

# Введение в DaqScript.

## Оглавление

Введение в DaqScript.....	1
Аннотация.....	3
Общий обзор и назначение DaqScript.....	3
Режимы работы интерпретатора DaqScript.....	4
Синтаксис интерпретатора DaqScript.....	5
Литералы.....	6
Переменные.....	7
Встроенные константы.....	7
Встроенные функции.....	8
Встроенные команды.....	10
Команда @echo.....	11
Команды @system и @global.....	11
Команда @async.....	11
Команда @voice.....	11
Команда @mmtimer.....	11
Оператор декларации переменных var.....	12
Оператор присвоения переменной =.....	12
Приоритеты операций.....	12
Оператор скобки ().....	12
Унарные операторы знака + -.....	13
Арифметические операторы + - * / % ^.....	13
Условный оператор if ... then .....	13
Оператор метки : и перехода goto.....	13
Оператор вызова подпрограмм gosub ... return и exit.....	14
Особенности реализации DaqScript.....	14
Рекомендации по программированию в DaqScript.....	15
Главная Консоль и Системный Калькулятор.....	18
Принцип работы Главной Консоли.....	18
Константы Системного Калькулятора.....	19
Команды Системного Калькулятора.....	20
Команда @adam.....	20
Команда @compile.....	21
Команда @daq.....	21
Команда @env.....	21
Команда @file.....	22
Команда @fonts.....	22
Команда @guard.....	23
Команда @help.....	23
Команда @if.....	23
Команда @integrity.....	24
Команда @list.....	26
Команда @log.....	27
Команда @memory.....	27
Команда @menu.....	28
Команда @note.....	28
Команда @onexception.....	29
Команда @open.....	30
Команда @pid.....	30
Команда @polling.....	31

Команда @run.....	32
Команда @silence.....	33
Команда @silent.....	33
Команда @sleep.....	34
Команда @speak.....	34
Команда @speech.....	35
Команда @sysinfo.....	35
Команда @term.....	35
Команда @testbench.....	37
Команда @textmetadata.....	37
Команда @tooltip.....	37
Команда @tty.....	39
Команда @view.....	40
Практика применения Главной Консоли.....	41
Ручное управление через Главную Консоль.....	41
Стартовые скрипты для инициализации программ.....	42
Вызов команд Главной Консоли из сценариев Script.....	43
Вызов команд Главной Консоли из программ DaqPascal.....	43
Вызов команд Главной Консоли из ядра CRW-DAQ.....	43
Сценарии контроля целостности @integrity.....	43
Удаленный вызов команд Главной Консоли.....	43
Применение DaqScript в DAQ системе.....	44
Применение DaqScript в программах DaqPascal.....	44
Применение DaqScript в сценариях Script.....	45
Объявление устройств Script.....	45
Описание конфигурации устройств Script.....	45
Дополнительные функции и команды устройств Script.....	46
Пример простой конфигурации устройства Script.....	49
Практика использования устройств Script.....	49
Применение DaqScript в мнемосхемах.....	50
Общие сведения о мнемосхемах.....	50
Интерпретатор Painter.....	52
Формула TagEval(v).....	53
Формула LedEval(v).....	53
Сценарий Painter(v).....	54
Применение DaqScript в расчетах.....	56
Инструмент Калькулятор.....	56
Инструмент Plot2D.....	56
Инструмент Plot3D.....	57
Макросы анализа данных в окнах с кривыми.....	57
Утилиты анализа данных в окнах с кривыми.....	59
Заключение.....	61
Список сокращений.....	62
Список источников.....	63

## Аннотация

Программный пакет **CRW-DAQ** [1,2,3] имеет несколько встроенных специализированных языков, ориентированных на решение разных задач. Например, язык **DaqConfig** служит для описания конфигураций измерительных систем, а **DaqPascal** – для создания алгоритмов сбора данных и управления. Этот документ содержит описание языка **DaqScript**, который создан для интерпретации «на лету» поступающих (**online**) в каналы связи команд в режиме исполнения (**runtime**).

## Общий обзор и назначение DaqScript

Язык **DaqScript** – простой командный интерпретатор, служащий для решения тех задач, которые, в силу своей специфики, принципиально не могут быть решены с помощью компиляторов. В первую очередь это касается интерпретации команд и выражений (формул), поступающих в виде сообщений по каналам связи или вводимых пользователем в интерактивном режиме (что является частным случаем канала связи). Также **DaqScript** применяется для расширения возможностей различных статически откомпилированных объектов пакета за счет добавления к ним возможности интерпретации команд или сообщений, поступающих от пользователя или других программ по каналам связи.

Говоря точнее, в пакете **CRW-DAQ** обычно работает много (десятки) экземпляров интерпретатора **DaqScript**, каждый из которых «привязан» к какому-то объекту и имеет (помимо общих) свои специфические функции, как показано на рисунке 1. Все экземпляры **DaqScript** имеют общий синтаксис и набор функций общего применения, но могут отличаться дополнительным набором функций, определяемых спецификой объекта привязки. В результате обработки команд, поступающих через каналы связи или пакетов (сценариев) экземпляр **DaqScript** оказывает воздействие на присоединенный объект. Он также выводит результаты в консоль (если она есть) и возвращает результат расчетов в виде вещественного числа (**double**).

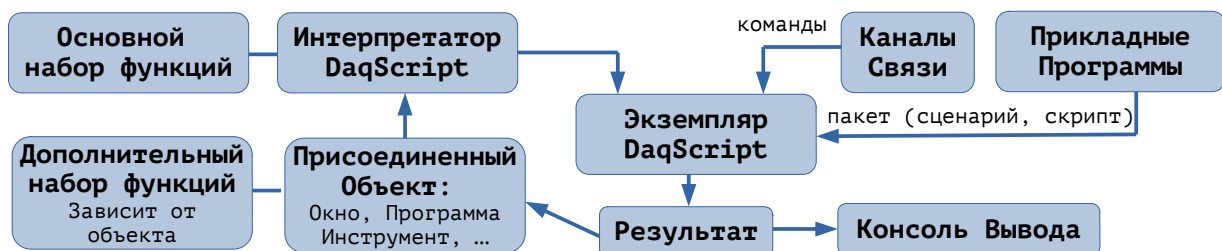


Рисунок 1. Общая схема работы интерпретатора **DaqScript**.

В таблице 1 перечислены сферы применения интерпретатора **DaqScript**. В окне **Главной Консоли** пакета **CRW-DAQ** объект **DaqScript** интерпретирует пользовательские команды, позволяя гибко управлять пакетом. В окне **Калькулятор** объект **DaqScript** позволяет выполнять простые расчеты в интерактивном режиме. В программах **DaqPascal** объект **DaqScript** позволяет расширить возможности компилятора за счет добавления интерпретации поступающих команд и выражений. В программах **DAQ Script** объект **DaqScript** позволяет быстро создавать простые прикладные сценарии для систем управления и сбора данных.

В окнах мнемосхем объект **DaqScript** позволяет рисовать фигуры, заданные пользовательским сценарием. В инструментах **Plot2D**, **Plot3D** объект **DaqScript** позволяет генерировать данные и рисовать графики функций. В окнах кривых объект **DaqScript** позволяет обрабатывать данные с помощью **макросов** (скриптов) на языке **DaqScript**. Как видим, **DaqScript** имеет много важных применений. Поэтому пакет **CRW-DAQ** сложно представить без этого интерпретатора.

*Таблица 1. Сферы применения интерпретатора **DaqScript**.*

№	Объект привязки	Выполняемые функции
1	<b>Главная Консоль</b>	Обработка пользовательского ввода, т. е. интерпретация и выполнение команд, вводимых пользователем в консоли или поступающих в консоль от других программ путем передачи сообщений.
2	<b>Инструменты\Калькулятор</b>	Вычисление выражений (формул), вводимых пользователем в интерактивном режиме.
3	Программа <b>DaqPascal</b> (device software program)	Добавление возможности интерпретации в скомпилированные программы <b>DaqPascal</b> . Каждая программа имеет свой экземпляр интерпретатора, доступный через функции <b>Eval</b> (вычисление выражений) и <b>Evar</b> (задание значений переменных).
4	Программа <b>DaqScript</b> (device software daqscript)	Выполнение простых программ (сценариев) сбора данных и управления на языке <b>DaqScript</b> . В основном используется для быстрого создания симуляторов.
5	Окно мнемосхемы ( <b>Circuit_Window</b> )	Рисование сенсоров в окне мнемосхем с помощью сценариев <b>Painter(v)</b> .
6	<b>Инструменты\График 2D</b>	Диалог построения графика кривой, заданной по формуле.
7	<b>Инструменты\График 3D</b>	Диалог построения графика поверхности, заданной по формуле.
8	Окно кривых ( <b>Curve_Window</b> ) <b>Кривая/Макрос анализа данных</b>	Выполнение макросов (сценариев) для обработки данных (кривых), заданных в окне.
9	Окно кривых ( <b>Curve_Window</b> ) <b>Кривая/Утилита анализа данных</b>	Интерпретация выражений (пользовательского ввода) в программах обработки данных на языке <b>Object Pascal</b> .

Следует отметить, что интерпретатор создавался с ориентацией на системы сбора данных и управления, поэтому он имеет ряд важных свойств – 1) предсказуемое время отклика, 2) большую надежность, 3) высокую устойчивость к ошибкам входных данных. Это позволяет использовать его в системах управления (мягкого) реального времени (**realtime**) и системах с высокими требованиями к надежности работы.

## Режимы работы интерпретатора **DaqScript**

Интерпретатор **DaqScript** может работать в двух режимах: **командном** и **пакетном**. В **командном** режиме каждая строка входного текста рассматривается как независимая, непосредственно выполняемая команда (директива), не связанная напрямую с другими командами. Она интерпретируется и выполняется для каждой строки отдельно. В таком режиме работают объекты, связанные с обработкой потоков событий, например, **Главная Консоль** и **Калькулятор**. В этом случае каждая команда, поступающая от пользователя или сообщение из канала связи, рассматривается как независимое событие. Функция **eval** в компиляторе **DaqPascal** также работает в командном режиме. Мнемосхемы (**Circuit\_Window**) обычно тоже работают в командном режиме, хотя могут работать и в пакетном режиме.

В **пакетном** режиме некоторый заданный текст (сценарий, скрипт) рассматривается как целое, в котором строки текста связаны друг с другом. Это позволяет выполнять ветвления (**if**) и переходы по меткам (**goto**), организовывать подпрограммы (**gosub**) и циклы. В пакетном режиме работает, например, программа **DaqScript** в системе сбора данных **DAQ**, а также графопостроители (**Инструменты\График 2D**, **Инструменты\График 3D**) и **макросы** в окнах кривых (**Curve\_Window**).

Режим работы конкретного экземпляра интерпретатора определяется характером решаемой задачи. Обработка потока событий всегда выполняется в командном режиме, т. к. в этом случае отдельные строки входного текста заранее не предсказуемы и не могут рассматриваться как единый текст. Если же выполняемые команды заранее известны и образуют единый текст (пакет, сценарий, скрипт), тогда их можно выполнять в пакетном режиме.

## Синтаксис интерпретатора DaqScript

Интерпретатор **DaqScript** имеет простой синтаксис, который можно описать в виде набора правил:

- Входной текст задан в виде **строк** в кодировке **ANSI** (8 бит на символ) в текущей кодовой странице операционной системы или в **UTF-8**. Строки разделяются символами **CR**, **LF**. Строки состоят из **слов**, разделенных **пробелами** и **специальными символами**.
- Интерпретатор **не чувствителен** к регистру символов, т. е. строки, заданные в верхнем и нижнем регистре рассматриваются интерпретатором как эквивалентные.
- Интерпретатор распознает символы «a»..**z**», «0»..**9**», «.», «\_», « », «**Tab**», «+», «-», «\*», «/», «%», «^», «(», «)», «,», «=», «:», «;», «\$», «@». Подробнее см. таблицу 2.
- **Идентификаторы** (слова, означающие названия переменных, констант, встроенных функций) могут содержать символы a..**z**, \_, 0..**9** и не должны начинаться с цифры, т. е. должны начинаться с буквы «a»..**z**» или знака подчеркивания «\_».
- Символ «;» точки с запятой отделяет **комментарий**, который продолжается до конца строки.
- Первый символ «@» является признаком **команды** (акции, директивы), которые обрабатываются специальным образом и могут содержать в качестве аргументов строковые **литералы**.
- Поддерживается один **тип данных** для переменных, констант и функций - числовой (вещественный, тип **double**, **IEEE 754**). Целые числа рассматриваются как частный случай (подмножество) вещественных чисел. Массивы и строки не предусмотрены.
- В **командном** режиме интерпретатор поддерживает следующие конструкции (см. таблицу 3): числовые **литералы**, **переменные**, встроенные **константы**, встроенные **функции**, встроенные **команды**, скобки «(...)», арифметические операторы «+-\*/^%», оператор присвоения «=», оператор **var** объявления переменных.
- В **пакетном** режиме интерпретатор (сверх того) поддерживает дополнительные конструкции (см. таблицу 3): условный оператор ветвления **if...then...**, оператор метки «:» и перехода **goto**, оператор вызова подпрограммы **gosub..return** и выхода **exit**.
- Интерпретатор возвращает численный результат (**double**) или значение **NaN** при ошибке интерпретации выражения.

Таблица 2. Алфавит интерпретатора DaqScript.

№	Символ(ы)	Область применения
1	0..9	Цифры (например, 12.34) и идентификаторы (например, x1, y2).
2	a..z, _	Идентификаторы, например: width, x1, _y1.
3	. (точка)	Отделяет целую и дробную часть числа.
4	Пробел, Tab	Разделитель слов.
5	CR, LF	Разделитель строк.
6	, (запятая)	Разделитель аргументов функций.
7	;	Разделитель комментария.
8	+ -	Знаки чисел. Операторы сложения и вычитания.
9	* / % ^	Операторы умножения, деления, остатка по модулю, возведения в степень.
10	( )	Скобки для изменения порядка операций и задания аргументов функций.
11	=	Оператор присвоения и создания/удаления переменных.
12	\$	Признак шестнадцатеричного числа (hexadecimal).
13	@	Признак команды (акции, директивы).

Таблица 3. Синтаксические конструкции интерпретатора DaqScript.

№	Конструкция	Описание
1	литералы	Числовые (вещественные или целые) константы.
2	переменные	Поименованные числовые данные, которые могут менять значения.
3	встроенные константы	Поименованные числовые данные, не меняющие значения.
4	встроенные функции	Поименованные действия с числовыми аргументами и результатом.
5	встроенные команды	Поименованные действия со строковым аргументом.
6	скобки ( )	Для изменения порядка операций и задания аргументов функций.
7	оператор var	Оператор объявления переменных или создания новых переменных.
8	оператор =	Оператор присвоения, а также создания/удаления переменных.
9	операторы +-*/^%	Арифметические операторы для действий с числами.
10	метки:	Помечает место для переходов (goto, gosub) в пакетном режиме.
11	if ... then ...	Оператор ветвления по условию в пакетном режиме.
12	goto L	Безусловный переход по метке L: в пакетном режиме.
13	gosub S	Вызов подпрограммы по метке S: в пакетном режиме.
14	return	Возврат из подпрограммы gosub в пакетном режиме.
15	exit	Досрочный выход из сценария в пакетном режиме.

## Литералы

Литералы – это постоянные значения, выраженные в явной нотации (т. е. способе записи), а не в виде символического имени. Интерпретатор **DaqScript** поддерживает числовые литералы, т. е. целые или вещественные числа, выраженные в виде чисел. Для целых чисел допускается десятичная или шестнадцатеричная нотация (с префиксом \$). Для вещественных используется десятичная нотация, в том числе с использованием научной нотации (с мантиссой и экспонентой после символа E). Строковые литералы допускаются только в аргументах команд (с префиксом @). Например:

1024	Целое число в десятичной нотации.
\$FE08	Целое число в шестнадцатеричной нотации.
3.1415	Вещественное число в фиксированной нотации.
1.024E+3	Вещественное число в научной нотации.
@echo Hello	Строковый литерал в аргументе команды @echo.

Независимо от типа нотации числовые литералы преобразуются в вещественные числа типа **double** (64 бит, **IEEE 754**). Целые числа рассматриваются как подмножество вещественных. При преобразовании целых чисел используется 32-битное представление числа.



## Переменные

**Переменные** в **DaqScript** – это поименованные данные (типа **double**), которые могут менять свои значения и доступны по своему символическому имени. Переменные могут динамически создаваться, удаляться и менять свое значение.

Создание переменных производится двумя способами: либо явно – оператором декларирования переменных «**var name1, name2, ...**», либо неявно – оператором присвоения «**name=value**». При явной декларации переменной «**var name**» значение существующей переменной не меняется, а несуществующая переменная создается с нулевым значением. Это свойство часто используется, например, для организации счетчиков событий с начальным нулевым значением.

Изменение значения существующей переменной производится путем присвоения нового значения «**name=value**».

Удаление переменной производится пустым присвоением «**name=**».

Например:

<code>x=1</code>	Создание переменной <code>x</code> с присвоением значения 1.
<code>var x,y</code>	Декларация переменных <code>x,y</code> . При этом будет <code>x=1,y=0</code> .
<code>x=x+1</code>	Переменная <code>x</code> получает значение 2.
<code>y=y+1</code>	Переменная <code>y</code> получает значение 1.
<code>@echo x=%x, y=%y.</code>	Печать переменных: <code>x=2, y=1</code> .
<code>x=</code>	Удаление переменной <code>x</code> .
<code>y=</code>	Удаление переменной <code>y</code> .

Следует отметить, что существует возможность множественного присвоения переменных в одном выражении, которое организуется с помощью скобок или в аргументах вызова функций, например:

<code>(x=1)+(y=2)+(z=3)</code>	Присвоение переменным значений <code>x=1, y=2, z=3</code> .
<code>x=sin(fi=2*pi*t)</code>	Вычисление выражения с присвоением значений <code>fi</code> и <code>x</code> .

Такое множественное присвоение значений переменных в одном выражении делает текст более компактным, а также увеличивает производительность за счет повышения скорости интерпретации.

Следует сказать, что скорость интерпретации зависит (хотя и незначительно) от длины имен, т. к. при интерпретации происходит поиск переменных в таблице переменных. Поэтому не следует использовать длинные имена переменных без необходимости.

## Встроенные константы

**Константы** в **DaqScript** – это поименованные числовые данные, которые не могут изменять свои значения после создания. В текущей версии интерпретатора имеются только **встроенные** константы, которые определяются при создании экземпляра объекта **DaqScript** и не могут быть определены пользовательским кодом.

В распечатке 1 показан основной набор констант **DaqScript**, который определен для любого экземпляра объекта **DaqScript**. При присоединении объекта к этому списку может быть добавлены также и другие константы. Некоторые константы (такие как **e**, **pi**, **false**, **true**) являются универсальными, другие (такие как **macheps**, **minint**, **maxint** и т.д.) определяются архитектурой компьютера. Есть также специальные константы (такие как **\_nan**, **\_inf**, **\_minusinf**), обозначающие исключительные значения «не число» и «бесконечность». Так, «не число» **\_nan** обозначает результат ошибки интерпретации или некорректной математической операции типа «логарифма от -1». «Бесконечность» **\_inf** возникает, например, при делении на ноль. Наконец, константы сеансового уровня (например, **processid**) действительны только в рамках данного сеанса работы программы.

**Распечатка 1. Основной набор встроенных констант DaqScript.**

Список констант:		
<code>_inf</code>	<code>= INF</code>	Значение «бесконечность».
<code>_minusinf</code>	<code>= -INF</code>	Значение «минус бесконечность».
<code>_nan</code>	<code>= NaN</code>	Значение «не число», признак ошибки.
<code>e</code>	<code>= 2.71828182845905</code>	Основание натурального логарифма.
<code>false</code>	<code>= 0</code>	Логический ноль или «ложь».
<code>macheps</code>	<code>= 2.22044604925031E-16</code>	Машинная точность типа double.
<code>maxbyte</code>	<code>= 255</code>	Максимальное значение типа Byte.
<code>maxdouble</code>	<code>= 1.7E308</code>	Максимальное значение типа Double.
<code>maxdword</code>	<code>= 4294967295</code>	Максимальное значение типа DWORD.
<code>maxint</code>	<code>= 2147483647</code>	Максимальное значение типа Integer.
<code>maxlongint</code>	<code>= 2147483647</code>	Максимальное значение типа LongInt.
<code>maxlongword</code>	<code>= 4294967295</code>	Максимальное значение типа LongWord.
<code>maxshortint</code>	<code>= 127</code>	Максимальное значение типа ShortInt.
<code>maxsingle</code>	<code>= 3.4E38</code>	Максимальное значение типа Single.
<code>maxsmallint</code>	<code>= 32767</code>	Максимальное значение типа SmallInt.
<code>maxword</code>	<code>= 65535</code>	Максимальное значение типа Word.
<code>mindouble</code>	<code>= 4.94065645841247E-324</code>	Минимальное значение типа Double.
<code>minint</code>	<code>= -2147483648</code>	Минимальное значение типа Integer.
<code>minlongint</code>	<code>= -2147483648</code>	Минимальное значение типа LongWord.
<code>minshortint</code>	<code>= -128</code>	Минимальное значение типа ShortInt.
<code>minsingle</code>	<code>= 1.5E-45</code>	Минимальное значение типа Single.
<code>minsmallint</code>	<code>= -32768</code>	Минимальное значение типа SmallInt.
<code>pi</code>	<code>= 3.14159265358979</code>	Число $\pi$ .
<code>processid</code>	<code>= 5004</code>	Идентификатор текущего процесса.
<code>true</code>	<code>= 1</code>	Логическая единица или «истина».

**Встроенные функции**

**Функции в DaqScript** – это поименованные действия над списком числовых аргументов, возвращающие числовой результат. Список аргументов помещается в скобки `()`, а сами аргументы в списке разделяются запятыми, например, `log(2,10)`. Скобки при вызове функций обязательны, даже если список аргументов пуст.

В **DaqScript** допускаются только **встроенные** функции, которые определяются на этапе компиляции пакета. Пользовательские функции в **DaqScript** не поддерживаются.

Основной набор встроенных функций, которые доступны в любом экземпляре объекта **DaqScript**, приведены на распечатке 2. Кроме этого основного набора объекты **DaqScript**, имеющие присоединенный объект, могут содержать и другие встроенные функции, дающие доступ к этому присоединенному объекту, как показано на рисунке 1.

Функции основного набора можно разбить на группы по назначению.

Группы арифметических функций (**abs**, **exp**, **gamma**, **hypot**, **ln**, **log**, **max**, **min**, **rand**, **random**, **sign**, **sqrt**), тригонометрических и гиперболических функций (**acos**, **asin**, **atan**, **acos**, **cos**, **cosh**, **deg**, **rad**, **sin**, **sinh**, **tan**, **tanh**) служат для математических расчетов. Группа функций округления (**ceil**, **floor**, **frac**, **int**, **round**, **trunc**) преобразуют вещественные числа в целые.

Группа логических функций и функций сравнения (**eq**, **ne**, **ge**, **gt**, **le**, **lt**, **not**, **and**, **or**, **xor**) служат для логических расчетов и отличаются тем, что возвращают только логические значения (**0** или **1**), что позволяет «смешивать» логические операции с арифметическими. При этом в логических расчетах значение **0** воспринимается как **false**, а любое ненулевое значение – как **true**.



**Распечатка 2. Основной набор встроенных функций DaqScript.**

Список функций:	
abs(a)	Модуль числа a.
acos(a)	Обратный косинус числа a.
and(a,b)	Логическое И: a AND b.
asin(a)	Обратный синус числа a.
atan(a)	Обратный тангенс числа a.
bitand(a,b)	Побитовое И: a AND b.
bitnot(a)	Побитовая инверсия числа a.
bitor(a,b)	Побитовое не исключающее ИЛИ: a OR b.
bitxor(a,b)	Побитовое исключающее ИЛИ: a XOR b.
ceil(a)	Округление числа a в большую сторону.
cga2rgb(a)	Вычислить RGB цвет для CGA цвета a.
cos(a)	Косинус числа a.
cosh(a)	Косинус гиперболический числа a.
cpu_clock()	Счетчик тактов CPU от момента вызова cpu_start.
cpu_count()	Счетчик числа доступных логических процессоров.
cpu_start()	Начать измерение счетчика тактов CPU.
deg(a)	Перевод радиан числа a в градусы.
eq(a,b)	1 если a=b или 0 – от «equal».
exp(a)	Натуральная экспонента числа a.
floor(a)	Округление числа a в сторону 0.
frac(a)	Дробная часть числа a.
gamma(a)	Гамма-функция числа a: $\gamma(a)=(a-1)!$
ge(a,b)	1 если a>=b или 0 – от «greater or equal».
getbitmask(a)	Число $2^a$ для целого числа a.
getclockres(a)	Узнать разрешение таймера в мс.
getpid()	Идентификатор текущего процесса.
getppid()	Идентификатор родительского процесса.
getticks()	Время BIOS в «тиках» (55 мс).
gt(a,b)	1 если a>b или 0 – от «greater then».
hypot(a,b)	Гипотенуза $\sqrt{a^2+b^2}$ .
int(a)	Округление числа a в сторону 0.
isbit(a,b)	1 если в a есть бит номер b или 0.
isinf(a)	1 если a=INF или 0.
isnan(a)	1 если a=NaN или 0.
le(a,b)	1 если a<=b или 0 – от «less or equal».
ln(a)	Натуральный логарифм числа a.
log(a,b)	Логарифм b по основанию a.
lt(a,b)	1 если a<b или 0 – от «less then».
max(a,b)	Максимальное из a,b.
min(a,b)	Минимальное из a,b.
mksecnow()	Время в микросекундах от старта.
msecnow()	Время в миллисекундах от Р.Х.
ne(a,b)	1 если a<>b или 0 – от «not equal».
not(a)	1 если a<>0 или 0 – логическая инверсия числа a.
or(a,b)	Логическое не исключающее ИЛИ: a OR b.
pidaffinity(a,b)	Задать маску b привязки процесса a к ядрам CPU.
rad(a)	Перевод градусов числа a в радианы.
rand()	Случайное число от 0 до 1.
random(a,b)	Случайное число от a до b.
rgb(a,b,c)	Вычислить цвет (R,G,B)=(a,b,c).
round(a)	Округление числа a до ближайшего целого.
secnow()	Время в секундах от Р.Х.
setclockres(a)	Задать разрешение таймера в a [мс].
sign(a)	Функция знака числа a: возвращает -1,0,+1.
sin(a)	Синус числа a.
sinh(a)	Синус гиперболический числа a.
sqrt(a)	Корень квадратичный числа a.
tan(a)	Тангенс числа a.
tanh(a)	Тангенс гиперболический числа a.
trunc(a)	Округление числа a в сторону 0.
xor(a,b)	Логическое исключающее ИЛИ – a XOR b.

Группа битовых функций (**bitand**, **bitnot**, **bitor**, **bitxor**) позволяет выполнять побитовые логические операции над целыми 32-битными числами. При этом аргументы функций неявно преобразуются в целые 32-битные числа, над которыми и совершаются действия.

Группа функций времени (**cpu\_clock**, **cpu\_start**, **getticks**, **getclockres**, **msecnow**, **mksecnow**, **secnow**, **setclockres**) позволяют измерять время и работать с таймерами.

Группа служебных функций **OC** (**cpu\_count**, **getpid**, **getppid**, **pidaffinity**) позволяет взаимодействовать с операционной системой.

## Встроенные команды

**Команды** (акции) в **DaqScript** — это поименованные действия над аргументами (включая числовые и строковые литералы), возвращающие числовой результат. В **DaqScript** поддерживаются только **встроенные** команды, определяемые на этапе компиляции.

Признаком команды является первый символ «@», за которым идет идентификатор команды и (необязательные) аргументы, идущие до конца строки. Аргументы рассматриваются (в общем случае) как строковые данные, подлежащие специальной интерпретации. Например:

@echo Привет мир!	Вывести сообщение «Привет мир!» в Главную Консоль.
@voice welcome	Проиграть звуковой файл с именем welcome.wav.

Команды возвращают числовое значение (**double**) с результатом выполнения. Кроме того, этот результат сохраняется в переменной **actionresult**. При ошибке интерпретации команды возвращают **NaN**.

Переменные и константы могут передаваться в аргументы команд с помощью **подстановки %n**, где **n** — имя переменной. При подстановке текущее (числовое) значение переменной преобразуется в строку (в формате по умолчанию **%g**) и подставляется в аргумент. При необходимости другого числового формата этот формат ставится непосредственно перед знаком подстановки. Например:

@echo x = %x	Подстановка значения переменной x.
@echo pi = %pi	Подстановка значения константы pi.
@echo pi = %7.2f%pi	Подстановка значения константы pi в формате %7.2f

Основной набор встроенных команд, которые доступны в любом экземпляре объекта **DaqScript**, приведены на распечатке 3. Кроме этого основного набора объекты **DaqScript**, имеющие присоединенный объект, могут содержать и другие встроенные команды, дающие доступ к этому присоединенному объекту, как показано на рисунке 1.

### Распечатка 3. Основной набор встроенных команд (акций) **DaqScript**.

Список акций:	Эти команды общие для всех объектов DaqScript.
@async a	Выполнить команду a асинхронно (послать в Главную Консоль).
@echo a	Вывод сообщения a в консоль (обычно в Главную Консоль).
@global a	Выполнение выражения a в Системном Калькуляторе.
@mmtimer a	Чтение\установка периода a мультимедийного таймера, [мс].
@system a	Выполнение выражения a в Системном Калькуляторе.
@voice a	Проигрывание звукового wav-файла с именем a.

Особую роль в пакете **CRW-DAQ** играет **Системный Калькулятор** — объект **DaqScript**, соединенный с **Главной Консолью**, как показано на рисунке 3. **Системный Калькулятор** отвечает за обработку потока команд, управляющих работой пакета и выполняется с соблюдением потоковой безопасности. Он также связан с **очередью событий (FIFO)**, в которую другие потоки могут помещать команды для асинхронной обработки в основном потоке.

Любой экземпляр **DaqScript** имеет связь с [Главной Консолью](#) и [Системным Калькулятором](#) через команды [@echo](#), [@system](#), [@global](#), [@async](#), описанные ниже.

### Команда **@echo**

Команда **@echo** помещает свой аргумент в консольный буфер **FIFO** для последующей печати в консольное окно (если оно подключено). Обычно это [Главная Консоль](#) или консоль **DAQ** устройства, с которым связан объект **DaqScript**. Печать всегда идет асинхронно, что резко снижает нагрузку **CPU** при большом потоке данных, но может вносить некоторую задержку исполнения, связанную с переключением потоков.

### Команды **@system** и **@global**

Команды **@system** и **@global** выполняют выражение, данное в аргументе, в [Системном Калькуляторе](#). Поскольку вызов команд может происходить в любом потоке, на время исполнения [Системный Калькулятор](#) блокируется и выполняет команды в критической секции кода, с соблюдением потоковой безопасности. Таким образом, каждый объект **DaqScript** имеет связь с [Главной Консолью](#) и [Системным Калькулятором](#), что позволяет использовать **DaqScript** как универсальный способ обмена данными и сообщениями между потоками и объектами пакета.

### Команда **@async**

Команда **@async** вместо непосредственного выполнения помещает свой аргумент в **FIFO** очередь входящих команд [Главной Консоли](#) для асинхронного выполнения в основном потоке программы, как показано на рисунке 3. Очередь команд обрабатывается по таймеру с периодом  $\approx 50$  мс в основном потоке пакета **CRW-DAQ**. Извлекаемые команды выполняются в [Системном Калькуляторе](#). В силу этого аргумент команды **@async** исполняется с некоторой задержкой, связанной с переключением потоков. Однако положительной стороной является то, что вызывающий поток не задерживается на выполнение операций.

### Команда **@voice**

Команда **@voice** проигрывает звуковой **wav** файл, имя которого передается в аргументе. Обычно это имя библиотечного файла без пути и расширения. Путь к файлам библиотеки звуков описывается в конфигурации пакета в переменной **Crw32.ini [System] SoundLibrary**.

Примеры:

<b>@echo</b> Привет, друг.	Печать сообщения в консоль.
<b>@voice</b> siren	Проиграть звук siren.wav в вызывающем потоке.
<b>@async @voice</b> siren	Проиграть звук асинхронно, в основном потоке.
<b>@system</b> ms=msecnow()	Записать время в переменную ms в Системном Калькуляторе.

### Команда **@mmtimer**

Команда **@mmtimer** управляет мультимедийным таймером, который непосредственно влияет на частоту системного таймера, который отвечает за переключение потоков. Обычно период системного таймера **10** или **16 мс**, а частота **100** или **64 Гц** для простых или многоядерных систем соответственно. С помощью вызова **@mmtimer n** можно задать период таймера **n** в диапазоне от **1** до **16 мс**, добиваясь частоты опроса потоков до **1000 Гц**.

Команда **@mmtimer** работает, но считается устаревшей. Вместо нее рекомендуется использовать функцию **setclockres**.

## Оператор декларации переменных var

Оператор **var** служит для декларации переменных, которые задаются именем или списком имен, разделенных запятой. Например:

<code>var a</code>	Декларация одной переменной <code>a</code> .
<code>var x,y,z</code>	Декларация списка переменных <code>x,y,z</code> .

При декларировании переменные создаются с начальным значением 0, если они еще не были созданы. Существующие переменные не меняют своего текущего значения. Это свойство часто используется для создания переменных (например, счетчиков) с начальным нулевым значением.

## Оператор присвоения переменной =

Оператор присвоения «**name=value**» служит для присвоения переменной с именем **name** значения выражения **value**, а также для создания переменной или её удаления. Непустое присвоение (при наличии выражения **value**) автоматически создает переменную **name**, если она не существовала. А пустое присвоение (без значения **value**) напротив удаляет переменную. Например:

<code>x=1</code>	Создает переменную <code>x</code> и присваивает ей значение 1.
<code>x=x+1</code>	Присваивает переменной <code>x</code> новое значение 2.
<code>x=</code>	Пустое присвоение удаляет переменную.

Скобки позволяют объединить несколько операторов присвоения в одной строке, что делает программу компактнее и быстрее. Например:

<code>(x=1)+(y=2)+(z=3)</code>	Объединение присвоений <code>x=1</code> , <code>y=2</code> , <code>z=3</code> .
<code>x=sin(fi=2*pi*t)</code>	Другой способ объединения присвоений <code>fi</code> , <code>x</code> .
<code>z=max(x=1,y=2)</code>	Так тоже можно присваивать.

## Приоритеты операций

Операции в выражениях **DaqScript** выполняются в соответствии с **уровнем приоритетов**, приведенных в таблице 4. При вычислении выражения сначала выполняются операции с наиболее высоким приоритетом, затем с более низким, пока все операции не будут исчерпаны. Операции с одинаковым приоритетом выполняются по мере поступления, т. е. слева направо.

Таблица 4. Таблица приоритетов операций DaqScript.

Уровень	Операции	Комментарии
6	<code>() f()</code>	Скобки <code>()</code> , аргументы функций <code>f()</code> .
5	<code>+ -</code>	Унарные операторы знака.
4	<code>^</code>	Возведение в степень.
3	<code>* / %</code>	Умножение, деление, остаток деления по модулю.
2	<code>+ -</code>	Сложение и вычитание.
1	<code>=</code>	Присвоение.

## Оператор скобки ()

Скобки служат для двух целей. Во-первых, скобки ограничивают список аргументов функций. Во-вторых, скобки меняют порядок выполнения операций при вычислении выражений. Например:

<code>x=log(2,pi)</code>	Скобки ограничивают список аргументов функции.
<code>x=(e+2)*5</code>	Скобки меняют порядок выполнения операций.

Порядок выполнения операций при интерпретации выражения определяется уровнем приоритета операций, в соответствии с таблицей 4. При необходимости изменить порядок вычисления используются скобки. Скобки имеют приоритет над всеми другими операциями и заставляют вычислять часть выражения, помещенную в скобки, как отдельное выражение, в приоритетном порядке. Кроме того, скобки в ряде случаев позволяют объединять несколько операций присвоения в одной строке. Такое объединение позволяет сделать текст программ более компактным, а также увеличивает скорость выполнения операций. Например:

<code>(x=1)+(y=2)+(z=3)</code>	Объединение присвоений <code>x=1, y=2, z=3</code> .
<code>x=sin(fi=2*pi*t)</code>	Другой способ объединения присвоений <code>fi, x</code> .

## Унарные операторы знака + -

Унарные операторы `+` `-` задают знак последующего выражения. Они имеют высший после скобок и функций приоритет, как показано в таблице 4. Например:

<code>x=-(a+b)</code>	Унарный оператор «минус».
<code>y=+(a-b)</code>	Унарный оператор «плюс».

## Арифметические операторы + - \* / % ^

Оператор возведения в степень `x^y` (`x` в степени `y`) имеет следующий приоритет после знаков, как показано в таблице 4.

Операторы `*` `/` `%` умножения, деления, остатка деления по модулю имеют следующий приоритет после степени.

За ними по приоритету следуют операторы `+` `-` сложения и вычитания.

Самый низкий приоритет имеет оператор `=` присвоения. Он выполняется в самом конце вычислений, присваивая переменной окончательный результат. Например:

<code>x=2^10</code>	Вычисляет 2 в степени 10.
<code>x=y*z+2</code>	Сначала выполняется умножение, потом сложение.

## Условный оператор if ... then ...

Условный оператор `if a then b` доступен только в пакетном режиме исполнения. Он вычисляет выражение `a` и затем, при условии истинности результата, выполняет выражение `b`. Условное выражение `a` считается истинным, если результат имеет любое ненулевое значение, и ложным при нулевом значении результата. При вычислении условия `a` удобно использовать логические функции. Например:

<code>if x then z=1</code>	Если <code>x &gt; 0</code> , то <code>z=1</code> .
<code>if le(x,0) then y=0</code>	Если <code>x &lt;= 0</code> , то <code>y=0</code> .

## Оператор метки : и перехода goto

Операторы метки `«:»` и перехода `«goto»` доступны только в пакетном режиме исполнения, когда входные выражения представляют собой связный текст (пакета, сценария, скрипта), а не независимые друг от друга команды (как при обработке потока событий).

Метка представляет собой идентификатор с последующим (без пробела) знаком двоеточия `«:»`. Назначение метки – отмечать место в сценарии, к которому возможен переход из другого места сценария.

Оператор перехода `goto` по метке принимает в качестве аргумента идентификатор метки (без двоеточия) и выполняет переход в то место, где расположена эта метка.

Оператор перехода зачастую используется совместно с условным оператором. Использование переменной, условного оператора, метки и перехода позволяет организовывать циклы. Например:

n=5	Число циклов.
i=0	Переменная цикла.
loop:	Метка начала цикла loop
i=i+1	Увеличиваем значение i в цикле
echo Iteration %i	Распечатываем номер итерации i
if lt(i,n) then goto loop	Проверяем условие для завершения цикла

## Оператор вызова подпрограмм **gosub ... return** и **exit**

Операторы вызова подпрограммы **gosub** и возврата **return** доступны только в пакетном режиме исполнения.

Оператор вызова подпрограммы **gosub** принимает в качестве аргумента идентификатор метки и выполняет переход по метке, подобно оператору **goto**. Однако, в отличие от **goto**, оператор **gosub** запоминает во внутреннем стеке адрес вызова. Это позволяет вернуться в точку вызова после завершения необходимых действий.

Оператор возврата **return** берет из стека вызовов адрес и возвращает управление в точку последнего вызова для продолжения вычислений.

Пример с циклом и вызовом подпрограммы:

n=5	Число циклов.
i=0	Переменная цикла.
loop:	Метка начала цикла loop.
gosub iteration	Вызов подпрограммы по метке iteration.
if lt(i,n) then goto loop	Проверяем условие для завершения цикла.
goto done	Переход по метке завершения работы done.
iteration:	Метка подпрограммы iteration.
i=i+1	Увеличиваем значение i в цикле.
echo Iteration %i	Распечатываем номер итерации i.
return	Возврат из подпрограммы.
done:	Метка завершения done.
@echo Выполнено %i циклов.	Печать результата.

Оператор вызова подпрограммы **gosub** не предполагает передачу аргументов. Все данные, используемые в подпрограмме, хранятся в общих переменных. Подпрограммы не имеют локальных переменных.

Оператор выхода **exit**, доступный в пакетном режиме, позволяет завершить выполнение сценария досрочно.

## Особенности реализации DaqScript

Интерпретатор **DaqScript** создавался для обработки событий в системе сбора данных и управления пакета **CRW-DAQ**. Интерпретатор специально разработан для работы в режиме опроса (**polling**), когда очередь событий обрабатывается путем многократного вызова (обычно по таймеру) интерпретатора. Интерпретатор сохраняет состояние всех своих внутренних данных после вызова, поэтому может продолжать вычисления при последующих вызовах.

Изначально ставилась задача обеспечить высокую надежность, отказоустойчивость и скорость выполнения команд. Поток входных данных изначально предполагается «агрессивным», т. е. содержащим возможность ошибок и некорректных данных. Никакие ошибки входных данных не должны приводить к аварийному завершению программы.

Простая структура интерпретатора и многоуровневая защита от ошибок (включая тщательную проверку всех входных данных и структурированную обработку исключений **SEH**) обеспечили высокую надежность и отказоустойчивость его работы. За более чем **20**-летнюю



практику работы пакета **CRW-DAQ** не зафиксировано случаев аварийного сбоя программ по вине **DaqScript**, несмотря на значительное число ошибок во входных данных, возникающих в процессе разработки, отладки и эксплуатации прикладных программ.

Для обеспечения высокой скорости работы предприняты следующие меры. Динамические переменные, константы, функции и команды хранятся в специальном списке имен, поиск в котором осуществляется с использованием методов хеширования и двоичного поиска. При поиске имен в списке сначала выполняется вычисление 32-битного хеша-кода имени, затем двоичный поиск в отсортированном списке хеш-кодов. Это значит, что при длине имени **L** и размере таблицы имен **N** поиск требует порядка  **$O(L + \log_2 N)$**  целочисленных операций **CPU**.

По факту, типичная простая операция, типа присвоения переменной значения, занимает порядка **1000** машинных циклов. Например, для процессора **i7-2.5GHz** это составит порядка **250** наносекунд на оператор **DaqScript**, что соответствует скорости около **4** миллионов простых операторов **DaqScript** в секунду. Этот результат мало зависит от размера программ и длины имен переменных. Поэтому интерпретатор пригоден для использования в программах, критичных к времени отклика, например, в системах управления (мягкого) реального времени.

Исходный код **\_EE.PAS** интерпретатора находится в каталоге **Resource\Dcc32\SYSLIB**. Интерпретатор можно, помимо прочего, использовать в качестве встроенного командного языка при создании автономных программ.

## Рекомендации по программированию в DaqScript

Здесь перечислены некоторые простые рекомендации, которые помогут эффективно использовать **DaqScript** для решения прикладных задач.

- Используйте скобки для объединения нескольких присвоений в одном выражении. Множественное присвоение повышает скорость работы и порой улучшает «читабельность» программ. Например:

<code>(x=1)+(y=2)+(z=3)</code>	Три присвоения в одной строке – компактно и быстро.
<code>c=max(a=1,b=2)</code>	Так тоже можно присваивать.

Для сравнения, эквивалентный код выглядит намного длиннее и выполняется медленнее:

```
x=1
y=2
z=3
a=1
b=2
c=max(a,b)
```

- Используйте скобки в сложных выражениях для повышения ясности и «читабельности» кода программ. Особенно в тех случаях, когда порядок выполнения операций не очевиден. Например:

<code>z=(2^x)*y</code>	Хотя скобки тут не нужны, они повышают «читабельность» кода.
<code>z=2^x*y</code>	А тут не сразу поймешь, в каком порядке выполняются операции.

Влияние введения скобок на производительность пренебрежимо мало, так что ставьте нужные скобки без лишних сомнений.

- Помните, что в **DaqScript** все переменные являются **глобальными**. Поэтому в подпрограммах, где используются временные «локальные» переменные, рекомендуется использовать имена, которые начинаются и заканчиваются знаком подчеркивания «\_», например, **\_x\_**, **\_y\_**. Это позволит легко отделить обычные переменные (где редко используются знаки подчеркивания) от «локальных» переменных из подпрограмм.

- Используйте оператор декларирования переменных **var** для повышения ясности и надежности программ. Декларация переменной ничему не мешает, но может устранить потенциальную ошибку обращения к несуществующей переменной.

<code>var count</code>	Декларирование счетчика count.
<code>count=count+1</code>	Следующее значение счетчика.
<code>@echo Счетчик = %count</code>	Печать счетчика в консоль.

- Имейте в виду, что создание и удаление переменной является сравнительно «дорогой» по времени выполнения операцией. Для повышения производительности избегайте необоснованного создания и удаления переменных, особенно в циклах. Переменные, созданные оператором **var** или присвоением (=), лучше сохранять на будущее и никогда не удалять. Присвоение новых значений уже существующим переменным идет намного быстрее, чем создание новых переменных.

- Имейте в виду, что логические функции (**eq**, **ne**, **le**, **lt**, **ge**, **gt**, **not**, **and**, **or**, **xor**) всегда возвращают только **false** или **true**, т. е. значения **0** или **1**. Поскольку в **DaqScript** логические значения одновременно являются еще и числовыми, их можно использовать также и в арифметических расчетах.

- Имейте в виду, что в аргументах логических функций (**not**, **and**, **or**, **xor**) значение **0** воспринимается как **false**, а любое ненулевое значение — как **true**. Поэтому арифметические выражения могут участвовать также и в логических расчетах.

- Используйте логические функции (**eq**, **ne**, **le**, **lt**, **ge**, **gt**, **not**, **and**, **or**, **xor**), в сочетании с арифметическими операторами (+ - \* /) и функциями для ведения логических и числовых расчетов. Поскольку логические функции всегда возвращают **0** или **1**, воспринимая в аргументах **0** как **false**, а любое другое значение как **true**, это позволяет смешивать в одном выражении логические и арифметические операторы, формируя довольно сложную логику работы. Например, для генерации меандра (серии периодических импульсов) с заданной скважностью **q** и периодом **p** можно использовать такие выражения:

<code>var t0</code>	Декларируем начальное время t0.
<code>t0=t0+secnow()*eq(t0,0)</code>	Определяем начальное время t0.
<code>meander=le(frac((secnow()-t0)/p),q)</code>	Формируем меандр с периодом p.

Такого рода выражения часто используются для генерации управляющих сигналов и формирования логики в системах управления. Например, для генерации 1-секундного управляющего импульса (**pulse**) для включения условного клапана на **20**-й секунде процесса, начавшегося в момент **t0**, можно использовать формулу:

<code>t=secnow()-t0</code>	Время от начала процесса в секундах.
<code>pulse=ge(t,20)*le(t,21)</code>	Импульс между 20-й и 21-й секундой процесса.

Подобного рода управляющие формулы часто вводятся оператором через графический интерфейс или загружаются их файла конфигурации.

- Из соображений производительности желательно по возможности заменять управляющие операторы (**if...then**, **goto**, **gosub...return**, **exit**) на эквивалентные арифметические выражения, которые выполняются быстрее. При такой замене большую помощь оказывают логические функции (**eq**, **ne**, **le**, **lt**, **ge**, **gt**, **not**, **and**, **or**, **xor**), в сочетании с арифметическими операторами (+ - \* /). Например:

if le(a,b) then c=0 c=c*le(a,b)	Это выражение обнуляет c, если $a \leq b$ . Его можно заменить эквивалентным простым выражением.
------------------------------------	---

Эквивалентное арифметическое выражение часто выглядит более компактным и вычисляется гораздо быстрее.

Другой пример. Допустим, надо сделать расчет цвета **color** для индикатора уровня с двумя порогами тревоги (**alarm1**, **alarm2**):

```
color=clGreen
if gt(v,alarm1) then color=clYellow
if gt(v,alarm2) then color=clRed
```

Условные операторы **if** в этом программном коде можно заменить одним эквивалентным арифметическим выражением:

```
color=clGreen*le(v,alarm1)+clYellow*gt(v,alarm1)*le(v,alarm2)+clRed*gt(v,alarm2)
```

Использование эквивалентных арифметических выражений может ускорить интерпретацию и сделать код более компактным и понятным.

- Имейте в виду, что команды (акции) сохраняют результат своего выполнения в предопределенной переменной **actionresult**. Например:

```
size=@file size Temp\Debug.Out
@echo Размер файла %size байт.
```

ЭКВИВАЛЕНТНО

```
@file size Temp\Debug.Out
@echo Размер файла %actionresult% байт.
```

Использование **actionresult** иногда позволяет избежать введения лишних переменных. Но надо помнить, что любая команда при вызове меняет значение **actionresult**.

- Имейте в виду, что при обнаружении любых ошибок интерпретации (например, в случае обращения к несуществующей переменной или при синтаксической ошибке) выполнение выражения прерывается, а в качестве результата интерпретации будет возвращено значение **NaN**.

- Имейте в виду, что при прерывании интерпретации выражения по причине ошибки какая-то часть выражения может успеть выполниться прежде, чем ошибка будет обнаружена. Это может иметь побочные эффекты. Например:

<b>a=</b>	Удаляем переменную а.
<b>b=</b>	Удаляем переменную b.
<b>fi=</b>	Удаляем переменную fi.
<b>x=max(a=pi,b=fi)</b>	Вычисляем такое выражение. Что получится?
Неизвестная переменная.	Возникла ошибка «Неизвестная переменная».
<b>x=max(a=pi,b=fi)</b>	Исходное выражение, где найдена ошибка.
^	Указание места обнаружения ошибки.
<b>fi</b>	Указание ошибочного выражения.
<b>a</b>	Проверяем значение переменной а.
3.14159265358979	Это число π. Присвоение успешно.
<b>b</b>	Проверяем значение переменной b.
Неизвестная переменная.	Не найдена. Присвоения не было.

В данном примере возникает ошибка интерпретации по причине обращения к неизвестной переменной **fi**. Однако присвоение значения переменной **a=pi** происходит успешно, т. к. оно выполняется до обнаружения ошибки.

## Главная Консоль и Системный Калькулятор

Окно «Главная Консоль», изображенное на рисунке 2, является одним из основных управляющих элементов пакета **CRW-DAQ**, наряду с «Меню», «Панелью Инструментов» и т. д. Роль этого окна больше, чем кажется на первый взгляд, т. к. оно служит не только удобным элементом графического интерфейса, но и является важной частью исполнительного механизма (**runtime**) пакета. Многие команды меню или панели инструментов выполняют свою работу не сами, а с помощью посылки программных сообщений в **Главную Консоль**. Поэтому окно **Главной Консоли** работает даже тогда, когда оно «свернуто» и не видно на экране.

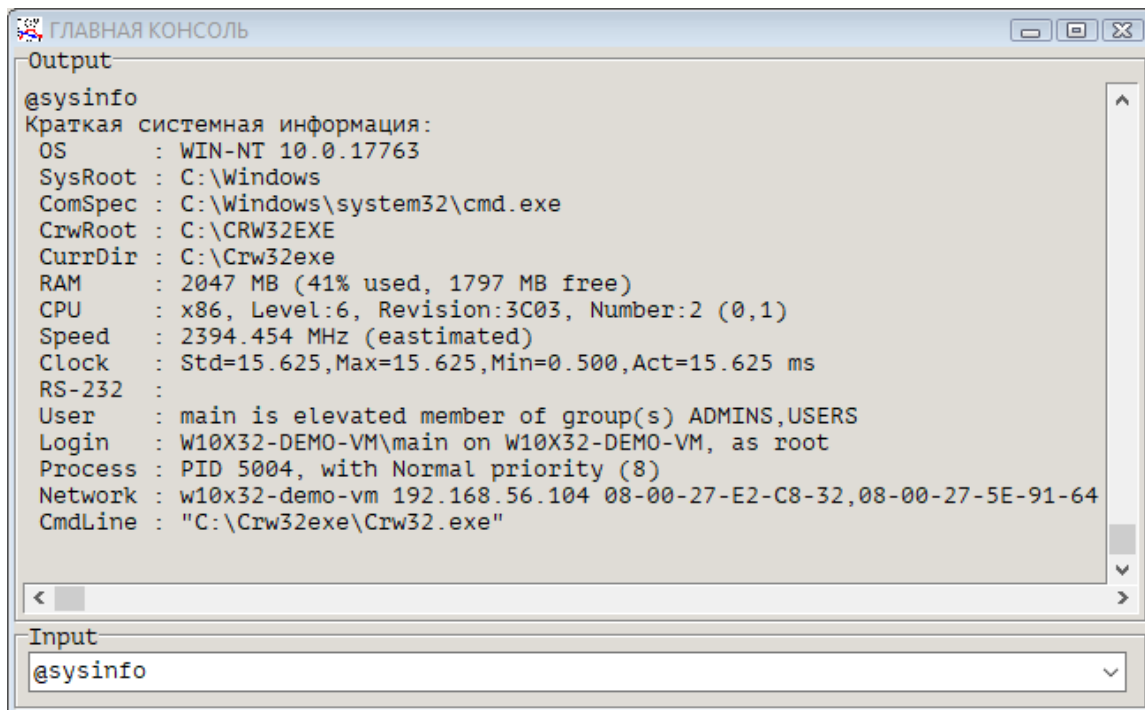


Рисунок 2. Внешний вид окна «Главной Консоли».

## Принцип работы Главной Консоли

Принцип работы **Главной Консоли** показан на рисунке 3. Главная Консоль владеет очередью **FIFO** для поступающих событий, в качестве которых выступают команды (строки текста), вводимые оператором, поступающие по каналам связи или посланные программно из других программных модулей.

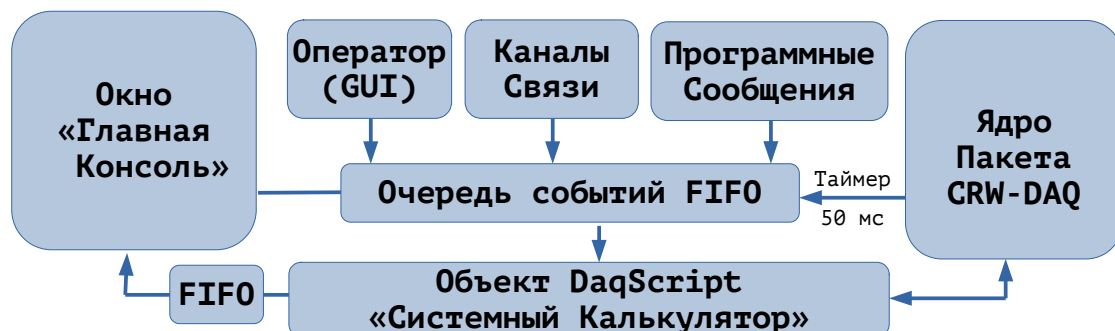


Рисунок 3. Схема работы «Главной Консоли».

**Главная Консоль** также присоединена к **Системному Калькулятору**, т. е. объекту **DaqScript**, выполняющему поступающие в **FIFO** команды. Ядро пакета **CRW-DAQ** каждые  $\approx 50$  мс генерирует сигналы таймера, по которым **Системный Калькулятор** опрашивает очередь событий, извлекает и обрабатывает поступившие команды. При работе **Системный Калькулятор** взаимодействует с ядром пакета **CRW-DAQ** и с окном **Главной Консоли**, выводя в неё результаты своих действий. Вывод в консоль также идет через **FIFO** буфер для снижения нагрузки на **CPU** за счет минимизации операций рисования в окне консоли.

Кроме основного набора **DaqScript**, **Системный Калькулятор** имеет дополнительный набор встроенных констант и команд. Собственно, этот набор и делает **Системный Калькулятор** полезным инструментом.

## Константы Системного Калькулятора

**Системный Калькулятор** поддерживает основной набор констант, указанный в распечатке 1, а также дополнительный набор констант, указанный в распечатке 4. Это константы **сеансового уровня**, содержащие параметры подключенных мониторов, «Рабочего Стола» и виртуального экрана, а также языков ввода и отображения. Они определяются в начале сеанса и могут обновляться в процессе работы командой `@view screen`. Параметры экрана используются для расчета геометрии окон при рисовании графического интерфейса. Языки ввода и отображения также могут использоваться для создания графического интерфейса. Например, в зависимости от языка отображения можно печатать сообщения на разных языках. В других объектах **DaqScript** эти данные доступны через команду `@system`.

### Распечатка 4. Дополнительный набор констант Системного Калькулятора.

<code>@list cons</code>	Команда печати списка констант.
Список констант:	Показаны только дополнительные константы.
<code>screen_colordepth</code> = 32	Разрядность цвета текущего монитора.
<code>screen_desktopheight</code> = 1013	Размер «Рабочего Стола» в пикселях.
<code>screen_desktopleft</code> = 0	Абсцисса левого края «Рабочего Стола».
<code>screen_desktoptop</code> = 0	Ордината верхнего края «Рабочего Стола».
<code>screen_desktopwidth</code> = 1871	Ширина «Рабочего Стола» в пикселях.
<code>screen_height</code> = 1013	Высота «Рабочего Стола» в пикселях.
<code>screen_monitor0flags</code> = 1	Флаги (свойства) основного монитора.
<code>screen_monitor0height</code> = 1013	Высота основного монитора в пикселях.
<code>screen_monitor0left</code> = 0	Абсцисса левого края основного монитора.
<code>screen_monitor0top</code> = 0	Ордината верхнего края основного монитора.
<code>screen_monitor0width</code> = 1871	Ширина основного монитора в пикселях.
<code>screen_monitor0workheight</code> = 1013	Высота рабочей области основного монитора.
<code>screen_monitor0workleft</code> = 0	Абсцисса левого края основного монитора.
<code>screen_monitor0worktop</code> = 0	Ордината верхнего края основного монитора.
<code>screen_monitor0workwidth</code> = 1809	Ширина рабочей области основного монитора.
<code>screen_monitorcount</code> = 1	Число доступных мониторов.
<code>screen_monitornumber</code> = 0	Номер текущего активного монитора.
<code>screen_monitorprimary</code> = 0	Номер основного монитора.
<code>screen_pixelsperinch</code> = 96	Разрешение монитора, пиксель на дюйм.
<code>screen_width</code> = 1871	Ширина виртуального экрана в пикселях.
<code>screen_workheight</code> = 1013	Высота рабочей области экрана в пикселях.
<code>screen_workleft</code> = 0	Абсцисса левого края рабочей области экрана.
<code>screen_worktop</code> = 0	Ордината верхнего края рабочей области экрана.
<code>screen_workwidth</code> = 180	Ширина рабочей области виртуального экрана.
<code>systemdefaultlcid</code> = 1049	Идентификатор языка ввода системы.
<code>systemdefaultuilanguage</code> = 1049	Идентификатор языка отображения системы.
<code>threadlocale</code> = 1049	Идентификатор языка ввода основного потока.
<code>threaduilanguage</code> = 1049	Идентификатор языка интерфейса для потока.
<code>userdefaultlcid</code> = 1049	Идентификатор языка ввода текущего пользователя.
<code>userdefaultuilanguage</code> = 1049	Идентификатор языка отображения пользователя.
<code>ansicodepage</code> = 1251	Номер кодовой страницы программ ANSI (Windows).
<code>oemcodepage</code> = 866	Номер кодовой страницы программ OEM (DOS).



## Команды Системного Калькулятора

**Системный Калькулятор** поддерживает основной набор **команд**, указанный в распечатке 3. Кроме того, он имеет дополнительный набор команд, указанный в распечатке 5.

### Распечатка 5. Дополнительный набор команд **Системного Калькулятора**.

@list acts	Команда печати списка доступных команд.
Список акций:	Здесь только дополнительные команды Системного Калькулятора.
@adam	Отладочные команды для модулей ADAM.
@compile	Компилировать указанный в аргументе файл.
@daq	Функции (compile,devsend,..) работы с DAQ-системой.
@env	Прочитать/установить переменную окружения.
@file	Работа с файлами (exist/size/kill).
@fonts	Работа с шрифтами: перечисление, регистрация, удаление.
@guard	Проверить\изменить уровень прав доступа.
@help	Вывод на консоль справки по теме ...
@if	Условное выполнение выражения.
@integrity	Утилита контроля целостности данных.
@list	Вывод на консоль списка переменных, констант, функций и т.д.
@log	Запись данных в журнальный файл.
@memory	Чтение\установка параметров памяти.
@menu	Выполнить команду меню.
@note	Печать метки времени и сообщения.
@onexception	Обработка исключений.
@open	Открыть файл ...
@pid	Работа с процессами (mini Task Manager).
@polling	Гистограммы периода опроса потоков.
@run	Запуск процесса из командной строки.
@silence	Задаёт режим работы команды @silent.
@silent	Выполнить команду в тихом режиме, без печати ввода\вывода.
@sleep	Приостановка программы на заданное время.
@speak	Выдать сообщение через речевой синтезатор.
@speech	Проверить\изменить параметры речевого синтезатора.
@sysinfo	Краткая информация о системе.
@term	Открыть терминальное окно.
@testbench	Модуль тестирования.
@textmetadata	Секция [MetaData] ...
@tooltip	Показать всплывающее сообщение.
@tty	Работа с виртуальной консолью (TTY): listen,close,status.
@view	Спрятать\показать визуальный элемент.

Следует заметить, что для получения краткой справки по большинству перечисленных команд следует вызвать их без аргумента. В справке будут перечислены аргументы вызова и приведены примеры.

### Команда @adam

Эта команда служит для наблюдения и отладки устройств **ADAM**, работающих по протоколу **DCON** [4]. Команда позволяет открыть консольное окно для вывода протокола обмена с устройствами и управлять этим выводом с помощью флагов. Например:

@adam console	Открывает отладочное окно «ADAM DEBUG CONSOLE».
@adam DebugMode	Выводит на печать текущие флаги отладки (debug mode).
@adam DebugMode n	Устанавливает битовую маску флагов отладки (debug mode).
	DebugMode Bit 0 – показать весь протокол ввода/вывода.
	DebugMode Bit 1 – показать только ошибки Timeout.
	DebugMode Bit 2 – показать только ошибки формата.
	DebugMode Bit 3 – показать время событий.
	DebugMode 0 - отключить отладочный вывод.
Примеры:	
@adam debugmode 15	Задать все флаги. Максимальная подробность вывода.
@adam console	Открыть консоль протокола обмена.
@adam debugmode 0	Отключить отладочный вывод.



## Команда @compile

Команда **@compile** служит для компиляции программ, написанных на языках **DaqPascal**, **Object Pascal** или **NSIS**. Она также дает способ перезапуска отдельных программ **DaqPascal** без перезапуска всей **DAQ** системы. При компиляции программы **DaqPascal** указывается имя **DAQ** устройства, которому принадлежит программа. При компиляции проекта **Object Pascal** или **NSIS** задается полное имя файла проекта. Например:

@compile	Команда компиляции.
Синтаксис:	
@compile	Показать справку по этой команде.
@compile device &name	Компиляция программы DaqPascal устройства &name.
@compile filename.dpr	Компиляция проекта Object Pascal.
@compile filename.nsi	Компиляция проекта NSIS.
Примечание: Для компиляции всегда используйте полное имя файла с путем и расширением.	
Примеры:	
@compile device &WebSrv	Компилировать (перезапустить) Web сервер &WebSrv.
@compile c:\demo\test.dpr	Компилировать указанный проект Object Pascal.
@compile c:\test\demo.nsi	Компилировать указанный проект NSIS.

## Команда @daq

Команда **@daq** работает с объектами **DAQ** системы. Позволяет послать **DAQ** устройству сообщение - строку текста, копируемую в консольный **FIFO** буфер устройства для последующей обработки. Также команда может компилировать программы **DaqPascal**, что дает способ перезапуска **DAQ** устройств без перезапуска всей системы.

@daq	Команда работы с устройствами DAQ системы.
Синтаксис:	
@daq	Показать справку по этой команде.
@daq compile &name	Компиляция программы DaqPascal устройства &name.
@daq devsend &name msg	Послать сообщение "msg" устройству &name.
@daq devmsg &name msg	Послать сообщение "msg" устройству &name.
@daq toplist wtype n	Свернуть окна типа wtype, кроме n последних.
Примеры:	
@daq compile &WebSrv	Компиляция (перезапуск) Web сервера.
@daq devsend &CronSrv @Warning Hello	Посылка сообщения серверу &CronSrv.

## Команда @env

Команда **@env** служит для работы с переменными окружения текущего процесса. Позволяет выполнять чтение, запись и очистку переменных окружения. Команда поддерживает подстановку значений переменных (типа **%TEMP%**). Например:

@env	Команда работы с переменными окружения.
Синтаксис:	
@env	Показать справку по этой команде.
@env name	Распечатать переменную окружения name.
@env name=	Очистить переменную окружения name.
@env name=value	Задать значение переменной окружения name.
@env /c name	Проверить (check) существование переменной name
@env /v PATH	Исправить (validate) список поиска PATH
@env /t PATH dir	Добавить dir в конец (tail) переменной PATH
@env /h PATH dir	Добавить dir в начало (head) переменной PATH
@env /d PATH dir	Удалить (delete) dir из списка поиска PATH
Примечание: Команда поддерживает подстановку переменных окружения (типа <b>%TEMP%</b> ).	
Примеры:	
@env dim_dns_node=%ComputerName%	Задать переменной значение.
@env dim_dns_node=main	Задать переменной другое значение.
@env dim_dns_node=	Очистить переменную.
@env dim_dns_node	Напечатать переменную.

## Команда @file

Команда **@file** служит для работы с файлами. Она позволяет проверять существование файла, узнавать его размер, удалять файл, менять текущий каталог программы.

@file	Команда работы с файлами.
Синтаксис:	
@file	Показать справку по этой команде.
@file check f	Проверка существования файла с именем f.
@file exist f	Проверка существования файла с именем f.
@file chdir d	Изменение текущего каталога на каталог d.
@file cd d	Изменение текущего каталога на каталог d.
@file size f	Возвращает размер файла с именем f или -1.
@file kill f	Убивает (удаляет) файл с именем f.
Примеры:	
@file exist Temp\Debug.Out	Проверка существования файла.
@file size Temp\Debug.Out	Проверка размера файла.
@file kill Temp\Debug.Out	Удаление файла.

При проверке существования файла команда возвращает статус 0/1, а при запросе размера – размер файла в байтах. Например:

```
size=@file size Temp\Debug.Out
flag=@file exist Temp\Debug.Out
if flag then @echo Файл существует и имеет размер %size
```

## Команда @fonts

Команда **@fonts** служит для работы с шрифтами. Она позволяет выводить список доступных в системе шрифтов, проверять наличие в системе заданного шрифта, регистрировать и отменять регистрацию шрифтов, а также управлять встраиванием шрифтов. Встроенные шрифты нужны для того, чтобы пакет мог использовать нужные библиотечные шрифты в системах, где они не установлены. Это уменьшает зависимость работы пакета от системного окружения. Например:

@fonts	Команда работы с шрифтами.
Синтаксис:	
@fonts	Показать справку по этой команде.
@fonts list uses	Список всех используемых шрифтов.
@fonts list screen	Список всех доступных экранных шрифтов.
@fonts list printer	Список всех доступных для принтера шрифтов.
@fonts list system c p	Список системных шрифтов с заданными charset, pitch.
@fonts enum c p n	Список шрифтов по заданным charset, pitch, name.
@fonts find a	Поиск шрифтов по имени (или псевдониму) a.
@fonts kill f	Удаление регистрации шрифта по файлу f.
@fonts add f	Добавление регистрации шрифта по файлу f.
@fonts embed info	Печать информации по встроенным шрифтам.
@fonts embed force	Принудительно встроит все доступные шрифты.
@fonts embed smart	Встроить только отсутствующие шрифты.
@fonts embed free	Освободить все встроенные шрифты.
@fonts embed save	Сохранить встроенные шрифты в системный каталог.
@fonts menu n	Установить свойство n шрифта для меню.
	n задает name/height/charset/pitch/style.
@fonts icon n	Установить свойство n шрифта для иконок.
@fonts hint n	Установить свойство n шрифта для подсказок.
Примеры:	
@fonts list screen	Список экранных шрифтов.
@fonts list printer	Список шрифтов принтера.
@fonts list system 204 1	Список русских моноширинных шрифтов.
@fonts add c:\Windows\xxx.ttf	Регистрация шрифта по файлу.
@fonts kill c:\Windows\xxx.ttf	Удаление регистрации шрифта.
@fonts enum 204 1	Список русских моноширинных шрифтов.
@fonts enum 204 * pt	Список русских шрифтов семейства PT.
@fonts enum * 1 co	Список моноширинных шрифтов по маске со*.
@fonts find ptmono	Найти шрифт PT Mono.

@fonts menu RUSSIAN_CHARSET	Задать в меню русский шрифт.
@fonts menu FIXED_PITCH	Задать в меню моноширинный шрифт.
@fonts menu PT Mono	Задать в меню шрифт PT Mono.
@fonts menu -13	Задать в меню размер шрифта.
@fonts menu bold	Задать в меню жирный стиль шрифта.

### Команда @guard

Команда **@guard** служит для определения текущего уровня доступа и управления уровнем доступа в пакете **CRW-DAQ**. Уровень доступа определяет наличие прав на использование тех или иных функций пакета, таких как запуск программ, редактирование программных кодов, загрузка и запуск измерительных систем и т.д. Например:

@help @guard	Получить справку по команде @guard.
@guard	Проверить уровень доступа 0/1/2/3=Lock/Guest/User/Root.
@guard Lock	Установить уровень доступа Lock (блокировка экрана).
@guard Guest	Установить уровень доступа Guest (гостевой уровень).
@guard User	Установить уровень доступа User (уровень пользователя).
@guard Root	Установить уровень доступа Root (полный доступ).

В результате вызова команда возвращает текущий уровень доступа в диапазоне **0...3**.

### Команда @help

Команда **@help** служит для получения справки по теме (команде), переданной в аргументе. Например:

@help	Получить справку по команде @help.
@help @help	Получить справку по команде @help.
@help list	Получить список доступных справок.
@help @guard	Получить справку по команде @guard.
@help @compile	Получить справку по команде @compile.

Многие команды также выдают более подробную справку при вызове этих команд без аргумента.

### Команда @if

Команда **@if a b** или **@if a then b** служит для выполнения **b** в зависимости от условия **a**. Сначала вычисляется условное выражение **a**. Если оно истинно, выполняется выражение **b**. Условие **a** считается ложным, если оно равно **0**, **NaN** или **INF**. Любое другое значение условия считается истинным, как и в логических функциях.

Условное выражение **a** не должно содержать пробелов. Оно также не должно быть командой. Желательно, чтобы это выражение было максимально простым, т. к. при ошибке в условном выражении интерпретатор не покажет причину и место возникновения ошибки. Лучше всего, если условие вычисляется заранее и сохраняется в переменной, которая затем проверяется в условии. Например:

exist=@file exist Temp\Demo.log	Вычисляем условие – существование файла.
@if exist then @echo Файл существует.	Проверяем условие и печатаем результат.

На условно выполняемое выражение **b** не накладывается особых ограничений, кроме того, что оно должно быть выражением командного режима интерпретатора, т. е. в нем не допускаются условный оператор **if**, переходы **goto** или вызовы подпрограмм **gosub**.

Команда **@if** на первый взгляд кажется избыточной, т. к. в **DaqScript** уже есть условный оператор **if...then...**, однако это не так. Условный оператор **if** работает только в пакетном режиме интерпретатора, а в командном режиме интерпретатора он недоступен. Для организации условного выполнения команд в командном режиме используется команда **@if**.

## Команда @integrity

Команда **@integrity** (от integrity - целостность) служит для обработки событий **системы контроля целостности данных (DICC - Data Integrity Control Center)**. Задача **DICC** – создание и проверка контрольных сумм файлов и обработка событий нарушения контроля целостности (**DIV - Data Integrity Violation**). Команда **@integrity** сложна и дает большие возможности для контроля целостности.

@integrity	Команда системы контроля целостности данных.
Синтаксис:	
@integrity	Показать справку по этой команде.
@integrity -a f	Синоним --add f.
@integrity -r f	Синоним --rule f.
@integrity -c f	Синоним --check f.
@integrity --add f	Добавить в файл с именем f контрольную сумму.
@integrity --check f	Проверить контрольную сумму файла с именем f.
@integrity --break	Прервать обработку событий, очистить очередь.
@integrity --clear	Очистить все правила обработки сообщений.
@integrity --load f s	Загрузить правила обработки из файла f, секции s.
@integrity --rule	Выдать список всех правил обработки сообщений.
@integrity --rule n	Удалить правило обработки сообщений с именем n.
@integrity --rule n t	Добавить правило с именем n и содержимым t.
@integrity n a	Обработать сообщение по правилу n с аргументами a.
Примеры:	
@integrity --rule save.crw	Очистить правило.
@integrity --rule save.crw @echo \$x1 file saved: \$1	Задать правило.
@integrity save.crw c:\demo.crw	Применить правило.

Контролю **DICC** подлежат файлы **DAQ** конфигураций \*.cfg, калибровок \*.cal, мнемосхем \*.crs, инициализации \*.ini, и ряд других. Это текстовые файлы, имеющие секционную структуру, в которых хранится важная информация по конфигурации **DAQ** системы.

Командой **@integrity --add f** в файл **f** с помощью внутренней утилиты **TextMetaData** добавляются контрольные суммы **MD5**, которые помещаются в параметре **[MetaData] @Checksum**. Добавление контрольных сумм к защищаемым файлам выполняется разработчиком прикладной **DAQ** системы после её отладки, когда содержимое файла принимает окончательный вид.

Командой **@integrity --check f** можно проверить сохранность файла **f**. По результатам проверки команда посылает в **Главную Консоль** сообщение со статусом **DIV** (смотри также распечатку 6):

@integrity SUCCESS:[MetaData]:@Checksum f	если проверка файла f прошла успешно
@integrity MISSING:[MetaData]:@Checksum f	если нет файла f или контрольной суммы
@integrity INVALID:[MetaData]:@Checksum f	если контрольная сумма файла f не совпала
@integrity FAILURE:[MetaData]:@Checksum f	если произошел сбой при чтении файла f

Для обработки сообщений **DICC** и событий **DIV** создаются **правила**, т. е. небольшие поименованные сценарии обработки данных. Правила создаются или загружаются в начале работы, при загрузке программы или конфигурации **DAQ** системы. В дальнейшем все поступающие команды **@integrity** обрабатываются с помощью этого набора правил. Командой **@integrity --clear** можно очистить список правил, а командой **@integrity --load f s** - загрузить набор правил из файла **f**, секции **s**. Обычно правила загружаются при старте пакета командой **@integrity --load Crw32.ini [@integrity.DefaultRules]** из файла конфигурации.

Сами же правила создаются командой **@integrity --rule n s**, где **n** - имя правила, **t** - содержимое (строка текста) правила. Правило может включать много строк текста, в этом случае последовательно выполняется каждая из них.

Для очистки (удаления) правила с именем **n** следует вызвать команду **@integrity --rule n** с пустой строкой текста содержимого.

При обработке сообщения вида **@integrity n a** команда воспринимает первое слово аргументов (**n**) как имя правила. По этому имени оно находит текст (**t**) этого правила в динамической таблице правил и применяет его к оставшимся (после первого слова) аргументам (**a**). В результате применения правила **n** с текстом **t** к аргументам **a** получается новая строка текста **t(a)**.

При обработке правил **t(a)** действуют следующие строковые подстановки аргументов в правила:

<b>\$*</b>	Все слова аргументов.
<b>\$1 ... \$9</b>	Слово 1 ... 9 извлеченное из аргументов.
<b>\$p1 ... \$p9</b>	Файловый путь из 1 ... 9 слова аргументов.
<b>\$n1 ... \$n9</b>	Файловое имя из 1 ... 9 слова аргументов.
<b>\$x1 ... \$x9</b>	Файловое расширение из 1 ... 9 слова аргументов.

После подстановки аргументов в правило получается строка сообщения **t(a)**, которая посылается в [Главную Консоль](#) для обработки. Затем итерации обработки строк продолжаются и далее до исчерпания очереди событий.

Приведем простой пример:

```
@Integrity --rule load.dat
@Integrity --rule load.dat @Integrity --check $*
@Integrity --rule Success:[MetaData]:@Checksum
@Integrity --rule Success:[MetaData]:@Checksum @Note SUCCESS: $1
```

В 1-й и 2-й строке очищается и добавляется правило с именем **load.dat** с текстом **@integrity --check \$\***. В строках 3 и 4 очищается и добавляется правило **Success:[MetaData]:@Checksum** с текстом **@Note SUCCESS: \$1**.

Посмотрим, что произойдет, например, при обработке сообщения **@integrity load.dat c:\demo.dat**. Сначала сообщение по первому правилу **load.dat** будет путем подстановки аргумента **\$\*** заменено сообщением **@integrity --check c:\demo.dat** и послано в [Главную Консоль](#). На второй итерации обработка сообщения приведет к проверке файла (допустим, успешной), по результатам которой будет послано новое сообщение **@Integrity Success:[MetaData]:@Checksum c:\demo.dat**. На третьей итерации это сообщение по правилу **Success:[MetaData]:@Checksum** будет путем подстановки **\$1** заменено сообщением **@Note SUCCESS: c:\demo.dat** и послано в [Главную Консоль](#). Наконец, обработка этого сообщения приведет к выводу в консоль сообщения, содержащее время, статус и имя файла. Таким образом, цепочка итераций обработки сообщений по заданным правилам позволяет добиться желаемого результата.

Контроль целостности файлов в пакете **CRW-DAQ** в основном основан на команде **@integrity** и загружаемых правилах обработки сообщений. При чтении или записи файлов в различных модулях пакета **CRW-DAQ** непосредственной проверки файлов не происходит. Вместо этого в [Главную Консоль](#) посылаются сообщения **@integrity**, содержащие тип операции и имя обрабатываемого файла, как показано в распечатке 6. Это позволяет, во-первых, централизовать всю обработку событий контроля целостности, а во-вторых, сделать её наиболее гибкой и настраиваемой (конфигурируемой). Вся работа по контролю целостности при этом происходит в соответствии с загруженными правилами обработки сообщений. Загрузка правил происходит, например, при запуске программы, а также при запуске конфигурации **DAQ**-системы.



**Распечатка 6. Список сообщений @integrity в пакете CRW-DAQ.**

@integrity SUCCESS:[MetaData]:@Checksum f	При успешной проверке файла f.
@integrity MISSING:[MetaData]:@Checksum f	При отсутствии файла или контрольной суммы
@integrity INVALID:[MetaData]:@Checksum f	При несовпадении контрольной суммы файла.
@integrity FAILURE:[MetaData]:@Checksum f	При ошибке чтения файла f.
@integrity DAQ:CrcReadBmp:Created f	При создании *.bmp файла f для мнемосхемы.
@integrity DAQ:CrcReadBmp:Failure f	При ошибке чтения *.bmp файла f.
@integrity save.crc f	При сохранении *.crc файла мнемосхемы f.
@integrity load.crc f	При загрузке *.crc файла мнемосхемы f.
@integrity save.cal f	При сохранении *.cal файла калибровки f.
@integrity load.cal f	При загрузке *.cal файла калибровки f.
@integrity save.crw f	При сохранении *.crw файла данных f.
@integrity load.crw f	При загрузке *.crw файла данных f.
@integrity save.dat f	При сохранении *.dat файла данных f.
@integrity load.dat f	При загрузке *.dat файла данных f.
@integrity save.txt f	При сохранении текстового файла f.
@integrity load.txt f	При загрузке текстового файла f.
@integrity load.pas f	При загрузке *.pas файла программы f.
@integrity enter.daq f	При загрузке DAQ конфигурации файла f.
@integrity leave.daq a	При завершении DAQ конфигурации файла f.
@integrity @DIM_EVENT: m	При нормальном сообщении m от DIM сервера.
@integrity @DIM_ALARM: m	При аварийном сообщении m от DIM сервера.
@integrity @OnException m	При возбуждении исключения с сообщением m.
@integrity @OnExceptionInfo I	При исключении с доп.информацией i.
@integrity CRW:@OPEN:Incomplete f	При неудачной попытке открыть файл f.
@integrity CRW:@OPEN:MissedFile f	При попытке открыть отсутствующий файл f.
@integrity WIN32:WM_QUERYENDSESSION l w	При запросе на завершение сессии Windows.
@integrity WIN32:WM_ENDSESSION l w	При завершении сессии Windows.
@integrity WIN32:WM_DISPLAYCHANGED w h c	При изменении параметров экрана, где: ширина w, высота h, глубина цвета c.

Таким образом, команда **@integrity** дает мощный и гибкий механизм для контроля целостности данных, включая создание и проверку контрольных сумм файлов, и для реакции на другие системные события.

Типичной реакцией на события системы контроля целостности данных является проверка файла и генерация сообщения (консольного или графического) с диагностикой по результатам проверки. Но при необходимости эта реакция может быть изменена на другую по желанию прикладного программиста.

**Команда @list**

Команда **@list** выводит список объектов программы или системы по теме, указанной в аргументе:

<b>@list</b>	Команда вывода списка объектов по указанной теме.
<b>Синтаксис:</b>	
<b>@list</b>	Вывод справки по этой команде.
<b>@list vars</b>	Список всех доступных переменных «Главной Консоли».
<b>@list cons</b>	Список всех доступных констант «Главной Консоли».
<b>@list funs</b>	Список всех доступных функций «Главной Консоли».
<b>@list acts</b>	Список всех доступных команд «Главной Консоли».
<b>@list pids</b>	Список всех работающих процессов в этой системе.
<b>@list threads</b>	Список всех работающих потоков текущего процесса.
<b>@list menus</b>	Список всех доступных команд «Главного Меню».
<b>@list keys</b>	Список всех горячих клавиш «Главного Меню».
<b>@list views</b>	Список всех визуальных компонентов (окон).
<b>@list users</b>	Список всех пользователей этого компьютера.
<b>@list hosts</b>	Список всех доступных по сети компьютеров.
<b>@list domains</b>	Список всех доступных по сети доменов.

С помощью **@list vars**, **@list cons**, **@list funs**, **@list acts** можно узнать текущее состояние **Системного Калькулятора**, связанного с **Главной Консолью**. Другие аргументы позволяют увидеть списки процессов и пользователей в системе, потоков и окон в процессе, компьютеров и доменов в сети.



## Команда @log

Команда **@log** выводит сообщение (строку текста) в указанный журнальный файл. Кроме того, команда позволяет удалять журнальные файлы и открывать их в различных программах:

@log	Команда записи сообщения в журнальный файл.
Синтаксис:	
@log	Вывод справки по этой команде.
@log --print f s	Записать строку s в журнальный файл с именем f.
@log --write f s	Записать строку s в журнальный файл с именем f.
@log --event f s	Записать время и строку s в журнальный файл с именем s.
@log --delete f	Удалить журнальный файл с именем f.
@log --lister f	Открыть журнальный файл f командой lister.
@log --tail f	Открыть журнальный файл f командой wintail.
@log --view f	Открыть журнальный файл f командой logview.

Имя журнального файла не должно содержать пробелов и обычно имеет расширение **\*.log**. Путь к файлу может быть неполным, если он задан относительно «домашнего» каталога пакета, где расположен исполняемый файл **Crw32.exe**. Журнальные файлы обычно располагаются в каталоге **Temp**. Например, команда:

```
@log --event Temp\test.log Тестовое сообщение от процесса %processid
```

запишет в файл Temp\test.log строку примерно такого вида:

```
2020.04.11-21:22:16 => Тестовое сообщение от процесса 5004
```

Если журнальный файл отсутствует, он создается автоматически. Если файл существует, запись производится в конец файла. Записи в журнал идут через внутренний **FIFO** буфер для снижения нагрузки **CPU**.

Журнальные файлы дают весьма удобный инструмент для ведения протоколов событий, происходящих в системе.

## Команда @memory

Команда **@memory** служит для работы с оперативной памятью (**RAM**) и буферами памяти. Команда позволяет узнавать различные параметры памяти и задавать некоторые из них.

@memory	Команда для работы с памятью.
Синтаксис:	
@memory Load	Узнать использование физической памяти, %.
@memory TotalPhys	Узнать полный объем физической памяти, байт.
@memory AvailPhys	Узнать объем доступной физической памяти, байт.
@memory TotalPageFile	Узнать общий размер файла подкачки, байт.
@memory AvailPageFile	Узнать размер доступной памяти в файле подкачки.
@memory TotalVirtual	Узнать общий объем виртуальной памяти, байт.
@memory AvailVirtual	Узнать объем доступной виртуальной памяти, байт.
@memory AllocMemSize	Объем выделенной памяти в процессе Crw32.exe, байт.
@memory AllocMemCount	Счетчик блоков выделенной памяти в процессе Crw32.
@memory min	Узнать минимальный объем рабочей памяти процесса.
@memory min n	Задать минимальный объем рабочей памяти процесса.
@memory max	Задать максимальный объем рабочей памяти процесса.
@memory max n	Задать максимальный объем рабочей памяти процесса.
@memory CfgCacheSize	Узнать размер буфера конфигурационных файлов.
@memory CfgCacheSize n	Задать размер буфера конфигурационных файлов.
@memory CfgCacheLim	Узнать предельный размер буфера конфиг. файлов.
@memory CfgCacheLim n	Задать предельный размер буфера конфиг. файлов.
@memory CfgCachePoll	Узнать период проверки размера буфера к.ф., мс.
@memory CfgCachePoll n	Задать период проверки размера буфера к.ф., мс.
@memory CfgCacheHold	Узнать «время удержания» буфера конфиг. файлов, мс.
@memory CfgCacheHold n	Задать «время удержания» буфера конфиг. файлов, мс.
@memory CfgCacheHash	Узнать метод хеширования конфиг. файлов.
@memory CfgCacheHash n	Задать метод хеширования конфиг. файлов, n=0..93.
@memory CfgCacheShot m	Увидеть буфер конфиг. файлов с уровнем m=0..2
@memory CfgCacheLook f	Увидеть буфер конфиг. файла с именем f.
Примеры:	

<code>n=@memory TotalPhys</code>	Узнать общий объем физической памяти RAM.
<code>@memory min 1024*1024*32</code>	Задать мин. объем рабочей памяти 32 МБ.
<code>@memory max 1024*1024*128</code>	Задать макс. объем рабочей памяти 128 МБ.
<code>@memory CfgCacheSize 1024*1024*2</code>	Задать размер буфера конфиг. файлов 2 МБ.

Несколько пояснений к параметрам команды. В системе есть физическая память (**RAM**), файл подкачки (на диске) и виртуальная память (суммарная). Часть этой памяти доступна для выделения, другая часть уже занята. У каждого процесса, включая **Crw32.exe**, есть «рабочая память», которую система будет пытаться удержать в физической памяти. Процесс выделяет себе для работы необходимый объем памяти в виде блоков, который можно контролировать. Для ускорения чтения и анализа конфигурационных файлов в процессе **Crw32.exe** выделяется буфер, который может расти до заданного предела. Программа будет пытаться удержать буфер в памяти некоторое время («время удержания»), чтобы повторные операции чтения делались из буфера, а не с более медленного диска. При этом объем буфера периодически проверяется и при необходимости память буфера освобождается.

Такая довольно сложная схема управления памятью ставит целью установить баланс между противоречивыми требованиями минимального использования памяти и максимального быстродействия. Увеличение «рабочей памяти» процесса и объема буферов памяти ускоряет работу программы, но может привести к нехватке памяти или к деградации производительности, если физической памяти недостаточно и система вынуждена задействовать файл подкачки. Команда **@memory** позволяет балансировать ресурсами системы, выделяя или освобождая буферы памяти по мере необходимости.

## Команда @menu

Команда **@menu** служит для просмотра списка и выполнения команд «Главного Меню».

<code>@menu</code>	Команда работы с «Главным Меню».
Синтаксис:	
<code>@menu list</code>	Показать список всех доступных команд «Главного Меню».
<code>@menu list keys</code>	Показать список всех горячих клавиш «Главного Меню».
<code>@menu run cmd</code>	Выполнить команду cmd из «Главного Меню».
Примеры:	
<code>@menu list</code>	Показать список команд.
<code>@menu run FormDaqControlDialog.ActionDaqInit</code>	Начать сеанс DAQ-системы.
<code>@menu run FormCw32.ActionFileExit</code>	Выполнить команду «Файл/Выход».

Несмотря на кажущуюся простоту, команда **@menu** дает большие возможности программисту, т. к. позволяет программно управлять работой программы так, как делает оператор клавиатурой и мышью.

## Команда @note

Команда **@note** выводит в «Главную Консоль» метку даты-времени и сообщение, переданное в аргументах. Можно сказать, что это вариант команды **@echo**, только с меткой даты-времени.

<code>@note</code>	Команда «заметка» вывода сообщения с меткой даты и времени.
Синтаксис:	
<code>@note</code>	Показать справку по этой команде.
<code>@note arg</code>	Печатает дату-время и сообщение arg.
Пример:	
<code>@note Start session.</code>	

Например:

```
@note Тестовое сообщение от процесса %processid
2020.04.12-03:42:19 => Тестовое сообщение от процесса 5004
```

Команда **@note** полезна для ведения журнала событий программы. Надо напомнить, что весь вывод **Главной Консоли** в текущем сеансе программы протоколируется в файле **Temp\Console.Out**. Поэтому все сообщения **@note** с меткой времени доступны в этом файле для анализа.

## Команда @onexception

Команда **@onexception** является частью механизма для обработки исключений. Исключения — это программные ошибки (например, деление на ноль или ошибка чтения файла), которые нарушают нормальную работу алгоритмов и требуют специальной обработки. При возбуждении исключения нормальное выполнение текущей процедуры прерывается и управление передается обработчику исключений.

<b>@onexception</b>	Команда обработки исключений.
Синтаксис:	
<b>@OnException</b>	Показать справку по этой команде.
<b>@OnException Msg</b>	Уведомление об исключении с сообщением Msg.
<b>@OnException --info Inf</b>	Уведомление об исключении с информацией Inf.
<b>@OnException --raise E Msg</b>	Возбудить исключение типа E с сообщением Msg.
<b>@OnException --raise Msg</b>	Возбудить тестовое исключение с сообщением Msg.
Примечание:	
В текущей версии посылает сообщение <b>@Silent @Integrity @OnException Msg</b>	
So « <b>@Integrity --rule @OnException ...</b> » rule should be defined.	
Also sends message <b>@Silent @Integrity @OnExceptionInfo Inf</b>	
So « <b>@Integrity --rule @OnExceptionInfo ...</b> » rule should be defined.	
Примеры:	
<b>@OnException --raise EDivByZero</b>	
<b>@OnException Exception «EDivByZero» in Crw32.exe PID 4736</b>	
<b>@OnException --info EDivByZero in Crw32.exe at 0022A2DA «Division by zero»</b>	

Обработчик исключения обычно освобождает ресурсы, требующие освобождения для предотвращения ошибки «утечки ресурсов», а затем посылает в **Главную Консоль** команду **@OnException Msg** с уведомлением **Msg** о причинах исключения. При наличии дополнительной информации (например, о месте возникновения исключения) также посылается команда **@OnException --info Inf** с этой дополнительной информацией. Эти команды посылаются с целью уведомления системы о возникшем исключении.

В текущей версии обработчик команды **@OnException Msg** посылает в **Главную Консоль** команду **@Silent @Integrity @OnException Msg**, а обработчик команды **@OnException --info Inf** посылает в консоль команду **@Silent @Integrity @OnExceptionInfo Inf** для дальнейшей обработки. Реакция на эти сообщения определяется правилами обработки событий **@OnException** и **@OnExceptionInfo**, загружаемыми вместе с другими правилами команды **@integrity**.

Стандартный набор правил для **@OnException** и **@OnExceptionInfo**, загружаемый при старте программы, вызывает вывод сообщения в консоль, а также вызов «всплывающего» окна сообщений командой **@tooltip** для уведомления оператора о возникшей ошибке. Кроме того, факт возникновения исключения фиксируется в системном журнальном файле **Temp\@OnException.log** для дальнейшего анализа.

Этот довольно сложный механизм обработки исключений сделан для того, чтобы сделать обработку исключений наиболее гибкой и легко конфигурируемой процедурой.

Для тестирования работы механизма обработки исключений служит команда **@OnException --raise Msg**, которая позволяет искусственно возбудить исключение с заданным сообщением.

## Команда @open

Команда **@open** открывает файлы для запуска или редактирования. Конкретные действия с открываемым файлом определяются его типом, который зависит от расширения имени файла. В аргументах команды **@open** допускаются только **полные** имена файлов, с полным путем и расширением. Опции **/r**, **/e**, **/d** перед именем файла уточняют, как именно следует открывать файл – для «запуска», «редактирования» или через диалог «Файл\Открыть» с возможностью задать имя файла.

@open	Команда открытия файла для запуска или редактирования.
Синтаксис:	
@open	Показать справку по этой команде.
@open /l ...	Блокировать повторный вызов @open и Drag&Drop.
@open /u ...	Разрешить повторный вызов @open и Drag&Drop.
@open /r f	Открыть файл f для запуска (run) без диалога «Файл\Открыть».
@open /e f	Открыть файл f для редактирования (edit) без диалога.
@open /d f	Открыть файл f с помощью окна диалога «Файл\Открыть».
Примеры:	
@open c:\test.cfg	Открыть конфиг. файл test.cfg через «Файл\Открыть».
@open /r c:\test.cfg	Запуск DAQ конфигурации test.cfg на исполнение.
@open /e c:\test.cfg	Открыть файл test.cfg для редактирования как текст.
@open /d c:\test.cfg	Открыть конфиг. файл test.cfg через «Файл\Открыть».
@open /r c:\test.cal	Открыть калибровку test.cal в Редакторе Калибровок.
@open /e c:\test.cal	Открыть калибровку test.cal в текстовом редакторе.
@open /r c:\test.bat	Запуск командный файл test.bat на исполнение.
@open /e c:\test.bat	Открыть файл test.bat в текстовом редакторе.

Команда поддерживает большое количество (более 40) расширений файлов, которые здесь нет смысла приводить. Вызов **@open** без аргументов выдает подробную информацию по каждому типу. Тип файла определяет конкретное содержание опций **/r**, **/e**, **/d**.

Кроме того, команда **@open** со специальными именами псевдо-файлов, начинающихся с префикса «**DAQ:**», позволяет запускать или редактировать объекты **DAQ** системы:

Специальные функции DAQ системы:	
@open DAQ:\DeviceList\&Dev\Calibration#n	- Открывает диалог калибровки номер n для DAQ устройства &Dev.
@open /e DAQ:\DeviceList\&Dev\Calibration#n	- Открывает файл калибровки номер n для DAQ устройства &Dev как тестовый.
@open DAQ:\DeviceList\&Dev\PropertyDialog	- Открывает диалог редактирования свойств устройства DAQ с именем &Dev.
@open DAQ:\DeviceList\&Dev\CommonPropertyDialog	- Открывает диалог редактирования общих свойств устройства DAQ с именем &Dev.

Это позволяет вызывать из прикладной программы (например, по кнопке) редактирование калибровок и свойств устройств **DAQ** системы.

Команда **@open** имеет важное значение для работы пакета. Например, обработка аргументов командной строки при старте программы выполняется с помощью посылки в [Главную Консоль](#) команды **@open** с именем файла. Наличие этой команды также позволяет прикладным программам открывать файлы аналогично тому, как это делает оператор через **Главное Меню**, что открывает большие возможности для прикладного программиста.

## Команда @pid

Команда **@pid** служит для работы с процессами операционной системы. Она позволяет распечатывать список работающих процессов, находить процессы по имени исполняемого файла или по идентифицирующему номеру (**PID**), а также завершать процессы по имени или по номеру, вместе с дочерними процессами до заданного уровня наследования.

@pid	Команда работы с процессами операционной системы.
Синтаксис:	
@pid	Показать справку по этой команде.
@pid list	Вывести в консоль список всех работающих процессов.
@pid find p	Найти процесс с именем/номером p, вернуть его номер PID.
@pid enum p	Найти все процессы с именем/номером p, вернуть счетчик.
@pid kill p e c	Убить (завершить) процесс p с кодом выхода e.
	p – номер или имя исполняемого файла процесса.
	e – код завершения процесса, по умолчанию 0.
	c – убить дочерние процессы до уровня c, по умолчанию 0.
Примеры:	
@pid list	Вывести список процессов.
@pid find cmd.exe	Найти первый попавшийся процесс cmd.exe.
@pid enum cmd.exe	Найти все работающие процессы cmd.exe.
@pid kill 123 1 0	Убить (завершить) процесс номер 123 с кодом ошибки 1.
@pid kill dns.exe	Убить (завершить) все процесс(ы) dns.exe.

Команда дает мощный инструмент для работы с процессами **ОС**, однако использовать его надо осторожно, чтобы не нарушить работу системы.

### Команда @polling

Команда **@polling** служит для работы со служебными потоками текущего процесса. В пакете **CRW-DAQ** работа большинства служебных потоков программы контролируется «сторожевым таймером» (watchdog), а также идет сбор статистики (счетчиков опроса) и набор гистограмм периода опроса потоков. Это позволяет обнаруживать «подвисание» потоков и анализировать работу потоков с точки зрения «реального времени». Команда **@polling** позволяет выводить список служебных потоков, наблюдать графики и таблицы гистограмм частоты опроса потоков.

@polling	Команда работы со служебными потоками программы.
Syntax:	
@polling	Показать справку по этой команде.
@polling list	Показать список служебных потоков программы.
@polling clear xxx	Очистить гистограмму опроса служебного потока xxx.
@polling clear all	Очистить все гистограммы опроса служебных потоков.
@polling clear log	Очистить журнал опроса служебных потоков.
@polling plot liny	Показать графики гистограмм опроса в линейной шкале.
@polling plot logy	Графики гистограмм опроса в логарифмической шкале.
@polling text liny	Показать таблицы гистограмм опроса в линейной шкале.
@polling text logy	Графики гистограмм опроса в логарифмической шкале.
@polling PriorityClass	Узнать текущий класс приоритета процесса.
@polling PriorityClass c p	Задать класс приоритета процесса.
	c – класс приоритета = (4,6,8,10,13,24)
	p – период проверки класса приоритета, мс
@polling monitor m	Открыть окно КОНСОЛЬ МОНИТОРА РЕСУРСОВ с флагами m:
	1:объекты,2:память,4:ссылки,8:потоки,*:сброс,-1:все.
Примеры:	
@polling list	Показать список служебных потоков.
@polling clear all	Очистить все гистограммы опроса потоков.
@polling plot logy	Открыть графики гистограмм частоты опроса потоков.
@polling monitor -1	Открыть КОНСОЛЬ МОНИТОРА РЕСУРСОВ и показывать всё.

Гистограмма периода опроса потока вычисляется так. В начале каждого опроса потока фиксируется время по часам **msecnow**. Период опроса - это разность времени текущего и предыдущего опроса, **мс**. Делением на 10 по периоду опроса вычисляется индекс ячейки гистограммы, и в неё добавляется 1 отсчет. Гистограмма имеет **5000** ячеек, что дает интервал **5** секунд с разрешением **10 мс**. Счетчик каждой ячейки, нормированный на общее число опросов, дает относительную вероятность соответствующего периода опроса.



Гистограммы удобно изображать в логарифмическом масштабе, так они выглядят нагляднее.

На рисунке 4 показан типичный вид графика гистограммы периода опроса потока программы **DaqPascal** и **Консоли Монитора Ресурсов**. Видно, что частота проса программы **&FAST\_POLL** составила **992 Гц**.

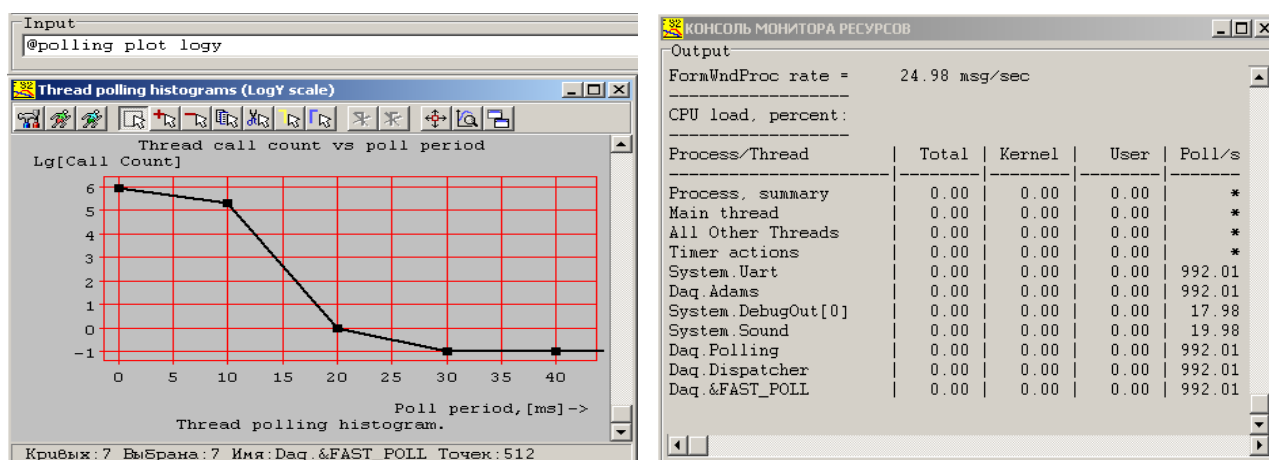


Рисунок 4. График гистограммы периода опроса потока (слева) и окно «Консоль Монитора Ресурсов» (справа).

Команда **@polling** позволяет также задавать класс приоритета процесса (**Idle, Lower, Normal, Higher, High, RealTime**)=(**4, 6, 8, 10, 13, 24**) и период его коррекции для предотвращения вмешательства извне. Вместе с относительными приоритетами потоков это позволяет максимально эффективно использовать возможности приоритетной вытесняющей многозадачности операционной системы **Windows**.

### Команда @run

Команда **@run** служит для запуска процессов **ОС** – программ, командных файлов и сценариев. Также позволяет открывать документы, если тип этих документов зарегистрирован в системе.

Перед аргументами можно указывать необязательные опции запуска, начинающиеся с символа «/» (слева) и определяющие вид окна, приоритет процесса и его рабочий каталог.

В аргументах команда **@run** принимает командную строку, которую надо выполнить. Исполняемый файл в командной строке может быть коротким именем файла, если он находится в пути поиска **PATH**. Допустимо также указывать неполные пути файлов относительно домашнего каталога программы.

@run	Команда запуска процессов – программ, скриптов и т.д.
Синтаксис:	
@run	Показать справку по этой программе.
@run [/opt] cmd	Запустить программу cmd с опциями /opt.
cmd	- командная строка запускаемой программы.
/opt	- опция запуска, одна из следующих:
/Hide	- запуск в скрытом окне.
/SW0../SW10	- запуск с режимом видимости ShowWindow=0..10. 0=HIDE, 1=NORMAL, 3=MIN, 4=SHOWNA, 5=SHOW, 6=MAX, 7=MINNA, 8=SHOWNA, 9=RESTORE, 10=DEFAULT
/Idle	- запуск с приоритетом Idle = 4
/Lower	- запуск с приоритетом Low = 6
/Normal	- запуск с приоритетом Normal = 8
/High	- запуск с приоритетом High = 13
/RealTime	- запуск с приоритетом RealTime = 24
/Cd d	- запуск с рабочим каталогом d.



/HomeDir d	- запуск с рабочим каталогом d.
/Path p	- запуск с списком путей поиска p.
/Term:opt	- запуск в терминала, см. команду @term
Примеры:	
@run did	Запуск программы did.exe, если она в PATH.
@run /hide dns.exe	Запуск программы dns.exe в скрытом окне.
@run /SW7 cmd.exe	Запуск cmd.exe свернутом неактивном окне.
@run /idle notepad	Запуск редактора Notepad с приоритетом Idle.
@run /cd c:\temp notepad.exe	Запуск Notepad в рабочем каталоге c:\temp.
@run /cd %temp% notepad.exe	Запуск Notepad в рабочем каталоге %temp%.
@run crw-daq.htm	Открыть справку по пакету CRW-DAQ.

В случае успеха команда **@run** возвращает номер **PID** запущенного процесса или **0**, если запуск не удался. Номер процесса можно сохранить в переменной, чтобы впоследствии иметь возможность отслеживать этот процесс или завершать его командой **@pid**.

Переменные окружения, задаваемые командой **@env**, наследуются дочерними процессами, поэтому они могут использоваться для неявной передачи параметров процессам, запускаемым командой **@run**.

Команда **@run** дает простой и удобный метод запуска дочерних процессов через посылку сообщения в **Главную Консоль**, который можно использовать для работы «гибридных» систем управления, в составе которых работает несколько независимых пакетов программ. В этом случае пакет **CRW-DAQ** может выступать супервизором, управляющим запуском и завершением всех совместно работающих процессов.

### Команда @silence

Команда **@silence** задает режим работы команды **@silent**, которая выполняет другие команды в «тихом» режиме. Режим включает в себя два логических флага (**0/1**) – входной **i** и выходной **o**. Входной флаг **i** определяет, должна ли команда **@silent** «подавлять» печать в **Главной Консоли** копии (эхо) входной команды. Выходной флаг **o** определяет, должна ли команда **@silent** «подавлять» печать в **Главной Консоли** результата выполнения этой команды.

@silence	Команда задания режима работы команды @silent.
Синтаксис:	
@silence	Показать справку по этой команде.
@silence i o	Задать режим команды @silent для входа i и выхода o.
	i = 0/1 - печатать/подавлять эхо (копию) входной команды.
	o = 0/1 - печатать/подавлять результат команды на выходе.
Примечание:	
@silence 1 1	Режим по умолчанию для команды @silent.
Примеры:	
@silence * *	Печать текущего состояния i/o режимов команды @silent.
@silence 0 0	Печатать копию входной команды и печатать результат.
@silence 1 0	Подавлять копию входной команды и печатать результат.
@silence 0 1	Печатать копию входной команды и подавлять результат.
@silence 1 1	Подавлять копию входной команды и подавлять результат.

### Команда @silent

Команда **@silent** выполняет другие команды в «тихом» режиме. Выполняемая «тихо» команда передается в аргументе вызова. При её выполнении происходит «подавление» вывода в **Главную Консоль** копии входной команды и результата её выполнения, так что в окне консоли ничего не печатается, если только это не предусмотрено логикой самой выполняемой команды.

Например, команда **@silent @run cmd** выполняет переданную ей в аргументе команду **@run cmd** и запускает процесс **cmd.exe**, но при этом ничего не выводит в консоль. Хотя в обычном режиме команда

**@run cmd** выводит в консоль сообщение о запуске процесса либо о возникшей ошибке запуска.

<b>@silent</b>	Команда выполнения другой команды в «тихом» режиме.
Синтаксис:	
<b>@silent</b>	Показать справку по этой команде.
<b>@silent cmd</b>	Выполнить команду cmd в «тихом» режиме, без печати в консоль.
Примечание: Поведение команды @silent зависит от режима, заданного командой @silence.	
Примеры:	
<b>@silent setclockres(1)</b>	«Тихо» задать таймер.
<b>@silent @daq devsend &amp;CronSrv @Warning Hello</b>	«Тихо» послать сообщение.

Команда **@silent** предохраняет [Главную Консоль](#) от печати лишних сообщений, выполняемых программой в фоновом режиме. Например, во время загрузки программы или запуска **DAQ** конфигурации стартовые сценарии могут запускать десятки и даже сотни консольных команд, наблюдать которые оператору нет необходимости. Такие действия выполняются в «тихом» режиме, не загромождая консоль большим объемом лишних сообщений. В то же время интерактивные команды оператора, выполняемые обычно без использования префикса **@silent**, будут выводиться на печать как обычно.

Режим работы команды **@silent** зависит от команды [@silence](#). По умолчанию программа настроена на использование «тихого» режима. Команда **@silence** позволяет временно отключать «тихий» режим, что бывает полезно во время отладки. После отладки «тихий» режим включается снова.

### Команда @sleep

Команда **@sleep** приостанавливает выполнение потока на заданное в аргументе время. Время «сна» задается в **мс** в диапазоне 0...10000.

<b>@sleep</b>	Команда «сна», т.е. приостановки потока на заданное время.
Синтаксис:	
<b>@sleep</b>	Показать справку по этой команде.
<b>@sleep n</b>	«Уснуть», т. е. приостановить поток программы на n миллисекунд.
	Допускается время «сна» n в диапазоне 0...10000 мс.

Команда **@sleep** заведена скорее в отладочных целях, и её не следует использовать без крайней необходимости, т. к. это приводит к задержке обработки очереди событий [Главной Консоли](#).

### Команда @speak

Команда **@speak** служит для вывода речевых сообщений с помощью речевого синтезатора **Speech API**. Настройка параметров синтезатора выполняется командой [@speech](#). При выводе можно использовать теги **SAPI**, задающие скорость, громкость, тембр и другие параметры.

<b>@speak</b>	Команда речевого синтезатора.
Синтаксис:	
<b>@speak</b>	Показать справку по этой команде.
<b>@speak phrase</b>	Сказать фразу «phrase» через речевой синтезатор Speech API.
Примеры:	
<b>@speech engine 1</b>	
<b>@speak Привет, Мир.</b>	
<b>@speak Hello, World.</b>	
Примеры использования тегов SAPI:	
<b>@speak \Rst\Reset all tags to default values.</b>	
<b>@speak Pause \Pau=1000\in speech for 1000 milliseconds.</b>	
<b>@speak Speak \Vol=65535\strong (loud) or \Vol=0\weak (silent).</b>	
<b>@speak \Spd=100\Speech speed 100 words per minute.</b>	
<b>@speak Pitch (speech tone) \Pit=100\low \Pit=200\high.</b>	
<b>@speak Speak \Pro=1\with intonation or \Pro=0\without intonation (monotone).</b>	
<b>@speak Speak \RmS=1\by chars \rms=0\by normal words.</b>	
<b>@speak Force \Emp\next word with strong intonation.</b>	

## Команда @speech

Команда **@speech** служит для настройки речевого синтезатора **Speech API**. Она позволяет выводить список доступных речевых «движков» (engine), выбирать для работы один из них, задавать уровень громкости, скорость и тембр голоса, приостанавливать, возобновлять и отключать речевой синтезатор.

@speech	Команда работы с речевым синтезатором Speech API.
Синтаксис:	
@speech	Показать справку по этой команде.
@speech engine	Узнать текущий номер «движка» Speech API.
@speech engine n	Задать номер «движка» Speech API, n>0.
@speech engine 0	Отключить «движок» Speech API, запретить речевой вывод.
@speech engines	Вывести список доступных «движков» Speech API.
@speech volume	Узнать текущий уровень громкости Speech API.
@speech volume n	Задать текущий уровень громкости Speech API, 0 ... 100.
@speech pitch	Узнать текущий тембр Speech API.
@speech pitch n	Задать текущий тембр Speech API.
@speech speed	Узнать текущий уровень скорости Speech API.
@speech speed n	Задать текущий уровень скорости Speech API, 50 ... 150.
@speech pause	Узнать текущее состояние паузы Speech API.
@speech pause n	Задать текущее состояние паузы Speech API, 0/1.
@speech stop	Остановить речевое сообщение.
Примеры:	
@speech engine 1	
@speak Hello, World.	

## Команда @sysinfo

Команда **@sysinfo** выводит в консоль минимальную информацию о системе, включая версию **ОС**, системный каталог, домашний каталог программы, объем физической памяти, версию и частоту **CPU**, список **COM** портов, имя пользователя и компьютера, **IP** и **MAC** адреса, идентификатор и приоритет текущего процесса. Например:

```
@sysinfo
Краткая системная информация:
OS       : WIN-NT 10.0.17763
SysRoot  : C:\Windows
ComSpec  : C:\Windows\system32\cmd.exe
CrwRoot  : C:\Crw32EXE
CurrDir  : C:\Crw32exe
RAM      : 2047 MB (36% used, 1957 MB free)
CPU      : x86, Level:6, Revision:3C03, Number:2 (0,1)
Speed    : 2394.453 MHz (eestimated)
Clock    : Std=15.625,Max=15.625,Min=0.500,Act=15.625 ms
RS-232   :
User     : main is elevated member of group(s) ADMINS,USERS
Login    : W10X32-DEMO-VM\main on W10X32-DEMO-VM, as root
Process  : PID 5064, with Normal priority (8)
Network  : w10x32-demo-vm 192.168.56.104 08-00-27-E2-C8-32,08-00-27-5E-91-64
CmdLine  : "C:\Crw32exe\Crw32.exe"
```

## Команда @term

Команда **@term** служит для работы с терминалом. Терминал – это канал связи и консольное окно (в рамках пакета **CRW-DAQ**), которое позволяет работать с этим каналом связи. Допускаются следующие каналы связи: **task** (программа с каналами ввода-вывода), **pipe** (именованный канал), **tcp** (канал связи через порт **TCP/IP**), **com** (последовательный **COM** порт). Каналы **pipe**, **tcp** могут работать в роли сервера или клиента, что указывается в аргументах вызова.

```

@term                                     Команда работы с терминалом.
Syntax:
@term                                     Показать справку по этой команде.
@term /opts args                         Открыть окно терминала с заданными опциями и аргументами.
  Args                                   — Список аргументов:
    | task t                             — Запуск программы t с каналом ввода-вывода.
    | pipe n                             — Открыть именованный канал (сервер) с именем n.
    | pipe h\n                           — Открыть именованный канал (клиент) n связи с h.
    | tcp port n server m                — Открыть TCP порт n (сервер) для m клиентов.
    | tcp port n client h                — Открыть TCP порт n (клиент) для связи с h.
    | com port n baudrate m              — Открыть COM порт n со скоростью m.
  /opts — Список опций:
    | /a — Использовать ANSI/OEM фильтр. Пример: @term /a task cmd
    | /d — Запускать task в видимом окне. Пример: @term /d task cmd
    | /c — Закрывать окно при отключении. Пример: @term /ca task cmd
    | /v — Показывать отладочный текст. Пример: @term /v tcp port 123 server 2
    | /x — Входящие данные в HEX. Пример: @term /hx com port 2 baudrate 9600
    | /h — Выходящие данные в HEX. Пример: @term /hx com port 2 baudrate 9600
    | /u — Выходящие данные в URL. Пример: @term /ux com port 2 baudrate 9600
    | /s — Контрольные суммы ADAM. Пример: @term /s com port 2 baudrate 9600
Примеры:
@term /ca task cmd
  - запустить cmd.exe, использовать ANSI/OEM, закрывать при завершении
@term /a task ping.exe
  - запустить ping.exe, использовать ANSI/OEM, не закрывать при завершении
@term pipe exchange
  - открыть именованный канал (сервер) с именем exchange
@term pipe crwbox\exchange
  - открыть именованный канал (клиент) с именем exchange на сервере crwbox
@term tcp port 1234 server 2
  - открыть TCP порт 1234 как сервер для 2 клиентов
@term tcp port 1234 client crwbox
  - открыть TCP порт 1234 как клиент для подключения к серверу crwbox
@term /s com port 2 baudrate 9600
  - открыть COM порт 2 на скорости 9600, с контрольными суммами ADAM
@term /hxs com port 2 baudrate 9600
  - открыть COM порт 2 на скорости 9600, с контрольной суммой, ввод-вывод в HEX формате

```

При открытии терминала **task** запускается указанная при вызове программа, стандартный ввод-вывод которой перенаправляется в терминальное окно. При вызове также указываются опции. Например, вызов `@term /ca task cmd.exe` приводит к запуску **cmd.exe** в терминальном окне. Опция **/c** (close) означает, что окно закрывается после разрыва связи, т. е. завершения работы **cmd.exe**. Опция **/a** (ansi) означает, что к потокам ввода-вывода надо применять фильтр **ANSI/OEM** для согласования кодировок.

При открытии терминала именованного канала **pipe** указывается канал (для сервера) или **сервер\канал** (для клиента). Например, серверный терминал `@term pipe demo` может связаться с клиентским терминалом `@term pipe localhost\demo`.

При открытии терминала **tcp** указывается номер порта, роль (**server** или **client**). Для сервера указывается также максимальное число клиентов, а для клиента — имя подключаемого сервера. Например, серверный терминал `@term tcp port 12345 server 2` может связаться через порт 12345 с двумя клиентскими терминалами `@term tcp port 12345 client localhost`.

При открытии терминала **com** указывается номер порта, скорость обмена и другие параметры связи (четность, число битов данных и стоповых битов). Например, команда `@term com port 2 baudrate 9600` открывает терминал связи по порту **COM2** на скорости **9600** бод. Терминалы **com** полезны, например, при работе с аппаратурой.

## Команда @testbench

Команда **@testbench** служит для проведения тестирования и оценки производительности системы.

@testbench	Команда тестирования и измерения производительности системы.
Syntax:	
@testbench	Показать справку по этой команде.
@TestBench ping-pong start t1 p1 t2 p2	Начать тест ping-pong.
@TestBench ping-pong stop	Завершить тест ping-pong.
@TestBench multi-polling start n t p	Начать тест multi-polling.
@TestBench multi-polling stop	Завершить тест multi-polling.
@TestBench RTC n	Запустить тест RTC с n итерациями.
Примечания:	
1) Тест ping-pong создает два потока с периодами t1,t2 и приоритетами p1,p2 которые непрерывно обмениваются сообщениями по принципу «ping-pong» для тестирования механизма обмена сообщениями.	
2) Тест multi-polling создает n потоков с периодом t мс, и приоритетом p для проверки производительности системы при большом числе потоков.	
3) Тест RTC (realtime clock) измеряет производительность функций времени. Число итераций n по умолчанию равно 10000000.	
Примеры:	
@TestBench ping-pong start 10 tpNormal 10 tpNormal	
@TestBench ping-pong stop	
@TestBench multi-polling start 20 10 tpNormal	
@TestBench multi-polling stop	
@TestBench RTC	

Например, тест **ping-pong** оценивает скорость работы механизма обмена сообщениями, создав два потока, непрерывно посылающих друг другу сообщения и измеряя частоту посылок. Тест **multi-polling** может создать много потоков, чтобы оценить эффективность работы многопоточного режима в системе. Тест **RTC** оценивает скорость работы функций измерения времени.

## Команда @textmetadata

Команда **@textmetadata** служит для контроля целостности конфигурационных файлов, т. е. текстовых файлов с секционной структурой (\*.cfg, \*.crc, \*.cal, \*.ini). Функция считается устаревшей, т. к. её функции теперь входит в команду [@integrity](#).

@TextMetaData	Команда контроля целостности конфигурационных файлов.
Синтаксис:	
@textmetadata	Показать справку по этой команде.
@textmetadata --add f	Добавить контрольную сумму в файл f.
@textmetadata --check f	Проверить контрольную сумму в файле f.

## Команда @tooltip

Команда **@tooltip** служит для вывода на экран сообщений во «всплывающем» окне в нижней части экрана, где обычно находится «область уведомлений» (system tray).

Уведомление содержит обязательный текст уведомления, а также большой набор параметров, таких как: имя и размер шрифта; цвет текста и фона; прозрачность; задержка; «иконка» (поясняющая картинка), рисуемая слева от текста уведомления; звуковой и видео файл, сопровождающие появление окна; текст и команда от 0 до 9 кнопок для выполнения каких-то действий. Несколько часто используемых (стандартных) наборов параметров (preset) поименованы и доступны по короткому имени. Например, для информационных уведомлений используется набор **preset stdNotify**, а для аварийных сообщений используется **preset stdAlarm**. Использование стандартных наборов позволяет сделать вызов команды простым и понятным.



<b>@tooltip</b>	Команда вывода «всплывающих» уведомлений.
<b>Syntax:</b>	
<b>@tooltip</b>	Показать справку по этой команде.
<b>@tooltip args</b>	Показать «всплывающее» окно уведомления.
<b>args</b>	Список аргументов в виде пар «имя» «значение»:
<b>text "..."</b>	Задать текст уведомления. Это обязательный параметр.
<b>preset s</b>	Задать стандартный набор параметров уведомления с именем s: stdOk, stdNo, stdHelp, stdStop, stdDeny, stdAbort, stdError, stdFails, stdSiren, stdAlarm, stdAlert, stdBreak, stdCancel, stdNotify, stdTooltip, stdSuccess, stdWarning, stdQuestion, stdException, stdAttention, stdInformation, stdExclamation
<b>delay d</b>	Задать задержку d мс, после которой окно само закроется.
<b>font "name"</b>	Задать имя шрифта для окна, по умолчанию "PT Mono Bold".
<b>fontSize n</b>	Задать размер шрифта для окна, pt. По умолчанию 16.
<b>textColor c</b>	Задать цвет текста для окна. По умолчанию black.
<b>bkColor</b>	Задать цвет фона для окна. По умолчанию blue.
<b>ico f.ico</b>	Задать имя файла «иконки» f.ico.
<b>audio f.wav</b>	Задать имя файла звука f.wav.
<b>wav f.wav</b>	Задать имя файла звука f.wav.
<b>avi f.wav</b>	Задать имя файла видео f.avi.
<b>trans n</b>	Задать прозрачность (transparency), 0 ... 255.
<b>noDouble n</b>	Задать 0/1 чтобы не/показывать дубликаты уведомления.
<b>OnClick "..."</b>	Задать команды для выполнения по нажатию на тексте окна.
<b>Guid g</b>	Задать GUID метку g для идентификации окна уведомления.
<b>sure n</b>	Если n=1 и окна с заданным GUID нет, оно создается.
<b>delete g</b>	Удалить окно с заданным GUID-меткой g.
<b>progress p</b>	Показать индикатор «ход выполнения» с p процентами.
<b>run cmd</b>	Запуск команды cmd при открытии окна уведомления.
<b>BtnN "..."</b>	Задать текст для кнопки номер N=1..9.
<b>cmdN "cmd"</b>	Задать выполняемую команду для кнопки N=1..9.
<b>xml "&lt;..&gt;"</b>	Задать XML выражение для отправки в канал FP-QUI.
<b>Пример:</b>	
<b>@tooltip text "Демонстрационное уведомление." preset stdNotify</b>	
<b>@tooltip text "Сообщение об ошибке." preset stdError delay 15000</b>	

На рисунке 5 показан вид «всплывающего» окна уведомления при вызове команды **@tooltip text "Демонстрационное уведомление." preset stdNotify**. Окно появляется в правом нижнем углу экрана, где находится «область уведомлений». Цвет, тип и размер шрифта, картинка и звук окна определяются параметром **preset stdNotify**.

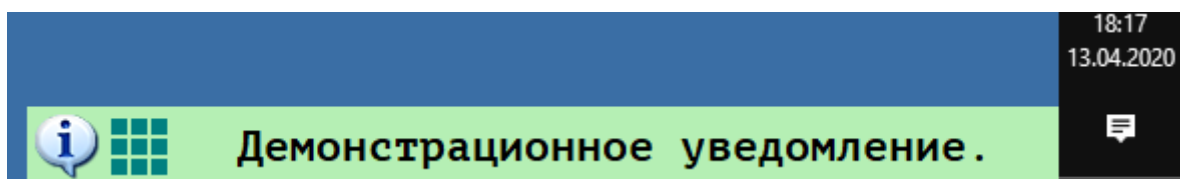


Рисунок 5. Внешний вид окна уведомления **@tooltip**.

Поскольку программы могут посылать много разных сообщений, которые могут повторяться или должны замещать друг друга, для сообщений можно вводить «метки» **GUID**, по которым они распознаются. Метки должны быть уникальными, чтобы окна с разными метками не мешали друг другу. Метки позволяют эффективно работать с окнами уведомлений. Так, можно удалять окна по заданной метке, обновлять их содержимое по метке вместо добавления нового окна и т. д.

Команда **@tooltip** основана на использовании модифицированной программы **FP-QUI** [5,6]. Она используется как система уведомлений в пакете **CRW-DAQ** для оповещения оператора о наступлении тех или иных событий, требующих внимания - например, в случае ошибок или аварийных ситуаций.

## Команда **@tty**

Команда **@tty** (это сокращение от teletype) служит для работы с программой **CRW-DAQ** через внешний (удаленный) терминал.

<b>@tty</b>	Команда доступа через удаленный терминал.
Синтаксис:	
<b>@tty status</b>	Показать статус TTY: открытые порты, статус защиты.
<b>@tty listen 0</b>	Слушать порт 0, доступ через утилиту «send2crwdaq».
<b>@tty close 0</b>	Заккрыть порт 0.
<b>@tty listen n</b>	Слушать TCP порт $n > 0$ , доступ через сокет TCP/IP.
<b>@tty close n</b>	Заккрыть TCP порт $n$ .
<b>@tty shield ON</b>	Включить защиту. Не выполнять входящие команды.
<b>@tty shield OFF</b>	Отключить защиту. Выполнять входящие команды.
<b>@tty verbose ON</b>	Включить режим подробной печати для протокола TCP.
<b>@tty verbose OFF</b>	Отключить режим подробной печати для протокола TCP.
<b>@tty panic</b>	Режим паники - закрыть все порты и включить защиту.
Примеры:	
<b>@tty status</b>	Показать статус удаленного терминала.
<b>@tty listen 0</b>	Открыть доступ через WM_COPYDATA (утилиту send2crwdaq).
<b>@tty listen 54321</b>	Открыть доступ через TCP порт 54321.
<b>@tty close 0</b>	Заккрыть доступ через WM_COPYDATA.
<b>@tty close 54321</b>	Заккрыть доступ через TCP порт 54321.

Команда **@tty** может открывать пронумерованные каналы связи и «прослушивать» (**listen**) порты связи, через которые внешние программы могут посылать сообщения в **Главную Консоль**. Эти сообщения выполняются на общих основаниях, как если бы они были введены локально. Это дает внешним программам большие возможности для организации совместной работы с пакетом **CRW-DAQ**.

Канал **@tty 0** реализован через механизм сообщений **WM\_COPYDATA**, которые можно посылать утилитой **unix send2crwdaq**, входящей в состав пакета. Сообщения **WM\_COPYDATA** передаются только локально, поэтому канал **@tty 0** доступен только локально работающим процессам. Зато этот канал не зависит от настроек **firewall**.

Каналы **@tty n** с номером  $n=1024...65535$  реализованы через порты **TCP/IP** (порты  $n=1...1023$  заняты под стандартные службы). Посылать команды в эти каналы можно, например, с помощью терминала **@term tcp port n client localhost** или команду **unix ncat**. Для удаленного доступа потребуется настроить **firewall**, чтобы разрешить нужный порт **TCP**.

Из соображений безопасности в начале все каналы **@tty** закрыты. Для открытия канала  $n$  надо вызвать **@tty listen n**. При этом канал открывается, а его счетчик увеличивается на 1. После работы с каналом его надо закрыть командой **@tty close n**. При этом счетчик канала уменьшается на 1. При нулевом счетчике канал закрывается. Счетчик позволяет правильно открывать и закрывать каналы в разных независимых друг от друга сеансах связи. При этом надо следить, чтобы каждый сеанс в конце работы закрывал открытый им канал.

Для временной приостановки выполнения удаленных команд **@tty** можно «поднять щит» командой **@tty shield on**, при этом каналы остаются открытыми, но удаленные команды игнорируются. Щит можно отключить командой **@tty shield off**.

Наконец, в случае угрозы безопасности можно вызвать команду «паника» **@tty panic** которая принудительно закрывает все каналы и сбрасывает их счетчики в 0, переводя **@tty** в безопасный режим.

В текущей версии каналы **@tty** работают только на чтение, т. е. через канал можно передавать команды в **Главную Консоль**, но нельзя получить по нему ответ. Однако обратную связь можно реализовать другими способами, а часто она и не требуется.

Проверить работу канала **@tty 0** с локальным каналом **0** можно с помощью утилиты **unix send2crwdaq**, например:

@tty listen 0	Открываем канал 0.
@TTY listen 0 count 1	Видим, что канал открылся.
@run unix send2crwdaq @echo Привет друг.	Запускаем тестовый процесс.
2020.04.14-00:53:19: Received 18 byte(s) ...	Принимаем от него команду.
@echo Привет друг.	Выполняем принятую команду.
Привет друг.	Видим результат выполнения.

В реальной практике внешние программы, конечно, передают более содержательные команды, но принцип остается тем же.

Проверить работу канала **@tty n** с портами **TCP/IP** можно через терминал **@term**. Например:

@tty listen 54321	Открыть порт 54321.
@term tcp port 54321 client localhost	Открыть клиентский терминал.
...	Терминал подключен к Главной Консоли.
...	Здесь можно передавать команды.
@tty close 54321	Закрывать порт 54321.

На практике пакет **CRW-DAQ** часто служит супервизором, который управляет запуском других программ. Вызываемые программы могут, в свою очередь, посылать ответные команды (например, уведомление о завершении работы) через каналы **@tty**, чтобы прикладная программа супервизора могла воспользоваться результатами вызванных программ. Это делает команду **@tty** полезным инструментом для организации программных комплексов, в которых работает несколько программ.

## Команда @view

Команда **@view** служит для работы с визуальными элементами — окнами, меню, статусными и инструментальными панелями. Она позволяет: узнать список визуальных элементов; показать или скрыть визуальный элемент с экрана; разрешить или запретить его; свернуть, нормализовать или распахнуть окно; изменить заголовок, размер и положение окна на экране, узнать параметры экрана.

<b>@view</b>	Команда работы с визуальными элементами.
<b>Синтаксис:</b>	
@view	Показать справку по этой команде.
@view list	Показать список визуальных элементов.
@view show n	Показать (сделать видимым) визуальный элемент «n».
@view hide n	Спрятать (сделать невидимым) визуальный элемент «n».
@view visible n	Узнать статус видимости визуального элемента «n».
@view enable n	Разрешить визуальный элемент «n».
@view disable n	Запретить визуальный элемент «n».
@view enabled n	Узнать статус разрешения визуального элемента «n».
@view min n	Свернуть (убрать с экрана) окно «n».
@view max n	Распахнуть (в максимальный размер) окно «n».
@view norm n	Вернуть (в нормальный размер) окно «n».
@view WindowState n	Узнать состояние окна визуального элемента «n».
@view title n t	Задать заголовок «t» окна «n».
@view pos n l t w h	Положение/размер left/top/width/height окна «n».
@view move n l t w h	Двигать/расширять left/top/width/height окно «n».
@view screen	Записать параметры экрана в константы screen_xxx.
<b>Примеры:</b>	
@view list	Вывод списка визуальных элементов.
@view hide FormCrw32.ToolBar	Спрятать Панель Инструментов.
@view show FormCrw32.ToolBar	Показать Панель Инструментов.
@view move FormCrw32 100 * * *	Сдвинуть главное окно влево.
@view pos FormCrw32 100 200 740 560	Задать ему новое положение/размер.
@view title FormCrw32 CRW32 #%ProcessId	Задать ему новый заголовок.
@view min FormDaqControlDialog	Свернуть окно «DAQ СИСТЕМА».
@view norm FormDaqControlDialog	Вернуть окно «DAQ СИСТЕМА».

Вызов **@view list** выводит список всех доступных визуальных элементов. В этом списке наиболее важную роль играют следующие элементы, которые доступны всегда:

FormCw32	Главное окно пакета CRW-DAQ.
FormCw32.ToolBar	Панель инструментов в главном окне.
FormCw32.StatusBar	Статусная строка в главном окне.
FormDaqControlDialog	Окно «DAQ СИСТЕМА».
FormMistimingService	Окно «Служба системного времени».
FormConsoleWindow	Окно «Главная Консоль».

Команды **@view show**, **@view hide**, **@view enable**, **@view disable**, **@view visible**, **@view enabled** позволяют показать, спрятать, разрешить, запретить и проверить статус визуальных элементов. Например:

@view hide FormCw32.ToolBar	Спрятать Панель Инструментов.
@view show FormCw32.ToolBar	Показать Панель Инструментов.

Команды **@view min**, **@view max**, **@view norm**, **@view pos**, **@view move**, **@view title** применимы к окнам (формам) и позволяют свернуть, развернуть в нормальный размер, распахнуть на максимум, задать положение/размер, сдвинуть/растянуть указанное окно и задать ему новый заголовок. Например:

@view pos FormCw32 100 200 740 560	Задать окну положение/размер.
@view title FormCw32 CRW32 #%ProcessId	Задать окну новый заголовок.

В командах **@view pos**, **@view move** при указании координат допустимо использовать формулы (вычисляемые выражения), а также подстановки переменных и констант.

Вызов **@view screen** записывает в константы из распечатки 4 текущие параметры экрана. Вместе с командой **@view pos** это позволяет правильно располагать и масштабировать окна, исходя из реальных размеров экрана. Например, можно задать главному окну размер в половину экрана и расположить его по центру:

```
@view pos FormCw32 %screen_width/4 %screen_height/4 %screen_width/2 %screen_height/2
```

Здесь использованы формулы с подстановкой констант экрана, например формула **%screen\_width/2** задает половину ширины экрана.

Команда **@view** дает хороший инструмент, позволяющий прикладным программам гибко управлять элементами графического интерфейса и адаптироваться к реальным размерам экрана.

## Практика применения Главной Консоли

Окно [Главной Консоли](#) и связанный с ним [Системный Калькулятор](#) являются мощными и полезными инструментами для прикладного программирования. Они применяются в самых разных вариантах. Здесь описаны некоторые наиболее важные их практические применения.

## Ручное управление через Главную Консоль

Самый простой и очевидный метод применения [Главной Консоли](#) — это ручной ввод команд в окне консоли. Его возможности не надо недооценивать, он играет огромную роль при разработке и отладке программ. Недостатком этого метода являются высокие требования к уровню подготовки, что затруднительно для рядового оператора.

Тем не менее, возможность оперативно проверить работу любой команды, просто набрав её в окне консоли, трудно переоценить. Эта возможность делает [Главную Консоль](#) мощным инструментом разработки прикладных программ. Все остальные методы применения [Главной Консоли](#) используют команды, которые проверяются и отлаживаются в ручном режиме, прежде чем попадают в программный код.

## Стартовые скрипты для инициализации программ

Другим важным методом применения [Главной Консоли](#) являются **стартовые скрипты** (сценарии), служащие для инициализации пакета, а также прикладных программ после их загрузки.

Стартовые скрипты обычно располагаются в тексте секций конфигурационных файлов и выполняются путем послыки находящихся в тексте команд в [Главную Консоль](#). Часто команды снабжены префиксом **@silent**, чтобы не загромождать консоль избыточным выводом.

Инициализация пакета **CRW-DAQ** делается, в частности, с помощью стартового скрипта **Crw32.ini [System.StartupScript]**. В этом скрипте, например, задаются нужные переменные окружения, свойства используемых программой шрифтов, параметры буферов памяти и других важных объектов программы, определяются параметры экрана. Этот скрипт играет большую роль в работе пакета.

Инициализация прикладных программ **DAQ** системы, написанных на языке [DaqPascal](#), тоже часто происходит с использованием стартовых скриптов. По сложившейся практике стартовые и завершающие скрипты для устройства с именем (например) **&DeviceName** располагаются в секциях конфигурационного файла, на которые указывают переменные **[&DeviceName] StartupScript** и **[&DeviceName] FinallyScript**. Обычно эти переменные указывают на секции **[&DeviceName.StartupScript]** и **[&DeviceName.FinallyScript]**. В текст секций помещаются необходимые команды, выполняемые с помощью функции **eval**. Надо помнить, что у программы [DaqPascal](#) есть свой отдельный программный поток и объект **DaqScript**, который связан с функцией **eval**. Доступ к [Главной Консоли](#) в этом случае делается через команду **@eval @system @async cmd**, которая не выполняет целевую команду **cmd** сама, а посылает её в [Главную Консоль](#) для асинхронного выполнения в главном потоке программы в общем цикле обработки событий.

Особенно важную роль играют стартовые скрипты сервера времени **&CronSrv**, для использования которого достаточно включить в **DAQ** конфигурацию файл **Resource\DaqSite\Default\CronSrv.cfg**:

```
[ConfigFileList]
; Включить стандартную конфигурацию сервера времени
ConfigFile = ~~\Resource\DaqSite\Default\CronSrv.cfg
[]
```

В стартовых скриптах сервера **&CronSrv** обычно располагаются команды для настройки общих параметров работы **DAQ** системы после её загрузки. Вот типичный пример такого скрипта:

<b>[&amp;CronSrv.StartupScript]</b>	Стартовый скрипт &CronSrv.
<b>;----- Set polling &amp; priority &amp; memory</b>	
<b>@Eval @System @Async SetClockRes(4)</b>	Задать период таймера.
<b>@Eval @System @Async @Memory CfgCacheSize 0</b>	Очистить буфер кэш-памяти.
<b>@Eval @System @Async @Memory min 1024*1024*64</b>	Задать мин. и макс. объем
<b>@Eval @System @Async @Memory max 1024*1024*512</b>	рабочей памяти процесса.
<b>@Eval @System @Async @Polling PriorityClass 24 1000</b>	Задать приоритет процесса.
<b>@Eval @System @Async pidaffinity(0,2^(cpu_count()-1))</b>	Задать привязку процесса к CPU
<b>;----- MainConsole</b>	
<b>@Eval @System @Async @View hide FormCrw32.ToolBar</b>	Убрать Панель Инструментов.
<b>@Eval @System @Async @View show FormCrw32.StatusBar</b>	Показать Строку Статуса.
<b>@Eval @System @Async @View min FormDaqControlDialog</b>	Свернуть окно «DAQ СИСТЕМА».
<b>;----- CronStat</b>	
<b>@cron.tab CronStat 0 0 0</b>	Создать задание CronStat,
<b>@cron.pul CronStat 10000</b>	выполняемое раз в 10 секунд.
<b>@cron.job CronStat @cron.cpu</b>	Для печати статуса &CronSrv.
<b>...</b>	... и так далее ...
<b>[]</b>	



## Вызов команд Главной Консоли из сценариев Script

Сценарии [Script](#) выполняются в пакетном режиме в своем отдельном программном потоке и работают со своим собственным объектом **DaqScript**. Они не могут непосредственно выполнять команды [Главной Консоли](#). Однако вызовом `@system @async cmd` они могут помещать команду `cmd` в очередь [Главной Консоли](#) для её асинхронного выполнения в главном потоке в общем цикле обработки событий.

Вызов `@system @async cmd` возвращает длину строки команды `cmd`, которая была скопирована в очередь событий [Главной Консоли](#) или 0 при ошибке (например, переполнении FIFO буфера). Это составная команда: `@system` дает доступ к Системному Калькулятору, а `@async` посылает её в очередь [Главной Консоли](#).

Таким образом, с помощью `@system @async` программы **DaqScript** получают доступ ко всем командам [Главной Консоли](#), что позволяет использовать их в любых прикладных программах **DAQ** системы.

## Вызов команд Главной Консоли из программ DaqPascal

В программный интерфейс [DaqPascal API](#) входит функция `eval` для вычисления выражения **DaqScript** в командном режиме, которая дает доступ к внутреннему объекту **DaqScript**, которым обладает каждая программа [DaqPascal](#).

Функция `eval` выполняется в потоке программы [DaqPascal](#) и не может непосредственно выполнять команды [Главной Консоли](#). Однако вызов `eval('@system @async '+cmd)` может помещать команду `cmd` в очередь [Главной Консоли](#) для её асинхронного выполнения в главном потоке в общем цикле обработки событий.

Таким образом, с помощью функции `eval` программы [DaqPascal](#) получают доступ ко всем командам [Главной Консоли](#), что позволяет использовать их в любых прикладных программах **DAQ** системы.

## Вызов команд Главной Консоли из ядра CRW-DAQ

Многие модули ядра пакета **CRW-DAQ** используют [Главную Консоль](#) как инструмент для выполнения тех функций, которые можно выполнить посылкой команд в [Главную Консоль](#). Это позволяет уменьшить объем кода, сделать его проще и надежнее.

## Сценарии контроля целостности @integrity

Команды [Главной Консоли](#) используются в сценариях [@integrity](#), определяющих реакцию системы контроля целостности данных на различные события, как подробно описано в справке по этой команде.

## Удаленный вызов команд Главной Консоли

Каналы связи [@tty](#) позволяют принимать команды [Главной Консоли](#) удаленно, как из других локальных процессов, так и из сети. Это позволяет создавать сложные распределенные программные комплексы, в которых совместно работает много независимых программ.

Командой `@run` из пакета можно запускать другие программы, нужные для работы комплекса. А эти программы, в свою очередь, могут посылать сообщения в [Главную Консоль](#) через каналы `@tty`, чтобы взаимодействовать с пакетом **CRW-DAQ**.

Таким образом, с помощью команд [Главной Консоли](#) пакет **CRW-DAQ** может выполнять роль супервизора, управляющего сложным программным комплексом, состоящим из нескольких независимых программ.

## Применение DaqScript в DAQ системе

Интерпретатор **DaqScript** применяется не только в окне **Главной Консоли**. Он активно используется в **DAQ** системе для создания алгоритмов управления и сбора данных исследовательских установок.

### Применение DaqScript в программах DaqPascal

Каждая программа на языке **DaqPascal** имеет свой экземпляр объекта **DaqScript** со своим набором данных. Для доступа к нему есть две функции – **eval** и **eval**. Также есть функция **global** доступа к **Системному Калькулятору** и окну **Главной Консоли**.

Функция **eval(s)** вычисляет строковое выражение **s** с помощью объекта **DaqScript** и возвращает результат вычисления. При ошибке интерпретации возвращается **NaN**. Вычисление идет в командном режиме интерпретатора.

Функция **eval(n,v)** задает переменной по имени **n** значение **v** типа **real** (что соответствует **double**) и возвращает статус операции. Она служит для неискаженной (без ошибок округления) записи в интерпретатор значения переменной. Для чтения значения переменной по имени **n** достаточно вызвать **eval(n)**.

Функция **global(s)** вычисляет строковое выражение **s** с помощью **Системного Калькулятора**. Вызов этой функции подобен вызову **eval('@global '+s)** или **eval('@system '+s)**. При её выполнении **Системный Калькулятор** блокируется для потоковой безопасности. Вызов **global('@async '+s)** позволяет выполнять команды **Главной Консоли** асинхронно, в главном потоке, помещая их в общую очередь событий.

Перечисленных функций вполне достаточно для работы с интерпретатором **DaqScript** из программ **DaqPascal**. За счет этого прикладные программы получают новое качество – возможность интерпретировать команды пользователя **online**, без необходимости остановки, компиляции и перезапуска программы.

Типичным применением **DaqScript** в прикладных программах **DaqPascal** является проведение расчетов по формулам, заданным оператором в интерактивном режиме, или загруженным из файла конфигурации или принятым из канала связи. Формулы могут также задавать параметры и алгоритм работы системы управления, например, определять время срабатывания исполнительных устройств. Например:

pulse:=0;	Инициализация
s:='ge(t,t1)*le(t,t2)';	Формула для расчета
if eval('t',msecnow) and eval('t1',20) and eval('t2',21)	Задать переменные
then pulse:=eval(s) else writeln('Ошибка вызова eval');	Расчет по формуле
if isnan(pulse)	Проверка результата
then writeln('Ошибка вызова eval')	Если NaN, то ошибка
else Execute(pulse);	Исполнение команды

Другим, более редким применением **DaqScript** являются задачи, требующие хранения большого объема данных. В языке **DaqPascal** нет динамических массивов, а в **DaqScript** число динамических переменных ничем не ограничено. Это позволяет использовать **DaqScript** как контейнер данных с доступом по ключу (имени переменной). Эта возможность существует, но считается устаревшей, т. к. для решения этих задач была создана группа функций работы с хэш-таблицами.

Объект **DaqScript**, связанный с программами **DaqPascal**, не имеет дополнительных констант, команд и функций, кроме основного набора.

## Применение DaqScript в сценариях Script

Прикладные программы для DAQ системы можно создавать (кроме **DaqPascal**) также и на языке **DaqScript**. Для этого служат устройства (device) типа **Script**, которые содержат в себе экземпляр объекта **DaqScript** для выполнения прикладных программ (сценариев) сбора и обработки данных. Сценарии устройств **Script** выполняются в пакетном режиме интерпретатора и работают в своем отдельном программном потоке. Здесь приводится обзорное описание устройств типа **Script**.

### Объявление устройств Script

Для использования устройства типа **Script** с именем (для примера) **&ScriptName** с программой на языке **DaqScript** надо включить в конфигурацию DAQ системы объявление, вид которого приведен на распечатке 7, где жирным выделены зарезервированные слова.

*Распечатка 7. Объявление устройства Script с программой DaqScript.*

<b>[DeviceList]</b>	В секции [DeviceList]
<b>&amp;ScriptName</b> = device software script	объявляем устройство script.
<b>[&amp;ScriptName]</b>	В секции с именем устройства
<b>InquiryPeriod</b> = 10	Задаем период опроса, мс.
<b>DevicePolling</b> = 1000, tpNormal	Задаем период и приоритет потока.
<b>ScriptSection</b> = <b>&amp;ScriptName</b> .Script	Задаем имя секции с кодом скрипта.
<b>Comment</b> = Демонстрационный пример	Задаем комментарий для устройства.
...	Задаем другие параметры.
<b>[ ]</b>	
<b>[&amp;ScriptName.Script]</b>	В секции с кодом скрипта
var runcount	помещаем прикладную программу
runcount=runcount+1	на языке DaqScript.
echo &ScriptName RunCount = %runcount	
<b>[ ]</b>	

В секции **[DeviceList]** объявлением **&ScriptName = device software script** создается устройство **&ScriptName** класса **Script**. В секции описания устройства задаются параметры, включая: имя секции **ScriptSection**, где расположен текст программы (скрипта) на языке **DaqScript**; период опроса **InquiryPeriod** устройства (мс); период опроса (мс) и приоритет потока **DevicePolling**. Приоритет потока может быть: **tpIdle**, **tpLowest**, **tpLower**, **tpNormal**, **tpHigher**, **tpHighest**, **tpTimeCritical**. Остальные параметры не обязательны и могут иметь значения по умолчанию.

### Описание конфигурации устройств Script

После объявления устройства **Script** надо задать его параметры конфигурации, кратко перечисленные в распечатке 8.

Кроме уже перечисленных обязательных параметров задается: комментарий **Comment**; размеры **FIFO** буферов для событий **AnalogFifo** и **DigitalFifo**; число подключаемых аналоговых и цифровых входов и выходов **AnalogInputs**, **AnalogOutputs**, **DigitalInputs**, **DigitalOutputs**, число калибровок **Calibrations**. Другие параметры меняются редко. Полное описание параметров содержится в **online** справке пакета.

При необходимости обработки измеряемых данных надо подключить данные (кривые) к аналоговым/цифровым входам/выходам устройства с помощью конструкции **Link ... with curve ...**, как указано в распечатке 8.

При необходимости к устройству также подключаются калибровки с помощью конструкции **Calibration#n** с номером калибровки **n**. Они нужны для обработки данных, поступающих от датчиков, имеющих калибровки.

**Распечатка 8. Параметры описания конфигурации устройства *Script*.****Обязательные параметры:**

ScriptSection = s	Имя секции, где расположен код скрипта на DaqScript.
InquiryPeriod = t	Период t (мс) опроса устройства.
DevicePolling = t, p	Период t (мс) и приоритет p программного потока. p=(tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical).

**Часто используемые параметры:**

Comment = s	Комментарий s - строка, поясняющая назначение устр-ва.
AnalogFifo = n	Размер n очереди FIFO аналоговых событий (в событиях).
DigitalFifo = n	Размер n очереди FIFO цифровых событий (в событиях).
AnalogInputs = n	Задаёт число n аналоговых входов. По умолчанию 0.
AnalogOutputs = n	Задаёт число n аналоговых выходов. По умолчанию 0.
DigitalInputs = n	Задаёт число n цифровых входов. По умолчанию 0.
DigitalOutputs = n	Задаёт число n цифровых выходов. По умолчанию 0.
Calibrations = n	Задаёт число n калибровок. По умолчанию 0.

**Редко используемые параметры:**

StartingOrder = n	Порядок (приоритет) n старта. По умолчанию 0.
StoppingOrder = n	Порядок (приоритет) n остановки. По умолчанию 0.
DispatcherFilter = s	Строка s задаёт параметры диспетчера событий.
GroupTag = n	Задаёт тег группы устройств. По умолчанию 0.
DebugMode = n	Задаёт режим отладки n. По умолчанию 0.

**Подключение данных (кривых):**

Link DigitalInput n with curve c bit b  
- Подключение кривой на цифровой вход

Link DigitalOutput n with curve c history h  
- Подключение кривой на цифровой выход

Link AnalogInput n with curve c smoothing w p k1 k2 history h  
- Подключение кривой на аналоговый вход

Link AnalogOutput n with curve c tolerance a r history h  
- Подключение кривой на аналоговый выход

- где:

- c - имя подключаемой кривой
- n - номер аналогового/цифрового входа/выхода
- b - номер бита данных для цифрового входа
- h - длина хранимой истории (макс. число точек)
- a, r - абсолютный и относительный квант округления
- w, p, k1, k2 - параметры сглаживания (см. справку пакета)

**Подключение калибровок:**

Calibration#n = file xname yname zname xscale yscale a b  
- где:

- n - номер калибровки, 0 ... Calibrations-1.
- file - ссылка на файл калибровки, \*.cal.
- xname - имя аргумента (независимой переменной).
- yname - имя значения (зависимой переменной).
- zname - имя дополнительного параметра.
- xscale - тип шкалы аргумента при создании калибровки.
- yscale - тип шкалы значения при создании калибровки.
- a b - область определения (диапазон изменения) аргумента.

**Дополнительные функции и команды устройств *Script***

Объект **DaqScript** в устройствах **Script** имеет, кроме основного набора [функций](#) и [команд](#), дополнительный набор функций и команд, приведенных на распечатке 9 и 10. Они позволяют создавать на языке **DaqScript** прикладные сценарии управления, сбора и анализа данных, графического интерфейса пользователя в **DAQ** системе. В силу ограничений языка, эти сценарии не могут служить полной заменой [DaqPascal](#), но простота создания сценариев делает их полезным инструментом разработки, в дополнение к [DaqPascal](#).

**Распечатка 9. Дополнительный набор функций устройства Script.**

<code>time()</code>	Узнать время от старта DAQ системы в единицах timeunits.
<code>timeunits()</code>	Узнать единицу измерения времени DAQ системы, в мс.
<code>crvlen(c)</code>	Узнать длину (число точек) кривой со ссылкой c.
<code>crvx(c,i)</code>	Узнать абсциссу x кривой c в точке номер i=(1..crvlen(c)).
<code>crvy(c,i)</code>	Узнать ординату y кривой c в точке номер i=(1..crvlen(c)).
<code>crvput(c,i,x,y)</code>	Поместить в кривую c в точке номер i данные (x,y).
<code>crvinteg(c,a,b)</code>	Узнать интеграл кривой c в интервале абсцисс (a..b).
<code>numais()</code>	Узнать Number of Analog Inputs, число аналоговых входов.
<code>numdis()</code>	Узнать Number of Digital Inputs, число цифровых входов.
<code>numaos()</code>	Узнать Number of Analog Outputs, число аналоговых выходов.
<code>numdos()</code>	Узнать Number of Digital Outputs, число цифровых выходов.
<code>refai(n)</code>	Узнать ссылку на аналоговый вход n или 0 если его нет.
<code>refdi(n)</code>	Узнать ссылку на цифровой вход n или 0 если его нет
<code>refao(n)</code>	Узнать ссылку на аналоговый выход n или 0 если его нет
<code>refdo(n)</code>	Узнать ссылку на цифровой выход n или 0 если его нет
<code>getai(n,t)</code>	Ордината y кривой на аналоговом входе n в момент времени t.
<code>getai_n(n)</code>	Число точек кривой на аналоговом входе n.
<code>getai_xn(n)</code>	Абсцисса x последней точки кривой на аналоговом входе n.
<code>getai_yn(n)</code>	Ордината y последней точки кривой на аналоговом входе n.
<code>getai_xi(n,i)</code>	Абсцисса x точки номер i кривой на аналоговом входе n.
<code>getai_yi(n,i)</code>	Ордината y точки номер i кривой на аналоговом входе n.
<code>getdi(n,t)</code>	Значение цифрового входа n в момент времени t.
<code>getdi_n(n)</code>	Число точек кривой на цифровом входе n.
<code>getdi_xn(n)</code>	Абсцисса x последней точки кривой на цифровом входе n.
<code>getdi_yn(n)</code>	Ордината y последней точки кривой на цифровом входе n.
<code>getdi_xi(n,i)</code>	Абсцисса x точки номер i кривой на цифровом входе n.
<code>getdi_yi(n,i)</code>	Ордината y точки номер i кривой на цифровом входе n.
<code>putev(w,c,t,d0,d1)</code>	Генерирует событие (what,chan,time,data0,data1).
<code>putao(n,t,d)</code>	Записать аналоговое событие: канал n, время t, данные d.
<code>putdo(n,t,d)</code>	Записать цифровое событие: канал n, время t, данные d.
<code>numcals()</code>	Узнать число подключенных к устройству калибровок.
<code>refcalibr(n)</code>	Ссылка на калибровку n или 0 если ее нет.
<code>calibr(n,d,p)</code>	Выполнить калибровочное преобразование. n - номер калибровки, d - данные, p - параметр.
<code>inportb(n)</code>	Ввод байта из порта ввода-вывода n.
<code>inportw(n)</code>	Ввод слова из порта ввода-вывода n.
<code>outportb(n,d)</code>	Вывод байта d в порт ввода-вывода n.
<code>outportw(n,d)</code>	Вывод слова d в порт ввода-вывода n.
<code>tm_new()</code>	Создать новый таймер и вернуть ссылку на него.
<code>tm_free(a)</code>	Удалить таймер со ссылкой a.
<code>tm_addint(a,b)</code>	Добавить в таймер a новый интервал b, [мс].
<code>tm_numint(a)</code>	Узнать число интервалов таймера a.
<code>tm_getint(a,b)</code>	Узнать длину интервала номер b таймера a, [мс].
<code>tm_setint(a,b,c)</code>	Задать длину c, [мс] интервала номер b таймера a.
<code>tm_gettime(a)</code>	Узнать время со старта таймера a, [мс].
<code>tm_start(a)</code>	Запуск таймера a.
<code>tm_stop(a)</code>	Остановка таймера a.
<code>tm_isstart(a)</code>	Узнать, запущен ли таймер a.
<code>tm_event(a)</code>	Генерация событий таймера a.
<code>tm_curint(a)</code>	Номер текущего интервала таймера a.
<code>igettag(a)</code>	Чтение значения integer тега a.
<code>rgettag(a)</code>	Чтение значения real тега a.
<code>isettag(a,b)</code>	Запись в integer тег a значения b.
<code>rsettag(a,b)</code>	Запись в real тег a значения b.
<code>typetag(a)</code>	Узнать тип тега a: 0/1/2/3=error/integer/real/string.
<code>clicktag()</code>	Узнать тег нажатого сенсора или 0 если сенсор не нажат.
<code>clickbutton()</code>	Узнать нажатую кнопку мыши: 1/2/4=Left/Right/Middle.
<code>vnew(a)</code>	Создать массив размерности a и вернуть ссылку на него.
<code>vfree(a)</code>	Удалить созданный vnew массив, заданный ссылкой a.
<code>vsize(a)</code>	Узнать размер (длину) массива заданного ссылкой a.
<code>vget(a,b)</code>	Чтение элемента массива a с номером b.
<code>vput(a,b,c)</code>	Запись в элемент массива a номер b значения c.

Примечание: напоминаем, что точки в кривых индексируются с 1, а аналоговые и цифровые входы и выходы - с 0.



**Распечатка 10. Дополнительный набор команд устройства Script.**

@inittag n t	Найти или создать тег с именем n и типом t=1..3.
@findtag n	Найти тег по имени n, вернуть ссылку в actionresult.
@clicksensor name	Проверить, нажат ли сенсор с именем name.
@action arg	Вызвать метод action для списка устройств arg.
@clear arg	Вызвать метод clear для списка устройств arg.
@cleardevice arg	Вызвать метод cleardevice для списка устройств arg.
@start arg	Вызвать метод start для списка устройств arg.
@stop arg	Вызвать метод stop для списка устройств arg.
@devmsg dev msg	Посылка сообщения msg устройству dev с синхронизацией.
@devsend dev msg	Посылка сообщения msg устройству dev с синхронизацией.
@devpost dev msg	Посылка сообщения msg устройству dev без синхронизации.
@devsendmsg dev msg	Посылка сообщения msg устройству dev с синхронизацией.
@devpostmsg dev msg	Посылка сообщения msg устройству dev без синхронизации.
@debugout msg	Вывод сообщения msg в файл отладки.
@clearcurve arg	Очистить кривые из списка имен arg.
@savecrw f n t l c	Выполнить сохранение списка кривых в файл CRW. f - имя *.crw файла (можно без пути и без расширения). n - имя окна под которым оно попадет в crw-архив. t - заголовок в верхней части окна кривых. l - метка в нижней части окна кривых. c - список имен сохраняемых кривых.
@specmarker win	Узнать Marker спектрометрического окна с именем win.
@specmarkerl win	Узнать MarkerL спектрометрического окна с именем win.
@specmarkerr win	Узнать MarkerR спектрометрического окна с именем win.
@specroil win	Узнать RoiL спектрометрического окна с именем win.
@specroir win	Узнать RoiR спектрометрического окна с именем win.
@windraw win	Перерисовать окно с именем win.
@winshow win	Показать (сделать видимым) окно с именем win.
@winhide win	Спрятать (убрать с экрана) окно с именем win.
@winselect win	Активизировать окно с именем win.

Ввиду большого объема подробное описание функций и команд здесь неуместно, оно есть в **online** справке пакета. Здесь можно дать лишь краткий обзор функций по группам.

Функции времени (**time**, **timeunits**) и таймера (**tm\_new**, **tm\_free**, **tm\_addint**, **tm\_numint**, **tm\_getint**, **tm\_gettime**, **tm\_start**, **tm\_stop**, **tm\_isstart**, **tm\_event**, **tm\_curint**) позволяют измерять время и организовывать периодические или запланированные по времени события.

Функции (**numais**, **numaos**, **numdis**, **numdos**) позволяют узнать число используемых входов и выходов, а (**refai**, **refao**, **refdi**, **refdo**) – получить ссылки подключенных к ним кривых. С кривыми можно работать по номерам входов-выходов (**getai**, **getai\_n**, **getai\_xn**, **getai\_yn**, **getai\_xi**, **getai\_yi**, **getdi**, **getdi\_n**, **getdi\_xn**, **getgi\_yn**, **getdi\_xi**, **getdi\_yi**, **putao**, **putdo**) или по ссылкам (**crvlen**, **crvx**, **crvy**, **crvput**, **crvinteg**).

Функции (**igettag**, **isettag**, **rgettag**, **rsettag**, **typetag**, **@inittag**, **@fintag**) позволяют работать с тегами данных **DAQ** системы.

Функции сообщений (**@devmsg**, **@devsend**, **@devsendmsg**, **@devpost**, **@devpostmsg**) и контроля (**@action**, **@clear**, **@cleardevice**, **@start**, **@stop**) помогут взаимодействовать и управлять устройствами **DAQ** системы.

Функции обработки нажатий сенсоров (**clicktag**, **clickbutton**, **@clicksensor**) и работы с окнами (**@windraw**, **@winshow**, **@winhide**, **@winselect**) позволяют решать задачи организации интерфейса пользователя.

Функции калибровки (**numcals**, **refcalibr**, **calibr**) помогут делать обработку данных с помощью калибровочных преобразований.

Команды (**@savecrw**, **@cleacurve**) служат для сохранения кривых в файлы формата **\*.CRW** и очистки кривых после сохранения.

Другие функции используются редко и описаны в **online** справке.

## Пример простой конфигурации устройства Script

В распечатке 11 содержится пример простой демонстрационной конфигурации устройства типа **Script**. В этом примере устройство **&WaveDemo** генерирует синусоидальную «волну» и записывает её в кривую Wave, подключенную к аналоговому выходу.

### Распечатка 11. Простейший пример конфигурации устройства **Script**.

[DeviceList]	Секция списка устройств
&WaveDemo = device software script	Объявление устройства
[&WaveDemo]	Секция описания устройства
Comment = Wave simulator for DEMO	Комментарий устройства
InquiryPeriod = 10	Период опроса устройства
DevicePolling = 100, tpNormal	Период и приоритет потока
ScriptSection = &WaveDemo.Script	Задать секцию скрипта
AnalogOutputs = 1	Задать число выходов
Link AnalogOutput 0 with curve Wave history 1000	Подключить кривую Wave
[ ]	
[&WaveDemo.Script]	Секция с кодом на DaqScript
var t,v	Объявление переменных
t=time()	Чтение текущего времени DAQ
v=sin(2*pi*t)	Вычисление сигнала «волны»
putao(0,t,v)	Запись результата в кривую
[ ]	

Обратите внимание, запись данных в кривую будет происходить с периодом 100 мс, хотя указан период опроса устройства 10 мс. Дело в том, что период опроса определяется максимальным значением из **InquiryPeriod** и **DevicePolling**. Опрос устройства не может происходить чаще, чем опрашивается программный поток, в котором оно работает.

## Практика использования устройств Script

Являясь интерпретатором, язык **DaqScript** значительно уступает компилятору **DaqPascal** в производительности **runtime** кода. Поэтому все серьёзные прикладные программы создаются обычно на **DaqPascal**. Однако у языка **DaqScript** и **DAQ** устройств типа **Script** есть сфера применения, в которой они используются часто и весьма успешно.

Это, в первую очередь, **симуляторы** и **отладочные сценарии**, которые являются важной частью инструментария разработчика прикладного ПО и практически всегда входят в цикл разработки и поддержки систем сбора данных и управления.

Важность симуляторов трудно переоценить. Разработка систем управления неизбежно требует создания симуляторов, т. к. вести разработку и отладку на реальном оборудовании зачастую просто нет возможности. Некоторые режимы работы оборудования (особенно аварийные) можно отлаживать только на симуляторах, т. к. создание опасных ситуаций ради проверки системы управления нецелесообразно.

В то же время, от симуляторов обычно не требуется высокая производительность или качество кода, поэтому использование **DaqScript** для этих целей вполне допустимо.

Удобство **DaqScript** состоит в том, что сценарии простых задач, которыми являются симуляторы, пишутся очень быстро, и могут быть расположены прямо в файле конфигурации, не требуя отдельного файла для кода программы. Это позволяет сэкономить время на разработку и поддержку симуляторов. По этой причине симуляторы очень часто пишутся на языке **DaqScript** в устройствах **Script**.

## Применение DaqScript в мнемосхемах

Мнемосхемы являются одним из основных элементов графического интерфейса **DAQ** системы пакета **CRW-DAQ**. Поскольку мнемосхемам посвящено отдельное [описание](#), здесь дается лишь минимальный объем сведений, касающийся использования **DaqScript** в мнемосхемах.

### Общие сведения о мнемосхемах

Окна мнемосхем – графические окна **DAQ** системы, содержащие постоянное **основное изображение** и массив изменяющихся **сенсоров**, т. е. изображений, чувствительных к действиям пользователя (нажатию кнопок клавиатуры и мыши) и меняющих вид в зависимости от значений подключенных **источников данных** (тегов или кривых).

Принципиальная схема работы мнемосхем показана на рисунке 6.

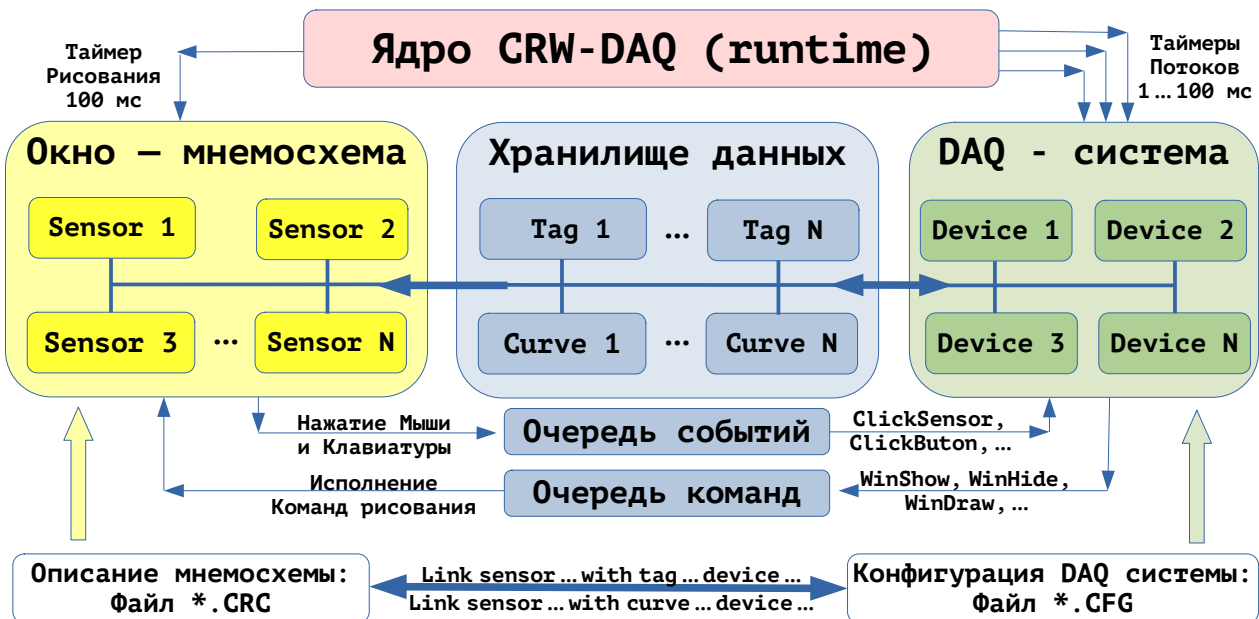


Рисунок 6. Принципиальная схема работы мнемосхем.

Окно – мнемосхема, описание которого находится в **\*.CRC** файле, содержит массив сенсоров (**sensor**), внешний вид которых зависит от значений подключенных источников данных из хранилища данных, т. е. тегов (**tag**) и кривых (**curve**). Значения тегов и кривых определяются алгоритмами работы массива устройств (**device**) **DAQ** - системы, т. е. драйверов устройств или прикладных программ на языках **DaqPascal**, **DaqScript**. Описание хранилища данных и устройств содержится в конфигурации **DAQ** системы, которая расположена в файле **\*.CFG**. Хранилище данных описывается в секциях **[TagList]**, **[DataStorage]**, а массив устройств объявляется в секции **[DeviceList]**.

Поскольку окна мнемосхем и устройства **DAQ** системы работают в разных программных потоках, они не должны взаимодействовать прямо – это противоречит потоковой безопасности. Поэтому предприняты все возможные меры для максимальной **изоляции** мнемосхем от устройств, т. е. прикладных программ сбора данных и управления. Для обеспечения изоляции мнемосхемы рисуются не вызовом функций рисования из прикладных программ, а только по системному таймеру (с периодом около 100 мс), который генерируется исполнительной системой (**runtime**), т. е. ядром пакета **CRW-DAQ**. Данные для рисования берутся из потокобезопасного хранилища данных.

Для изоляции окон мнемосхем, события сенсоров (нажатия мыши и клавиатуры) помещаются в потокобезопасный **FIFO** буфер **очереди событий**, а в прикладных программах извлекаются из очереди и обрабатываются с помощью функций **API** (**ClickSensor**, **ClickButton**, ...). При необходимости прикладные программы с помощью функций **API** (**WinShow**, **WinHide**, **WinDraw**, ...) могут посылать команды рисования, которые помещаются в потокобезопасный **FIFO** буфер **очереди команд**, а выполняются по системному таймеру в основном потоке программы, где и происходит рисование всех окон. Так потоки устройств (прикладных программ) полностью изолируются от потока рисования мнемосхем.

Большое внимание к изоляции потоков связано с тем, что пакет **CRW-DAQ** является **многопоточным** и требует дисциплины при работе.

Помимо потоковой безопасности, изоляция улучшает **модульность** и **структуру** прикладного кода, т. к. алгоритмы управления четко отделяются от задач представления и отображения данных.

На распечатке 12 приведен краткий список основных параметров описания мнемосхем в файлах **\*.CRC**.

### Распечатка 12. Краткий список параметров описания мнемосхем.

#### Конфигурация мнемосхемы в файле **\*.CRC**:

<b>[Circuit]</b>	Секция описания общих параметров мнемосхемы.
<b>GeneralMap = f</b>	Обязательная ссылка на *.BMP файл основного изображения f.
<b>Name = s</b>	Имя окна мнемосхемы s. По умолчанию – имя секции WindowName.
<b>Hint = s</b>	Строка «подсказки» s, заданная в URL кодировке. По умолчанию нет.
<b>StartupScript = s</b>	Имя стартовой секции s интерпретатора Painter. По умолчанию нет.
<b>ToolBarHeight = h</b>	Высота h панели инструментов или 0 если её нет. По умолчанию 0.
<b>[ConfigFileList]</b>	Включение стандартной конфигурации Painter, где заданы обычные значения по умолчанию.
<b>ConfigFile = ~~\Resource\DaqSite\Default\Painter.crc</b>	
<b>[Circuit.StartupScript]</b>	Стартовая секция для интерпретатора Painter.
<b>... = ...</b>	Здесь задаются начальные значения всех используемых переменных.
<b>[SensorList]</b>	Секция для объявления списка сенсоров мнемосхемы.
<b>Sensor = SensorName</b>	Объявление (создание) сенсора по имени SensorName.
<b>[SensorName]</b>	Секция описания сенсора по имени SensorName.
<b>Pos = x, y</b>	Обязательные координаты левого верхнего угла сенсора.
<b>Name = s</b>	Имя сенсора s. По умолчанию это SensorName.
<b>Tag = n</b>	Начальное значение n тега изображения. По умолчанию 0.
<b>Tag#n = t, f, s</b>	Обязательная таблица N изображений для фона сенсора, n=1 ... N. t – значение тега, при котором будет активен этот фон. f – ссылка на *.BMP файл изображения фона сенсора. s – необязательная строка надписи на фоне сенсора.
<b>LED = w, d, v, f, s</b>	Описание формата поля вывода текста (light emitted display). w – ширина поля надписи, 0=«нет надписи». По умолчанию 0. d – число цифр для вещественного числа после запятой. v – начальное значение надписи. По умолчанию 0. f – формат отображения. По умолчанию * (не задан). Формат имеет вид как в функции printf(), например %7.3f. s – описание шрифта надписи. Например: Name:PT_Mono\Size:10.
<b>Hint = s</b>	Строка «подсказки» s, в URL кодировке. По умолчанию нет.
<b>ToolBarKey = k</b>	Флаг (k=0/1) помещения сенсора на панель ToolBar. По умолчанию 0.
<b>ToolBarOpaque = o</b>	Флаг (o=0/1) непрозрачности фона сенсора ToolBar. По умолчанию 0.
<b>ToolBarBorder = b</b>	Флаг (b=0/1) наличия рамки сенсора ToolBar. По умолчанию 1.
<b>ToolBarSpace = s</b>	Промежуток (пикселей) вокруг сенсора ToolBar. По умолчанию 2.
<b>TagEval(v) = s</b>	Формула s расчета тега изображения. По умолчанию нет.
<b>LedEval(v) = s</b>	Формула s расчета значения цифрового поля LED. По умолчанию нет.
<b>Painter(v) = s</b>	Добавляет строку s в сценарий рисования сенсора с помощью интерпретатора Painter. По умолчанию нет.

Поскольку мнемосхемы берут данные для отображения из тегов и кривых из хранилища данных, а события отправляют для обработки устройствам **DAQ** системы, то помимо описания мнемосхемы необходима процедура **подключения**, которая выполняет привязку (**link**) сенсоров к источникам данных и к устройствам. Эта привязка делается с помощью конструкций **Link sensor...with tag...device...** и **Link sensor...with curve...device...**. Кроме того, окна мнемосхем нужно включить в общую конфигурацию **DAQ** системы, что делается в секции **[Windows]**. На распечатке 13 приведен список параметров подключения мнемосхем в файлах **\*.CFG**.

*Распечатка 13. Краткий список параметров подключения мнемосхем.*

**Подключение мнемосхемы в файле \*.CFG:**

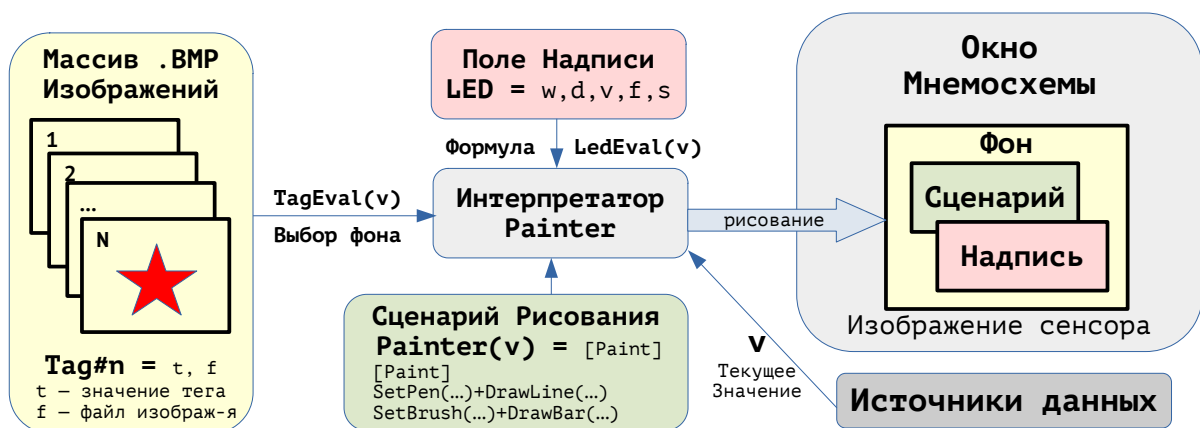
```
[Windows]
WindowName = Circuit_Window
[WindowName]
Name = s
Circuit = f
Link sensor s with tag t device &d
Link sensor s with curve c device &d
[]
```

В секции описания окон:  
Объявить окно мнемосхемы по имени WindowName.  
В секции мнемосхемы по имени WindowName:  
Задать имя окна. По умолчанию WindowName.  
Задать имя \*.CFG файла f для мнемосхемы.  
Связать сенсор s с тегом t и устройством &d.  
Связать сенсор s с кривой c и устройством &d.

Подробное описание мнемосхем дается в отдельном [документе](#). Здесь же рассмотрено только применение **DaqScript** в мнемосхемах.

## Интерпретатор Painter

Каждое окно мнемосхемы содержит в себе связанный с ним объект **DaqScript**, имеющий дополнительный набор переменных, констант функций, и команд указанных в распечатке 14. Этот объект служит для рисования графических изображений на сенсоре, и потому называется **Painter** (художник).



*Рисунок 7. Принципиальная схема процедуры рисования сенсоров.*

Принципиальная схема процедуры рисования сенсоров показана на рисунке 7. Изображение сенсора в окне мнемосхемы формируется из изображения **фона** на который накладывается **сценарий** и **надпись**, вид которых определяется текущим значением (**v**) из источника данных.

Изображение фона сенсора формируется путем выбора элемента из таблицы файлов **\*.BMP**, заданной выражениями **Tag#n = t, f**, где **n=1...N** – порядковый номер элемента таблицы, **t** – значение тега изображения, **f** – имя файла изображения **\*.BMP**, выбираемого при этом значении.



Выбор изображения **фона** из таблицы осуществляется при помощи тега изображения, вычисляемого по формуле **TagEval(v)**. Если эта формула не задана, берется значение тега данных **v**. Вычисленное значение тега изображения округляется до целого. Если элемента с таким значением тега в таблице нет, берется начальное значение тега, заданное в параметре **Tag** описания сенсора.

Затем сверху на фон накладывается сценарий рисования **Painter(v)**. Этот сценарий (если он задан) рисует на фоновом изображении различные геометрические фигуры (линии, стрелки, прямоугольники, многоугольники, эллипсы) с помощью вызова функций рисования, например, **DrawLine**, **DrawBar**, **DrawEllipse** и т.д.

Затем на фоновое изображение и сценарий сверху накладывается **надпись**, значение которой вычисляется по формуле **LedEval(v)**. Если эта формула не задана, берется значение тега данных **v**. Вычисленное значение форматируется в соответствии с параметрами, указанными в списке параметров **LED=w,d,v,f,s**, где **w** – ширина поля, **d** – число цифр после запятой, **v** – начальное значение, **f** – формат, **s** – строка описания шрифта для надписи. В частности, при нулевом значении ширины поля **w** надпись не рисуется. Это значение по умолчанию, используемое для сенсоров, содержащих изображение без надписи.

Для формирования изображения сенсора строго обязательным является только фон, а сценарий и надпись могут отсутствовать. Поэтому в описании сенсора обязаны присутствовать: параметр **Pos**, определяющий положение сенсора в окне, и непустая таблица **Tag#n**.

Строка (**v**) в названии формул **TagEval(v)**, **LedEval(v)**, **Painter(v)** напоминает о том, что вычисляемые формулы зависят от значения (**v** от слова value) источника данных, т. е. подключенного тега или кривой.

Рассмотрим формулы **TagEval(v)**, **LedEval(v)**, и сценарий **Painter(v)** более подробно.

### Формула TagEval(v)

Формула **TagEval(v)** служит для вычисления **тега изображения**, т. е. текущего номера изображения фона, рисуемого на сенсоре и взятого из таблицы тегов, описанных в параметрах **Tag#n**. Переменная **v** (от value) в момент вызова содержит значение присоединенного к сенсору источника данных (тега или кривой). Если формула не указана, используется значение **v** источника данных. Например:

[SensorList]	В списке сенсоров:
Sensor = STATUS.BIT0	Объявить сенсор
[STATUS.BIT0]	В секции сенсора:
Pos = 550, 130	Задать его положение
TagEval(v) = isbit(v,0)	Задать формулу тега
Tag#1 = 0, ..\Bitmaps\ledbmp_3_10_4_silver_ptmono.bmp	Картинка для тега=0
Tag#2 = 1, ..\Bitmaps\ledbmp_3_10_4_lime_ptmono.bmp	Картинка для тега=1
Hint = STATUS.BIT0: Нулевой бит статусного регистра	Задать текст подсказки
[ ]	

Здесь сенсор рисует серым/зеленым фоном состояние **0**-го бита статуса, т. к. формула **TagEval(v)=isbit(v,0)** выделяет **0**-й бит данных.

### Формула LedEval(v)

Формула **LedEval(v)** служит для вычисления числового поля **LED** (от light emission display), отображаемого в текстовом виде на сенсоре в указанном в конфигурации формате. Переменная **v** (от value) в момент вызова содержит значение присоединенного к сенсору источника данных (тега или кривой). Если формула не указана, отображается значение **v** источника данных. Например:

[SensorList]	В списке сенсоров:
Sensor = NET.ADDR	Объявить сенсор
[NET.ADDR]	В секции сенсора:
Pos = 450, 70	Задать его положение
LedEval(v) = bitand(v,255)	Задать формулу надписи
LED = 3, 0, 0, %3.0f, Name:PT_Mono\Size:10	Задать шрифт надписи
Tag#1 = 0, ..\Bitmaps\ledbmp_4_10_4_silver_ptmono.bmp	Задать фон сенсора
Hint = Сетевой адрес устройства (младший байт адреса)	Задать текст подсказки
[ ]	

В этом примере сенсор отображает младший байт адреса, т. к. выражение `LedEval(v)=bitand(v,255)` выделяет 8 младших бит данных.

### Сценарий Painter(v)

Сценарий **Painter(v)** позволяет (с помощью графических функций, показанных в распечатке 14 и на рисунке 8) рисовать на фоновом изображении сенсора разные геометрические фигуры – точки, линии, стрелки, прямоугольники, многоугольники, эллипсы, сегменты, дуги и хорды эллипсов, а также текст с разными атрибутами.

#### Распечатка 14. Краткий список функций интерпретатора **Painter**.

##### Список дополнительных переменных:

v	Переменная v (value) содержит значение источника данных (тега или кривой) в момент вызова процедуры рисования.
---	--

##### Список дополнительных функций:

SensorWidth()	Узнать ширину сенсора в пикселях (px).
SensorHeight()	Узнать высоту сенсора в пикселях (px).
LinkedValue()	Узнать значение источника данных (тега или кривой).
LinkedCurve()	Узнать ссылку подключенной кривой источника данных.
LinkedTag()	Узнать ссылку подключенного тега источника данных.
Bookmark()	Узнать текущий тег (номер) фонового изображения.
LedWidth()	Узнать ширину (в символах) поля надписи LED.
LedDigit()	Узнать число знаков после запятой в поле надписи LED.
MoveLed(dx,dy)	Сдвинуть надпись LED на (dx,dy) единиц px.
SetCursor(x,y)	Установить позицию текстового курсора для @print.
SetPen(pc,ps,pm,pw)	Задать для линий цвет pc, стиль ps, режим pm, толщину pw.
SetBrush(bc,bs)	Задать для заливки цвет bc и стиль bs.
DrawPoint(x,y,c,s)	Рисовать точку в координатах x,y, цветом c и стилем s.
DrawLine(x1,y1,x2,y2)	Рисовать линию из точки x1,y1 в x2,y2 текущим стилем Pen.
DrawArrow(x1,y1,x2,y2,d1,d2)	Рисовать линию со стрелкой с длиной/шириной d1,d2.
DrawRect(x1,y1,x2,y2)	Рисовать прямоугольник с текущим стилем Pen и Brush.
DrawBox(x1,y1,x2,y2)	Рисовать прямоугольник с текущим стилем Pen и Brush.
DrawBar(x1,y1,x2,y2)	Рисовать прямоугольник с текущим стилем Pen и Brush.
DrawRoundRect(x1,y1,x2,y2,rx,ry)	Рисовать прямоугольник со скругленными углами rx,ry.
DrawRoundBox(x1,y1,x2,y2,rx,ry)	Рисовать прямоугольник со скругленными углами rx,ry.
DrawRoundBar(x1,y1,x2,y2,rx,ry)	Рисовать прямоугольник со скругленными углами rx,ry.
DrawEllipse(x1,y1,x2,y2)	Рисовать эллипс, вписанный в заданный прямоугольник.
DrawPie(x1,y1,x2,y2,x3,y3,x4,y4)	Рисовать сектор эллипса, см. рисунок 8.
DrawArc(x1,y1,x2,y2,x3,y3,x4,y4)	Рисовать дугу эллипса, см. рисунок 8.
DrawChord(x1,y1,x2,y2,x3,y3,x4,y4)	Рисовать хорду эллипса, см. рисунок 8.
DrawPoly(x,y,pl)	Рисовать полигональную фигуру (путем нескольких вызовов). x,y – координаты (очередной) точки полигональной фигуры, pl – операция [plNone, plClear, plAddPoint, plPolyLine, plPolygon, plPolyBezier].
SetSimulation(m)	Задать режим m=(0/1)=(рисование/тест-симуляция).
SetDebugFlags(f)	Задать флаги DebugFlags для отладочного вывода.
@print msg	Рисовать текст (msg) в позиции курсора, см. SetCursor(...).
@textwidth msg	Узнать ширину текста (msg) в пикселях.
@textheight msg	Узнать высоту текста (msg) в пикселях.
@font n v	Задать свойству n значение v для шрифта команды @print. n – имя=(name/charset/color/size/height/pitch/style). v – значение, например: @font name PT Mono

### Памятка функций Painter(v)

#### Рисование линий и стрелок

```

drawLine(x1,y1,x2,y2)
drawArrow(x1,y1,x2,y2,d1,d2)

```

#### Вывод текста

```

x,y v=314.159
@textwidth ...
@textheight ...
@font ...
setCursor(x,y)
@print v=%7.3f%v

```

#### Прямоугольники и эллипсы

```

drawBar(x1,y1,x2,y2)
drawRoundBar(x1,x2,y1,y2,rx,ry)
drawEllipse(x1,x2,y1,y2)
drawPie(x1,x2,y1,y2,x3,y3,x4,y4)
drawArc(x1,y1,x2,y2,x3,y3,x4,y4)

```

#### Основные соглашения о передаче параметров

v - текущее значение привязанной кривой/тега

sensorWidth() sensorHeight() - размеры сенсора

#### Стили рисования линий и заполнения фигур

```

setPen(color, style, mode, width)
setBrush(color, style)
color = clBlack..clWhite (RGB)

```

Fuchsia	Aqua	Yellow	White
Purple	Teal	Olive	Silver
Blue	Lime	Red	Gray
Navy	Green	Maroon	Black

```

style = {psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear, psInsideFrame}
mode = {pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy, pmMergePenNot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge, pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor}
style = {bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross}

```

Рисунок 8. Памятка по графическим функциям Painter.

При рисовании также используются константы, наиболее важные из которых перечислены на распечатке 15. Полный набор констант весьма обширен и доступен в каталоге **Resource\DaqSite\Painter**. Эти константы нужны для указания цвета и стиля линий (**SetPen**) и заполнения фигур (**SetBrush**).

Распечатка 15. Краткий список констант интерпретатора *Painter*.

#### Список цветовых констант (cl=color):

clBlack	clMaroon	clGreen	clOlive	clNavy	clPurple	clTeal	clGray
clWhite	clRed	clLime	clYellow	clBlue	clFuchsia	clAqua	clSilver

#### Список констант стиля для линий (ps=penstyle, pm=penmode):

```

SetPen(**, ps, **, **) psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear, psInsideFrame
SetPen(**, **, pm, **) pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy, pmMergePenNot,
pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge, pmNotMerge,
pmMask, pmNotMask, pmXor, pmNotXor

```

#### Список констант стиля заливки (bs=barstyle):

```

SetBrush(**, bs) bsSolid, bsClear, bsHorizontal, bsVertical,
bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross

```

#### Список констант для параметров шрифтов команды @print:

```

@font charset ANSI_CHARSET, DEFAULT_CHARSET, SYMBOL_CHARSET, MAC_CHARSET,
SHIFTJIS_CHARSET, HANGEUL_CHARSET, JOHAB_CHARSET, GB2312_CHARSET,
CHINESEBIG5_CHARSET, GREEK_CHARSET, TURKISH_CHARSET, HEBREW_CHARSET,
ARABIC_CHARSET, BALTIC_CHARSET, RUSSIAN_CHARSET, THAI_CHARSET,
EASTEUROPE_CHARSET, OEM_CHARSET
@font style fsBold, fsItalic, fsUnderline, fsStrikeOut
@font pitch fpDefault, fpVariable, fpFixed

```

При рисовании фигур используется общепринятая концепция **пера** (**pen**), которое задает цвет и стиль линий, и **кисти** (**brush**), которая задает цвет и стиль заполнения фигур. То есть для рисования сначала задается перо и кисть, а затем выполняется рисование нужной фигуры. При этом с помощью арифметических операторов вызовы функций можно объединять в одно выражение, например:

```

setPen(clRed, psSolid, pmCopy, 1) + drawLine(10, 20, 300, 400)
setPen(clRed, psSolid, pmCopy, 2) + setBrush(clAqua, bsSolid) + drawBar(10, 10, 100, 150)

```

Здесь сначала задается перо (**setPen**) и кисть (**setBrush**), а затем вызывается рисование (**drawLine**, **drawBar**, ...). Такая форма вызова не только компактнее записана, но и быстрее выполняется.

## Применение DaqScript в расчетах

Кроме описанных выше применений в окне [Главной Консоли](#) и [DAQ системе](#), **DaqScript** применяется в расчетных инструментах, которые кратко описаны здесь.

### Инструмент Калькулятор

Меню «**Инструменты/Калькулятор**» вызывает окно **Калькулятор**, вид которого показан на рисунке 9. Он служит для проведения простых расчетов с помощью встроенного экземпляра объекта **DaqScript** и выгодно отличается от обычного калькулятора **Windows** наличием переменных, скобок и большим числом доступных функций. Экземпляр **DaqScript**, встроенный в **Калькулятор**, поддерживает основной набор [констант](#), [функций](#), [команд](#) и не имеет дополнительных.

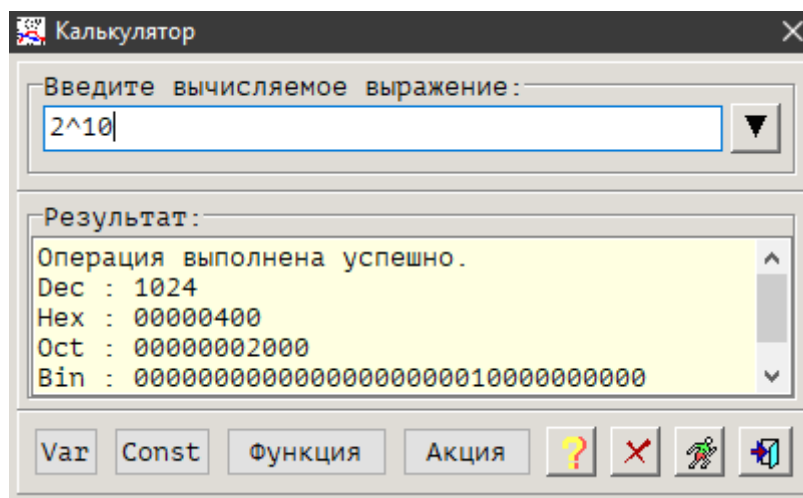


Рисунок 9. Внешний вид окна **Калькулятор**.

Для удобства результат расчетов для целых чисел выводится в нескольких популярных системах счисления: десятичной (**decimal**), шестнадцатеричной (**hexadecimal**), восьмеричной (**octal**) и двоичной (**binary**). Это бывает полезно для анализа цифровых систем, где эти системы счисления часто используются.

Кнопки **Var**, **Const**, **Функция**, **Акция** позволяют вывести в окно **Результат** текущий список переменных, констант, функций и команд. Это позволяет использовать **Калькулятор** как справочник по языку **DaqScript**, а также, например, проверять работу отдельных выражений при разработке сценариев на языке **DaqScript**.

Инструмент **Калькулятор** имеет чисто прикладное назначение, позволяя пользователю проводить простые расчеты в удобной форме.

### Инструмент Plot2D

Меню «**Инструменты/Plot2D**» вызывает «**Диалог построения графика функции**», вид которого показан на рисунке 10. Он позволяет строить графики функций или параметрически заданных кривых. В диалоге задаются: число точек кривой, начало и конец интервала независимой переменной **t**, а также текст сценария **DaqScript**, вычисляющего абсциссу **x(t)** и ординату **y(t)** кривой как функцию **t**. Результатом расчетов является новое окно с искомой кривой. Объект **DaqScript**, встроенный в **Plot2D**, имеет только основной набор функций.

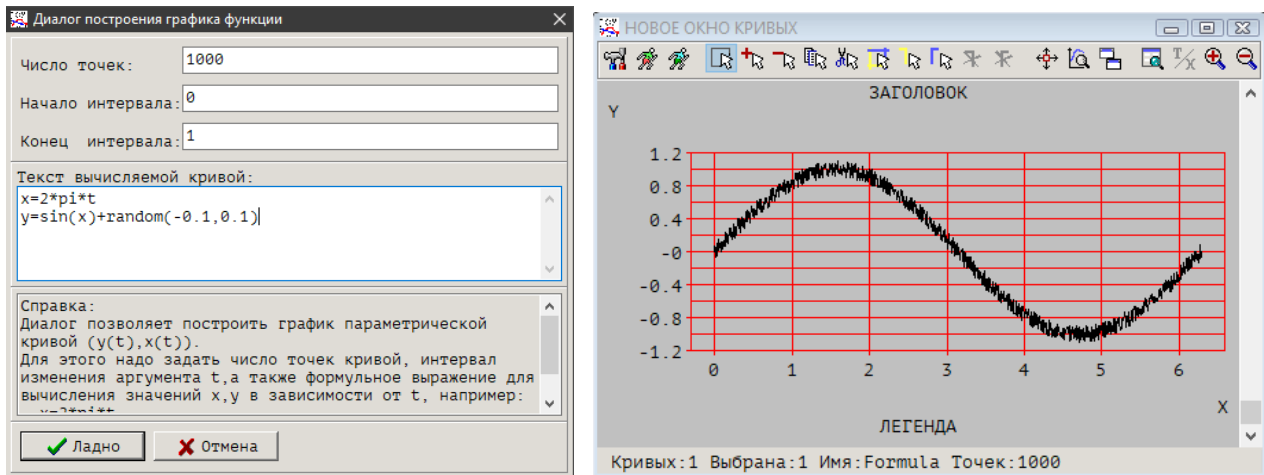


Рисунок 10. Внешний вид диалога **Plot2D** и результат его работы.

Инструмент **Plot2D** играет роль вспомогательной утилиты для генерации расчетных данных (кривых) и для проверки работы выражений языка **DaqScript** перед их использованием в сценариях.

### Инструмент Plot3D

Меню «**Инструменты/Plot3D**» вызывает «**Диалог построения графика поверхности  $z(x,y)$** », вид которого показан на рисунке 11. Он может строить графики функций  $z(x,y)$  от двух аргументов. В диалоге задаются: число точек вдоль  $x$  и  $y$ , начало и конец интервалов независимых переменных  $x$  и  $y$ , а также текст сценария **DaqScript**, вычисляющего  $z$  как функцию  $(x,y)$ . Результатом расчетов является новое окно с искомой поверхностью. Объект **DaqScript**, встроенный в **Plot3D**, имеет только основной набор функций.

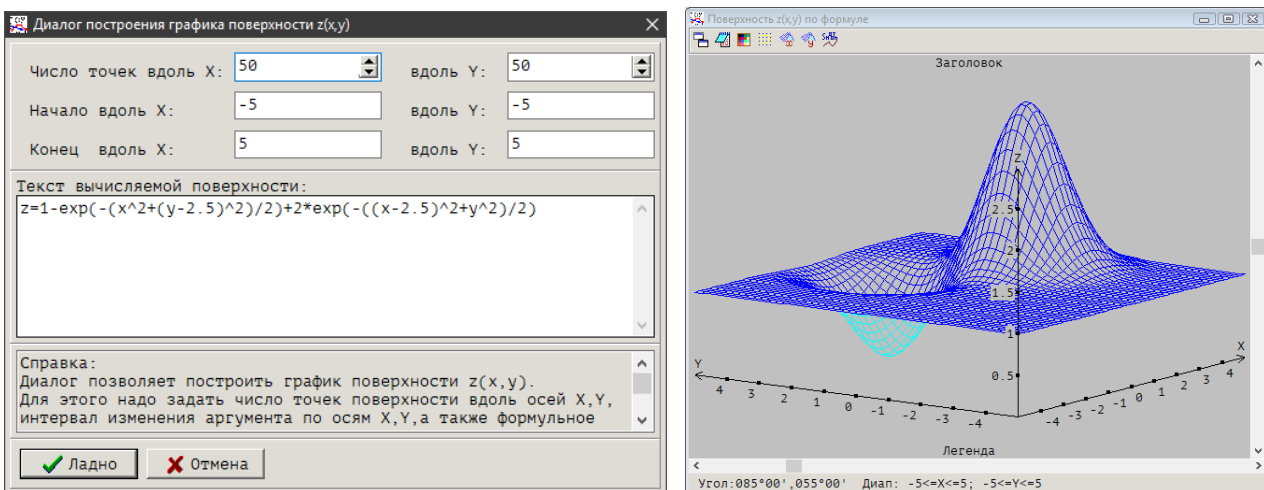


Рисунок 11. Внешний вид диалога **Plot3D** и результат его работы.

Инструмент **Plot3D** играет роль вспомогательной утилиты для генерации расчетных данных (поверхностей) и для проверки работы выражений языка **DaqScript** перед их использованием в сценариях.

### Макросы анализа данных в окнах с кривыми

Окна с кривыми в пакете **CRW-DAQ** имеют команду «**Кривая/Макрос анализа данных**», как показано на рисунке 12, которая позволяет обрабатывать исходные кривые, получая результат в новых кривых.



**Распечатка 16. Дополнительный набор функций макросов DaqScript.**

<code>iocount(a)</code>	Узнать число кривых в окне а источника( $a>0$ )/приемника( $a<0$ )/буфера( $a=0$ ).
<code>ioselected(a)</code>	Узнать ссылку выделенной кривой в окне а источника( $a>0$ )/приемника( $a<0$ ).
<code>ioselect(a)</code>	Выделить кривую а в окне источника( $a>0$ )/приемника( $a<0$ ).
<code>ioroix(a)</code>	Узнать абсциссу x ROI-маркера окна а.
<code>ioroiy(a)</code>	Узнать ординату y ROI-маркера окна а.
<code>iosetroi(a,b,c)</code>	Задать координаты ROI-маркера окна а равными (b,c).
<code>iox(a,b)</code>	Узнать абсциссу x точки номер b кривой а в окне источника( $a>0$ )/приемника( $a<0$ )/буфера( $a=0$ ).
<code>ioy(a,b)</code>	Узнать ординату y точки номер b кривой а в окне источника( $a>0$ )/приемника( $a<0$ )/буфера( $a=0$ ).
<code>iolen(a)</code>	Узнать длину кривой а в окне источника( $a>0$ )/приемника( $a<0$ ).
<code>iosetlen(a)</code>	Задать длину b кривой а в окне источника( $a>0$ )/приемника( $a<0$ )/буфера( $a=0$ ).
<code>ioadd(a,b,c)</code>	Добавить точку (b,c) к кривой а в окне источника( $a>0$ )/приемника( $a<0$ )/буфера( $a=0$ ).
<code>ioput(a,b,c,d)</code>	Задать точку (c,d) номер b в кривой а в окне источника( $a>0$ )/приемника( $a<0$ )/буфера( $a=0$ ).
<code>iocreate(a,b)</code>	Создать в окне приемника кривую цвета а, стиля b и вернуть ее ссылку $<0$ .
<code>ioclone(a)</code>	Создать копию кривой а из окна источника( $a>0$ )/приемника( $a<0$ )/буфера( $a=0$ ) и вернуть ее ссылку $<0$ .
<code>iokill(a)</code>	Уничтожить кривую а из окна источника( $a>0$ )/приемника( $a<0$ )/буфера( $a=0$ ).
<code>iosort(a,b,c,d,e)</code>	Сортировка кривой а, возвращает ссылку кривой $<0$ , b=Flags, c=AbsEps, d=RelEps, e=method.
<code>iosorted(a,b)</code>	Проверяет, отсортирована ли кривая а, а при $b<>0$ проверяет также, нет ли дубликатов по абсциссе X.
<code>iofindindex(a,b)</code>	Узнать номер ближайшей точки кривой а в точке b. Кривая должна быть отсортирована по абсциссе X.
<code>iointerpol(a,b)</code>	Линейная интерполяция кривой а в точке b. Кривая должна быть отсортирована по абсциссе X.
<code>iosmooth(a,b,c,d,e,f)</code>	Сглаживание кривой а в точке b с окном с степени d и ядром $(1-e)^f$ . Кривая должна быть отсортирована по X.
<code>iomediana(a,b,c)</code>	Найти индекс медианы массива ординат у кривой а в интервале индексов (b..c).

**Примечание:**

- 1) Кривые идентифицируются целочисленными ссылками (a). Модуль ссылки равен порядковому номеру кривой в окне.
- 2) Кривые из окна источника имеют положительные ссылки ( $a>0$ ).
- 3) Кривые из окна приемника имеют отрицательные ссылки ( $a<0$ ).
- 4) Кривая в буфере обмена имеет нулевую ссылку ( $a=0$ ).
- 5) Окна источника/приемника идентифицируются знаком +/- ссылки (a).

**Распечатка 17. Дополнительный набор команд макросов DaqScript.**

<code>@caption s</code>	Задаёт заголовок s для окна – приемника.
<code>@title s</code>	Задаёт «титул» s для окна – приемника в верхней части окна.
<code>@legend s</code>	Задаёт «легенду» s для окна – приемника в нижней части окна.
<code>@error s</code>	Вывод сообщения s об ошибке и остановка выполнения скрипта.
<code>@yesno s</code>	Вывод модального окна диалога с запросом на подтверждение s.
<code>@warning s</code>	Вывод модального окна диалога с предупреждающим сообщением s.
<code>@information s</code>	Вывод модального окна диалога с информационным сообщением s.

Макросы выполняют обработку данных с помощью встроенного объекта **DaqScript**, имеющего дополнительный набор функций и команд, приведенный на распечатках 16 и 17. Макросы предполагают, что есть окно – **источник** с исходными кривыми, имеющими положительные ссылки (порядковые номера), и новое окно – **приемник** с кривыми результатов расчетов, имеющими отрицательные ссылки (порядковые номера с минусом). Ссылка **0** означает кривую в **буфере обмена**.

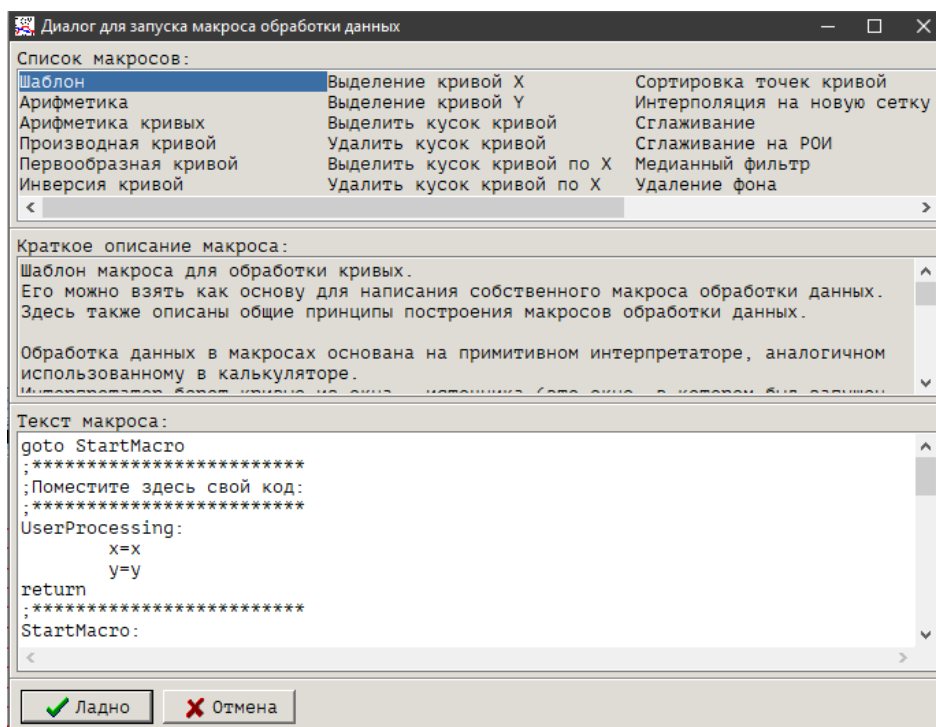


Рисунок 12. Внешний вид диалога вызова макроса обработки данных.

С помощью указанного набора функций и команд макросы выполняют обработку исходных кривых и (возможно) буфера обмена, создавая новые кривые в окне приемника с результатами расчетов.

В дистрибутив пакета **CRW-DAQ** входит библиотека (из более 15) готовых к работе макросов для решения типичных задач обработки данных, расположенная в каталоге **Resource\Macro**. Также есть шаблон, позволяющий быстро создать новый макрос обработки данных, если нужного макроса в библиотеке не нашлось.

### Утилиты анализа данных в окнах с кривыми

Окна с кривыми в пакете **CRW-DAQ** имеют команду «Кривая/Утилита анализа данных», как показано на рисунке 13, которая позволяет обрабатывать исходные кривые, получая результат в новых кривых, аналогично макросам.

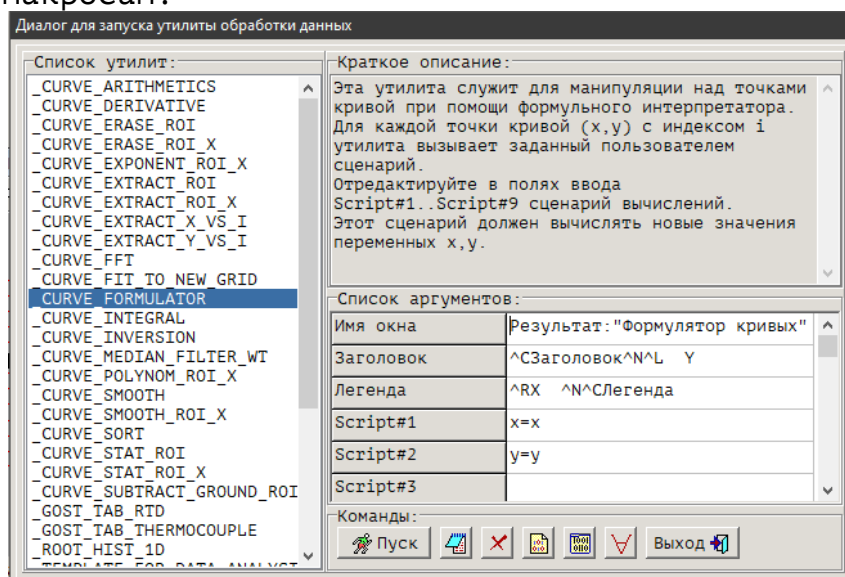


Рисунок 13. Внешний вид диалога вызова утилиты обработки данных.

Утилиты анализа данных создаются на языке **Object Pascal** (версии 13.0), компилятор которого интегрирован в пакет **CRW-DAQ**. Для создания утилит обработки используется специальная библиотека классов **CrwApi**, расположенная в **Resource\Dcc32\SYSLIB\\_CRWAPI.PAS**. В составе этой библиотеки есть класс **TScriptInterpreter**, который фактически является «оберткой» для интерпретатора **DaqScript**. Он позволяет (в рамках вызываемой утилиты) создавать экземпляры объектов **DaqScript** и выполнять с их помощью нужные действия по интерпретации выражений в командном или пакетном режиме. Таким образом пользователю предоставляется возможность вести нужные ему расчеты по формулам, вводимым в полях ввода при вызове утилиты.

В каталоге **Resource\DataAnalysisPlugins** пакета **CRW-DAQ** есть библиотека (из более 25) готовых утилит анализа данных, а также имеются шаблоны для создания новых утилит обработки данных, если библиотечных утилит не хватает. Поскольку все утилиты доступны в исходных кодах, а интерфейсный файл библиотеки **\_CRWAPI.PAS** хорошо документирован (в комментариях), то здесь достаточно указать на широкие возможности и высокую скорость обработки данных, которую предоставляют утилиты анализа данных. Всю нужную информацию для их создания можно найти в указанных выше каталогах и файлах.

## Заключение

Интерпретатор **DaqScript** является важнейшим компонентом пакета **CRW-DAQ**, выполняющим целый ряд критически значимых функций.

Он обеспечивает возможность интерпретацию данных, вводимых пользователем в полях ввода в виде формул или арифметических выражений или поступающих **online** из каналов связи.

Он обеспечивает возможность проведения разных математических расчетов в интерактивном режиме.

Он обеспечивает возможность генерации данных (кривых) по формулам и построения графиков функций в интерактивном режиме.

Он обеспечивает работу окна Главной Консоли, которая выполняет обработку сообщений и команд, поступающих в режиме **online**.

Он обеспечивает обработку данных в окнах кривых с помощью макросов и утилит анализа данных.

Он обеспечивает возможность быстрой разработки надежных и гибких прикладных программ и сценариев для сбора и обработки данных, а также симуляторов в **DAQ** системе.

Практика показала высокую надежность и эффективность работы интерпретатора **DaqScript**.

Ввиду всего перечисленного, освоение языка **DaqScript** является необходимой частью подготовки разработчика прикладного **ПО** в пакете **CRW-DAQ**.

Остается пожелать читателям успеха в освоении **DaqScript**.

**Спасибо за внимание.**

## Список сокращений

<b>DAQ</b>	Data AcQuisition – сбор данных. Общепринятый термин.
<b>CRW</b>	CuRves in Windows – окна с кривыми. Форма представления данных.
<b>CRW-DAQ</b>	CuRves in Windows for Data AcQuisition – название пакета [1].
<b>DAQ система</b>	Система разработки и исполнения для задач сбора данных и управления.
<b>СКУ, АСУ, АСКУ</b>	Автоматизированная Система Контроля и Управления.
<b>ОС, OS</b>	Операционная система. Общепринятый термин.
<b>ПО</b>	Программное обеспечение. Общепринятый термин.
<b>ms, мс</b>	Millisecond. Миллисекунда. Единица измерения времени.
<b>px</b>	Pixel. Пиксель. Единица измерения координат растровых изображений.
<b>pt</b>	Point. Пункт. Единица измерения размера шрифтов.
<b>FIFO</b>	First In, First Out. Очередь типа «первым вошел – первым вышел».
<b>API</b>	Application Program Interface. Прикладной программный интерфейс.
<b>GUI</b>	Graphical User Interface. Графический интерфейс пользователя.
<b><u>DaqConfig</u></b>	Интерпретатор языка описания конфигураций в пакете CRW-DAQ.
<b><u>DaqScript</u></b>	Интерпретатора командного языка в пакете CRW-DAQ.
<b><u>DaqPascal</u></b>	Компилятор языка прикладного программирования в пакете CRW-DAQ.
<b>online</b>	«На линии» – режим обработки событий при подключении каналов связи.
<b>runtime</b>	«Исполнительная система» – код программы во время выполнения задач.
<b>realtime</b>	«Реальное время» – задачи, критично зависящие от времени отклика <b>ПО</b> .
<b>интерактивный</b>	Режим работы программы при взаимодействии с пользователем <b>online</b> .
<b>NaN</b>	«Not a Number». Значение «не число» для обозначения ошибок.
<b>INF</b>	«Infinity». Значение «бесконечность» для обозначения ошибок.



## Список источников

1. А.В.Курякин, Ю.И.Виноградов. Программа для автоматизации физических измерений и экспериментальных установок (CRW-DAQ). // Свидетельство РФ об официальной регистрации программы для ЭВМ № 2006612848 от 10.08.2006 г.  
Домашний сайт пакета CRW-DAQ: <http://crw-daq.su>
2. А.В.Курякин. Автоматизация физических экспериментов на тритиевых комплексах исследовательских установок «ТРИТОН», «АКУЛИНА» и «ПРОМЕТЕЙ». Диссертация на соискание степени кандидата физико-математических наук, Саров, 2010.  
Диссертация: <http://crw-daq.su/download/documents/alexey-kuryakin-disser-final.pdf>  
Автореферат: <http://ftp.jinr.ru/dissertation/Kuryakin.pdf>
3. А.В.Курякин, Ю.И.Виноградов. Программное обеспечение автоматизированных измерительных систем в области тритиевых технологий. // ВАНТ, серия «Термоядерный синтез», 2008 г., выпуск 2, с.80-90.
4. Протокол DCON. [https://www.bookasutp.ru/Chapter2\\_10.aspx](https://www.bookasutp.ru/Chapter2_10.aspx)  
[https://www.icpdas.com/download/7000/whatisdconprotocol\\_eng.htm](https://www.icpdas.com/download/7000/whatisdconprotocol_eng.htm)
5. FP-QUI notification tool for Windows.  
<https://sourceforge.net/projects/fp-qui/>  
<https://github.com/bfour/FP-QUI>  
<http://fp-qui.sourceforge.net/>
6. Система «всплывающих» уведомлений FPQUI.  
<https://sourceforge.net/projects/fpqui/>
7. Clara Gaspar. DIM - Distributed Information Management System.  
<https://dim.web.cern.ch/>
8. UTF-8, a transformation format of ISO 10646.  
<https://tools.ietf.org/html/rfc3629>
9. Н.Вирт. «АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ». М.: Мир, 1989, 360 стр.