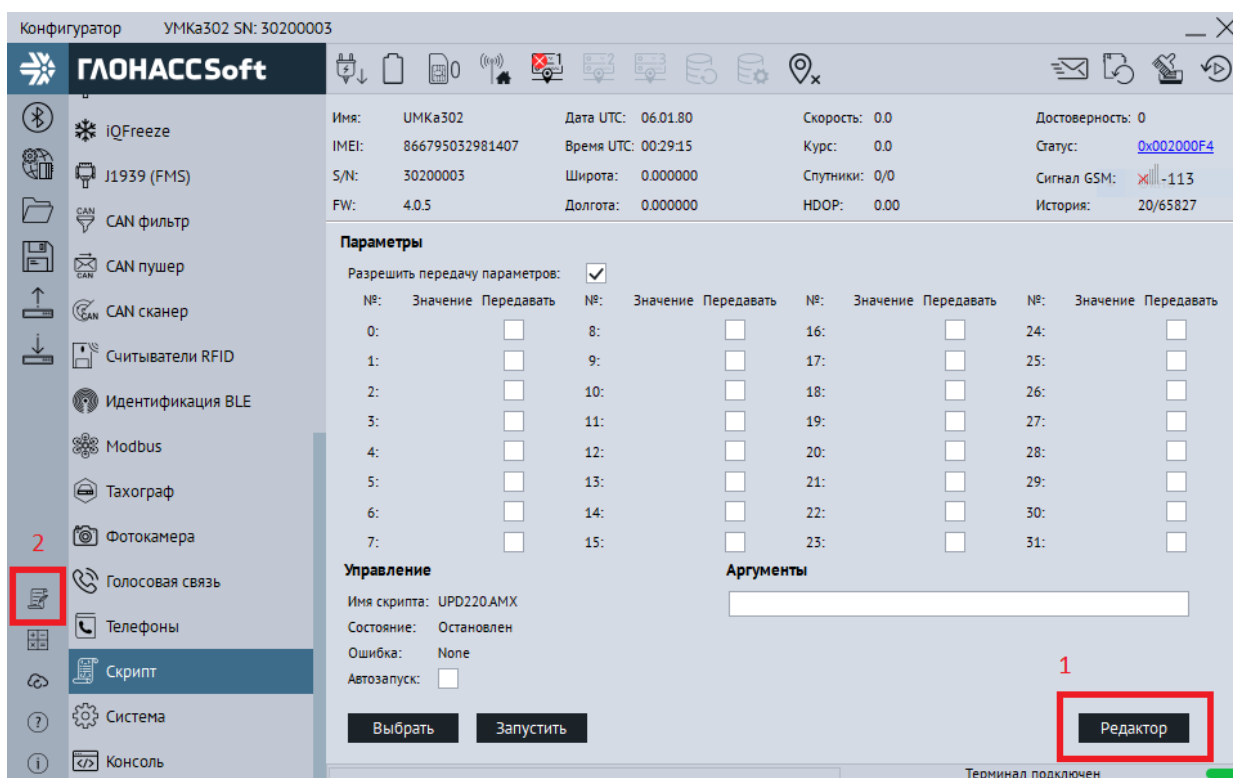
Для выбора скрипта Нажмите на кнопку «Выбрать». В появившемся окне (выбор скрипта), если необходимый файл ещё не загружен, нажмите на **+** и укажите путь к файлу скрипта.

ВАЖНО! В устройствах используется формат имён файла 8.3 (максимум восемь символов для имени файла и максимум три символа для расширения). Скрипт с именем, не соответствующим данному требованию, не будет загружен в память устройства.

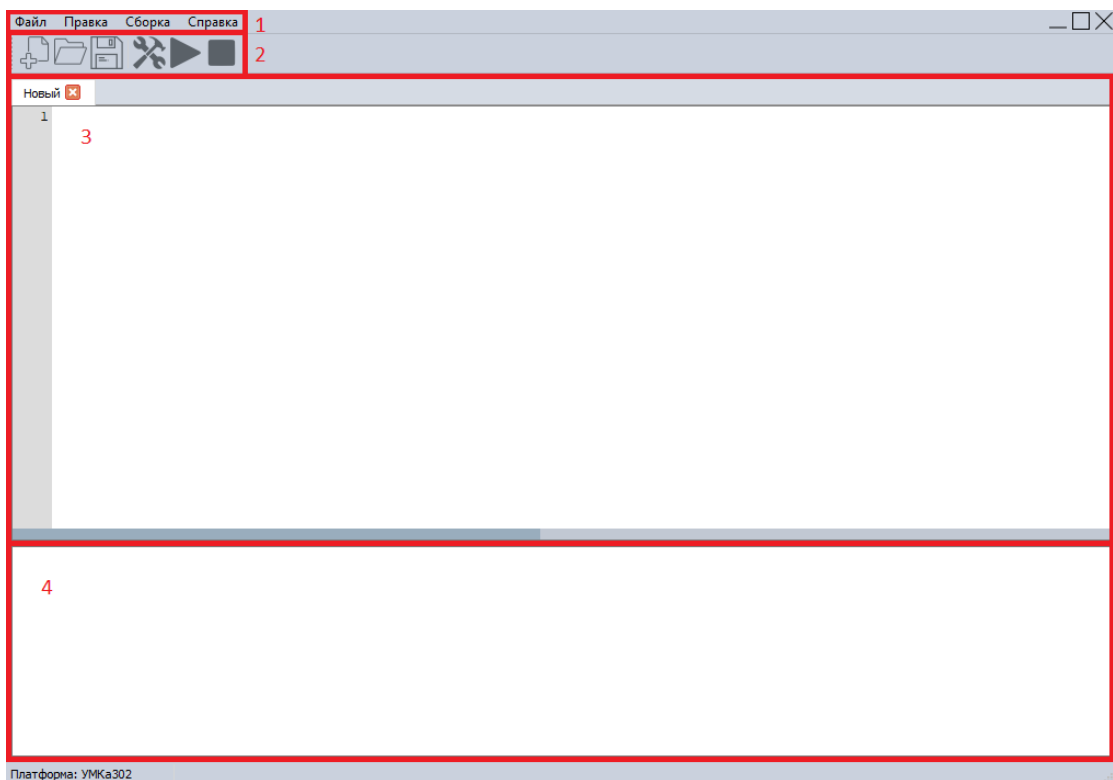
После загрузки выберите необходимый скрипт и нажмите «Выбрать». Для начала работы нажмите на кнопку «Запустить». Рекомендуется установить галочку «Автозапуск». В данном случае скрипт будет запускаться на исполнение при включении и перезагрузке терминала.

Редактор скриптов

Для написания скриптов в конфигураторе реализован редактор. Он самодостаточен и позволяет выполнять некоторые функции по управлению скриптами, такие как загрузка скрипта в память устройства и запуск на выполнение. Для запуска редактора необходимо нажать кнопку «Редактор» (1) на вкладке «Скрипты» (при этом терминал должен быть подключён) или на кнопку вызова редактора на боковой панели (2).



Общий вид редактора:

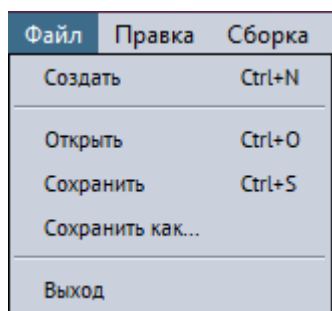


Редактор состоит из областей:

- Меню (1);
- Панель инструментов (2);
- Поле редактирования (3);
- Поле вывод результата компилирования (4);

Состав и описание пунктов меню

Пункт меню «Файл» реализует механизмы работы с файлами исходных текстов скриптов и библиотек.

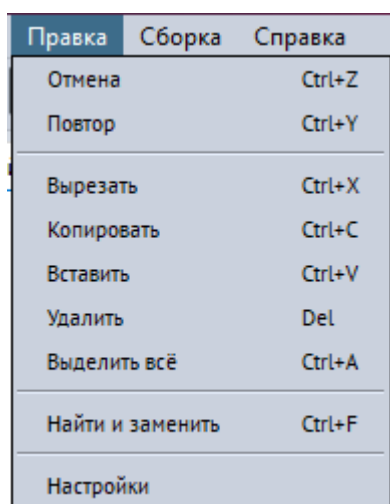


Пункт меню содержит следующие подпункты:

- «Создать» (Ctrl+N) - создать новый файл с исходным текстом скрипта или библиотеки
- «Открыть» (Ctrl+O) - открыть ранее созданный файл с исходным текстом скрипта или библиотеки

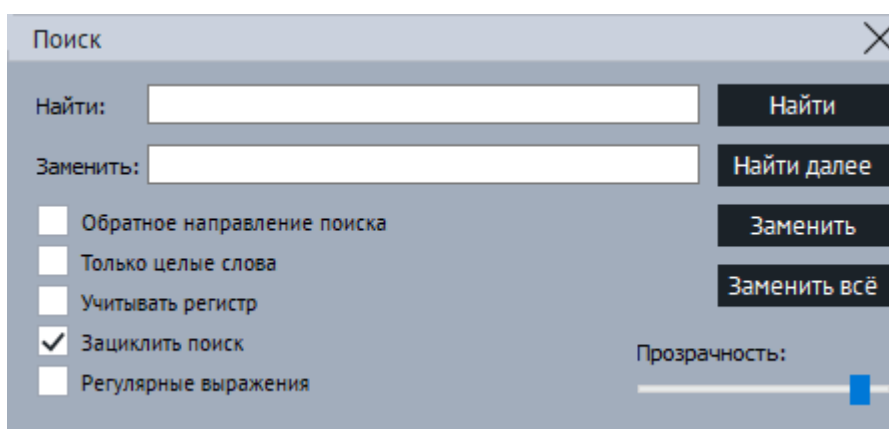
- «Сохранить» (Ctrl+S) - сохранить файл с исходным текстом скрипта или библиотеки
- «Сохранить как...» - сохранить файл с исходным текстом скрипта или библиотеки с новым именем
- «Выход» - закрыть редактор скриптов

Пункт меню «**Правка**» - в данном пункте меню реализованы механизмы работы с исходным текстом.



Пункт меню состоит из следующих подпунктов: «Отмена» (Ctrl+Z), «Повтор» (Ctrl+Y), «Вырезать» (Ctrl+X), «Копировать» (Ctrl+C), «Вставить» (Ctrl+V), «Удалить» (Del), «Выделить всё» (Ctrl+A), «Найти и заменить» (Ctrl+F), «Настройки».

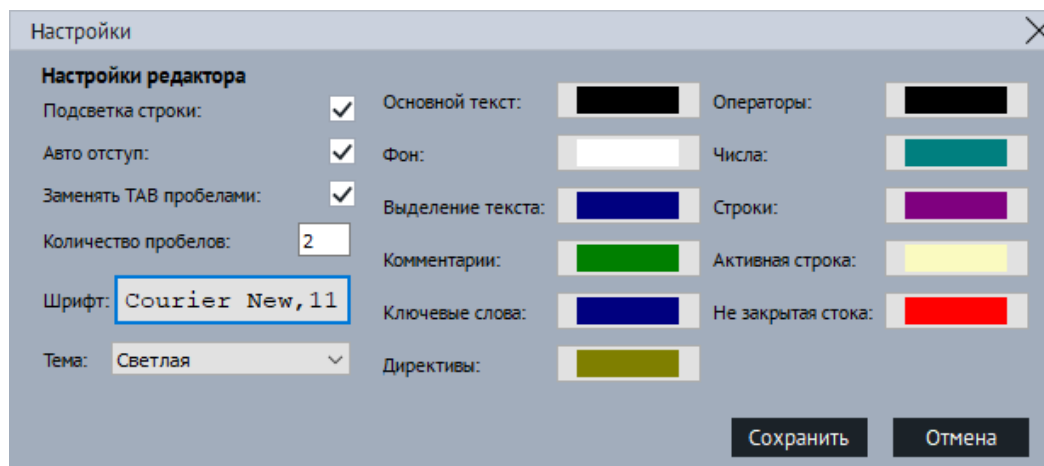
При выборе пункта «Найти и заменить» (Ctrl+F) будет вызвано окно:



В поле «Найти» необходимо ввести искомый текст. При осуществлении замены текста в поле «Заменить» нужно вставить замещающий текст. Устанавливая или снимая галочки напротив соответствующих опций можно настроить правила поиска и замены. Для начала поиска необходимо нажать кнопку «Найти». Для

продолжения поиска «Найти далее». Для замены текста необходимо нажать кнопку «Заменить», для замены сразу всех найденных фрагментов – кнопку «Заменить всё».

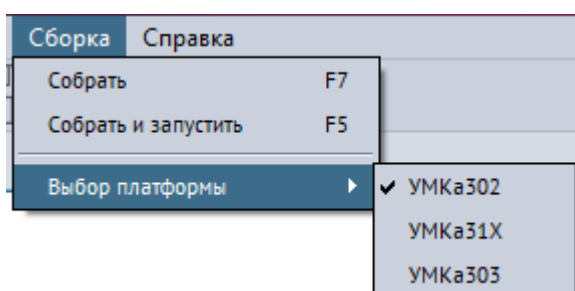
При выборе пункта «Настройки» будет вызвано окно настроек редактора:



При нажатии на поле «Шрифт» откроется диалог выбора шрифта.

В поле «Тема» можно выбрать варианты «Светлая», «Темная», «Пользовательская». «Пользовательская» тема может быть кастомизирована с помощью полей выбора цвета элементов интерфейса.

Пункт меню «Сборка» реализует функционал по компиляции скрипта и загрузке его в память терминала.



Пункт меню содержит следующие подпункты:

- «Собрать» (F7), - производит компиляцию скрипта.
- «Собрать и запустить» (F5) - производит компиляцию скрипта, загрузку в подключённое устройство и запуск на выполнение.

Для обоих подпунктов в поле вывода результата компилирования будет отображена информация о результате компиляции скрипта, а также найденные

ошибки и предупреждения. Готовый файл скрипта будет находиться в той же папке, что и исходный с тем же именем и расширением «.atx».

- «Выбор платформы» - для корректной работы скрипта необходимо указать, для какой платформы осуществляется компиляция.

Пункт меню «Справка». Состоит из одного подменю - «Каталог документов». При выборе подменю в проводнике откроется каталог с файлами, в которых описаны особенности языка, библиотек и т.п. Перечень файлов приведён в приложение Б.

Состав и описание панели инструментов

Панель инструментов дублирует часто используемые функции меню.

Состав панели инструментов:



«Создать» (Ctrl+N);



«Открыть» (Ctrl+O);



«Сохранить» (Ctrl+S);



«Собрать» (F7);



«Собрать и запустить» (F5);

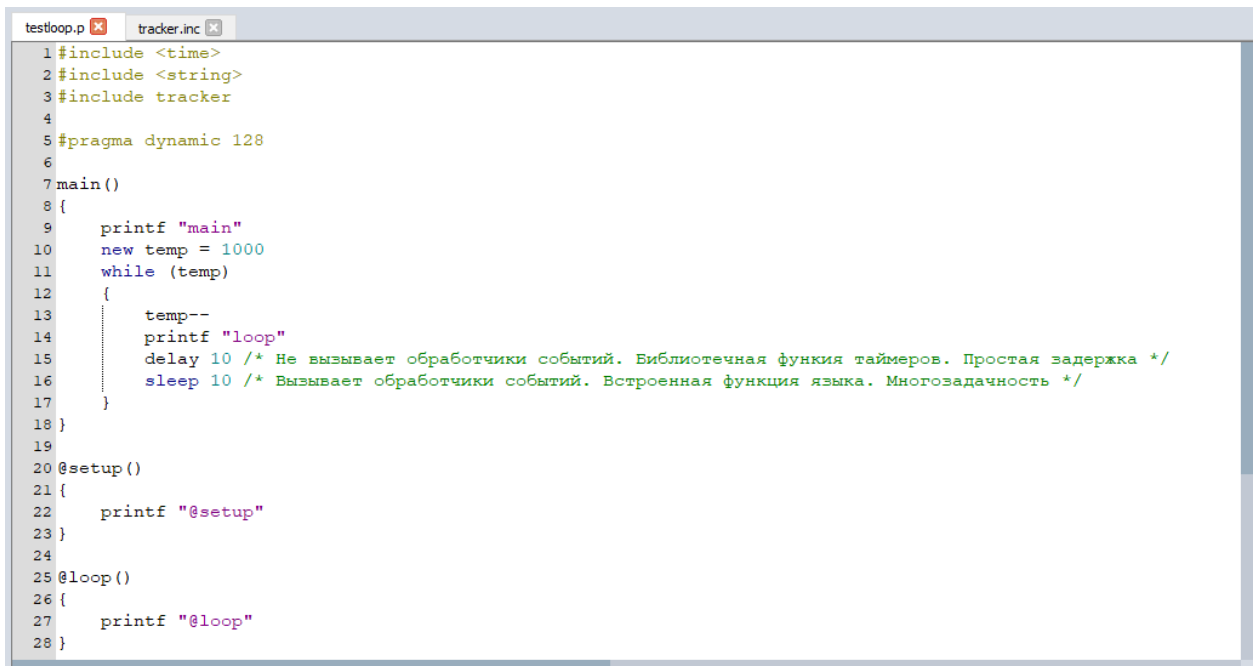


«Остановить».

Поле редактирования скрипта

Поле предназначено для ввода и редактирования исходного текста скрипта и библиотек. Одновременно может быть открыто несколько скриптов и библиотек. Осуществляется компиляция активной открытой страницы.

Пример поля редактирования с открытым скриптом:





```
1#include <time>
2#include <string>
3#include tracker
4
5#pragma dynamic 128
6
7main()
8{
9    printf "main"
10    new temp = 1000
11    while (temp)
12    {
13        temp--
14        printf "loop"
15        delay 10 /* Не вызывает обработчики событий. Библиотечная функция таймеров. Простая задержка */
16        sleep 10 /* Вызывает обработчики событий. Встроенная функция языка. Многозадачность */
17    }
18}
19
20@setup()
21{
22    printf "@setup"
23}
24
25@loop()
26{
27    printf "@loop"
28}
```

Различный по назначению текст выделен цветом (тема «Светлая»):

- коричневым – директивы компилятора;
- фиолетовым – строки;
- бирюзовым – числа;
- голубым – ключевые слова;
- зелёным – комментарии;
- чёрным – функции и операторы.

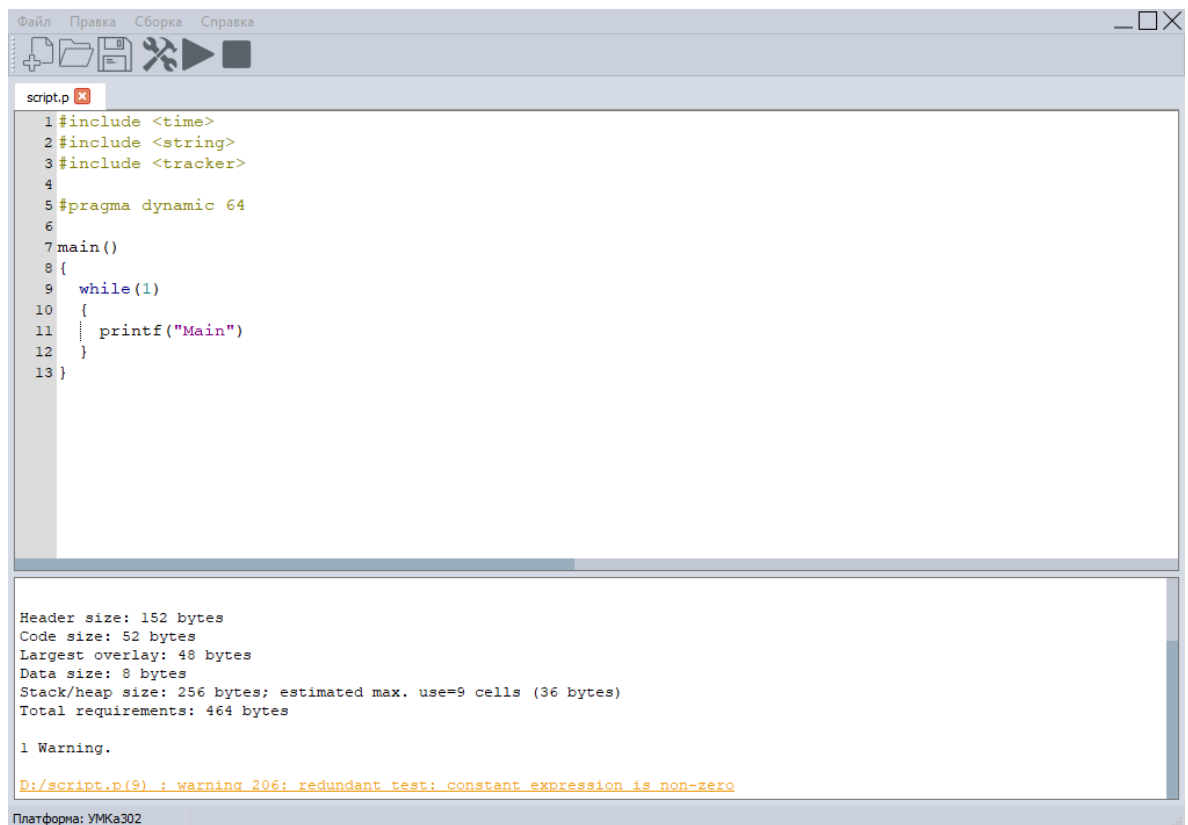
В комментариях допускается использовать текст на кириллице, но названия переменных, функций, библиотек и т.п. должно вводиться только латиницей.

Поле вывод результата компилирования

После вызова функции меню **«Сборка->Собрать»** или **«Сборка->Собрать и запустить»** или аналогичных по действию кнопок панели инструментов  и  исходный файл будет передан компилятору. После компиляции в поле вывода результата будет выведен текст, содержащий информацию о размере используемой памяти, предупреждениях и ошибках компиляции.

Кнопка  останавливает выполнение выполняющегося в терминале скрипта.

Пример вывода:



The screenshot shows a window titled "script.p" with a menu bar (Файл, Правка, Сборка, Справка) and a toolbar. The script content is as follows:

```
1#include <time>
2#include <string>
3#include <tracker>
4
5#pragma dynamic 64
6
7main()
8{
9    while(1)
10    {
11        printf("Main")
12    }
13}
```

Below the script, the compilation output is displayed:

```
Header size: 152 bytes
Code size: 52 bytes
Largest overlay: 48 bytes
Data size: 8 bytes
Stack/heap size: 256 bytes; estimated max. use=9 cells (36 bytes)
Total requirements: 464 bytes

1 Warning.
D:/script.p(9) : warning 206: redundant test: constant expression is non-zero
```

At the bottom left, it says "Платформа: УМКа302".

Значение полей:

Header size: 152 bytes - размер заголовка скрипта;
Code size: 52 bytes - размер исполняемой части скрипта;
Largest overlay: 48 bytes - максимальный размер оверлея;
Data size: 8 bytes - размер глобальных переменных;
Stack/heap size: 256 bytes; estimated max. use=9 cells (36 bytes) - выделенный и используемый размер стека;
Total requirements: 464 bytes - всего необходимо памяти для запуска скрипта;

.../pawn/samp4.p(9) : warning 206: redundant test: constant expression is non-zero - Предупреждение. В данном случае в строке под номером 9 встречено предупреждение с кодом 206.

Создание скрипта

Структура скрипта

При написании скриптов для УМКа3ХХ возможно использование нескольких подходов: с использованием функции `main` (Си подобный), с использованием функций `@setup` и `@loop` (Arduino подобный), использование функций вызова событийной модели. Данное деление условное и допускается комбинация разных подходов.

Скрипт рекомендуется «разбивать» на функции - это упрощает его структуру, поиск ошибок и модификацию кода. Подробнее про функции описано в разделе «Функции».

Си подобный подход

Выполнение программы начинается с вызова функции `main`.

В примере показан бесконечный цикл, но это не обязательное требование. С выходом из функции `main` работа со скрипта не завершается при использовании событийной модели (см раздел «Передача параметров на сервер»).

```
1 /* Глобальные переменные */
2 new GlobalVar;
3
4 /* функции */
5 MyFunc(A)
6 {
7     /* Локальные переменные функции */
8     new Result
9
10    Result = (A * 2);
11    return Result
12 }
13
14 main()
15 {
16     /* Локальные переменные */
17     new Count = 0;
18
19     printf("setup")
20
21     for(;;)
22     {
23         Count++;
24         GlobalVar = MyFunc(Count);
25         printf("loop. Result 2*A = %d", GlobalVar);
26     }
27 }
```

Arduino подобный подход

В скрипте Вы можете объявить две основные функции @setup и @loop

```
1 #include tracker
2
3 /* Глобальные переменные */
4 new GlobalVar;
5 new Count;
6
7 @setup()
8 {
9     printf("@setup")
10    Count = 0;
11 }
12
13 @loop()
14 {
15     Count++;
16     GlobalVar = MyFunc(Count);
17     printf("loop. Result 2*A = %d", GlobalVar);
18 }
19
20 /* функции */
21 MyFunc(A)
22 {
23     /* Локальные переменные функции */
24     new Result
25
26     Result = (A * 2);
27     return Result
28 }
```

Функция @setup запускается один раз при запуске скрипта. Используйте её, чтобы инициализировать переменные, установить режимы работы цифровых портов, и т.д.

Функция @loop в бесконечном цикле последовательно раз за разом исполняет команды, которые описаны в её теле. Т.е. после завершения функции снова произойдёт её вызов.

Эти первые примеры показывают несколько различий между языком MyLogic и языком C:

- обычно нет необходимости включать какой-либо заголовочный файл. Но при Arduino подобном подходе обязательно подключение библиотеки tracker (#include tracker), т.к. используются функции обратного вызова, объявленные в данной библиотеке;
- точки с запятой не обязательны кроме случаев написания нескольких операторов в одной строке;
- когда тело функции представляет собой одну инструкцию, фигурные скобки необязательны;
- когда не используются результат функции в выражении или присвоение, круглые скобки вокруг аргумента функции по желанию;

- функции могут быть объявлены после места их вызова.

Комментарии

Комментарии в скрипте служат для описания фрагмента кода. Это облегчает поиск нужных участков кода и позволяет легче в нем разобраться, при необходимости правки, особенно если это нужно сделать спустя какое-то время. Также рекомендуется в начале скрипта описывать какие задачи он решает. Комментарии не попадают в итоговой скомпилированный и загружаемый код.

Комментарии бывают двух типов: однострочные и многострочные.

Однострочные комментарии начинаются с «//». Текст, который будет располагаться, за этими знаками будет являться комментарием и в редакторе он окрасится зелёным цветом. Пример:

```
3 // Однострочный комментарий
```

Многострочный комментарий начинается со знаков «/*». Все строки текста после этих знаков будут также окрашены зелёным цветом. Заканчивается многострочный комментарий знаками «*/». Пример:

```
5 /* Многострочный
6 комментарий
7 пример*/|
```

Так же не стоит злоупотреблять многострочными комментариями или комментировать каждую строчку кода. Вместо упрощения можно получить ситуацию, когда исходный текст будет «перегружен» информацией и в нем будет сложнее разобраться.

Переменные и константы

Переменные – это именованные ячейки памяти для хранения данных, которые могут изменяться в процессе исполнения программы. Переменная должна быть объявлена до её использования.

В языке MyLogic существует несколько типов переменных: целочисленные, числа с плавающей точкой и логические. Для объявления новой переменной используется оператор new. Имена переменных чувствительны к регистру.

Целочисленная переменная объявляется следующим образом:

```
44 new Var;
```

При таком объявлении начальное значение переменной содержит нулевое значение. В случае, если необходимо присвоить начальное значение переменной, то достаточно ей его присвоить при объявлении.

```
46 new Var = 25;
```

Вещественная переменная может хранить в себе только дробные числа и объявляется следующим образом (одним из двух подходящих):

```
41 new Float:Var;  
42 new Float:Var = 123.456;
```

Логическая переменная может хранить в себе только два логических состояния «истина» - true и «ложь» - false. Примеры объявления:

```
38 new bool:Var;  
39 new bool:Var = true;
```

По области видимости переменные могут быть глобальными или локальными.

Глобальные переменные - переменные к которым можно обратиться из любой части кода.

Локальные переменные - переменные, которые находятся внутри тела какой-либо функции, и доступны только внутри данной функции.

```
30 new GlobalVar; // Это глобальная переменная  
31 MyFunc (A)  
32 {  
33     new Result // Это локальная переменная  
34  
35     Result = (A * 2);  
36     return Result  
37 }
```

Константа – величина, не меняющаяся в процессе выполнения скрипта. В скриптах константы объявляются почти так же, как и переменные, однако каждое объявление начинается со слова `const` и нужно обязательно указывать значение:

```
88 const IntConst = 100;
89 const Float:FloatConst = 100.21;
```

Массивы

Массив – именованная область памяти, предназначенная для хранения нескольких переменных одного типа. Обращение к элементам массива выполняется через один или несколько индексов.

Массив может быть проинициализирован и может быть объявлен как константа.

Количество используемых индексов массива может быть различным: массивы с одним индексом называют одномерными, с двумя – двумерными и т.д. В MyLogic массивы могут быть одномерными, двумерными и трехмерными.

Примеры объявления массивов:

```
11 new IntArray2[3] = [1210, 1220, 1230] /* Одномерный массив целочисленного типа
12                                     инициализирован */
13
14 new Float:FloatArray[3] = [123.45, 123.56, 321.56] /* Одномерный массив вещественного типа
15                                                     инициализирован */
16
17
18
19 new const ConstArray[3] = [1230, 1231, 1232] /* Одномерный массив целочисленного типа
20                                               константа */
21
22
23 new Int2Array[10][20] /* Двумерный массив целочисленного типа
24                       не инициализирован */
25
26 new Int2Array2[4][5] = [ /* Двумерный массив целочисленного типа, инициализирован */
27 [347,782,632,437,721], [847,722,612,137,725], [347,782,632,437,721], [146,954,445,463,1024]
28 ];
```

Обращение к элементам массива осуществляется по индексу, начиная с 0:

```
68 FloatArray[2] = 1.256
69 IntArray[1] = 568
```

Для целочисленных массивов по умолчанию один элемент массива занимает одну ячейку (32 бита). При необходимости можно объявить массив с побайтовым доступом. В данном случае для объявления массива и доступа к его элементам используются фигурные скобки { }.

Пример:

```
71 new CharArray{3} = {121,122,123}; // Объявление
72 CharArray{1} = 128; // Пример обращения
```


Строки

Строка – это последовательность символов.

По сути строки – это те же массивы. А поскольку символы в MyLogic могут храниться в целочисленных переменных, то строка – это массив целых чисел. Строки могут быть не упакованными и упакованными. В первом случае каждый символ строки занимает одну ячейку (32 бита), во втором – в каждой ячейке хранится 4 символа.

Строки заканчиваются неявным символом `'\0'` – завершающий ноль. По этому символу функции работы со строками (к примеру - `printf`) определяют, на какой позиции закончилась строка.

Примеры объявления строк:

```
9 new String[] = "String" // Объявление строки, размер не указан
10
11 new String2[20] = "String" // Объявление строки с указанием размера
12
13 new String3{} = "String" // Объявление упакованной строки
```

Структуры

Скриптовый язык MyLogic не поддерживает структуры напрямую, но можно объявить массивы, которые будут «имитировать» простые структуры. Для создания такой структуры индексам массива присваиваются индивидуальный «идентификатор». В структуре можно сформировать подмассив.

Пример объявления:

```
74 new Acc[.x, .y, .z];
75
76 new position[Float: .lat, Float: .lon, Float: .height, Float: .course, Float: .speed, Float: .hdop]
77
78 new msg[.text{40}, .priority]
```

При многократном использовании структуры, её состав можно объявить через директиву `#define`:

```
79
80 #define ACC[.x, .y, .z]
81
82 new Acc[ACC]
```

Обращение к элементам таких структур осуществляется по их именованному индексу.

К примеру:

```
84 Acc.x = 20;  
85 position.lon = 12.658
```

Операторы

Оператор – это элемент языка, задающий описание действия, которое необходимо выполнить.

Все операторы MyLogic, условно разделены на следующие категории:

- - математические операторы;
- - операторы сравнения;
- - логические операторы;
- - условные операторы, к которым относятся оператор условия `if` и оператор выбора `switch`;
- операторы цикла (`for`, `while`, `do while`);
- операторы перехода (`break`, `continue`, `return`);

Математические операторы

Как и в большинстве других языков программирования, скриптовый язык поддерживает основные математические операторы: умножение (*), деление (/) сложение (+), вычитание (–) и остаток от деления (%).

Операторы сравнения

К операторам сравнения, относятся: «меньше» (<), «меньше или равно» (<=), «больше» (>), «больше или равно» (>=), «равно» (==) и «не равно» (!=).

Логические операторы

Довольно часто возникает необходимость проверять не одно условное выражение, а несколько. Для проверки нескольких условных выражений существуют логические операторы: «И» (&&), «ИЛИ» (||) и «НЕ» (!).

Условные операторы

Условные конструкции позволяют проверить, удовлетворяют ли данным условиям выражения или нет и в зависимости от результата, выполняет соответствующий код.

`if` – это условный оператор, в скобках после которого указывается условие. В фигурных скобках пишется код, который выполнится, если условие истинно.

`else` – это также условный оператор, но он выполняет свои функции, только в том случае, если условие в `if` ложно. То есть, оператор `if` можно назвать как оператор «если», а `else` как оператор «иначе».

Если необходимо проверить несколько условий, то используется конструкция `else if` - «иначе если».

Пример:

```
15 new a = 5;
16 new b = 25;
17 if(a < b) // Если a меньше b
18 {
19     printf("a less b"); // Выводим сообщение "a меньше b"
20 }
21 else if (a == b) // Иначе если a равно b
22 {
23     printf("a equals b"); // Выводим сообщение "a равно b"
24 }
25 else // Иначе
26 {
27     printf("a more b"); // Выводим сообщение "a больше b"
28 }
```

Оператор `switch` используется в скриптах для выполнения действий по нескольким (чаще всего 2 и более) ветвям:

```
30 switch(<выражение>)
31 {
32     case <диапазон1>:
33     {
34         <действия1>;
35     }
36     case <диапазон2>:
37     {
38         <действия2>;
39     }
40     default:
41     {
42         <действия_иначе>;
43     }
44 }
```

Пример:

```
47 /* функция определения дня недели по дате */
48 weekday(day, month, year)
49 {
50     if (month <= 2)
51         month += 12, --year
52     new j = year % 100
53     new e = year / 100
54     return (day + (month+1)*26/10 + j + j/4 + e/4 - 2*e) % 7
55 }
56
57 @loop()
58 {
59     switch (weekday(03,12,2020))
60     {
61     case 0, 1: /* 0 == Saturday, 1 == Sunday */
62         printf("Weekend")
63     case 2:
64         printf("Monday")
65     case 3:
66         printf("Tuesday")
67     case 4:
68         printf("Wednesday")
69     case 5:
70         printf("Thursday")
71     case 6:
72         printf("Friday")
73     default:
74         printf("Invalid week day")
75     }
76 }
```

Действия в метке `default` выполняются в том случае, когда значение выражения в `switch` не принадлежит ни одному из интервалов. Метка `default` не обязательна.

Главное преимущество `switch` перед `if` - эффективность: если в конструкции `switch` указано несколько диапазонов, принадлежность к одному из них определяется быстрее, чем в `if`.

Операторы цикла

Цикл `while` как и любой другой цикл, повторяет свою функцию до тех пор, пока его условие истинно. Если его условие будет ложно, цикл прекращает работу. Цикл `while` выглядит следующим образом:

```

17     new a = 0; // Объявляем переменную со значением 0
18     while(a < 3) // Проверяем условие выполнение
19     {
20         /* Выполняем действие */
21         a++;
22         printf("Hello, World!");
23     }

```

Цикл `do while`. Если `while` сначала проверяет условие, а затем выполняет тело цикла, то `do while` делает все с точностью до наоборот. Сначала он выполняет тело цикла, а потом проверяет условие. Пример:

```

6     new a = 0; // Объявляем переменную со значением 0
7     do
8     {
9         /* Выполняем действие */
10        a++; // Увеличиваем на 1 значение в переменной
11        printf("Hello, World!");
12    }
13    while(a < 3); // Проверяем условие выполнение

```

Цикл `for` - цикл со счётчиком, в котором некоторая переменная изменяет своё значение от заданного начального значения до конечного значения с некоторым шагом, и для каждого значения этой переменной тело цикла выполняется один раз. Начальное значение, условия выполнения и шаг цикла задаётся при объявлении:

```

27     /* Цикл со счётчиком.
28     Начальное значение счетчика 0,
29     условие выполнения - значение счетчика меньше 3,
30     счетчик увеличивается на 1 */
31     for (new i = 0; i < 3; i++)
32     {
33         printf("Hello, World!");
34     }
35 }

```

Для всех трёх примеров вывод в отладочную консоль будет одинаковым:

```

> I11:21:47 PAWN: Hello, world!
< I11:21:47 PAWN: Hello, world!
< I11:21:47 PAWN: Hello, world!
< I11:21:47 PAWN: Hello, world!

```

Операторы перехода

Операторы `break` и `continue`. Данные операторы используются в циклах: `for`, `while`, `do while`. Оператор `break` прерывает работу цикла пропустив оставшиеся действия. Оператор `continue`, пропускает оставшееся действие цикла и переходит к проверке условия.

Пример:

```
69 new a = 0; // Объявляем переменную со значением 0
70 while(a < 5) // Проверяем условие выполнение
71 {
72     /* Выполняем действие */
73     a++;
74     if(a == 2) // Если a равно 2
75     {
76         break; // Выходим из цикла
77     }
78     printf("Check a");
79 }
80
81 new b = 0; // Объявляем переменную со значением 0
82 while(b < 5) // Проверяем условие выполнение
83 {
84     /* Выполняем действие */
85     b++;
86     if(b == 2) // Если b равно 2
87     {
88         continue; // завершаем действие и переходим к проверке условия
89     }
90     printf("Check b");
91 }
```

Вывод в отладочную консоль:

```
< I11:43:44 PAWN: Check a
< I11:43:44 PAWN: Check b
< I11:43:44 PAWN: Check b
< I11:43:44 PAWN: Check b
< I11:43:44 PAWN: Check b
```

Для первого примера, как только переменная `a` станет равной 2 цикл завершится. Т.к. инкремент `a` происходит до проверки условия, то будет выведена одна строка "Check a". Во втором примере строка "Check b" выведена 4 раза т.к. при выполнении условия «`b` равно 2» функция `printf` будет пропущена.

Функции

Функция – именованная последовательность операторов, которая определена и записана только в одном месте скрипта, однако ее можно вызвать для выполнения из одной или нескольких точек скрипта. Функции в MyLogic могут быть объявлены после места их вызова.

Синтаксис:

```
67 <имя> (<параметры>)  
68 {  
69     <действие1>;  
70     <действие2>;  
71     <действие3>;  
72     <действиеN>;  
73     <return value>  
74 }
```

Оператор `return` имеет два назначения. Во-первых, может использоваться для возврата значения. Во-вторых, выполняет немедленный выход из функции. То есть прерывает работу функции и осуществляет выход в вызывавший функцию код.

Пример простой функции:

```
74 /* Функция sum, получает на входе целые числа a и b,  
75    затем складывает их и возвращает результат */  
76 sum(a, b)  
77 {  
78     new result = a+b;    // Сложить 2 числа и записать результат в result  
79     return result;      // Вернуть результат сложения в место вызова  
80 }
```

Аргументы в функцию передаются в виде значений. Таким образом использование и изменение переданных переменных не отразится на исходных данных. Если необходимо, чтобы функция их изменила, параметры в функцию необходимо передавать по ссылке. Для этого используется символ `&` в объявлении аргументов функции:

```
83 sum(&a, b)  
84 {  
85     new result = a+b;    // сложить 2 числа и записать результат в result  
86     a = result;         // В a также записали результат выполнения  
87     return result;      // вернуть результат сложения в место вызова  
88 }
```

Массивы в функцию всегда передаются по ссылке. При этом размер массива может быть указан явно:

```
2
3 #define ARRAY_SIZE      3
4 new IntArray[ARRAY_SIZE] = [1210,1220,1230];
5
6 /* функция вывод значений массива.
7  Размер переданных данных передается как параметр */
8 PrintArray1(Buff[], size = sizeof Buff)
9 {
10     printf("PrintArray1")
11     for (new i = 0; i < size; i++)
12     {
13         printf("Array[%d] = %d", i, Buff[i]);
14     }
15 }
16
17 /* функция вывод значений массива. Размер массива указан явно */
18 PrintArray2(Buff[ARRAY_SIZE])
19 {
20     printf("PrintArray2")
21     for (new i = 0; i < ARRAY_SIZE; i++)
22     {
23         printf("Array[%d] = %d", i, Buff[i]);
24     }
25 }
26
27 main()
28 {
29     PrintArray1(IntArray) // Передаем массив в функцию, размер определяется автоматически
30     PrintArray1(IntArray, ARRAY_SIZE) // Передаем массив в функцию, размер задаем
31     PrintArray2(IntArray) // Передаем массив в функцию. Размер должен быть строго ARRAY_SIZE
32 }
33
34 #if 0
35
```

Некоторые параметры в функции могут быть объявлены с параметрами по умолчанию (необязательными параметрами). Если параметр(ы) последний(ие) то можем их не указывать. Если параметр находится между обязательными или в начале тогда вместо параметра, который нужно пропустить, указываем символ подчёркивания «_». На примере выше при первом вызове функции PrintArray1 мы не передали размер массива т.к. он задан по умолчанию равным размеру передаваемого массива (определяется через `sizeof`).

Функции в скриптовом языке условно можно разделить на следующие виды:

- функции, непосредственно реализованные в скрипте;
- библиотечные функции;
- функции платформы (нативные функции);
- функции скрипта, вызываемые платформой (callback функции);

При объявление библиотечных функции рекомендуется использовать ключевое слово `stock`. Это позволяет объявлять функции, которые не будут использоваться в скрипте и при этом компилятор не будет выдавать предупреждение о неиспользовании.

Функции с ключевым словом `native` являются функциями платформы и реализуют механизмы взаимодействия скрипта с аппаратным и программными средствами платформы. Описание данных функций приведено в приложении А.

Публичные функции, которые вызывает платформа по событиям начинаются с слова `public` или символа «@». Данные функции как правило реализуют событийную модель функционирования.

Событийная модель

В скриптах для абонентских терминалов УМКа3ХХ возможна реализация событийно - ориентированной модели программирования - парадигма программирования в которой выполнение скрипта определяется системными событиями.

Реализация модели осуществляется через обработчики событий, которые для устройства могут быть:

- ✓ `@timer` - срабатывание таймера;
- ✓ `@setup` - запуск скрипта;
- ✓ `@loop` - периодически вызываемая функция скрипта;
- ✓ `@accupdate` - поступление новых данных от акселерометра;
- ✓ `@gnssupdate` - поступление новых данных от GNSS;
- ✓ `@bleadvertrecv0-3` - поступление данных от BLE;
- ✓ `@canrecv` и `@canrecv0-1` - поступление данных по CAN шине;
- ✓ `@chat` - поступление специальной команды;
- ✓ `@bboxupdate` – добавление точки в черный ящик.

Для выполнения определённых действий с заданной периодичностью в скриптах предусмотрена возможность использования программного таймера.

Для управления таймером предусмотрена следующая функция:

```
native settimer(milliseconds, bool: singleshoot=false);
```

Аргументы функции:

- `milliseconds` - период таймера в миллисекундах;
- `singleshoot` - однократный запуск, по умолчанию `false`;

После вызова данной функции с указанным периодом будет вызываться обработчик события `@timer`.

Пример:

```
1 #include <tracker>
2 #include <time>
3
4 #pragma dynamic 32
5
6 new CountEventTimer = 0;
7
8 main()
9 {
10     settimer(1000) // Запускаем таймер с периодом 1 сек
11 }
12
13 @timer()
14 {
15     CountEventTimer++; // Увеличиваем значение счетчика событий таймера
16     printf("Timer event #%d", CountEventTimer) // Выводи в отладочный вывод
17 }
18
```

Более подробно остальные обработчики событий платформы описаны в приложении А.

ВАЖНО! Выполнение кода осуществляется в одном потоке. Обработчики событий не должен содержать в себе бесконечный цикл и допускать длительного выполнения, поскольку скрипт не может реагировать на другие события, пока не завершился обработчик текущего.

В случае необходимости возможна комбинация основного цикла и событийной модели. При этом возможны два подхода:

Использование бесконечного цикла и оператора `sleep`:

```
99 @Timer()
100 {
101     /* Обработка события - таймер*/
102 }
103
104 main()
105 {
106     for(;;)
107     {
108         /* Выполняем действия*/
109         sleep(100) // Обязательно!!! Вызывает обработчики событий. Встроенная функция языка.
110     }
111 }
```

Использование @loop:

```
99 @Timer()  
100 {  
101     /* Обработка события - таймер*/  
102 }  
103  
104 @loop()  
105 {  
106     /* Выполняем действия*/  
107 }
```

Передача параметров на сервер

Все параметры, предназначенные для передачи на сервер в устройствах УМКа3ХХ сохраняются в чёрном ящике устройства. ЧЯ состоит из записей в которых хранится срез значений всех параметров в момент формирования записи. Каждая запись состоит из заголовка и набора данных (параметров, тэгов).

Параметр, Тэг - данные, однозначно характеризующие состояние одного из параметров устройства (системное время, импульсный вход и т.п.) на момент формирования записи.

Для скриптов в системе предусмотрено 32 целочисленные знаковые 32-х битные ячейки (параметра, тэга). Для каждого тэга возможно указание смещения запятой (параметр `pow`) и достоверности данных (параметр `valid`).

Запись значений в ячейки из скрипта осуществляется функцией `settag(indx, value, bool: valid = true, pow = 0)`. Описание приведено в приложении А.

Для передачи параметров с плавающей запятой их нужно привести к целочисленному виду. Для этого необходимо преобразовать значение параметра в целое число при помощи функции `floatround`.

Важно понимать, что запись значения в ячейку не означает его передачу. Реально передано в последней точке будет только записанное значение на момент формирования точки. С принципом формирования и передачи данных от терминала можно ознакомиться в РЭ на УМКа302/УМКа303/УМКа31Х.

При необходимости точку можно принудительно сформировать, вызвав функцию `pushpoint`.

Пример записи параметров:

```
1#include <tracker>
2#include <float>
3
4#pragma dynamic 128
5
6main()
7{
8    new Float: Sin; // Значение синуса
9    new Float:Degrees = 0.0; // Градус
10    new TagValue;
11    new Step = 0
12
13    for(;;)
14    {
15        Sin = floatsine(Degrees, degrees); // Вычисляем синус
16        printf("Sin(%.01f) = %.03f", Step, Degrees, Sin); // Выводим
17        settag(0, Step, true, 0); // В ячейку 0 пишем значение шага
18        /* Значение градуса выводим с точностью до десятых */
19        TagValue = floatround((Degrees * 10), floatround_floor);
20        settag(1, TagValue, true, 1); // В ячейку 1 пишем значение градуса
21        /* Значение синуса выводим с точностью до тысячных */
22        TagValue = floatround((Sin * 1000), floatround_floor);
23        settag(2, TagValue, true, 3); // В ячейку 2 пишем значение синуса
24        if ((Step % 100) == 0) pushpoint(); // Сохраняем в истории каждые 100 шагов
25        Degrees += 0.1; // Увеличиваем на 0,1 градус
26        Step++; // Увеличиваем номер шага
27        sleep(100);
28    }
29 }
```

После загрузки и запуска скрипта на вкладке «Скрипты» можно проконтролировать изменение данных:

Параметры передаваемые на сервер:				
Разрешить передачу параметров:				<input checked="" type="checkbox"/>
№:	Значение	Передавать	№:	
0:	437	<input checked="" type="checkbox"/>	8:	
1:	43.6	<input checked="" type="checkbox"/>	9:	
2:	0.689	<input checked="" type="checkbox"/>	10:	
3:		<input type="checkbox"/>	11:	
4:		<input type="checkbox"/>	12:	
5:		<input type="checkbox"/>	13:	
6:		<input type="checkbox"/>	14:	
7:		<input type="checkbox"/>	15:	

Проконтролировать запись параметров в архив можно на вкладке «История» (Параметры Amx0 - Amx2):

Конфигуратор УМКа302 SN: 20054669

ГЛОНАССSoft

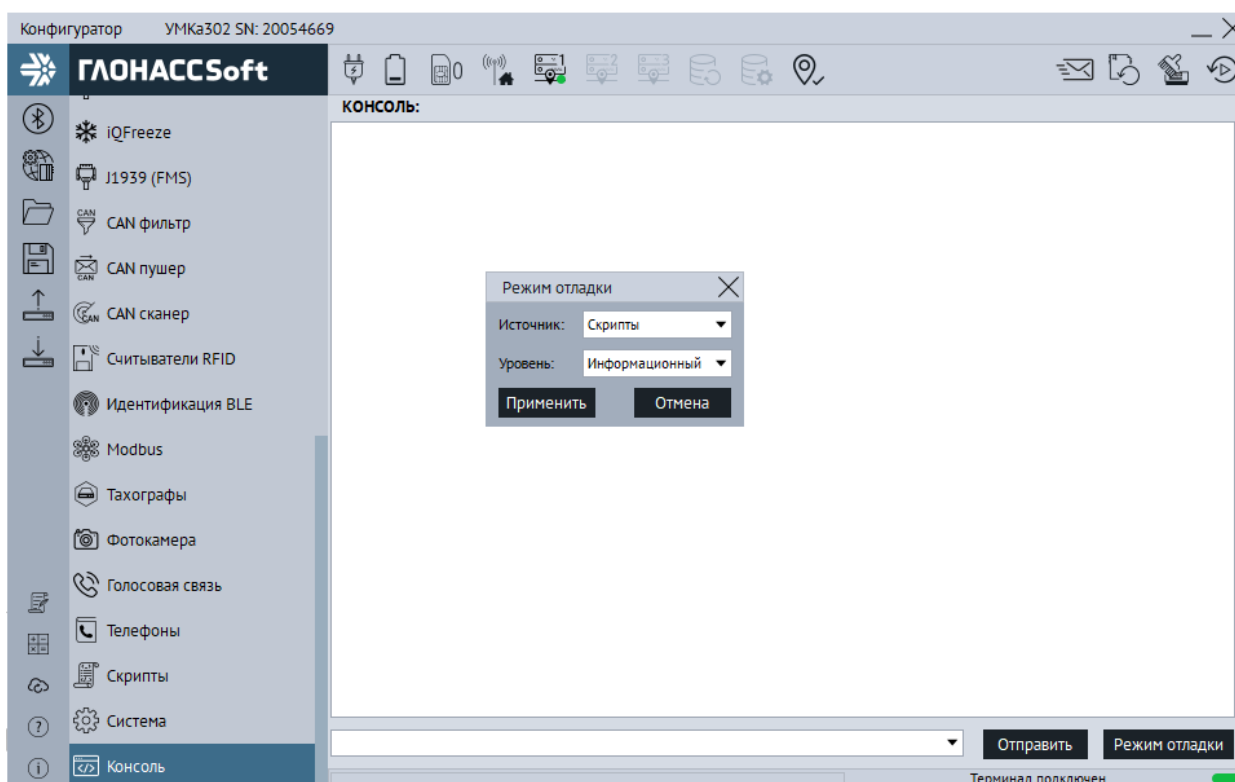
История:

Light	Course	Speed	Hdop	Sats	Status	Uext	Uakb	Ain0	Ain1	Din0	Din1	Fuel0	Ft0	Amx0	Amx1	Amx2
0.2	0.0	0.0	0.78	7+6	0x00200014	13.527		10 (10)	10 (10)	0 (0)	0 (0)			68	6.7	0.118
0.2	0.0	0.0	0.78	7+6	0x00200014	13.499		10 (10)	10 (10)	0 (0)	0 (0)			100	10.0	0.173
0.3	138.0	0.0	0.76	8+6	0x00200014	13.421		10 (10)	10 (10)	0 (0)	0 (0)			144	14.4	0.248
0.3	138.0	0.0	0.76	8+6	0x00200014	13.527		10 (10)	10 (10)	0 (0)	0 (0)			200	20.0	0.342
0.3	138.0	0.0	0.82	8+6	0x00200014	13.518		20 (20)	0 (0)	0 (0)	0 (0)			0	0.0	0.000
0.3	138.0	0.0	0.77	8+5	0x00200010	13.520		10 (10)	0 (0)	0 (0)	0 (0)			100	10.0	0.173
0.3	138.0	0.0	0.75	8+5	0x00200010	13.518		20 (20)	10 (10)	0 (0)	0 (0)			200	20.0	0.342
0.3	157.8	0.0	0.84	7+5	0x00200010	13.518		10 (10)	10 (10)	0 (0)	0 (0)			291	29.1	0.486
0.3	157.8	0.0	0.84	7+5	0x00200010	13.509		20 (20)	10 (10)	0 (0)	0 (0)			300	30.0	0.500

Отладка скрипта

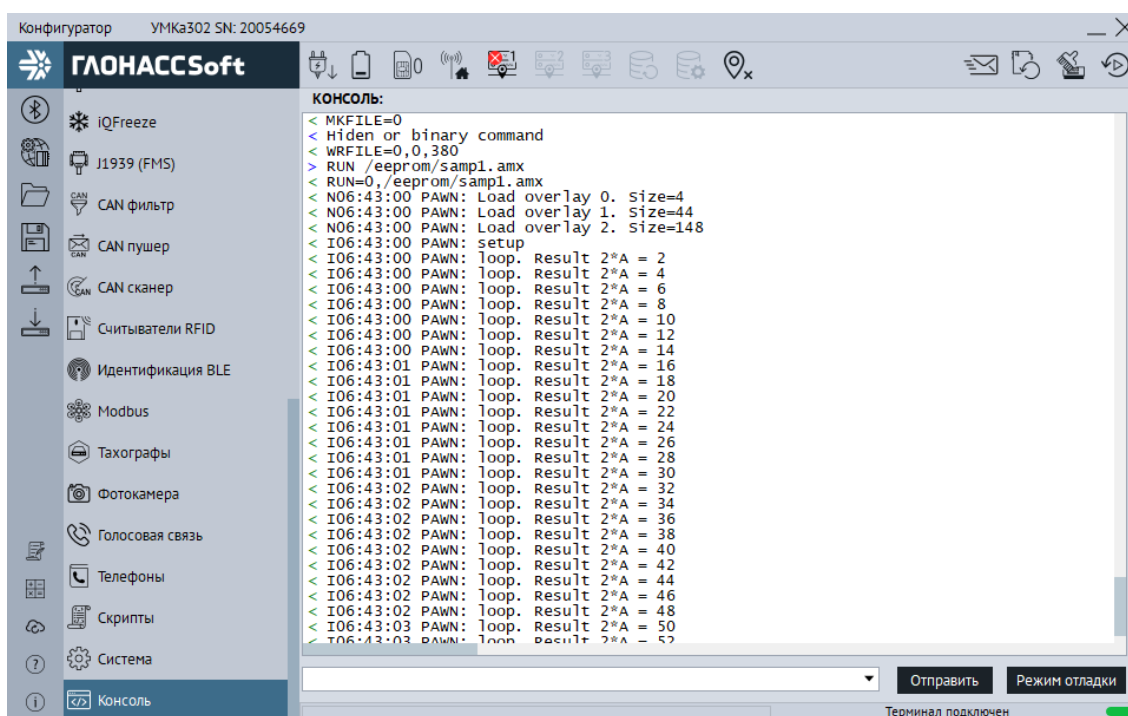
Отладка скрипта может осуществляться с помощью отладочных сообщений, заблаговременно расставленных разработчиком в ключевых местах скрипта.

Для вывода отладочных сообщений в скриптах используются функция `printf`. Чтобы увидеть в консоли отладочные сообщения необходимо перейти на вкладку «Консоль», нажать кнопку «Режим отладки». В появившемся окне указать источник - «Скрипты», уровень - «Информационный». Нажмите кнопку «Применить».




Пример тестового скрипта и его отладочного вывода:

```
1 /* Глобальные переменные */
2 new GlobalVar;
3
4 /* функции */
5 MyFunc (A)
6 {
7     /* Локальные переменные функции */
8     new Result
9
10    Result = (A * 2);
11    return Result
12 }
13
14 main()
15 {
16     /* Локальные переменные */
17     new Count = 0;
18
19     printf("setup")
20
21     for(;;)
22     {
23         Count++;
24         GlobalVar = MyFunc(Count);
25         printf("loop. Result 2*A = %d", GlobalVar);
26     }
27 }
```



В случае, если необходимо внести изменения в работающий скрипт (включён вывод отладочной информации) и при этом оперативно проконтролировать его работу при помощи отладочного вывода то можно поступит следующим образом:

- Внести необходимые изменения в скрипт. При этом не нужно переключаться с вкладки «Консоль».

- Скомпилировать скрипт и запустить на выполнение кнопкой  панели инструментов.
- Если ошибок нет, то скрипт будет загружен в память устройства и запущен на выполнение. При этом в консоли появятся новые отладочные сообщения.

Описание функции `printf`:

```
native printf(const format[], {Float,Fixed,_}:...);
```

Параметр `const format[]` - Строка которая состоит из объектов двух различных назначений. Во-первых, это символы, которые сами должны быть выведены на экран. Во-вторых, это спецификаторы формата, определяющие вид, в котором будут выведены аргументы.

Поддерживаются следующие спецификаторы формата:

<code>%c</code>	Символ типа <code>char</code>
<code>%d</code>	Десятичное число целого типа со знаком
<code>%f</code>	Десятичное число с плавающей точкой
<code>%q</code>	Десятичное число с фиксированной точкой
<code>%s</code>	Строка символов
<code>%u</code>	Десятичное число целого типа без знака
<code>%x</code>	Шестнадцатеричное целое число без знака (буквы нижнего регистра)
<code>%%</code>	Выводит символ <code>%</code>

Директивы препроцессора

Первым этапом компиляции исходного файла скрипта `MyLogic` в исполняемый байт-код является «предварительная обработка»: удаляются комментарии и «условно скомпилированные» блоки, выполняются операции поиска и замены текста исходного файла, обрабатываются директивы компилятора.

Все директивы препроцессора начинаются с символа «`#`»

В данном разделе перечислены часто используемые директивы. С полным перечнем и описанием можно ознакомиться в документе «`Pawn_Language_Guide.pdf`» стр. 114.

Директивы `#if`, `#endif` - директивы условного компилирования. При их использовании в скомпилированный скрипт может быть включены или исключены

определённые участки кода в зависимости от условия. У каждой директивы `#if` в исходном файле должна быть соответствующая закрывающая директива `#endif`.

```
7 #if константное выражение
8 // Группа операций
9 #endif
10
```

Пример:

```
1 #include tracker
2
3 /* Глобальные переменные */
4 new GlobalVar;
5
6
7 /* функции */
8 MyFunc(A)
9 {
10     /* Локальные переменные функции */
11     new Result
12
13     Result = (A * 2);
14     return Result
15 }
16
17 main()
18 {
19     /* Локальные переменные */
20     new Count = 0;
21
22     printf("setup")
23
24     for(;;)
25     {
26         Count++;
27         GlobalVar = MyFunc(Count);
28 #if 0 // Исключаем вывод логов
29     printf("loop. Result 2*A = %d", GlobalVar); // Выводим только результат
30 #endif
31         sleep(100);
32     }
33 }
34
```

В данном примере конструкцией `#if 0`, `#endif` мы исключили вывод в консоль отладочного сообщения от скрипта. При необходимости, изменив `#if 0` на `#if 1` вывод можно включить обратно.

Директива `#define` позволяет вводить в текст программы константы и макроопределения. Общая форма записи: `#define <Идентификатор> <Замена>`

Директива `#define` указывает компилятору, что нужно подставить строку, определённую аргументом «замена», вместо каждого аргумента «идентификатор» в исходном файле.

Есть два основных типа #define: define-функции (марсосы) и define-объекты (константы):

```
121 // константа
122 #define BUFFER_SIZE 1024
123 // макрос
124 #define max(a, b) ((a) > (b) ? (a) : (b) )
125 // После замены макроса код будет выглядеть следующим образом:
126 z = max(x, y)
127 z = ( (x) > (y) ? (x) : (y) );
```

Директива #include подключает файл в скрипт, что позволяет его структурировать или использовать ранее написанные библиотеки. Если расширение не указано, то по умолчанию используется расширение «.inc».

Имя файла может быть указано в угловых скобках < и > поиск файла будет осуществляться в заранее заданном списке каталогов. Для УМКа302 это каталоги «...\UMKa3XX\pawm\include» и «...\UMKa3XX\pawm\include\302», для УМКа303 это каталоги «...\UMKa3XX\pawm\include» и «...\UMKa3XX\pawm\include\303», для УМКа31X - «...\UMKa3XX\pawm\include» и «...\UMKa3XX\pawm\include\31X».

Если имя файла задано в двойных кавычках или без кавычек вообще - файл ищется в текущем каталоге.

Примеры:

```
1 #include <time>
2 #include <string>
3 #include <tracker>
4 #include <serial>
5 #include mercury
```

Директива #pragma. Через данную директиву настраиваются дополнительные настройки, такие как уровни предупреждения или дополнительные возможности:

#pragma dynamic <value> – задаёт размер стека и кучи в ячейках. По умолчанию размер равен 4096 ячеек (16384 байта). Рекомендуется уменьшать данное значение до реально необходимого с некоторым запасом.

Как это сделать:

Скомпилировать скрипт. В окне вывода результата компиляции найти строку вида: Stack/heap size: 16384 bytes; estimated max. use=8 cells (32 bytes). Последнее значение – это оценочное значение максимально используемого размера стека/кучи. Реальный необходимый размер стека/кучи составляет 8 ячеек (32 байта).

Задаём #pragma dynamic 16 и компилируем скрипт повторно.

Опять находим строку с оценкой используемого размера стека/кучи:

Stack/heap size: 64 bytes; estimated max. use=8 cells (32 bytes)

При внесении изменений в скрипт всегда убеждаемся, что выделенный размер больше чем оценочное значения использования стека. Не рекомендуется делать запас более чем в 2 раза.

Более подробно о скриптовом языке можно прочитать в документе «Pawn_Language_Guide.pdf»

Аргументы скрипта

При решении некоторых задач необходима настройка параметров работы скрипта (например, задания границ срабатывания для аналогового канала, указать сетевой адрес подключаемого оборудования). Для этих целей могут быть использованы аргументы скрипта.

Аргументы скрипта – это строка длиной до 80 символов, в которой параметры разделены запятой. Поиск параметров возможен как по индексу, так и по имени. Именованные аргументы передаются как «name=value» или «name:value», где name – наименование параметра, value – его значение (число или строка).

Для использования аргументов к скрипту должна быть подключена библиотека «args.inc». Более подробно с функциями работы с аргументами можно ознакомиться в документе «Arguments_Support.pdf».

Рассмотрим работу с аргументами на примере скрипта «testarg.p».

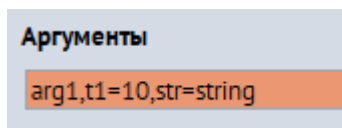
Текст скрипта:

```
18 main()
19 {
20     new opt{100}
21     new TextArg{100}
22     new IntArg = 25;
23
24     printf("Argument count = %d", argcount() /* Получаем общее количество аргументов */
25
26     for (new index = 0; argindex(index, opt); index++)
27     {
28         printf("Argument %d = \"%s\"", index, opt) /* Выводим последовательно все аргументы */
29     }
30     if (argvalue(0, "t1", IntArg) == true) /* Получаем числовое значение аргумента t1 и присваиваем его переменной IntArg */
31     {
32         printf("Argument t1 = %d", IntArg); /* Выводим числовое значение аргумента t1 из IntArg */
33     }
34     if (argstr(0, "str", TextArg) == true) /* Получаем строковое значение аргумента str и присваиваем его переменной TextArg */
35     {
36         printf("Argument str = \"%s\"", TextArg); /* Выводим строковое значение аргумента str из TextArg */
37     }
38     printf("IntArg = %d", IntArg); /* Выводим числовое значение переменной IntArg */
39 }
```

В строке 24 осуществляется вывод количества переданных в скрипт аргументов. В строках с 26 по 29 выводятся все введенные аргументы. В строках 30 – 33 ведется

поиск и вывод числового аргумента с именем «t1». В строках 34 – 37 ведется поиск и вывод тестового аргумента с именем «str».

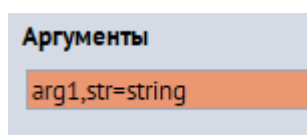
Пример записи параметров скрипта в конфигураторе:



При запуске скрипта с данными аргументами вывод в консоль будет следующий:

```
> RUN /eeprom/testarg.amx,arg1,t1=10,str=string
< RUN=0,/eeprom/testarg.amx,arg1,t1=10,str=string
< I11:21:20 PAWN: Argument count = 3
< I11:21:20 PAWN: Argument 0 = "arg1"
< I11:21:20 PAWN: Argument 1 = "t1=10"
< I11:21:20 PAWN: Argument 2 = "str=string"
< I11:21:20 PAWN: Argument t1 = 10
< I11:21:20 PAWN: Argument str = "string"
< I11:21:20 PAWN: IntArg = 10
```

Если параметр не найден, то у переменной, переданной в функцию, значение не изменяется. К примеру, если задать следующие аргументы:



Вывод в консоль будет следующий:

```
> RUN /eeprom/testarg.amx,arg1,str=string
< RUN=0,/eeprom/testarg.amx,arg1,str=string
< I11:53:25 PAWN: Argument count = 2
< I11:53:25 PAWN: Argument 0 = "arg1"
< I11:53:25 PAWN: Argument 1 = "str=string"
< I11:53:25 PAWN: Argument str = "string"
< I11:53:25 PAWN: IntArg = 25
```

Параметр с именем «t1» не был найден и соответственно не выведен на консоль. Значение переменной «IntArg» осталось равным 25.

Данная особенность позволяет задавать конфигурацию работы скрипта по умолчанию, изменяя только те параметры, которые необходимо.

Приложения А. Встроенные функции платформы

Таблица поддерживаемых функций

Функция	Платформа						
	302	303	310/.В	311	312	314	315
Библиотека tracker.inc							
settag	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
settagf	4.3.7	1.0.0	1.7.0	1.7.0	1.7.0	0.11.0	2.3.6
getinfo	4.5.2	1.2.9	1.8.1	1.8.1	1.8.1	0.11.0	2.3.6
pushpoint	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
getqueue	3.1.5	0.5.0	1.3.12	1.3.12	1.3.12	0.11.0	2.3.6
getprioqueue	3.1.5	0.5.0	1.3.12	1.3.12	1.3.12	0.11.0	2.3.6
getinput	2.15.5	0.5.0	1.2.3	-	1.2.3	0.11.0	2.3.6
getstate	2.15.5	0.5.0	1.2.3	-	1.2.3	0.11.0	2.3.6
setout	2.15.5	0.5.0	1.2.3	-	1.2.3	0.11.0	2.3.6
getposition	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
getspeed	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
getacc	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
getmove	3.1.21	1.0.0	1.3.12	1.3.12	1.3.12	0.11.0	2.3.6
geteep	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
seteep	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
getstatus	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
getincline	2.15.5	0.5.0	-	-	-	-	-
snapshot	2.15.5	0.5.0	-	-	-	-	-
setpwrstate	2.15.8	0.5.0	1.3.3	1.3.3	1.3.3	0.11.0	2.3.6
setpwrwindow	3.1.21	1.0.0	1.3.12	1.3.12	1.3.12	0.11.0	2.3.6
getrawfuel	3.0.6	0.5.0	1.3.3	1.3.3	1.3.3	0.11.0	2.3.6
getibutton	3.0.6	0.5.0	-	-	-	-	-
getowrtemp	3.1.25	1.0.0	-	-	-	-	-
getrfid	3.0.6	0.5.0	-	-	-	-	-
getmdbtag	3.1.16	1.0.0	-	-	-	-	-
getcanlogtag	4.3.3	1.0.0	-	-	-	-	-
@setup	2.15.5	0.5.0	1.3.3	1.3.3	1.3.3	0.11.0	2.3.6
@loop	2.15.5	0.5.0	1.3.3	1.3.3	1.3.3	0.11.0	2.3.6
@chat	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
@accupdate	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
@gnssupdate	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
@bboxupdate	3.1.1	0.5.0	1.3.12	1.3.12	1.3.12	0.11.0	2.3.6
Библиотека camera.inc							
snapshot	-	1.0.0	-	-	-	-	-
setcampwr	-	1.0.0	-	-	-	-	-
getcamera	-	1.1.0	-	-	-	-	-
setcamstr	-	1.2.3	-	-	-	-	-
@snapdone	-	1.1.0	-	-	-	-	-
Библиотека geofence.inc							

Функция	Платформа						
	302	303	310/.B	311	312	314	315
getgeofence	3.1.18	0.5.0	-	-	-	-	-
getgeofname	3.1.18	0.5.0	-	-	-	-	-
Библиотека serial.inc							
rsopen	2.15.5	0.5.0	1.2.6	1.2.6	1.2.6	0.11.0	2.3.6
rsclose	2.15.5	0.5.0	1.2.6	1.2.6	1.2.6	0.11.0	2.3.6
rssend	2.15.5	0.5.0	1.2.6	1.2.6	1.2.6	0.11.0	2.3.6
rsrecv	2.15.5	0.5.0	1.2.6	1.2.6	1.2.6	0.11.0	2.3.6
Библиотека can.inc							
addcanfilter	2.15.5	0.5.0	-	-	-	-	-
cansend	2.15.5	0.5.0	-	-	-	-	-
getcantag	3.0.6	0.5.0	-	-	-	-	-
getfmstag	3.0.6	0.5.0	-	-	-	-	-
getbasetag	4.2.2	1.0.0	-	-	-	-	-
@canrecv	2.15.5	0.5.0	-	-	-	-	-
@canrecv0	3.1.9	1.0.0	-	-	-	-	-
@canrecv1	3.1.9	1.0.0	-	-	-	-	-
Библиотека ble.inc							
bleadvertsubscribe	2.15.5	0.5.0	-/1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
bleadvertsubsmac	2.15.5	0.5.0	-/1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
bleadvertunsubscribe	2.15.5	0.5.0	-/1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
bleadvertbroadcast	4.4.0	1.2.0	-/1.7.0	1.7.0	1.7.0	0.11.0	2.3.6
bleadverttradiosilence	4.4.0	1.2.0	-/1.7.0	1.7.0	1.7.0	0.11.0	2.3.6
getbletag	3.0.6	0.5.0	-/1.3.4	1.3.4	1.3.4	0.11.0	2.3.6
getbleidtag	3.0.6	0.5.0	-/1.3.4	1.3.4	1.3.4	0.11.0	2.3.6
getbleid	2.15.5	0.5.0	-/1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
@bleadvertrecv0-4	2.15.5	0.5.0	-/1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
Библиотека modem.inc							
sendsms	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
sendsmsnum	3.0.6	0.5.0	-	-	-	-	-
sendmsg	4.4.4	1.5.3	-	-	-	-	-
doanswer	3.0.6	0.5.0	-	-	-	-	-
dohang	3.0.6	0.5.0	-	-	-	-	-
iscalling	3.0.6	0.5.0	-	-	-	-	-
getcallerid	3.0.6	0.5.0	-	-	-	-	-
@callerid	3.0.6	0.5.0	-	-	-	-	-
Библиотека args.inc							
argcount	3.0.6	0.5.0	1.9.3	1.9.3	1.9.3	0.11.0	2.3.6
argindex	3.0.6	0.5.0	1.9.3	1.9.3	1.9.3	0.11.0	2.3.6
argstr	3.0.6	0.5.0	1.9.3	1.9.3	1.9.3	0.11.0	2.3.6
argvalue	3.0.6	0.5.0	1.9.3	1.9.3	1.9.3	0.11.0	2.3.6
Библиотека time.inc							
settimer	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
delay	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6
gettime	3.0.6	0.5.0	1.3.12	1.3.12	1.3.12	0.11.0	2.3.6

Функция	Платформа						
	302	303	310/.B	311	312	314	315
getdate	3.0.6	0.5.0	1.3.12	1.3.12	1.3.12	0.11.0	2.3.6
cvttimestamp	3.0.6	0.5.0	1.3.12	1.3.12	1.3.12	0.11.0	2.3.6
@timer	2.15.5	0.5.0	1.2.3	1.2.3	1.2.3	0.11.0	2.3.6

«tracker.inc» - основные функции платформы

Определения:

TAGS_COUNT 32 - максимальное количество параметров для передачи.

EEPROM_COUNT 64 - максимальное количество ячеек, которые можно сохранить в EEPROM.

EEPROM_COUNT 64 - Количество параметров Modbus.

Константы:

```
const status:
{
    Sim1 = 1,           – Номер активной SIM карты. 0-SIM0, 1-SIM1
    NoCon0 = 2,         – Отсутствует соединение с первым сервером
    NoAkb = 4,          – Признак низкого напряжения АКБ (0-норма, 1-низкое)
    NotVal = 5,         – Признак недействительности координат (0 - валидны,
                        1 - невалидны)
    StatNav = 6,        – Координаты зафиксированы при отсутствии движения
    NoExt = 7,          – Признак низкого напряжения питания терминала (0-
                        норма, 1-низкое)
    JamGnss = 9,        – Глушилка Gnss (Обрыв GNSS антенны)
    NetErr = 10,        – Проблема с сотовой сетью (0-норма, 1-проблема)
    HiExt = 11,         – Признак высокого напряжения питания терминала (0-
                        норма, 1-высокое)
    SdCard = 12,        – Данные черного ящика пишутся на SD
    Tamper = 13,        – Обнаружено вскрытие корпуса
    SOS = 15,           – SOS (Тангента)
    DOut0 = 17,         – Состояние дискретного выхода 0
    NoCon1 = 19,        – Отсутствует соединение со вторым сервером
    RemCon = 20,        – Терминал подключен к серверу конфигурирования
    UsbCon = 21,        – Терминал подключен по USB
    UpdCon = 22,        – Терминал подключен к серверу обновлений
    IButton = 23,       – Подключен ключ IButton
    Roaming = 24,       – Находимся в роуминге
    Hosting = 25,       – Терминал привязан к хостингу
    Trimble = 26,       – Координаты получены от внешнего источника NMEA
    BbCorr = 28,        – Черный ящик поврежден
    PwrIdle = 29,       – Режим энергосбережения IDLE
    NoCon2 = 30,        – Отсутствует соединение с третьим сервером
```

```

    PwrSndby = 31,      – Режим энергосбережения Standby
}

const queue_name:
{
    QUEUE_PRIMARY = 0,      - Очередь основного сервера
    QUEUE_SECONDARY = 1,    - Очередь второго сервера
    QUEUE_THIRD = 2,        - Очередь третьего сервера
    QUEUE_HISTORY = 3,      - Очередь истории
}

const input_name:
{
    PWR_AKB = -2,  – Канал напряжения АКБ
    PWR_EXT = -1,  – Канал напряжения питания
    IN0_AIN0 = 0,  – Аналоговый вход 0
    IN1_AIN1 = 1,  – Аналоговый вход 1 (только для УМКа30Х)
    IN1_DIN0 = 1,  – Дискретный вход 0 (только для УМКа31Х)
    IN2_DIN0 = 2,  – Дискретный вход 0 (только для УМКа30Х)
    IN3_DIN1 = 3,  – Дискретный вход 1 (только для УМКа30Х)
}

const pwr_state:
{
    PWR_STATE_RUN = 0,      - Минимальное энергосбережение. Работа.
    PWR_STATE_IDLE = 1,     - Среднее энергосбережение. Бездействие.
    PWR_STATE_STANDBY = 2,  - Максимальное энергосбережение. Ожидание.
}

const fuel_name: – для УМКа302 и УМКа303
{
    /* 0-6 – ДУТ RS-485 */
    FUEL_RS485_0 = 0,
    FUEL_RS485_1 = 1,
    FUEL_RS485_2 = 2,
    FUEL_RS485_3 = 3,
    FUEL_RS485_4 = 4,
    FUEL_RS485_5 = 5,
    FUEL_RS485_6 = 6,
    /* 7-14 – ДУТ BLE */
    FUEL_BLE_0 = 7,
    FUEL_BLE_1 = 8,
    FUEL_BLE_2 = 9,
    FUEL_BLE_3 = 10,
    FUEL_BLE_4 = 11,
    FUEL_BLE_5 = 12,
    FUEL_BLE_6 = 13,
    FUEL_BLE_7 = 14,
}

```



```

/* 15-16 - Аналоговые ДУТ */
FUEL_ANALOG_0 = 15,
FUEL_ANALOG_1 = 16,
/* 17-18 - Частотные ДУТ */
FUEL_FREQUENCY_0 = 17,
FUEL_FREQUENCY_1 = 18,
/* 19-20 - Уровень топлива CAN FMS */
FUEL_FMS_0 = 19,
FUEL_FMS_1 = 20,
/* 21-22 - Уровень топлива CAN Filter */
FUEL_CAN_0 = 21,
FUEL_CAN_1 = 22,
/* 23 - Уровень топлива CAN Base */
FUEL_BASE_0 = 23,
}

const fuel_name: – для УМКа31Х
{
    /* 0-2 - ДУТ RS-485 */
    FUEL_RS485_0 = 0,
    FUEL_RS485_1 = 1,
    FUEL_RS485_2 = 2,
    /* 7-10 - ДУТ BLE */
    FUEL_BLE_0 = 7,
    FUEL_BLE_1 = 8,
    FUEL_BLE_2 = 9,
    FUEL_BLE_3 = 10,
    /* 15 - Аналоговый ДУТ */
    FUEL_ANALOG_0 = 15,
}

const rfid_channel:
{
    RFID_CARD = 0,      – Номер карты
    RFID_RADIO = 1,     – Номер радиометки
    RFID_TEMP = 2,      – Температура радиометки
}

const info_str:
{
    INFO_TYPE = 0,      – Тип терминала
    INFO_NAME = 1,      – Имя, задаваемое пользователем
    INFO_SN = 2,        – Серийный номер
    INFO_FW = 3,        – Версия прошивки
    INFO_IMEI = 4,      – IMEI терминала
}

const canlog_name: – только для УМКа302 и УМКа303
{
    CANLOG_S = 0,       – Security state flags

```

CANLOG_A_B = 1,	- Полное время работы двигателя от 000000.00 до 999999.99 ч
CANLOG_C_D = 2,	- Полный пробег транспортного средства от 0000000.00 до 9999999.99 км
CANLOG_E_F = 3,	- Полный расход топлива от 0000000.0 до 9999999.9л
CANLOG_G_R = 4,	- Уровень топлива в баке от 000.0 до 100.0 % или от 000.0 до 999.9 л
CANLOG_H = 5,	- Скорость оборотов двигателя от 0000 до 9999 об/мин
CANLOG_I = 6,	- Температура двигателя
CANLOG_J = 7,	- Скорость тр средства от 000 до 999 км/ч
CANLOG_K = 8,	- Нагрузка на ось 1 от 00000.0 до 99999.9 кг
CANLOG_L = 9,	- Нагрузка на ось 2 от 00000.0 до 99999.9 кг
CANLOG_M = 10,	- Нагрузка на ось 3 от 00000.0 до 99999.9 кг
CANLOG_N = 11,	- Нагрузка на ось 4 от 00000.0 до 99999.9 кг
CANLOG_O = 12,	- Нагрузка на ось 5 от 00000.0 до 99999.9 кг
CANLOG_P = 13,	- Контроллеры аварии
CANLOG_U_V = 14,	- Уровень жидкости AdBLUE от 000.0 до 100.0 % или от 000.0 до 999.9 л
CANLOG_WA_LO = 15,	- Состояние сельхозтехники. Младшие 32 бита
CANLOG_WA_HI = 16,	- Состояние сельхозтехники. Старшие 32 бита
CANLOG_WB = 17,	- Время жатки от 000000.00 до 999999.99 ч
CANLOG_WC = 18,	- Убранная площадь от 000000.00 до 999999.99 га
CANLOG_WE = 19,	- Количество собранного урожая от 000000.00 до 999999.99 т
CANLOG_WF = 20,	- Влажность зерна от 000.0 до 100.0 %
CANLOG_XB = 21,	- Нагрузка на двигатель от -512 до +511 %

}

Функции:

```
native bool: settag(indx, value, bool: valid = true, pow = 0)
```

Записать значение параметра для дальнейшей передачи.

Аргументы функции:

- `indx` - номер ячейки от 0 до (`TAGS_COUNT` - 1);
- `value` - значение параметра;
- `valid` - достоверность параметра, значение по умолчанию «true» - достоверны;
- `pow` - смещение десятичной точки. К примеру, при `value = 105` и `pow = 1` будет передано значение 10.5. По умолчанию `pow = 0` - число целое;

Функция возвращает true если выполнена успешно и false в случае сбоя.

```
stock bool: settagf(indx, Float: value, bool: valid = true, pow = 0)
```

Записать значение параметра с плавающей точкой для дальнейшей передачи.

Аргументы функции:

- `indx` - номер ячейки от 0 до (`TAGS_COUNT` - 1);
- `value` - значение параметра типа `Float`;
- `valid` - достоверность параметра, значение по умолчанию «true» - достоверны;
- `pow` - смещение десятичной точки. К примеру, при `value = 105` и `pow = 1` будет передано значение 10.5. По умолчанию `pow = 0` - число целое;

Функция возвращает true если выполнена успешно и false в случае сбоя.

```
native pushpoint(bool: hiprio = false)
```

Записать точку в архив для дальнейшей передачи.

Аргументы функции:

- `hiprio` - признак высокоприоритетной точки, по молчанию `false`.

```
native bool: getinfo(info_str: info, str{}, size = sizeof(str))
```

Получить информацию о терминале.

Аргументы функции:

- `info` - тип запрашиваемой информации о терминале, может принимать значения, описанные константой `info_str`;
- `str{}` - упакованный массив, в который осуществляется запись полученной информации о терминале;
- `valid` - длина упакованного массива `str{}`;

Функция возвращает true если выполнена успешно и false в случае сбоя.

```
native getqueue(queue_name: q = QUEUE_PRIMARY)
```

Получить количество точек в очереди на передачу.

Аргументы скрипта:

- `queue_name: q` - очередь, может принимать значения, описанные константой `queue_name`;

```
native getprioqueue(queue_name: q = QUEUE_PRIMARY)
```

Получить количество высокоприоритетных точек в очереди на передачу.

Аргументы скрипта:

- `queue_name: q` - очередь, может принимать значения, описанные константой `queue_name`;

```
native getinput(input_name: in = IN0_AIN0)
```

Получить значение на входе.

Аргументы функции:

- `input_name: in` – номер входа, может принимать значения, описанные константой `input_name`.

Функция возвращает значение, зависящее от настройки входа. Если вход настроен как аналоговый - напряжения на входе в мВ, счётный - значение счётчика и т.д.

```
native bool: getstate(input_name: in = IN0_AIN0)
```

Получение состояния входа.

Аргументы функции аналогичны функции `getinput`.

Функция возвращает `true` - на входе лог «1», `false` - на входе лог «0».

```
native bool: setout(out = 0, bool: st = false, bool: save = false)
```

Установить состояние выхода.

Аргументы функции:

- `out` - номер выхода. Может принимать значения 0 и 1;
- `st` – новое состояние выхода. `true` - выход включён (замкнут), `false` - выключен (разомкнут);
- `save` – сохранить значение выхода в энергонезависимой памяти. `true` - после перезагрузки выход будет в последнем установленном состоянии. Не стоит злоупотреблять записью состояния выхода в энергонезависимой памяти без необходимости.

ВНИМАНИЕ. Для корректной работы функций `getstate`, `getinput` и `setout` входы и выход должны быть соответствующим способом настроены. См. РЭ на УМКа302, УМКа303 и УМКа31Х.

```
native bool: getposition(position[
    Float: .lat, Float: .lon, Float: .height,
    Float: .course, Float: .speed, Float: .hdop])
```

Получить текущие навигационные данные.

Аргументы функции:

- `position` - массив с навигационными данными, где:
 - `.lat` - широта в градусах;
 - `.lon` - долгота в градусах;
 - `.height` - высота над уровнем моря в м;
 - `.course` - направление в градусах;
 - `.speed` - скорость в км/ч;
 - `.hdop` - фактор потери точности в горизонтальной плоскости `hdop`;

Функция возвращает `true` – навигационные данные достоверны, `false` – недостоверны.

```
native getspeed()
```

Получить текущее значение скорости в км/ч.

Аргументы функции: Нет.

Функция возвращает скорость в км/ч или -1 если скорость недостоверна.

```
native bool: getacc(acc[.x, .y, .z])
```

Получить текущие значения ускорений с акселерометра.

Аргументы функции:

- `acc` - массив с ускорениями по осям, где:
 - `.x` - ускорение по оси X в тысячных от g
 - `.y` - ускорение по оси Y в тысячных от g
 - `.z` - ускорение по оси Z в тысячных от g

Функция возвращает `true` - значения достоверны, `false` – недостоверны.

```
native bool: getmove()
```

Получить признак движения от акселерометра.

Аргументов нет.

Функция возвращает `true`, если присутствует признак движения.

```
native geteep(indx)
```

Прочитать значение переменной из EEPROM.

Аргументы функции:

- `indx` - номер ячейки от 0 до `EEPROM_COUNT - 1`

Функция возвращает текущее значение в ячейке EEPROM с запрошенным индексом.

```
native bool: seteep(indx, value, bool: savenow = false)
```

Записать значение в EEPROM.

Аргументы функции:

- `indx` - номер ячейки от 0 до `EEPROM_COUNT - 1`;
- `value` – записываемое значение;
- `savenow` – принудительная запись если `true` и по возможности если `false`. Значение по умолчанию `false`. Запись в EEPROM достаточно продолжительный процесс и желательно что бы система сама определила подходящее для записи время.

Функция возвращает `true` - запись завершилась корректно, `false` - сбой при записи.

ВНИМАНИЕ. Для функций `geteep` и `seteep` одна ячейка всегда занимает 32-бита (4 байта). Сохранять в каждой ячейке можно любое целое число в диапазоне от -2147483648 до 2147483647.

```
native bool: getstatus(status: s)
```

Прочитать значение бита статуса состояния.

Аргументы функции:

- `s` - номер запрашиваемого бита. Номера бит статуса и их названия определены константой `status`.

Функция возвращает `true` - бит статуса установлен, `false` - бит статуса сброшен.

```
native bool: getincline(incline[.x, .y, .z])
```

Получить текущие углы наклона с инклинометра.

Аргументы функции:

- `incline` - массив с наклонами по осям, где:
 - `.x` - угол наклона по оси X, град;
 - `.y` - угол наклона по оси Y, град;
 - `.z` - угол наклона по оси Z, град.

Функция возвращает `true` - значения достоверны, `false` – недостоверны.

```
native bool: snapshot(serv = -1) для УМКа302
```

Сделать фотографию с подключённой камеры и передать ее на указанный сервер.

Аргументы функции:

- `serv` - номер сервера: 0 – основной сервер, 1 – дополнительный сервер, 2 – альтернативный сервер, -1 - сохранить во внутренней памяти;
- `cam` - номер канала фотокамеры. По умолчанию 0.

Функция возвращает `true` - снимок сделан и поставлен в очередь передачи, `false` - ошибка выполнения функции.

ВНИМАНИЕ. К терминалу должна быть подключена камера из списка поддерживаемых и настроен соответствующий интерфейс. Подробнее описано в РЭ на терминал.

```
native bool: setpwrstate(pwr_state: s = PWR_STATE_RUN)
```

Установить режим энергосбережения.

Аргументы функции:

- `pwr_state: s` – устанавливаемый режим. Возможные значения описаны константой `pwr_state`.

Функция возвращает `true` – режим энергосбережения установлен, `false` - ошибка выполнения функции.

```
native bool: setpwrwindow(bool: window = false)
```

Установить режим окна активности.

Аргументы функции:

- `window` – состояние режима окна активности.

Функция возвращает `true`, если режим окна активности успешно установлен.

```
native bool: getrawfuel(fuel_name: in, &fuel)
```

Получить сырой (до применения фильтрации) уровень топлива.

Аргументы функции:

- `fuel_name: in` – номер канала. Может принимать зачения, описанные константой `fuel_name`.
- `&fuel` – переменная, в которую будет записано значение уровня топлива.

Функция возвращает `true` – уровень топлива получен, `false` - ошибка выполнения функции или нет значения.

```
native bool: getibutton(indx = 0, &value)
```

Получить значение канала `iButton`.

Аргументы функции:

- `indx = 0` – номер канала, на текущий момент всегда 0;
- `&value` - переменная, в которую будет записано значение канала `iButton`;

Функция возвращает `true` – значение канала получено, `false` - ошибка выполнения функции или нет значения.

```
native bool: getowrtemp(indx, &Float: value)
```

Получить значение канала температуры 1-wire.

Аргументы функции:

- `indx` – номер канала;
- `&value` - значение температуры типа `Float`;

Функция возвращает статус канала. `true` - канал достоверный.

```
native bool: getrfid(indx = 0, rfid_channel: chan, &value)
```

Получить значение канала RFID-считывателя.

Аргументы функции:

- `indx = 0` – номер считывателя от 0 до 3;
- `rfid_channel: chan` – тип канала данных. Может принимать значения, описанные константой `rfid_channel`.
- `&value` - переменная, в которую будет записано значение канала RFID выбранного считывателя;

Функция возвращает `true` – значение канала получено, `false` - ошибка выполнения функции или нет значения.

```
native bool: getmdbtag(indx, tag[.val, .pow])
```

Получить значение канала Modbus.

Аргументы скрипта:

- `indx` - номер канала;
- `tag` - тег канала, где `val` - значение, `pow` - позиция запятой.

Функция возвращает статус канала. `true` - канал достоверный.

```
native bool: getcanlogtag(canlog_name: indx, tag[ .val, .pow ])
```

Получить значение канала CanLog.

Аргументы скрипта:

- `indx` - номер канала. Может принимать значения, описанные константой `canlog_name`;
- `tag` - тег канала, где `val` - значение, `pow` - позиция запятой.

Функция возвращает статус канала. `true` - канал достоверный.

Функции событийной модели:

```
forward @setup()
```

Инициализация. Вызывается однократно при старте скрипта.

```
forward @loop()
```

Основной цикл. Вызывается периодически.

```
forward @chat(string{128})
```

Командный интерфейс в скриптах.

Передаётся строка, введённая пользователем по команде chat.

Результат выполнения может быть при необходимости помещён в данную строку.

```
forward @accupdate()
```

Событие по обновлению данных от акселерометра.

```
forward @gnssupdate()
```

Событие по обновлению данных о координатах.

```
forward @bboxupdate()
```

Чёрный ящик обновлен (записана точка).

«camera.inc» – функции работы с камерой (только для УМКа303)

Константы:

```
const CamChan:
{
    CAM_ANALOG_0 = 0,    - Аналоговая камера 0
    CAM_ANALOG_1 = 1,    - Аналоговая камера 1
    CAM_DIGIT = 2,       - Цифровая камера
}
```

```
const CamServ:
{
    CAM_SERV_NONE = -1,      - Память терминала
    CAM_SERV_MAIN = 0,      - Основной сервер
    CAM_SERV_ALT = 1,       - Альтернативный сервер
    CAM_SERV_ADD = 2,       - Дополнительный сервер
}
```

Функции:

```
native bool: snapshot(CamServ: serv = CAM_SERV_NONE,
    CamChan: cam = CAM_ANALOG_0, bool: async = false)
```

Сделать фотоснимок и отправить на сервер.

Аргументы скрипта:

- `serv` – номер сервера. Может принимать значения, описанные константой `CamServ`;
- `cam` – канал фотокамеры. Может принимать значения, описанные константой `CamChan`;
- `async` – асинхронный режим съемки.

Функция возвращает `true`, если получение снимка завершилось успешно или процесс съемки успешно запущен при `async = true`.

```
native bool: setcampwr(bool: enable = false)
```

Управление питанием камер.

Аргументы скрипта:

- `enable` – состояние питания камер.

Функция возвращает `true`, если установка состояния питания камер завершилась успешно.

```
native bool: getcamera(camera[ .timestamp, .channel ])
```

Получить данные камеры.

Аргументы скрипта:

- `camera` – данные камеры (`.timestamp` – время последнего снимка, `.channel` – номер канала камеры, совершившей последний снимок).

Функция возвращает `true`, если данные достоверны.

```
native bool: setcamstr(const text{}, CamChan: cam = CAM_ANALOG_0)
```

Установить текст, который будет отображаться на снимке.

Аргументы скрипта:

- `text{}` – текст;
- `cam` – канал фотокамеры. Может принимать значения, описанные константой `CamChan`.

Функция возвращает `true`, если текст успешно установлен.

Функции событийной модели:

```
forward @snapdone(bool: status, serv, cam)
```

Обработчик событий окончания асинхронного снимка.

Аргументы функции:

- `status` – состояние результата съемки;
- `serv` – номер сервера. Может принимать значения, описанные константой `CamServ`;
- `cam` – канал фотокамеры. Может принимать значения, описанные константой `CamChan`.

«geofence.inc» – функции работы с геозонами

Константы:

```
const GEOFENCE_NONE = 0
```

Не в геозоне.

Функции:

```
native bool: getgeofence(&geofence)
```

Получить текущую геозону.

Аргументы скрипта:

- `&geofence` – номер геозоны.

Функция возвращает `true`, если получение геозоны завершилось успешно. Если устройство не находится ни в одной из геозон, по ссылке `&geofence` будет записана константа `GEOFENCE_NONE`.

```
native bool: getgeofname(name{}, size = sizeof(name))
```

Получить имя текущей геозоны.

Аргументы скрипта:

- `name` – буффер имени геозоны;
- `size` – размер строки имени геозоны.

Функция возвращает `true`, если получение имени текущей геозоны завершилось успешно.

«serial.inc» – функции работы с последовательными портами

ВНИМАНИЕ. Перед использованием последовательного порта в скриптах необходимо их настроить на странице «Интерфейсы» указав режим: «Скрипт» и скорость работы по умолчанию.

Константы:

Номер используемого порта:

```
const SerialPort:
{
    SERIAL_PORT_0 = 0,
    SERIAL_PORT_1 = 1,    - только для УМКа302 и УМКа303
}
```

Если для работы со скриптом настроен только один порт (RS485 или RS232) обращение к порту осуществляется по `SERIAL_PORT_0`. В случае, если для

работы со скриптом настроены оба порта, то `SERIAL_PORT_0` соответствует интерфейсу RS485, `SERIAL_PORT_1` соответствует интерфейсу RS232.

ВНИМАНИЕ. Для текущей версии ПО возможность работы с несколькими портами в скрипте не реализована. Во всех функциях для параметра «`port`» рекомендовано использовать значение по умолчанию.

Скорость обмена. `SB_AUTO` соответствует скорости обмена установленной в конфигураторе при настройке интерфейса:

```
const SerialBaud:
{
    SB_AUTO = 0,
    SB_1200 = 1200,
    SB_2400 = 2400,
    SB_4800 = 4800,
    SB_9600 = 9600,
    SB_19200 = 19200,
    SB_38400 = 38400,
    SB_57600 = 57600,
    SB_115200 = 115200,
    SB_230400 = 230400, - только для УМКа302 и УМКа303
    SB_460800 = 460800, - только для УМКа302 и УМКа303
    SB_921600 = 921600, - только для УМКа302 и УМКа303
}
```

Функции:

```
native bool: rsopen(
    SerialBaud: baud = SB_AUTO, SerialPort: port = SERIAL_PORT_0)
```

Открыть порт для обмена.

Аргументы функции:

- `baud` - скорость интерфейса, по умолчанию используется значение, настроенное в конфигураторе;
- `port` - используемый порт, по умолчанию `SERIAL_PORT_0`.

Функция возвращает `true` - порт открыт успешно, `false` - не удалось открыть порт.

```
native bool: rsclose(SerialPort: port = SERIAL_PORT_0)
```

Заккрыть порт.

Аргументы функции:

- `port` - номер порта, который необходимо закрыть, по умолчанию `SERIAL_PORT_0` (RS485).

Функция возвращает `true` - порт закрыт успешно, `false` - не удалось закрыть порт.

ВНИМАНИЕ. Функция открытия порта настраивает скорость порта и блокирует работу по всем интерфейсам RS-485 и RS-232 для остальных задач пока не будет вызвана соответствующая функция закрытия. Рекомендуется регулярно закрывать порт на некоторое время, тем самым освобождать его для использования другими задачами.

```
native bool: rsend(  
    const buff[], size = sizeof(buff), SerialPort: port =  
    SERIAL_PORT_0)
```

Передача данных через порт.

Аргументы функции:

- `buff` - буфер с данными. Массив должен быть не упакованным;
- `size` - размер данных для передачи. По умолчанию размер буфера с данными;
- `port` - порт, через которые необходимо передать данные, по умолчанию `SERIAL_PORT_0` (RS485).

Функция возвращает `true` - данные переданы успешно, `false` - сбой при передаче данных.

ВНИМАНИЕ. Порт перед передачей данных должен быть открыт.

```
native rsrecv(  
    buff[], size = sizeof(buff), first = 0,  
    next = 0, SerialPort: port = SERIAL_PORT_0)
```

Чтение данных из порта.

Аргументы функции:

- `buff` - буфер для чтения данных. Массив должен быть не упакованным.
- `size` - максимальный размер данных который необходимо прочитать. По умолчанию размер буфера.

- `port` - порт, через которые необходимо получить данные, по умолчанию `SERIAL_PORT_0` (RS485).
- `first` - максимальное время ожидания первого байта данных в мсек. По умолчанию 0.
- `next` - максимальное время ожидания последующих байт данных в мсек. По умолчанию 0.

Функция возвращает количество полученных данных.

ВНИМАНИЕ. Порт перед чтением данных должен быть открыт.

Пример использования:

```

1#include <time>
2#include serial
3
4#pragma dynamic      256
5
6main()
7{
8    new Buff[128];
9    new Size;
10
11    for(;;)
12    {
13        if (rsopen() == true) // Открываем порт RS485 на скорости по умолчанию и проверяем успешность операции
14        {
15            Size = rsrecv(Buff,_, 100, 100) // Читаем данные из порта
16            rsend(Buff, Size); // Прочитанные данные выводим в порт
17            rsclose();
18        }
19        delay(100); // Повторяем каждые 100 мсек
20    }
21}

```

Скрипт открывает порт, каждые 100 мсек считывает данные из него и выводит полученные данные обратно в порт.

«can.inc» – функции работы с CAN (только для УМКа302 и УМКа303).

Определения:

`CANFILERS_COUNT 32` – количество CAN фильтров.

Константы:

Номер используемого порта:


```
const CanPort:
{
    CAN_PORT_0 = 0,
    CAN_PORT_1 = 1,
}
```

Индексы параметров FMS:

```
const fms_indx:
{
    FMS_INDX_TFU = 0,      - Полный расход топлива
    FMS_INDX_FL = 1,      - Уровень топлива в баке
    FMS_INDX_ECT = 2,     - Температура двигателя
    FMS_INDX_ES = 3,      - Обороты двигателя
    FMS_INDX_ETH = 4,     - Время работы двигателя
    FMS_INDX_HRTVD = 5,   - Пробег транспортного средства
    FMS_INDX_EPL = 6,     - Нагрузка на двигатель
    FMS_INDX_APP = 7,     - Позиция педали акселератора
    FMS_INDX_AW1 = 8,     - Нагрузка на ось 1
    FMS_INDX_AW2 = 9,     - Нагрузка на ось 2
    FMS_INDX_AW3 = 10,    - Нагрузка на ось 3
    FMS_INDX_AW4 = 11,    - Нагрузка на ось 4
    FMS_INDX_AW5 = 12,    - Нагрузка на ось 5
    FMS_INDX_HRLFC = 13,  - Полный расход топлива высокой точности
    FMS_INDX_FL2 = 14,    - Уровень топлива во втором баке
    FMS_INDX_VS = 15,     - Скорость транспортного средства
}
```

Индексы параметров CAN BASE:

```
const base_indx:
{
    CANBASE_INDX_HOUR = 0, - Время работы двигателя в часах
    CANBASE_INDX_VDHR = 1, - Пробег транспортного средства
    CANBASE_INDX_LFC = 2,  - Полный расход топлива
    CANBASE_INDX_LFE = 3,  - Расход топлива в л/ч
    CANBASE_INDX_FL = 4,   - Уровень топлива в л или %
    CANBASE_INDX_RPM = 5,  - Скорость вращения двигателя
    CANBASE_INDX_ET = 6,   - Температура двигателя
    CANBASE_INDX_VS = 7,   - Скорость транспортного средства
}
```

Функции:

```
native bool: addcanfilter(id, CanPort: port = CAN_PORT_0)
```

Добавить фильтр сообщений по идентификатору сообщения CAN.

Аргументы функции:

- `id` - идентификатор CAN сообщения. Возвращаемое значение: `true` - фильтр успешно добавлен, `false` - фильтр не добавлен.
- `CanPort: port` - CAN порт. Варианты описаны константой `CanPort`.

```
native bool: cansend(id, dlc, data[8], CanPort: port = CAN_PORT_0)
```

Передача сообщения в CAN шину.

Аргументы функции:

- `id` - идентификатор CAN сообщения;
- `dlc` - размер сообщения от 0 до 8;
- `data` – данные;
- `CanPort: port` - CAN порт.

Функция возвращает `true` - сообщение успешно добавлено в очередь на передачу, `false` – ошибка параметров или в очереди на передачу не было свободного места.

```
native bool: getcantag(indx, tag[.val, .pow])
```

Получить значение канала датчика (фильтра) CAN.

Аргументы функции:

- `indx` – номер канала от 0 до 31;
- `tag[.val, .pow]` – данные канала, `val` - значение, `pow` - позиция запятой.

Функция возвращает `true` – значение канала получено, `false` - ошибка выполнения функции или нет значения.

ВНИМАНИЕ. Перед использованием функции, в устройстве должны быть настроены CAN фильтры для вашего транспортного средства. Подробнее описано в РЭ на терминал.

```
native bool: getfmstag(fms_indx: indx, tag[ .val, .pow ])
```

Получить значение канала датчика FMS.

Аргументы функции:

- `fms_indx: indx` – номер канала, может принимать значения, описанные константой `fms_indx`;
- `tag[.val, .pow]` – данные канала, `val` - значение, `pow` - позиция запятой.

Функция возвращает `true` – значение канала получено, `false` - ошибка выполнения функции или нет значения.

```
native bool: getbasetag(base_indx: indx, tag[.val, .pow])
```

Получить значение канала датчика CAN BASE.

Аргументы функции:

- `base_indx: indx` - номер канала, может принимать значения, описанные константой `base_indx`;
- `tag` - тег канала, где `val` - значение, `pow` - позиция запятой.

Функция возвращает `true`, если значение успешно получено.

Функции событийной модели:

```
forward @canrecv0(id, dlc, data[8])
```

```
forward @canrecv1(id, dlc, data[8])
```

Обработчик сообщений от соответствующего CAN (0 или 1).

Аргументы функции:

- `id` - идентификатор принятого CAN сообщения;
- `dlc` - размер поля данных;
- `data` - принятые данные.

```
forward @canrecv(id, dlc, data[8])
```

Общий обработчик для всех необработанных ранее сообщений от всех CAN-интерфейсов.

Аргументы функции:

- `id` - идентификатор принятого CAN сообщения;
- `dlc` - размер поля данных;
- `data` - принятые данные.

«ble.inc» - функции работы с BLE

Определения:

Определения, упрощающие обработку данных:

`BLE_GAP_LEN_OFFS` — позиция длины сообщения в буфере
`BLE_GAP_TYPE_OFFS` — позиция типа сообщения в буфере
`BLE_GAP_DATA_OFFS` — позиция начала данных в буфере

`BLEID_COUNT` — количество каналов идентификации по BLE
`BLEDEV_COUNT` — количество устройств BLE

Константы:

Типы сообщения:

```
const ble_gap_type:
{
    BLE_GAP_FLAGS = 0x01,           — Flags
    BLE_GAP_SHORTENED_LOCAL_NAME = 0x08, — Shortened Local Name
    BLE_GAP_COMPLETE_LOCAL_NAME = 0x09, — Complete Local Name
    BLE_GAP_MFR_SPEC_DATA = 0xff,   — Manufacturer Specific Data
}
```

```
const ble_offs:
{
    BLE_OFFS_FUEL = -2, - Канал топлива
    BLE_OFFS_TEMP = -1, - Канал температуры
    BLE_OFFS_PARAM0 = 0,
    BLE_OFFS_PARAM1 = 1,
    BLE_OFFS_PARAM2 = 2,
    BLE_OFFS_PARAM3 = 3,
```

```

BLE_OFFS_PARAM4 = 4,
BLE_OFFS_PARAM5 = 5,
BLE_OFFS_PARAM6 = 6,
BLE_OFFS_PARAM7 = 7,
}

```

Функции:

```

native bool:bleadvertisubscribe(
    ble_gap_type:gap = BLE_GAP_MFR_SPEC_DATA, bleadvertrecv = 0)

```

Подписка на все BLE сообщения указанного типа.

Аргументы функции:

- `gap` - тип сообщения (значения описаны константой `ble_gap_type`);
- `bleadvertrecv` - номер обработчика сообщения от 0 до 3. По умолчанию 0.

Функция возвращает `true` - подписка добавлена, `false` - ошибка добавления подписки.

```

native bool: bleadvertisubsmac(
    mac[6], ble_gap_type: gap = BLE_GAP_MFR_SPEC_DATA,
    bleadvertrecv = 0)

```

Подписка на все BLE сообщения указанного типа по указанному MAC-адресу.

Аргументы функции:

- `mac` – MAC-адрес BLE датчика;
- `gap` – тип сообщения (значения описаны константой `ble_gap_type`);
- `bleadvertrecv` – номер обработчика сообщения от 0 до 3. По умолчанию 0.

Функция возвращает `true` - подписка добавлена, `false` - ошибка добавления подписки.

```

native bool: bleadvertunsubscribe(bleadvertrecv)

```

Отписать обработчик от всех сообщений.

Аргументы функции:

- `bleadvertrecv` – номер обработчика сообщения от 0 до 3. По умолчанию 0.

Функция возвращает `true` - подписки удалены, `false` - ошибка удаления подписок.

```
native bool: bleadvertbroadcast(data[], len = sizeof(data),
    ble_gap_type: gap = BLE_GAP_MFR_SPEC_DATA)
```

Начать или завершить вещание рекламного сообщения.

Аргументы функции:

- `data[]` – данные рекламного сообщения;
- `len` – длина рекламного сообщения от 1 до 26 байт или 0 что бы завершить вещание;
- `gap` – тип сообщения (значения описаны константой `ble_gap_type`).

Функция возвращает `true` – вещание успешно начато, `false` - ошибка установления вещания.

```
stock bool: bleadvertradiosilence()
```

Завершить вещание рекламного сообщения.

Аргументов нет.

Функция возвращает `true` – вещание успешно остановлено, `false` - ошибка прекращения вещания.

```
native bool: getbletag(dev = 0, ble_offs: offs, tag[ .val, .pow ])
```

Получить значение канала датчика BLE.

Аргументы функции:

- `dev = 0` – номер устройства от 0 до 7, по умолчанию 0;
- `ble_offs: offs` – номер канала данных, может принимать значения, описанные константой `ble_offs`;
- `tag[.val, .pow]` – данные канала, `val` - значение, `pow` - позиция запятой.

Функция возвращает `true` – значение канала получено, `false` - ошибка выполнения функции или нет значения.

```
native bool: getbleidtag(indx, tag[.val, .dist])
```

Получить идентификатор и расчетное расстояние iBeacon устройства по индексу.

Аргументы функции:

- `indx` - канал прослушивания от 0 до 3;
- `tag[.val, .dist]` - данные канала, `val` - идентификатор iBeacon, `dist` - расстояние;

Функция возвращает `true` - успешное чтение идентификатора, `false` - идентификатор не считан.

```
native bool: getbleid(indx, &value)
```

Получить идентификатор iBeacon устройства по индексу.

Аргументы функции:

- `indx` - канал прослушивания от 0 до 3;
- `&value` - ссылка на переменную для записи идентификатора iBeacon.

Функция возвращает `true` - успешное чтение идентификатора, `false` - идентификатор не считан.

Функции событийной модели:

```
forward @bleadvertrecv0(mac[6], rssi, data[31], len)
```

```
forward @bleadvertrecv1(mac[6], rssi, data[31], len)
```

```
forward @bleadvertrecv2(mac[6], rssi, data[31], len)
```

```
forward @bleadvertrecv3(mac[6], rssi, data[31], len)
```

Соответствующая функция будет вызвана при поступлении BLE сообщения согласно ранее настроенным подпискам.

Аргументы функции:

- `mac` – mac адрес устройства, передавшего сообщение;
- `rssi` – уровень сигнала dbm;
- `data` – принятые данные;
- `len` – длина сообщения.

«modem.inc» - функции работы с модемом

Функции:

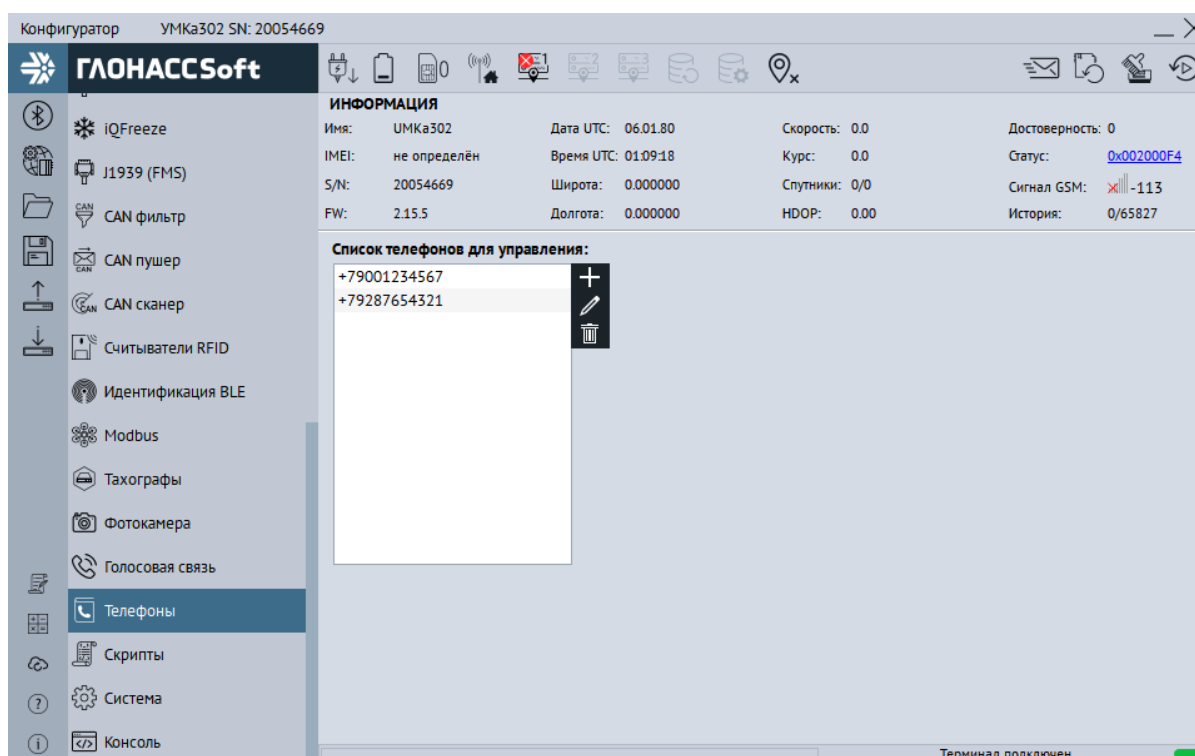
```
native bool: sendsms(const text{}, indx = 0)
```

Отправить SMS на номер из списка авторизованных. Для УМКа31Х функция находится в библиотеке tracker.inc.

Аргументы функции:

- `text{}` - текст SMS, только латинскими буквами;
- `indx` - номер записи от 0 до 4 в списке телефонов для управления, по умолчанию 0.

Функция возвращает `true` - SMS поставлена в очередь передачи, `false` - ошибка. Необходимо убедиться, что список номеров заполнен и обращение осуществляется к записи с корректным номером.



```
native bool: sendsmsnum(const text{}, const number{})
```

Отправить SMS на произвольный номер.

Аргументы функции:

- `text{}` - текст sms, только латинскими буквами;
- `number{}` - телефонный номер в виде строки, к примеру "+79871234567".

Функция возвращает `true` - SMS поставлена в очередь передачи, `false` - ошибка.

```
native bool: sendmsg(const text{}, indx = 0)
```

Отправить сообщение на сервер.

Аргументы функции:

- `text{}` – текст сообщения;
- `indx` – номер сервера.

Функция возвращает `true` – успешно, `false` - ошибка.

```
native bool: doanswer()
```

Ответить на входящий звонок.

Аргументов нет.

Функция возвращает `true` – успешно, `false` - ошибка.

```
native bool: dohang()
```

Сбросить входящий звонок.

Аргументов нет.

Функция возвращает `true` – успешно, `false` - ошибка.

```
native bool: iscalling()
```

Проверить, есть ли входящий звонок или установлено голосовое соединение.

Аргументов нет.

Функция возвращает `true` – есть входящий вызов или установлено голосовое соединение, в противном случае функция возвращает `false`.

```
native bool: getcallerid(number{}, size = sizeof(number))
```

Авто определение номера входящего звонка или установленного голосового соединения.

Аргументы функции:

- `number{}` – строка в которую будет записан номер в случае успешного выполнения функции; Строка должна быть размером не менее 13 символов;
- `size = sizeof(number)` – размер буфера для номера, по умолчанию равен размеру строки `number{}`.

Функция возвращает `true` – если есть входящий вызов или установлено голосовое соединение и удалось определить номер, в противном случае функция возвращает `false`.

Функции событийной модели:

```
forward @callerid(const number{})
```

Функция будет вызвана, если имеется входящий вызов и номер определен.

Аргументы функции:

- `const number{}` – номер звонящего абонента.

Функции работы с временем

```
native gettime(&hour=0, &minute=0, &second=0)
```

Получить текущее время.

Аргументы функции:

- `hour` - час;
- `minute` - минута;
- `second` - секунда.

Функция возвращает количество секунд, прошедших с 1 января 1970 года (unix time).

```
native getdate(&year=0, &month=0, &day=0)
```

Получить текущую дату.

Аргументы функции:

- `year` – год;
- `month` – месяц;
- `day` – день.

Функция возвращает количество дней, прошедших с начала года.

```
native cvttimestamp(  
    seconds1970, &year=0, &month=0, &day=0, &hour=0, &minute=0,  
    &second=0)
```

Преобразовать unix time в дату и время.

Аргументы функции:

- `seconds1970` - unix time;
- `year, month, day, hour, minute, second` – ссылка на переменные, куда будут записаны дата и время;

Функция всегда возвращает 0.

Более подробно работа с временем средствами языка Pawn описана в файле `Time_Functions.pdf`.

Приложение Б. Список прилагаемых файлов с описанием языка

Документация

- Pawn_Getting_Started.pdf - общее описание языка;
- Pawn_Language_Guide.pdf - подробное руководство по языку Pawn;
- Floating_Point_Support - описание библиотеки работы с числами с плавающей запятой;
- Fixed_Point_Support.pdf - описание библиотеки работы с числами с фиксированной запятой;
- String_Manipulation.pdf - описание библиотеки работы со строками;
- Time_Functions.pdf - описание библиотеки работы с временем и таймером;
- Arguments_Support.pdf – описание библиотеки работы с аргументами;

Примеры

- demo.p - демонстрация работы скриптовой системы «My logic»;
- timer.p - пример работы с таймером;
- testsms.p - пример работы с входом и отправкой sms;
- testport.p - пример работы с последовательным портом;
- testinp.p - пример чтения значения входа и управления выходом;
- testcan.p- пример работы с CAN шиной;
- testblea.p - пример работы с BLE;
- testacc.p - пример работы с акселерометром по событию;
- tagfloat.p - пример работы с числами с плавающей точкой и записью параметров в ЧЯ;
- pp1.p - скрипт по чтению параметров с 4-х датчиков ПП-01. Пример подключения пользовательских библиотек, работы с реальным оборудованием;
- du_label.p - Скрипт реализующий дополнительный функционал датчика угла наклона DU BLE. Использование датчика угла как метки;
- antijam.p - пример чтения бита статуса устройства (JamGnss - глушение GNSS);
- и управления выходом в зависимости от изменения статуса;
- openble.p - управление выходом устройства по BLE;
- testarg.p – примеры работы с аргументами скрипта;
- testmdm.p – примеры использования функций библиотеки работы с модемом;
- testtime.p – пример работы с функциями времени;