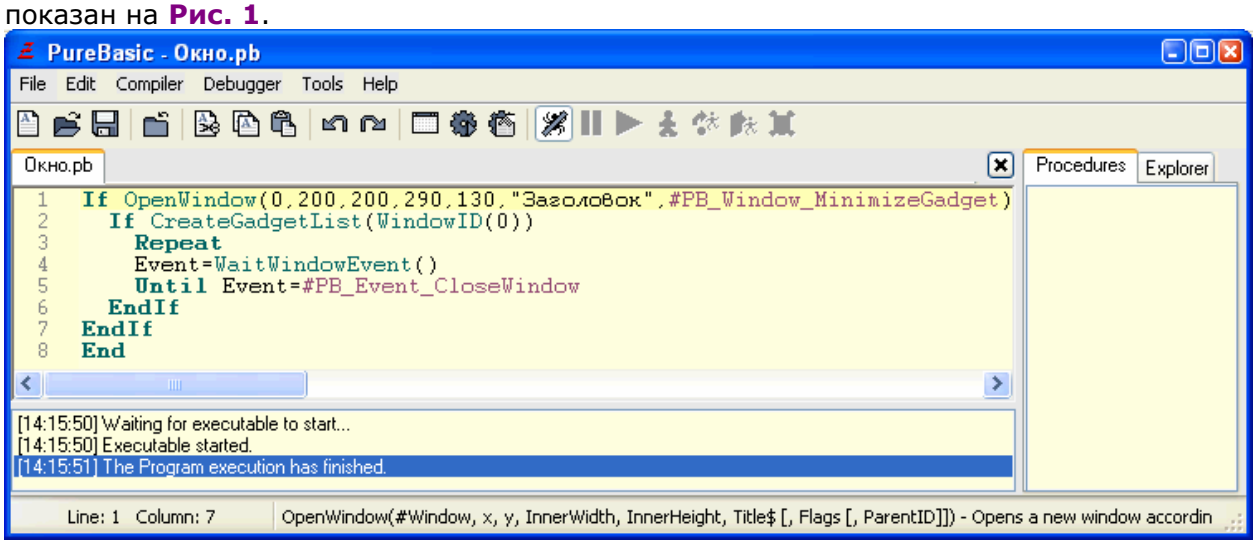
Основы языка PureBasic 4.00

П. Высочанский, г. Рыбница, Приднестровье (Молдавия).

Считается, что составить приличную программу под Windows, можно, лишь изучив один из алгоритмических языков высокого уровня: Delphi, C++, Perl и так далее. Это справедливо для больших проектов, но если предполагается разрабатывать в основном небольшие программы, то можно выбрать язык программирования с относительно простым синтаксисом, например, бейсик. Об одной из разновидностей бейсика – программе "PureBasic" далее пойдёт речь. У этой программы довольно большие возможности, которые можно расширить, используя так называемые API функции или библиотеки с функциями, которые можно найти на страничке http://www.purearea.net/pb/english/userlibs.php. Приложения, будут работать в любой операционной системе семейства Windows. Минимальный размер исполняемого файла получается относительно небольшим – примерно 10Кб.

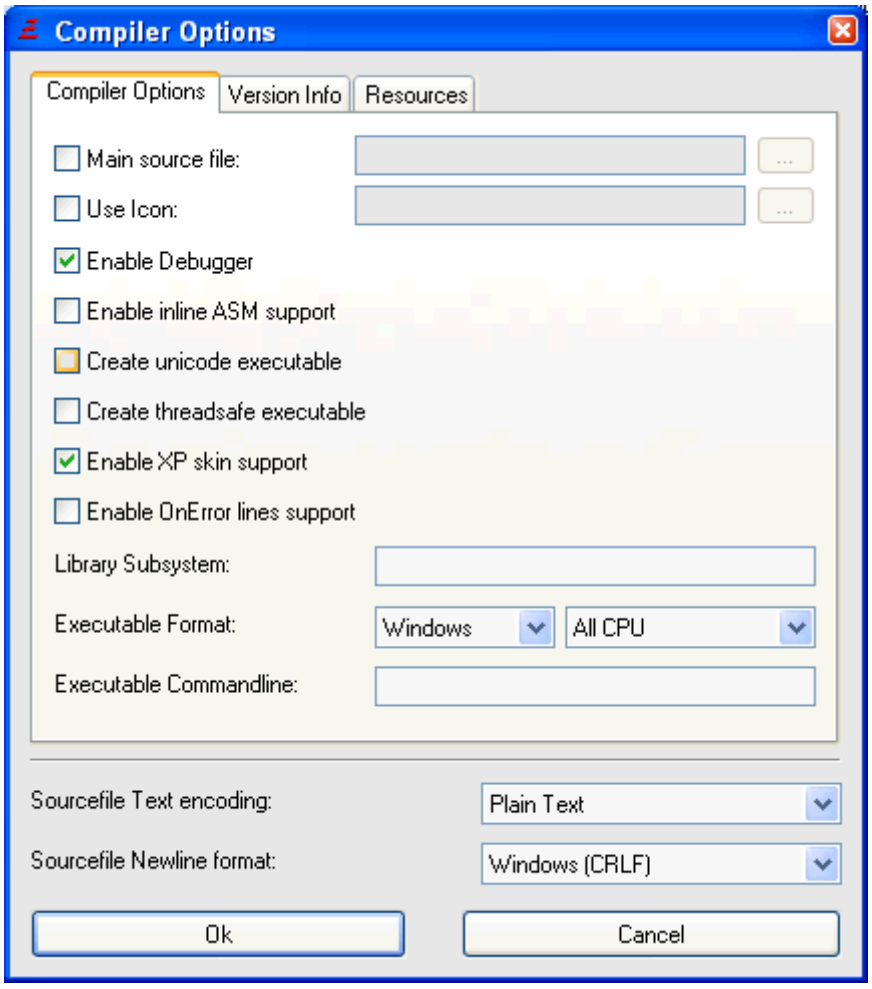
К сожалению, полная версия программы платная, но можно скачать бесплатную демо-версию **[1]**. На момент написания статьи по этой ссылке была доступна версия 4.10, возможно сейчас появилась более свежая версия. Поскольку у демо-версии есть некоторые ограничения (одно из которых, невозможно скомпилировать программу, если исходный текст имеет более 800 строк кода) поэтому не гарантируется работа всех примеров программ, которые будут описаны ниже.

Установка программы никаких особенностей не имеет. Внешний вид главного окна показан на **Рис. 1**.



В его центральной части находится редактор исходных текстов программ. Причём одновременно может быть открыто несколько файлов. Внизу, отображается процесс компиляции, а в правой части окна отображается список процедур либо содержимое дисков в зависимости от того, какая закладка была выбрана. Исходный текст (файлы с расширением "pb") загружается с помощью пункта "**Open**" из меню "**File**". Когда исходный текст загружен, его можно скомпилировать. Для этого в меню "**Compiler**" выберите пункт "**Compile/Run**". Это приведёт к компиляции программы и если в исходном тексте отсутствуют ошибки, программа запуститься. Для того, чтобы создать исполняемый файл, следует в меню "**Compiler**" выбрать пункт "**Create Executable**". При этом откроется окно, в котором следует выбрать место сохранения файла. Перед созданием исполняемого файла можно произвести некоторые настройки, например, выбрать иконку для файла. Для этого в меню "**Compiler**" следует выбрать пункт "**Compiler Options**". Это приведёт к открытию окна, показанного на **Рис. 2**. Если поставить галочку в пункте "Use Icon", будет активирована строка напротив этого пункта, в которой можно указать путь к требуемой иконке, которая будет у скомпилированной программы. Если нужно чтобы в созданном приложении все элементы управления программой (в языке "PureBasic" они называются гаджетами) отображались в стиле Windows XP, следует поставить галочку в пункте "Enable XP skin support". Перейдя на закладку "Version Info", можно указать версию файла, его название и т.д. Эта информация носит справочный характер и необязательна.

При желании, программу "PureBasic" можно русифицировать (русификатор – файл "PB_Rus 4.00.exe" к статье прилагается). Он позволяет полностью русифицировать версию 4.00 и частично 4.10. Если после русификации, русский язык не был установлен текущим, следует открыть окно "Preferences" из меню File, перейти на закладку "Language" и выбрать с помощью выпадающего списка русский язык.



Синтаксис

Начнём изучение синтаксиса с переменных. Переменные это ячейки памяти, в которых может храниться какая-либо информация определяемая типом переменной **Табл. 1**. От типа зависит то, какая информация может храниться в переменной. Например, в переменной типа "String" можно хранить строку текста почти неограниченного размера, а в переменной типа "Double" дробные числа. Если тип переменной не указан, будет установлен по умолчанию тип "Long". Имена переменных могут состоять из букв английского алфавита, цифр и символа "_" (нижний пробел), причём строчные и заглавные буквы не различаются, т. е. **variable** и **VARIABLE** это одна и та же переменная. В имени переменной недопустимы кириллические символы и символы *, %, № и т. д. Имя переменной не должно начинаться с цифры.

Таблица 1

Имя типа.	Способ объявления переменной.	Занимаемая память.	Возможные значения чисел.
Byte	a.b	1 Байт	от -128 до +127
Character	a.c	1 Байт	от 0 до 255
Word	a.w	2 Байта	от -32768 до +32767
Long	a.l	4 Байта	от -2147483648 до +2147483647
Float	a.f	4 Байта	Дробные числа.
Quad	a.q	8 Байт	От –9223372036854775808 до +9223372036854775807
Double	a.d	8 Байт	Дробные числа.
String	a.s	длина строки + 1 байт	Строка текста.

В полной мере всё выше сказанное об имени переменной относится и к константам, в которые можно занести информацию только один раз. Имя константы отличается от имени переменной только наличием в начале имени символа "#". Если в константу данные будут записаны более одного раза, при компиляции программы появится сообщение об ошибке. Тип константы указывать не требуется. Следует отметить, что информация в некоторые константы записываются самим компилятором.

Если требуется хранить в переменных большое количество однотипной информации, в этом случае удобнее использовать массив, который представляет собой некоторое

количество однотипных переменных, объединённых под одним именем. Он объявляется с помощью оператора **Dim**
Пример: **Dim** array.w(10)

array – имя массива, после которого указан тип входящих в него переменных. Если тип массива не указан, входящим в его состав переменным будет присвоен тип "Long". В скобках задан размер массива, означающий количество переменных в массиве. В нашем случае его составе 11 переменных, поскольку нумерация начинается с нуля. Для изменения размера уже созданного массива служит оператор **ReDim**
Пример: **ReDim** array.w(5)

В скобках задан требуемый размер массива.
Для записи информации в массив, в его скобках указываем номер переменной и с помощью оператора присваивания "=" записываем требуемое значение.
Пример: array.w(2) = 5

В переменную с номером 2 массива, будет записано число 5.
При чтении информации из массива, с помощью оператора присваивания копируем текущее содержимое определённой переменной массива.
Пример: x = array.w(4)

В переменную "x" будет записана информация из переменной с номером 4 массива.

Циклы.

В основном циклы служат для выполнения части кода определённое количество раз, до тех пор, пока не будет выполнено определённое условие. В составе языка PureBasic есть несколько операторов, организующих цикл.

Операторы: **For** - **Next**.
Формат:
For <переменная> = <выражение1> **to** <выражение2> [step <шаг>]
Выполняемый код
Next [<Переменная>]

При первом выполнении оператора **For**, <переменной> присваивается значение <выражения1>. Затем, происходит дальнейшее выполнение программного кода, пока не встретится оператор **Next**. Как только это произойдёт, <переменная> увеличивается на величину <шага> (или уменьшается если <шаг> отрицательный) и происходит сравнение <переменной> с <выражением2>. Если они равны, цикл прерывается, в противном случае будет снова выполнен код, расположенный между операторами **For** и **Next**, и так будет продолжаться до тех пор, пока они не станут равны. Квадратные скобки выделяют необязательный параметр. Этих скобок в реальной программе не должно быть, это следует запомнить, поскольку в дальнейшем, этими скобками будут выделяться все необязательные параметры. В нашем случае такими параметрами являются оператор "step" (если он отсутствует, шаг будет равен одному) и <Переменная> после оператора **Next**.
Пример:

```
For y=1 To 5
Debug y
Next y
```

В этом примере в начале цикла, переменной "y" будет присвоено число 1. Далее выполняется оператор **Debug** (отладчик) создающий окно, в котором отображается текущее содержимое переменной. При выполнении оператора **Next** происходит увеличение числа в переменной и сравнение с числом указанным после оператора **To**, в нашем случае с числом 5. Если равенства нет, программный код внутри цикла будет выполнен снова. Так будет происходить до тех пор, пока на условие не будет выполнено. При текущих значениях чисел, код внутри цикла будет выполнен 5 раз, после чего, работа программы завершится.

Операторы: **Repeat** - **Until**
Формат:
Repeat
Выполняемый код
Until <выражение>

Код внутри этого цикла будет выполняться до тех пор, пока <выражение> не станет истинным.
Если требуется бесконечный цикл, то в место оператора **Until** следует использовать оператор **ForEver** без параметров. В операторе **Until** может быть несколько выражений, разделённых операторами **And** (<выражение> и <выражение>) и/или **Or** (<выражение> или <выражение>) Допускается неограниченное число вложенных циклов.

Пример:

```
a=0
Repeat
a=a+1
Debug a
Until a>10
```

Программа зациклится до тех пор, пока число в переменной "a" не станет больше десяти. Это произойдёт на одиннадцатом витке цикла.

Операторы: **While** - **Wend**
Формат:
While <выражение>
Выполняемый код
Wend

В отличие от цикла **Repeat** – **Until**, <выражение> проверяется в начале цикла. В операторе **While** может быть несколько выражений, разделённых операторами **And** (<выражение> и <выражение>) и/или **Or** (<выражение> или <выражение>) Допускается неограниченное число вложенных циклов.
Пример:

```
a=1
While a<>8
Debug a
a=a+1
Wend
```

Программный код внутри этого цикла будет выполнен 7 раз, до тех пор, пока в переменной "a" не окажется число 8, поскольку при этом условие уже не выполняется. В нашем случае, условие будет выполняться во всех случаях, кроме того, когда в переменной будет число 8.

Если требуется досрочно прервать один из вышеуказанных циклов, следует использовать оператор **Break**.
Пример:

```
a=0
Repeat
a=a+1
Debug a
 If a=3
```

```
  Break
 EndIf
Until a>10
```

В этом случае выполнение цикла будет прервано, как только в переменной "a" окажется число 3. Проверку выполняют операторы **If** – **EndIf**.

Операторы ветвления программы

Только в редких случаях программа выполняется последовательно с верху в низ. Обычно в программе присутствуют операторы, изменяющие последовательность выполнения кода в зависимости от условия.

Операторы: **If** - **Else** - **EndIf**
Формат:
If <выражение1>
Выполняемый код при выполнении условия в операторе **If**
[**ElseIf** <выражение2>]
Выполняемый код при невыполнении условия в операторе **If** и выполнении условия в операторе **ElseIf**
[**Else**]
Выполняемый код при невыполнении условия ни в одном из операторов
EndIf

Условный оператор **If** служит для управления порядком выполнения программного кода. Следующее за оператором **If**, <выражение1> играет роль условия, от верности которого зависит порядок выполнения кода. Необязательный оператор **ElseIf** предназначен для проверки <выражения2>, в том случае, если условие в операторе **If** было неверно. В операторах **If** и **ElseIf** можно указать несколько выражений, разделяя их операторами **And** (<выражение> и <выражение>) и **Or** (<выражение> или <выражение>). Необязательный оператор **Else** служит для выполнения части программного кода, если все выражения оказались неверными. В любом случае программный код будет выполнен только после одного из этих операторов. Конец тела оператора **If** определяется оператором **EndIf**.
Пример:

```
For count = 0 To 10
 If count = 5
  Debug "Счётчик равен пяти"
 Else
  Debug count
 EndIf
Next
```

В этом примере организован цикл **For** – **Next**, в котором значение переменной "count" увеличивается от 0 до 10 с шагом 1. Условие в операторе **If** будет выполнено, когда в переменной окажется число 5. В этом случае будет выполнен код после этого оператора, что приведёт к появлению в отладочном окне текстовой надписи. В тех случаях, когда значение переменной отличается от пяти, будет выполнен код после оператора **Else**. Отладочное окно, после выполнения этой программы показано на Рис 3.

Операторы: **Select** - **Case** - **EndSelect**
Формат:
Select <выражение1>
Case <выражение2>
Выполняемый код при совпадении <выражение1> и <выражение2>
[**Case** <выражение3>]
Выполняемый код при совпадении <выражение1> и <выражение3>
[**Default**]
Выполняемый код если не было ни одного совпадения в операторах Case
EndSelect

При выполнении оператора **Select** запоминается <выражение1>, значение которого будет сравниваться с выражениями во всех операторах **Case**. При обнаружении совпадения в одном из операторов **Case**, будет выполнен расположенный после него программный код. Если нет ни одного совпадения, будет выполнен код после оператора **Default**. Оператор **EndSelect** завершает операцию выбора, начатую оператором **Select**.
Пример:

```
For count=1 To 8
 Select count
  Case 2
   Debug "Счётчик равен двум"
  Case 7
   Debug "Счётчик равен семи"
  Default
   Debug count
 EndSelect
Next count
```

В этом примере организован цикл **For** – **Next**, в котором значение переменной "count" изменяется от 1 до 8. При этом код, расположенный между этими операторами будет выполнен 8 раз. При каждом выполнении кода, значение переменной "count" будет запоминаться для последующего сравнения в операторах **Case**. Совпадение произойдёт в первом операторе **Case**, когда в переменной будет число 2, а во втором, когда в переменной будет число 7. В этих случаях в отладочном окне будут выведены соответствующие текстовые надписи. При других числах в этой переменной, совпадений нет, поэтому будет выполняться код после оператора **Default**.

Оператор: **Goto**
Формат:
Goto <метка>

Оператор **Goto** служит для безусловного перехода в область программы обозначенную меткой. Пользоваться этим оператором следует с большой осторожностью, поскольку некорректное его использование может привести к зависанию программы. Требования к имени меток такие же, как и у имён переменных.
Пример:

```
x=0
Repeat
Debug x
 If x=4
  Goto metka
 EndIf
x=x+1
Until x>8
metka:
End
```

Эта программа завершит свою работу, после того как число в переменной "x" станет равным 4, поскольку в этом случае выполнится условие в операторе **If**, что в свою очередь приведёт к выполнению оператора **Goto**. Программа начнёт выполняться с кода, расположенного после метки. Поскольку там расположена директива **End**, работа программы завершится. Обратите внимание, что на конце имени метки, находящейся в программе присутствует двоеточие для того, чтобы отличать метки от переменных, а в имени метки, после оператора **Goto** двоеточия нет, поскольку в этом случае это однозначно метка.

Подпрограммы и процедуры

Предположим, что нужно одинаковую часть кода выполнить несколько раз. Не совсем рационально помещать один и тот же код несколько раз в программу ведь это приведёт к

увеличению её размера. Специально для такого случая была введена поддержка процедур и подпрограмм. Начнём с подпрограмм.

Вызывать подпрограмму можно только с помощью оператора **Gosub**, а возвращаться из неё следует с помощью оператора **Return**. При этом управление будет передано на следующую инструкцию после вызвавшего эту подпрограмму оператора **Gosub**. Переходить из подпрограммы в основную программу с помощью оператора **Goto** можно, но предварительно следует использовать оператор **FakeReturn**, который эмулирует возврат из подпрограммы без фактического возврата. Это требуется для правильной работы программы. Из подпрограммы можно вызывать другие подпрограммы, но при этом нужно следить за тем, чтобы количество вызовов, равнялось количеству возвратов из подпрограмм.
Пример:

```
a=4
b=5
Gosub plus
Debug c
End
Plus:
c=a+b
Return
```

В первых двух строках этого примера, переменным "a" и "b" присваиваются значения 4 и 5 соответственно. Далее, с помощью оператора **Gosub** вызывается подпрограмма, имеющая метку "Plus". В подпрограмме происходит сложение этих переменных, результат которого помещается в переменную "c". Затем происходит возврат из подпрограммы, (поскольку встретился оператор **Return**) на следующую команду после оператора **Gosub**. Следующей командой является оператор **Debug**, создающий отладочное окно, в котором будет отображено число из переменной "c". Затем выполняется директива **End**. Почти во всех предыдущих примерах эта директива отсутствовала, поскольку она автоматически добавляется в конец программы. Но в этом случае, если директивы **End** не будет в программе после оператора **Debug**, произойдёт выполнение подпрограммы без её вызова, чего допускать нельзя!

Процедуры, как и подпрограммы можно использовать для многократного использования одного и того же кода. Начало процедуры объявляется с помощью оператора **Procedure**, после которого следует тип процедуры (он обозначает тип возвращаемой переменной с помощью оператора **ProcedureReturn**), а затем имя процедуры и её параметры в круглых скобках. Если параметров несколько, их следует разделять запятыми. Скобки обязательно должны быть, даже если у процедуры нет никаких параметров. Далее помещают требуемый программный код. Если в результате его работы требуется возвратить какой-либо результат, следует использовать оператор **ProcedureReturn**, после которого помещается имя переменной или выражение. Конец процедуры обозначается с помощью оператора **EndProcedure**. Вызывают процедуру, просто указав её имя и все параметры в скобках. При составлении программ, имеющих в своём составе процедуры, следует помнить, что процедура должна быть объявлена до её вызова. Это можно сделать, поместив процедуру в начале программы, то есть до её первого вызова или объявив её с помощью оператора **Declare**. При объявлении с помощью этого оператора следует указать тип процедуры, её имя и все параметры. Следует помнить, что все переменные, используемые в процедуре, как бы изолированы от основной программы, поскольку они являются локальными. Если есть необходимость сделать какую-либо переменную из основной программы доступной в процедуре, следует использовать оператор **Global**, указав после которого, имя переменной либо массива.
Пример:

```
Procedure Plus(a , b)
c=a+b
ProcedureReturn c
EndProcedure
x=8
y=2
```

```
Result=Plus(x , y)
Debug result
```

Переменным "x" и "y" присваиваются значения 8 и 2 соответственно. Далее вызывается процедура с именем "Plus". Как отмечалось ранее, в процедуре недоступны переменные, находящиеся в основной программе, поэтому мы передаём их через параметры процедуры. В переменную "a" процедуры будет записано число из переменной "x", а в переменную "b" будет записано число их переменной "y". Далее в процедуре производится сложение чисел из переменных "a" и "b", результат которого, помещается в переменную "c". С помощью оператора **ProcedureReturn** производится возврат значения переменной "c" как результат работы процедуры. Далее следует оператор **EndProcedure**, который завершает процедуру. Результат переписывается в переменную "Result" и выполняется следующая команда после имени процедуры, которой является оператор **Debug**, выводящий на экран число из переменной "Result".

Окна

Окно открывается с помощью функции OpenWindow(). У неё следующий формат.

Result = **OpenWindow**(Window, x, y, InnerWidth, InnerHeight, Title$ [, Flags])

Window – уникальный идентификатор окна — любое целое положительное число. Этот идентификатор используется для последующей работы с окном.

x и y - координаты верхнего левого угла окна, заданые в пикселях, отсчет начинается с верхнего левого угла экрана.

InnerWidth – ширина окна в пикселях (точках)

InnerHeight – высота окна в пикселях (точках)

Title$ – Текст, который будет отображаться в заголовке окна.

Flags – Один или несколько флагов позволяющих изменять вид окна. Флаги представляют собой константы, в которые компилятор предварительно записал определённые значения. Если флагов несколько, они должны быть отделены друг от друга оператором "логического или" (символом |). Обращаю Ваше внимание на то, что слово Flags взято в квадратные скобки условно, в реальной программе их не должно быть! Эти скобки означают, что этот параметр является необязательным и его можно проигнорировать.
Флаги могут быть следующими:

#PB_Window_SystemMenu – Добавляет меню к заголовку окна.
#PB_Window_MinimizeGadget – Добавляет кнопку "свернуть" к заголовку окна.
#PB_Window_MaximizeGadget. – Добавляет кнопку "развернуть" к заголовку окна.
#PB_Window_SizeGadget – Этот флаг разрешает изменять размер окна с помощью курсора "мышки".
#PB_Window_Invisible – Этот флаг позволяет сделать "невидимое" окно.
#PB_Window_ScreenCentered – Если будет использован этот флаг, окно будет располагаться в центре экрана, параметры "X " и "Y" проигнорированы.
#PB_Window_Maximize – Если будет использован этот флаг, после создания, окно будет раскрыто на весь экран.
#PB_Window_Minimize – Если будет использован этот флаг, после создания, окно будет свёрнуто.

Теперь рассмотрим простую программу, создающую окно (Программы\Окно\Окно_1.pb) После её компиляции откроется окно, которое можно свернуть на панель задач или закрыть. Теперь разберемся, как эта программа работает. В самом начале вызывается функция **OpenWindow**(), создающая окно. Далее следует цикл "Repeat – Until", в котором программа зациклится на всё время своей работы. Внутри этого цикла расположена только одна функция **WaitWindowEvent**(), которая на каждом витке цикла записывает в переменную

"Event" идентификатор текущего события в программе. Например, событием является нажатие на кнопки мыши или клавиатуры, но только лишь в том случае если курсор располагается в пределах окна. В этой программе отслеживается только одно событие (в операторе **Until**) – попытка закрыть программу. Если "щёлкнуть мышкой" по кнопке "закрыть" в заголовке окна, это приведёт к записи в переменную "Event" числа эквивалентному числу в константе #PB_Event_CloseWindow. Условие в операторе **Until** будет выполнено и цикл прервётся. Что приведёт завершению работы программы.

Итак, окно создавать научились, только оно "пустое", значит следующий шаг, его заполнение элементами управления программой. В языке PureBasic они называются гаджетами. Добавим к окну кнопку (Программы\Окно\Окно_2.pb). В начале программы открывается окно и создаётся новый список гаджетов с помощью функции **CreateGadgetList**. Для её правильной работы требуется системный идентификатор окна, который получаем с помощью функции **WindowID**(), в качестве параметра, используем идентификатор требуемого окна. Далее с помощью функции **ButtonGadget** создаём кнопку. У неё следующий формат:

Result = **ButtonGadget**(Gadget, x, y, Width, Height, Text$ [, Flags])

Первый параметр (Gadget) – идентификатор создаваемого гаджета, который используется в дальнейшем для работы с гаджетом. Второй параметр (x) - расстояние от левого края окна до начала гаджета. Третий параметр (y) – расстояние он верхнего края окна до начала гаджета. Четвёртый параметр (Width) – ширина гаджета. Пятый параметр (Height) – высота гаджета. Шестой параметр (Text$) – текст на кнопке. Необязательный параметр "Flags" предназначен для изменения свойств кнопки. Забегая вперёд, скажу, что подобный формат имеют большинство гаджетов. Остальная часть программы такая же, как в предыдущем примере. После компиляции программы появится окно, в центре которого будет кнопка. Щёлкать мышкой по ней бесполезно, поскольку мы ещё не "объяснили" программе, что нужно делать после нажатия на кнопку. В следующем примере (Программы\Окно\Окно_3.pb), программа уже реагирует на нажатие по кнопке. В начале программы открывается окно и создаётся кнопка. Далее следует цикл "Repeat – Until", в котором будем обрабатывать события от кнопки, то есть производить ответную реакцию программы на её нажатие. На каждом витке цикла в переменные "Event" и "Gadget" будут записываться идентификаторы события в программе и активного гаджета соответственно. Далее, с помощью оператора **If** проверяется, чтобы в переменной "Event" было такое же число, как и в константе #PB_Event_Gadget, а в переменной "Gadget" было число равное двум, поскольку именно такой идентификатор был присвоен кнопке (первый параметр в функции **ButtonGadget**). Это означает, что произошло событие в гаджете с идентификатором 2. Другими словами, была нажата экранная кнопка. В этом случае будет выполнен код, расположенный между операторами **If** и **EndIf**. Там располагается функция **MessageRequester**(), выводящая на экран окно с сообщением.

Гаджеты

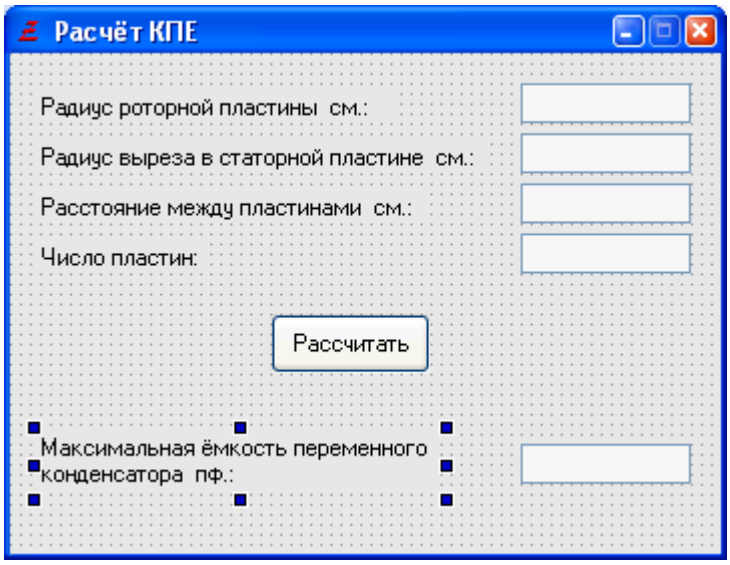
Описания функций создающих гаджеты находятся в разделе "Gadget" справки языка PureBasic. Учтите, что не все функции из этого раздела предназначены для создания гаджетов, некоторые (примерно половина) нужны для управления уже созданными гаджетами. К сожалению, с этой статье нет возможности рассматривать все функции по отдельности, поскольку это займёт очень много места, поэтому работа функций будет рассматриваться на практических примерах. Одним из таких примеров будет программа (Программы\Гаджет\Плюс.pb), которая складывает числа. В самом начале программы объявляется тип некоторых переменных. Например, переменной "string1" присваивается тип String, что означает, что в этой переменной будет храниться строка текста, а в переменной "result", которой присваивается тип Quad можно хранить целые числа с разрядностью 8 байт. Далее открывается окно и создаётся новый список гаджетов. Обратите внимание, что перед функциями **OpenWindow**() и **CreateGadgetList**() расположены операторы **If**, а завершающие их тело операторы **EndIf** располагаются в конце программы, перед директивой **End**. Это требуется для того, чтобы если не удалось открыть окно либо создать новый список гаджетов сразу же завершить работу программы. Далее следуют функции, создающие гаджеты: **StringGadget**() – создаёт поле для ввода данных.
TextGadget() – создаёт надпись в окне.

ButtonGadget() – создаёт кнопку.

Далее следует цикл "Repeat – Until". На каждом витке которого, с помощью функций **WaitWindowEvent**() и **EventGadget**() узнаём текущее событие в программе и идентификатор активного гаджета соответственно. С помощью оператора **If** узнаём, была ли нажата экранная кнопка "Равно". Представим, что она была нажата. В переменной "event" будет число эквивалентное числу из константы #PB_Event_Gadget, а в переменной "gadget" будет число 3, поскольку именно такой идентификатор у этой кнопки. Условие в операторе **If** будет справедливо, что приведёт к выполнению кода, расположенного между этим оператором и оператором **EndIf**. Этот код производит складывание чисел. Сначала считывается текст с помощью функции **GetGadgetText**() из гаджетов с идентификаторами 0 и 2, в которых хранятся введённые числа. Текст из гаджета с идентификатором 0 помещается в переменную "string1", а из гаджета с идентификатором 2 в переменную "string2". Далее, преобразуем, текст в число с помощью функции **ValQ**(), потому что если попытаемся сложить числа в текстовом формате, то получим совсем не тот результат, что требуется. Затем, складываем числа и помещаем результат в переменную "result". Теперь число из этой переменной следует вывести на экран посредством гаджета с идентификатором 4, но сначала преобразовываем число из переменной "result" в текстовый формат с помощью функции **StrQ**(). Остальная часть программы такая же, как в предыдущих примерах.

Для визуального проектирования внешнего вида окна можно использовать программу "Visual Designer", которая вызывается с помощью одноимённого пункта из меню "Tools". После её запуска появятся несколько окон, одно из которых имеет точечную структуру – это заготовка окна, которую предстоит наполнить требуемыми гаджетами. Окно "PureBasic Visual Designer" содержит все доступные гаджеты, которые можно "перетащить" с помощью мыши на заготовку. В окне "Properties" можно изменять свойства выделенного объекта. Окно "Objects viewer" отображает все объекты в заготовке окна.

В качестве примера используем программу **[2]**, которая, по-видимому, была написана на ранней версии бейсика. Сначала следует создать новый проект, выбрав в меню "File" пункт "New project", а затем сохранить его с помощью пункта "Save project" (Программы\Visual Designer\Проект.pbv). Проект следует сохранять, как можно чаще, чтобы не потерять данные при случайном сбое. Далее заготовку перетаскиваем на центр экрана, чтобы удобней было работать. Затем ширину (Width) и высоту (Height) окна делаем равными 355 и 250 соответственно, следя за размером в пунктах Width и Height окна "Properties". В поле "Caption" вводим строку "Расчёт КПЕ", которая станет заголовком окна. На панели "Flags" следует отметить пункты MinimizeGadget и ScreenCentered это добавит одноимённые флаги к функции **OpenWindow**(). Далее наполняем заготовку окна гаджетами. Для этого сначала добавим несколько надписей. В окне "PureBasic Visual Designer" щёлкаем по компоненту изображённому как большая и маленькая буквы "A" (TextGadget). Теперь можно этот гаджет добавить в наше окно. Для этого просто располагаем курсор мыши в нужной позиции заготовки и нажав на левую кнопку, перемещаем мышь. Добавляем таким образом ещё 4 этих компонента. Затем помещаем в окно пять полей для ввода данных (StringGadget), размещая из напротив текстовых надписей и одну кнопку (ButtonGadget). В конечном итоге должно получиться, как показано на **Рис. 4**. Текст вводится в помощью пункта "Text" в окне "Properties". Перейдя на вкладку "Extra" можно задать дополнительные параметры, например, выбрать другой шрифт или добавить всплывающую подсказку. Сделаем это на примере кнопки. Для этого щёлкнем по кнопке, тем самым, выделив её. После чего в поле "ToolTip" введём текст "Узнаём результат расчета", который будет появляться при наведении курсора на область кнопки. Чтобы в этом убедится, выберем в меню "Project" пункт "Run". Появится почти не отличающееся окно, разве что исчезла точечная структура. Если навести



курсор на кнопку, появится всплывающее окошко в только что введённым текстом. Теперь закроем это окно. После того как на заготовке окна были размещены все компоненты, можно экспортировать данные в окно редактора PureBasic. Для этого, в меню "Project", выберем пункт "PureBasic editor", после чего, программу "Visual Designer" можно закрыть. При этом в редакторе программы PureBasic откроются два файла. В первом, всего две строки, и он нам в дальнейшем не понадобится, а во втором, будет собственно исходный текст созданного окна (Программы\Visual Designer\КПЕ_1.pb). Его следует сохранить в любом доступном месте с помощью пункта "Save As" из меню "File". В начале программы с помощью операторов **Enumeration** и **EndEnumeration**, в константы, находящиеся внутри их тела записываются числа, начиная с нуля. Эти константы используются как идентификаторы окна (#Window_0) и идентификаторы гаджетов. Далее следует процедура "Open_Window_0()", в которой находится программный код, открывающий окно и создающий гаджеты. Во всех этих функциях в качестве идентификаторов используются константы. Запускать эту программу ещё рано, поскольку она представляет собой только полуфабрикат. Для того чтобы увидеть окно на экране нужно добавить вызов функции "Open_Window_0()" и главный цикл программы (Программы\Visual Designer\КПЕ_2.pb). Теперь уже окно можно увидеть, но программа пока ещё не понимает, что ей следует делать, когда будет нажата экранная кнопка "Рассчитать". Значит следующий шаг – добавление кода, выполняющего это действие (Программы\Visual Designer\КПЕ_3.pb). Поскольку изменения произошли в цикле "Repeat – Until", то именно с этого места рассмотрим работу программы. В начале цикла в переменные "Event", "Gadget" и "Type" будут записаны, идентификатор события в программе, идентификатор активного гаджета и тип события, соответственно. Далее с помощью оператора **If** проверяется наличие события в каком-либо гаджете и если события нет, то рабочая точка программы "перепрыгнет" на оператор **EndIf**, расположенный перед оператором **Until**. Если событие в гаджете произошло, условие будет выполнено. Далее проверяется, была ли нажата экранная кнопка "Рассчитать" и если она была нажата, с помощью функций **GetGadgetText**() считываются, информация из полей для ввода данных. После чего с помощью функций **ValF**() производится преобразование текстовых данных в числовые. Эти данные помещаются в переменные с типом Float. Далее в операторе **If** проверяется, чтобы в переменных были числа отличные от нуля. Если хоть в одной переменной будет число равное нулю, условие будет выполнено и на экране появится окно с предупреждением, созданное функцией MessageRequester(). В том случае, если в переменных будут числа неравные нулю, условие выполнено не будет и программа начнёт выполняться после оператора **Else**, что приведёт к вычислению выражения и выводу результата на экран с помощью функции **SetGadgetText**(). С помощью функции SetGadgetColor() изменяется цвет фона гаджета с идентификатором заданным константой #String_4. Далее следует ещё один обработчик событий, изменяющий цвет в гаджете с идентификатором #String_4 при любом изменении исходных данных по которым производится расчёт. Это производится следующим образом: В операторе **If**, отслеживаются события во всех полях для ввода данных, кроме того, в котором отображается результат. Условие будет выполнено, если станет активным один из этих гаджетов. С помощью следующего оператора **If** узнаём тип события. Для выполнения условия требуется, чтобы в переменной "Type" было такое же число, как и в константе #PB_EventType_Change. Это означает, что данные в гаджете были изменены. При выполнении условия будет изменён цвет фона гаджета с идентификатором #String_4.

Файлы

Вся информация на дисках компьютера находится в файлах и для доступа к ней следует использовать функции из раздела "File" справки – если требуется работать с содержимым файла или функции из раздела " FileSystem " справки – если нужно, скажем, переместить файл в другую папку или переименовать. Здесь будет рассмотрена работа только функций из раздела "File". Последовательность работы с файлом следующая: С начала следует открыть (создать) файл, затем оттуда что-то прочитать или записать, после чего, нужно закрыть файл.

Файл открывается с помощью одной из трёх функций в зависимости от предполагаемых дальнейших действий. Результат работы функций будет больше нуля в случае успешного открытия или создания файла или равен нулю в противном случае.

ReadFile() – Если требуется открыть файл только для чтения.

OpenFile() – Если требуется открыть файл для редактирования (чтение и запись). Если файла с таким именем не существовало, он будет создан.
CreateFile() – Если требуется создать новый файл или очистить предыдущий с таким же именем.
У этих функций внутри скобок одинаковые параметры. Первый параметр это присваиваемый идентификатор к открываемому файлу, который будет использоваться для дальнейшего обращения к файлу. Второй параметр – имя файла.

Для чтения информации из файла предназначены функции, имя которых начинается с "Read" (кроме ReadFile), а функции, начинающиеся с Write, предназначены для записи информации в файл. Вторая часть имени функции означает тип данных. Например, ReadByte – чтение одного байта из файла, WriteString – запись строки текста в файл. Когда больше не предполагается работать с файлом, его следует закрыть с помощью функции **CloseFile**(), указав в её параметре идентификатор закрываемого файла.

Теперь проведём несколько экспериментов с файлами. Для начала, создадим пустой файл с именем "Проба.txt" в корне диска "C:" (Программы\Файл\Проба.pb). Программа работает так: С помощью функции **CreateFile**() создаём файл, который затем закрываем с помощью функции **CloseFile**(). Директива **End** завершает работу программы. С помощью оператора **If** проверяется был ли открыт файл. Условие будет выполнено, если результат работы функции не равен нулю. Как уже отмечалось ранее функция **CreateFile**() стирает предыдущий файл с таким же именем. В этом очень легко убедится. Откройте созданный файл в любом текстовом редакторе и наберите какой-либо текст. Затем опять запустите программу. После этого размер файла станет равным 0 байт, то есть предыдущие данные из этого файла были уничтожены. Теперь запишем в файл строку текста (Программы\Файл\Проба_1.pb). По сравнению с предыдущим примером, здесь была добавлена функция **WriteString**(), которая запишет в файл слово "Текст". В этом можно убедиться, открыв файл в любом текстовом редакторе.

Очень часто для сохранения настроек программы используют файлы, которые обычно имеют расширение INI. В языке PureBasic для этого существуют специальные функции, описание которых можно найти в разделе "Preference" справки программы. Как и в предыдущем случае, файл сначала следует открыть, затем выполнить необходимые действия с его содержимым, после чего, закрыть. Файл открывается с помощью одной из двух функций. **OpenPreferences**() – Файл доступен для чтения и записи. Эта функция не создаёт файл в случае его отсутствия!
CreatePreferences() – Файл доступен только для записи. В случае отсутствия файла с указанным именем он будет создан. Если файл уже существовал, его содержимое будет стёрто!
Результат работы функций будет больше нуля в случае успешного открытия или создания файла или равен нулю в противном случае.

Для чтения информации из файла предназначены функции, имя которых начинается с "Read", а для записи используются функции с началом имени "Write". Середина имени содержит слово "Preference", а окончание имени, означает тип читаемых или записываемых данных. Закрывается файл с помощью функции **ClosePreferences**(). Теперь рассмотрим пример сохранения размера окна в файле (Программы\Файл\Настройки.pb). Если запустить программу и изменить размеры окна, то при следующем запуске программы, у окна будут те же размеры. Программа работает следующим образом: В начале вызывается подпрограмма "LoadPreferences", которая читает данные из файла "PreferencesWindow.ini" расположенного в одной папке с программой. Файл открывается с помощью функции **OpenPreferences**(). Функция **PreferenceGroup**() определяет раздел, с которого будут читаться данные. С помощью функций **ReadPreferenceLong**() считываются записанные в файл данные. Первый параметр этой функции – ключевое слово в файле, второй параметр – значение по умолчанию, которое будет использовано, в случае если не удалось открыть файл (например, он отсутствует) либо не найдено ключевое слово. После чтения всех данных следует закрывающая файл функция **ClosePreferences**() и оператор **Return**, завершающий подпрограмму. Программа начнёт выполняться со следующей строки после оператора **Gosub**, вызвавшего подпрограмму. В функции **OpenWindow**() в качестве параметров x, y, Width, Height задающих положение и размеры окна использованы одноимённые переменные, данные в которые были записаны в выше рассмотренной подпрограмме. Далее следует цикл "Repeat – Until", работа которого была рассмотрена ранее. При закрытии программы цикл прервётся и с помощью оператора **Gosub**, будет выполнен переход на подпрограмму,

начинающеюся с метки "SavePreferences". В начале которой, открывается файл с помощью функции **CreatePreferences**(). Условие в операторе **If** будет выполнено, если удалось создать файл. Функция **PreferenceGroup**() определяет раздел, в которой будут записываться данные. Функции **WritePreferenceLong**() записывают данные в файл. Первый параметр функции – ключевое слово, второй параметр – записываемые данные, которые получаем с помощью функций возвращающих в качестве результата, параметры окна. Далее файл закрывается и происходит возврат из подпрограммы, после чего работа программы завершается.

Меню

Меню создаётся с помощью функций из раздела "Menu" справки. Последовательность его создания следующая: Сначала, с помощью функции **CreateMenu**(), создаётся пустое меню. Далее, с помощью функции **MenuTitle**(), создаём новый заголовок меню и заполняем его пунктами, с помощью функции **MenuItem**(). Когда в этом меню созданы все пункты, создаём ещё один заголовок (конечно, если это нужно) и его тоже заполняем пунктами и так далее. На этом создание меню можно считать оконченным. В качестве примера рассмотрим файл (Программы\Меню\Меню.pb). После функций создающих окно, следует функция **CreateMenu**(), которая создаст ниже заголовка окна, область меню. Первый параметр этой функции – идентификатор создаваемого меню, второй параметр – системный идентификатор окна, который получаем с помощью функции **WindowID**(). Оператор **If** проверяет, было ли создано меню. Функция **MenuTitle**() создаёт заголовок с именем "Файл". Затем, с помощью функций **MenuItem**() добавляются три пункта. Первый параметр функции – идентификатор пункта, а второй это текст пункта. Функция **MenuBar**() создаёт разделитель между пунктами. Далее **MenuTitle**() создаёт ещё один заголовок, к которому добавляется один пункт. Остальная часть программы рассматривалась в предыдущих примерах. Запустив эту программу, вы увидите окно, с двумя заголовками меню. Если щёлкнуть по одному из них, меню раскроется, но если щёлкнуть по пункту то ничего не произойдет, кроме того, что открытое меню исчезнет. Это происходит потому что, как и в случае с гаджетами, программа не знает, как ей реагировать на клик по пункту меню. В следующем примере добавлена обработка событий от меню (Программы\Меню\Меню_1.pb). После запуска программы появится окно по внешнему виду не отличающееся от предыдущего. При щелчке по любому пункту (кроме пункта "Выход", который завершит работу программы) будет появляться окошко с именем этого пункта. Программа работает следующим образом: После функций открывающих окно и меню, следует цикл "Repeat – Until", на каждом витке которого, в переменные "Event" и "Menu" будут записываться идентификаторы текущего события в программе и активного пункта меню, соответственно. Условие в строке "If Event=#PB_Event_Menu" будет выполнено при активации какого-либо пункта меню. В операторе **Select** запоминается значение из переменной "Menu" для последующего сравнения в операторах **Case**. Далее следуют четыре оператора **Case**, после имени которых, находятся сравниваемые значения. Программный код будет выполнен только после одного из этих операторов в зависимости от того, по какому пункту был произведён щелчок мышкой. Например, если щёлкнуть по пункту "Сохранить", будет выполнен код после строки "Case 1". Это произойдёт потому, что данный пункт имеет идентификатор 1 и такое же число является условием в операторе **Case**.

Панель инструментов

Панель инструментов создаётся с помощью функций из раздела "ToolBar" справки. Панель предназначена для быстрого доступа к некоторым пунктам меню. Она создаётся с помощью функции **CreateToolBar**(). Первый параметр которой – идентификатор создаваемой панели, а второй, системный идентификатор окна, на которое следует поместить панель. Кнопки добавляются с помощью функции **ToolBarStandardButton**(), её первый параметр – идентификатор пункта меню, который будет дублировать данная кнопка. Второй параметр – константа, определяющая рисунок на кнопке. Следующий пример (Программы\Панель инструментов\Панель.pb) отличается от предыдущего лишь тем, что в нем присутствует код создающий панель. Он расположен между оператором **EndIf** завершающим создание меню и оператором **Repeat**. Как уже отмечалось ранее, панель инструментов создаётся с помощью функции **CreateToolBar**(). Далее следуют функции **ToolBarStandardButton**(), которые создают кнопки, дублирующие пункты меню, идентификаторы которых, заданны в первом параметре функции. С помощью функций

ToolBarToolTip() добавляются всплывающие подсказки к кнопкам. Первый параметр функции – идентификатор панели, второй параметр – идентификатор кнопки, который в тоже время является идентификатором дублирующего пункта меню. Третий параметр – текст всплывающей подсказки. Запустив программу, Вы увидите, что ниже меню появилась панель с двумя кнопками. При наведении курсора мыши на одну из кнопок будет появляться всплывающее окошко с подсказкой. Если щёлкнуть по кнопке, то появится сообщение, что был щелчок по пункту меню. В примере (Программы\Панель инструментов\Редактор.pb) обобщён весь материал пройденный ранее. Эта программа является простейшим текстовым редактором. Она работает следующим образом: После объявления процедур LoadFile и SaveFile, открывается невидимое окно и создаётся новый список гаджетов. Функция **EditorGadget**() создаёт редактор текста. Далее создаются меню и панель инструментов. Функция **CreatePopupMenu**() создаёт контекстное меню, в котором есть два пункта, дублирующие пункты из основного меню "Файл". Функция **HideWindow**() разрешает отображать ранее созданное, невидимое окно. Далее следует цикл "Repeat – Until". На каждом витке этого цикла в переменные "Event", "Menu", "Gadget", "Window" с помощью функций, будут записываться идентификаторы: текущего события в программе, активного пункта меню, гаджета в котором произошло событие и активного окна, соответственно. Условие строке "If window=0" будет выполнено, если активным является окно с идентификатором 0. Только в этом случае будут обрабатываться события от всех его объектов. Далее в строке "If Event=#PB_Event_SizeWindow" проверяется, изменился ли размер окна и если он изменился, будет выполнена функция **ResizeGadget**(), которая изменит размер редактора (EditorGadget) таким образом, чтобы он занимал необходимую часть окна. Константы #PB_Ignore означают, что эти параметры изменять не нужно. Условие в строке "If Event=516" будет справедливо при щелчке правой кнопкой мышки в пределах окна. В этом случае будет выполнена функция **DisplayPopupMenu**(), отображающая контекстное меню. Первый параметр этой функции – идентификатор меню, а второй – системный идентификатор окна, в котором это меню следует отобразить. Далее производится обработка событий от меню. Код после строки "Case 0" будет выполнен, если щёлкнуть по пункту меню "Открыть". Функция **OpenFileRequester**() открывает стандартное окно выбора файла, путь к которому будет помещён в строковую переменную "File". Далее оператор **If** проверяет, чтобы в этой переменной была строка текста. Если условие выполняется, будет вызвана процедура LoadFile, открывающая файл. В начале этой процедуры, функция **ReadFile**() открывает выбранный файл. Далее следует цикл While – Wend, код внутри которого выполняется до тех пор, пока результат работы функции **Eof**() не станет отличным от нуля. Это произойдёт по достижению конца файла. Данные из файла читаются построчно с помощью функции **ReadString**() и помещаются в строковую переменную "Text". Когда весь файл будет прочитан, информация перепишется с помощью функции **SetGadgetText**() из переменной "Text" в гаджет заданный в переменной "Gadget". При вызове процедуры, в эту переменную было записано число 1, что означает, что данные будут помещены в текстовый редактор, поскольку у него именно такой идентификатор. Далее, файл закрывается, и работа процедуры завершается.

Если в меню выбрать пункт "Сохранить", будет выполнен код после строки "Case 1". Функция **SaveFileRequester**() вызывает стандартное окно сохранения файла. В строковую переменную "File" будет записан путь к сохраняемому файлу, либо "пустая" строка, если в окне была нажата экранная кнопка "Отмена". Далее, оператор **If** проверяет, чтобы в переменной "File" была строка текста. В следующей строке, с помощью функции **GetExtensionPart**() определяем, есть ли расширение у сохраняемого файла. Если расширения нет, оно будет добавлено. После этого, вызывается процедура SaveFile. В этой процедуре создаётся новый файл, считываются данные из редактора и записываются как одна большая строка, с помощью функции **WriteString**(). Затее файл закрывается.

При выборе в меню пункта "О программе", будет выполнен программный код после строки "Case 3". Это код создаст ещё одно окно. В начале, с помощью функции **IsWindow**() проверяется, существует ли это окно. Если оно есть, то с помощью функции **SetActiveWindow**() делается активным. Если окна нет, то будет выполнен код после оператора **Else**, что приведёт к созданию окна и нового списка гаджетов. Далее создаются два гаджета. Обработка событий происходит внутри главного цикла "Repeat – Until" и находится после строки "If window=1" Там отслеживается попытка закрыть окно и нажатие на экранную кнопку "Ok". В любом случае, окно будет закрыто с помощью функции **CloseWindow**().

Строка состояния

Строкой состояния называется специальная панель, находящаяся в нижней части окна, на которой может отображаться различная информация. Она создаётся с помощью функций из раздела "StatusBar" справки. Рассмотрим создание строки состояния и работу с ней на примере (Программы\Строка состояния\Строка состояния.pb). Эта программа отображает текущее время.

В начале программы находится процедура "DateStatusBar", работу которой рассмотрим позже. Далее, открывается окно. После чего с помощью функции **CreateStatusBar**() создаётся строка состояния. Первый параметр функции – идентификатор создаваемой строки состояния, а второй – системный идентификатор окна. С помощью функции **AddStatusBarField**() она делится на несколько частей (областей), первой автоматически присваивается идентификатор 0, следующей 1 и так далее. Параметр функции определяет длину области. Далее, функция **StatusBarText**() записывает текст в первую область строки состояния с идентификатором ноль. Функция **OpenLibrary**() открывает файл "User32.dll" находящийся в системной папке операционной системы для последующего использования его функций. Следующая функция **CallFunction**() предназначена для доступа к функциям из ранее открытого файла. Она вызывает так называемую API функцию "SetTimer", организующую таймер, который будет запускать процедуру "DateStatusBar" через каждые 500 миллисекунд. Символ "@" перед именем процедуры означает, что на самом деле вызываемой функции будет передано не имя процедуры, а её физический адрес в памяти. Далее, следует цикл "Repeat – Until", работа которого рассматривалась ранее. При закрытии окна этот цикл прервётся и будет выполнена строка "CallFunction(0,"KillTimer",WindowID(0),1)", что приведёт к прерыванию работы ранее созданного таймера. Следующая функция **CloseLibrary**() закрывает файл "User32.dll".

Теперь разберёмся с процедурой "DateStatusBar", которая как уже отмечалось ранее, вызывается два раза в секунду (500 мс) по таймеру. С помощью строки "Time.s = FormatDate("%hh:%ii:%ss", Date())" узнаём текущее время. Функция **StatusBarText**() записывает его в область с идентификатором ноль, строки состояния.

Трей

Трей расположен в правой части панели задач Windows. Обычно его используют некоторые программы для размещения своих иконок. Рассмотрим принцип добавления иконки в трей на примере программы (Программы\Трей\Трей.pb). После запуска программы, в верхней части экрана появится небольшое окно, а в трее иконка в виде принтера. Если щёлкнуть правой кнопкой мышки по этой иконке, появится всплывающее меню с единственным пунктом "Выход". Работает программа следующим образом: После функции **OpenWindow**() открывающей окно, находится функция **LoadImage**(), с помощью которой загружается в память рисунок (иконка) их файла SysTray.ico, находящийся в одной папке с программой. Функция **AddSysTrayIcon**() добавляет иконку в трей. Первый параметр этой функции – идентификатор иконки в трее, требуемый для дальнейшей работы с ней. Второй параметр – системный идентификатор окна. Третий параметр – системный идентификатор иконки, который был получен с помощью предыдущей функции. Функция **SysTrayIconToolTip**() добавляет всплывающую подсказку к иконке. Первый параметр функции – идентификатор иконки в трее, а второй – текст подсказки. Далее, создаётся всплывающее меню с помощью **CreatePopupMenu**(). Обработка событий от окна и трея, осуществляется в цикле "Repeat – Until". Условие в строке "If Event=#PB_Event_SysTray And Type=#PB_EventType_RightClick" будет справедливо при щелчке правой кнопкой мышки по иконке в трее. Это приведёт к появлению всплывающего меню, поскольку будет выполнена функция **DisplayPopupMenu**(). Обработку события от пункта "Выход" меню осуществляет строка "If Event=#PB_Event_Menu And Menu=2". Если щёлкнуть по этому пункту, то будет выполнен оператор **Break**, прерывающий цикл "Repeat – Until" и тем самым завершающий работу программы.

COM порт

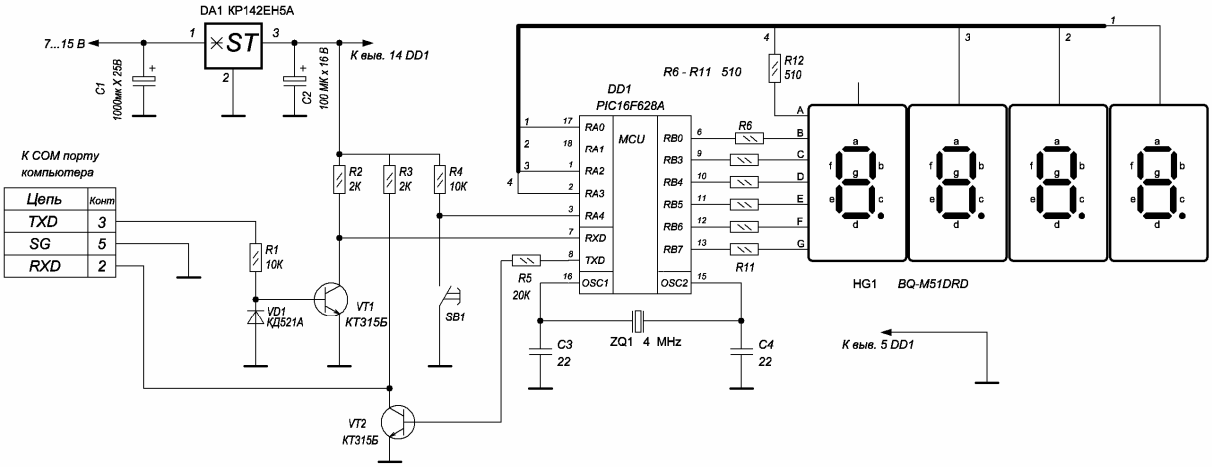
Последовательный (Com) порт предназначен для связи компьютера с внешними устройствами, например, модемом. В операционных системах семейства Windows 9x, основанных на ядре MS-DOS можно обращаться к портам, используя ассемблерные инструкции Inp и Out. К сожалению, эти инструкции заблокированы для программ

пользователя, в операционных системах на основе ядра NT (Windows NT, 2000, XP и др.). Это сделано для большей стабильности системы. Поэтому для доступа к порту, следует использовать специальный драйвер, например тот, что есть в системе. К этому драйверу можно обращаться посредством так называемых API функций. Чтобы облегчить задачу, будем использовать дополнительную библиотеку с функциями к программе PureBasic, которая для связи с портом использует API функции. Скачать её можно по адресу http://www.purearea.net/pb/download/userlibs/MVCOM_LIBRARYV12.zip. Она содержит инсталлятор, требуется только указать путь к программе PureBasic. После её установки, в меню "Help" появится пункт "External Help" содержащий вложенное меню, в котором должен появиться пункт "MVCOM.chm". Это файл справки рассматриваемой библиотеки. Все далее рассматриваемые функции, имя которых, начинается с Com, принадлежат этой библиотеке.

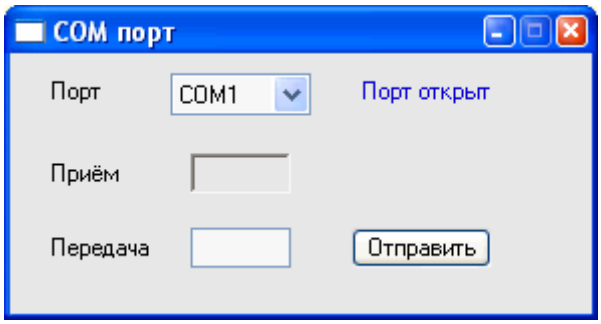
Для доступа к COM порту, его следует сначала открыть. Это делает функция **ComOpen**(). В первом параметре которой, указываются свойства порта: его имя, скорость, чётность, количество бит и длина стопового бита. Все эти данные должны быть помещены в строковую переменную. Например, строка "COM2:2400,N,8,1" означает, что будет использоваться порт с именем "COM2", скорость обмена информацией равна 2400 бод (бит в секунду), проверка чётности не производится, количество передаваемых и принимаемых байт равно 8, длина стоп бита, равна 1. Второй параметр функции, отвечает за способ управления потоком информации. Третий и четвёртый параметры это размеры в байтах буфера приёма и передачи. Они нужны для временного хранения информации, пока она не будет прочитана программой или отправлена в порт. Эта функция при успешном подключении к порту возвращает системный идентификатор порта, который будет использоваться для доступа к порту посредством остальных функций их этой библиотеки.

Для чтения данных из открытого порта служит функция **ComRead**(). Первый её параметр – системный идентификатор порта, второй – адрес в памяти буфера для приёма данных, третий параметр – размер буфера, заданный в байтах. Записываются данные в порт с помощью функции **ComWrite**(). У неё параметры такие же, как и у предыдущей функции, единственное отличие в том, что данные из буфера отправляются в порт. Порт закрывается с помощью функции **ComClose**(), в качестве параметра которой, следует указать системный идентификатор закрываемого порта.

Для примера используем программу (Программы\COM\Com порт.pb). Она позволяет побайтно обмениваться информацией с внешним устройством, схема которого показана на Рис 5.



Внешний вид окна программы изображён на Рис 6. После её запуска, выбираем порт, и в поле "Передача" вводим любое целое положительное число от 0 до 255, после чего нажимаем на экранную кнопку "Отправить". На индикаторе HG1 будет отображено переданное число. Если нажать на кнопку SB1, то последний принятый байт будет отправлен в компьютер, и отобразится в поле "Приём". Работает программа следующим образом: В самом начале программы, переменную "ComId" объявляем глобальной. В ней будет храниться системный идентификатор открытого



порта или ноль, если выбранный порт недоступен. Процедура InCom() принимает данные из порта. Она вызывается по таймеру через каждые 100 миллисекунд. В начале процедуры проверяется, чтобы число в переменной "ComId" было больше нуля, это означает наличие открытого порта. Далее, с помощью функции **ComInputBufferCount**() узнаем, сколько байт было принято, но ещё не прочитано. Если поступила информация в порт, тогда считываем её с помощью функции **ComRead**(). В качестве приёмного буфера выступает переменная "Buf". Символ "@" перед её именем позволяет узнать адрес в памяти этой переменной. Функция **SetGadgetText**() записывает принятый байт в десятичном формате в гаджет с идентификатором 4.

Выполнение программы начнётся с функции **OpenWindow**(), открывающей окно. Далее, создаётся гаджеты. После чего, с помощью функции **GetGadgetText**() получаем текст из текущего пункта, выпадающего списка. Далее вызывается подпрограмма, начинающаяся с метки "SelectPort", в которой осуществляется открытие порта. В начале подпрограммы производится подготовка к открытию. В строковую переменную "Port" записывается имя порта (из переменной "ComboBox"), скорость его работы, отключается проверка чётности, количество бит устанавливается равным 8, длина стоп бита, равна 1. Далее проверяется, есть ли открытый порт. В случае положительного результата выполняется функция **ComClose**(), закрывающая текущий порт и обнуляется переменная "ComId". Далее открывается порт с помощью функции **ComOpen**(). В результате работы этой функции в переменную "ComId" будет помещён идентификатор порта либо число 0, случае неудачной попытки открыть порт. После этого выводим сообщение о результатах открытия порта. Далее следует оператор **Return**, означающий конец подпрограммы и программа начнёт выполняться со следующей команды после оператора **Gosub**, вызвавшего подпрограмму. Далее следует функция **OpenLibrary**(), открывающая файл "user32.dll", для использования функций входящих в его состав. Из этого файла будут вызваны две функции "SetTimer" и "SendMessageA". Первая создаст таймер, по которому будет вызываться процедура InCom() через каждые 100 миллисекунд, а последняя ограничивает количество веденных символов в гаджет с идентификатором 6. Далее следует цикл "Repeat – Until", в котором будут обрабатываться события программы. В начале, с помощью строки "If Event=#PB_Event_Gadget", убеждаемся, что произошло событие в одном из гаджетов. Далее, в операторе **Select** запоминается текущее значение переменной "Gadget", для последующего сравнения в операторах **Case**. Программный код после строки "Case 1" будет выполнен при любых манипуляциях с выпадающим списком, с помощью которого, можно выбрать текущий порт. В начале считывается текст из текущего пункта, выпадающего списка. Затем в операторе **If** производится проверка считанных данных. Условие будет выполнено, если данные при текущей проверке отличаются от предыдущей, иными словами, был выбран другой порт. В этом случае, будет вызвана подпрограмма "SelectPort", работа которой рассматривалась ранее. В ней закрывается текущий порт и открывается выбранный.

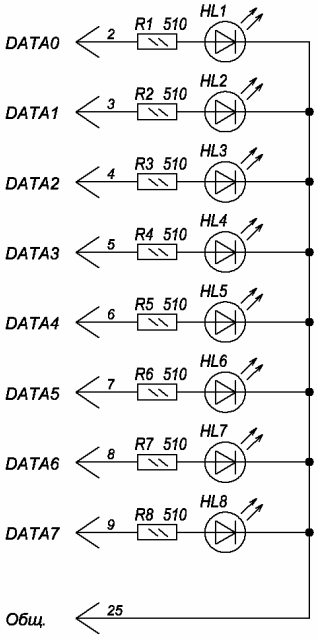
Код после строки "Case 7" будет выполнен, при нажатии на экранную кнопку "Отправить". Сначала считывается текст из поля "Отправить", у которого идентификатор 6. Далее вызывается подпрограмма "ComOut", передающая один байт во внешнее устройство. В начале подпрограммы, проверяется наличие открытого порта, и если его нет, на экране появится сообщение об ошибке. Далее текст из строковой переменной "String" преобразуется в числовой вид и помещается в переменную "Out". После чего проверяется, чтобы число не превышало 255 и если оно превышает, делаем его равным этому значению. Функция **ComWrite**() передаёт данные из переменной "Out" в порт. Далее происходит возврат их подпрограммы.

При закрытии программы, цикл "Repeat—Until" прервётся. После чего закрывается открытый порт, выключается таймер с помощью API функции "KillTimer" из файла "user32.dll" и закрывается этот файл с помощью функции **CloseLibrary**(). Директива **End** завершает работу программы, что приводит к закрытию окна и освобождению всех ресурсов, занятых программой.

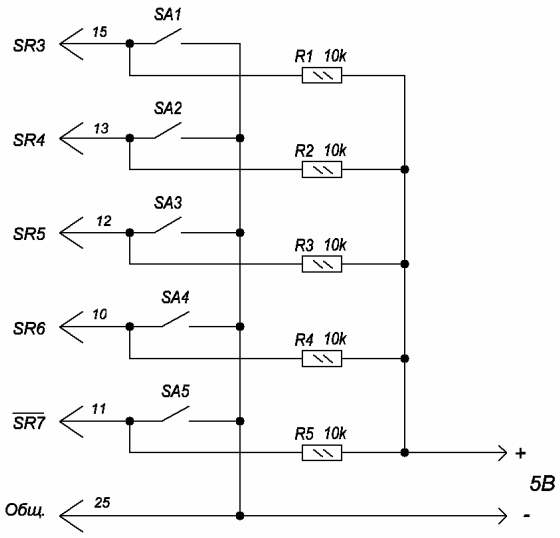
LPT порт

Параллельный (LPT) порт предназначен для связи с принтером. Этот порт можно рассматривать как три восьмиразрядных регистра в пространстве ввода—вывода: регистр данных по адресу 378H, регистр состояния по адресу 379H и регистр управления по адресу 37AH. Эти адреса относятся к порту LPT1. У других портов эти адреса будут иными, например, порт LPT2 зенимает адреса 278H—27AH. Более подробно об этом порте можно

узнать здесь [**3**]. Для связи с портом будем использовать драйвер "inpout32.dll". Для примера возьмём программу (Программы\LPT\LPT порт.pb). Она работает с портом LPT1 и позволяет записывать информацию в регистр данных и считывать информацию из регистра состояния. Визуально наблюдать за изменением информации в регистре данных можно с помощью устройства, схема которого показана на Рис 7.



Светодиоды будут отображать переданный байт в двоичном виде. Для изменения логических уровней на входах регистра статуса, можно использовать устройство, схема которого показана на Рис 8. Нажав на экранную кнопку "Принять" получаем данные из этого регистра. При этом следует учитывать, что три младших разряда регистра не имеют связей с разъёмом, а самый старший разряд SR7 инвертирован!



Теперь рассмотрим работу программы. В её начале находятся две процедуры: LPT_Inp и LPT_Out, которые предназначены для обмена информацией с портом, посредством драйвера "inpout32.dll". Далее с помощью функции **OpenLibrary**() к программе подключается файл "inpout32.dll", который может располагаться в системной папке, либо в одной папке с программой. Если этот файл не был обнаружен, будет выведено сообщение об ошибке и работа программы прервётся, поскольку будет выполнена директива **End**. Следующая функция **OpenLibrary**(), подключает к программе файл "user32.dll" из системной папки. Далее открывается окно и создаются гаджеты, после чего, следует цикл "Repeat – Until". Условие в строке "If Event=#PB_Event_Gadget" будет выполнено при событии в одном из гаджетов. Оператор **Select**, запоминает текущее значение переменной "Gadget" для последующего сравнения в операторах **Case**. Если нажать на экранную кнопку "Принять", будет выполнен код, после строки "Case 3". С помощью процедуры LPT_Inp считывается байт из регистра состояния, расположенного по адресу 379H. Результат помещается в переменную "Inp". Данные из этой переменной переписываются в поле "Приём" с помощью функции **SetGadgetText**(). Нажатие на экранную кнопку "Отправить" приведёт к выполнению кода после строки "Case 6". Данные будут считаны, преобразованы в числовой вид и помещены в переменную "Out". Процедура LPT_Out записывает информацию из их этой переменной в регистр данных порта LPT1. Первый параметр процедуры – адрес в шестнадцатеричном виде, а второй записываемый байт.

При завершении работы программы, будут закрыты файлы "inpout32.dll" и "user32.dll" с помощью функции **CloseLibrary**().

Литература

1. Демо-версия языка PureBasic – http://www.purebasic.com/download/PureBasic_Demo.exe
2. **Воронин В**. КПЕ для усилителя мощности – Радио, 2005, № 10, с. 64 – 65
3. **Захаров Д**. Программирование LPT порта в Visual Basic – Радио, 2007, №9, с. 61-62

Оператор	Описания / Примеры
=	Этот оператор обычно используется для присваивания переменной определённого значения. **Пример**: A=b+c A=10
+	Этот оператор используется для суммирования нескольких переменных или прибавления определённого значения к переменной. **Пример**: A=v+r S=d+20
–	Этот оператор используется для вычитания из переменной определённого значения. **Пример**: m=20-1 n=#const-10
*	Этот оператор выполняет умножение. **Пример**: R=20*5 F=b*3
/	Этот оператор выполняет деление (делить на ноль нельзя!). **Пример**: w=x / 4 f= d / s
&	Этот оператор выполняет поразрядное "И" на уровне двоичных чисел. LHS | RHS | Результат ---------- 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1 **Пример**: b.w = %1100 & %1010 ; Результат %1000 s= d & w
|	Этот оператор выполняет поразрядное "ИЛИ" на уровне двоичных чисел. LHS | RHS | Результат ---------- 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 1 **Пример**: b.w = %1100 | %1010 ; Результат %1110 g= h | t
!	Этот оператор производит поразрядное "Исключающее- ИЛИ на уровне двоичных чисел. LHS | RHS | Результат ---------- 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 0 **Пример**: b.w = %1100 ! %1010 ; Результат %0110 u= e ! f
~	Производится поразрядное логическое отрицание (инвертирование) на уровне двоичных чисел. RHS | Результат ---------- 0 | 1 1 | 0 **Пример**: b.w = ~%1010 ; Результат %0101

()	С помощью скобок можно изменить последовательность математических действий. **Пример**: d = 5 * (2 + 3) ; сначала будет произведено сложение а затем умножение.
<	Оператор "Меньше", используется для сравнения переменных.
>	Оператор "Больше", используется для сравнения переменных.
<=	Оператор "Меньше или равно", используется для сравнения переменных.
>=	Оператор "Больше или равно", используется для сравнения переменных.
<>	Оператор "Не равно", используется для сравнения переменных
And	Логическое "И"
Or	Логическое "ИЛИ"
Xor	"Исключающее ИЛИ"
Not	Логическое отрицание (инвертирование)
<<	Сдвиг содержимого переменной влево, на уровне двоичных чисел **Пример**: v.b = %00000001 << 1 ; Сдвиг на один разряд влево, результат %00000010 с последующей записью результата в переменную "v" v.b = %00000001 <<3 ; Сдвиг на три разряда влево, результат %00001000 с последующей записью результата в переменную "v" a= d << 2 ; Сдвиг содержимого регистра "d" на два разряда влево
>>	Сдвиг содержимого переменной вправо, на уровне двоичных чисел **Пример**: x = %01001000 >> 2 ; Сдвиг на 2 разряда вправо, результат %00010010 p=f.b >>3 ;Сдвиг содержимого регистра "d" на три разряда вправо