Оглавление

Описание библиотеки PIP....2
Общие сведения....2
Контейнеры....3
Байтовый массив....4
Битовый массив....4
Строка....4
Файл....4
Поток....5
Таймер....5
Математика....5
Вычислитель....7
Захват клавиатуры....10
Консоль....10
Конфигурационный файл....14
Последовательный порт....16
Сетевой интерфейс....18
Протокол....18
Сигналы....21
Анализ командной строки....22
Процесс....23
Описание классов....26

Описание библиотеки PIP

Общие сведения

Целью создания является потребность в инструменте, позволяющем создавать консольные программы общего назначения, в том числе для работы с COM-портами и Ethernet, с минимальным объемом кода.

Библиотека получила название PIP, сокращенно Platform-Independent Primitives. К основным достоинствам можно отнести работоспособность кода, написанного на базе PIP, минимум на 3 ОС:

1. Windows (любой версии);
2. Linux (проверено на Gentoo и Ubuntu);
3. QNX (проверено на версиях 6.3.0 и 6.3.2),
   а также на 2 компиляторах под Windows: MinGW и MS Visual C.

Библиотека представляет собой набор классов, некоторые из них базовые, т. е. используются как предок класса-наследника. При создании библиотеки использовался стиль Qt.

При подключении любого файла библиотеки подключается файл «piincludes.h», который содержит:

- макрос LINUX, QNX или WINDOWS, определяющий рабочую ОС;
- макрос CC_GCC или CC_VC, определяющий компилятор;
- макрос FOREVER – бесконечный цикл;
- макрос FOREVER_WAIT для бесконечного ожидания;
- краткие названия типов:
  - `llong – long long;`
  - `uchar – unsigned char;`
  - `ushort – unsigned short int;`
  - `uint – unsigned int;`
  - `ulong – unsigned long;`
  - `ullong – unsigned long long;`
  - `ldouble – long double;`
- функцию `string errorString()`, возвращающую текст последней системной ошибки;
- шаблонные функции:
  - `void piSwap(Type & f, Type & s)` – меняет местами значения f и s;
  - `Type piMin(const Type & f, const Type & s)` – возвращает минимальное значение из f и s;
  - `Type piMax(const Type & f, const Type & s)` – возвращает максимальное значение из f и s;
  - `Type piMin(const Type & f, const Type & s, const Type & t)` – возвращает минимальное значение из f, s и t;
  - `Type piMax(const Type & f, const Type & s, const Type & t)`

– возвращает максимальное значение из f, s и t;
    - `Type piClamp(const Type & v, const Type & min, const Type & max)` – возвращает значение `v` в пределах `min` и `max`.

Все классы библиотеки оперируют контейнерами, определенными в файле «picontainers.h» и строками, определенными в файле «pistring.h».

Контейнеры

- **PIVector** – быстрый доступ к любому элементу, быстрое добавление в конец;
- **PIList** – быстрый обход всех элементов;
- **PISet** – набор уникальных элементов;
- **PIStack** – реализует функции стека;
- **PIDeque** – быстрое добавление в начало и конец;
- **PIQueue** – реализует функции очереди;
- **PIHash** – набор пар ключ-значение, ключи уникальны.

Для удобного обхода контейнеров существует макрос `piForeach(i,c)`, позволяющий получить поэлементный доступ к любому типу контейнера. Этот макрос расширяется набором окончаний:

- C – приставка const к переменной;
- R – обход в обратном порядке;
- A (только GCC) – автоматическое определение типа переменной.

Например:

```
PIVector<int> v;
v << 1 << 4 << 8;
piForeachC (int i, v) \\ == piForeach (const int i, v)
    cout << i << " ";
cout << endl;
```

В консоли будет «1 4 8».

```
PIVector<int> v;
v << 1 << 4 << 8;
piForeachRA (i, v) { \\ == piForeachR (int & i, v)
    i--;
    cout << i << " ";
}
cout << endl;
```

В консоли будет «7 3 0».

Для удобного использования перечислений существует шаблонный класс **PIFlags**, позволяющий использовать его вместо переменной целого типа и имеющий оператор [] для проверки вхождения элемента

перечисления.

Например:

```
enum TestEnum {First, Second};
PIFlags<TestEnum> flags;
flags = First | Second;
if (flags[Second]) cout << "second ";
flags &= First;
if (flags[First]) cout << "first";
if (flags[Second]) cout << "second";
cout << endl;
```

В консоли будет «second first».

Байтовый массив

Байтовый массив в PIP, класс **PIByteArray**, – это наследный от PIVector<uchar> класс, имеющий функции перевода содержимого в base64 и обратно, а также упаковку/распаковку по алгоритму RLE.

Битовый массив

Байтовый массив в PIP, класс **PIBitArray**, реализует хранение битовых данных и все присущие контейнерам операции. Содержимое всегда выравнено по 4-х байтовой границе. Существует потоковый оператор вывода в std::ostream, группирующий вывод по 8 бит.

Строка

Строка в PIP, класс **PIString**, работает с кодировкой UTF8.

Строка представляет собой динамический массив символов **PIChar**, каждый из которых – это 4-х байтное целое число. Т. е. строка в может содержать символы в юникоде. Строка имеет все необходимые функции для работы с текстовой строкой. Полная совместимость со строками:

- char * ;
- wchar_t * ;
- std::string;
- std::wstring.

Строку можно создавать из любого числа, а также переводить в числа, причем числа с приставкой «0x» и «0» воспринимаются как шестнадцатеричные и восьмеричные соответственно.

Файл

Файл в PIP, класс **PIFile**, является расширением файлового потока std::fstream. Он может выступать в качестве текстового и бинарного файла одновременно: для побайтового чтения/записи существуют

функции writeData/readData, а для текстового чтения/записи потоковые операторы.

Поток

Поток в PIP, класс **PIThread**, реализован через pthread в *NIX\QNX средах и через WinAPI в Windows. Для использования потока необходимо наследоваться от класса PIThread и переопределить функцию void run(); Также можно переопределить функции, вызывающиеся при старте и завершении потока: void begin() и void end(). функция run() вызывается внутри созданного потока до тех пор, пока не будет вызвана функция stop() или terminate(), т. е. нет необходимости создавать внешний цикл внутри run(). Поток имеет приоритет и собственный мьютекс, который при включенном needLockRun() блокирует вызов run(). Этот мьютекс можно блокировать/разблокировать функциями потока lock()/unlock(). Поток запускается функцией start(int delay), где delay – задержка в миллисекундах между вызовами run(). Для одиночного зауска run() используется функция startOnce().

Таймер

Таймер в PIP, класс **PITimer**, реализован через события системы в *NIX\QNX средах и через поток с задержкой в Windows. В конструкторе таймера передается указатель на функцию формата void function(void * data, int derlim), которую таймер будет вызывать через фиксированные отрезки времени. В эту функцию он передает данные, заданные функцией setData(void * data) или в конструкторе и делитель частоты, вызвавший функцию ('1' для основной частоты). Также таймер можно использовать для измерения отрезков времени: функции elapsed_n(), elapsed_u(), elapsed_m() и elapsed_s() возвращают количество нано-, микро-, мили- или секунд с момента последнего вызова reset() или запуска таймера.

Таймеру можно задавать делители частоты с помощью функции void addDelimiter(int delim, TimerEvent slot), где delim – количество делитль частоты, slot – вызываемая по делителю функция, по умолчанию = 0, при этом будет вызываться основная функция таймера.

Например:

```
int ticks = 0;

void timerFunc(void * d, int delim) {
     cout << "tick " << ticks << ", delim = "
          << delim << ", elapsed "
          << ((PITimer*)d)->elapsed_m()
          << " ms" << endl;
     ticks++;
};

int main(int argc, char * argv[]) {
```

```
    PITimer timer(timerFunc, &timer);
    timer.addDelimiter(2);
    timer.addDelimiter(5);
    timer.start(50.);
    while (ticks < 10) msleep(1);
};
```

В консоли будет:

```
«tick 0, delim = 1, elapsed 50.3072 ms
 tick 1, delim = 1, elapsed 100.31 ms
 tick 2, delim = 2, elapsed 100.432 ms
 tick 3, delim = 1, elapsed 150.269 ms
 tick 4, delim = 1, elapsed 200.283 ms
 tick 5, delim = 2, elapsed 200.407 ms
 tick 6, delim = 1, elapsed 250.293 ms
 tick 7, delim = 5, elapsed 250.417 ms
 tick 8, delim = 1, elapsed 300.27 ms
 tick 9, delim = 2, elapsed 300.394 ms»
```

Математика

В PIP реализованы векторный-матричный аппараты, решение дифференциальных уравнений различными методами, а также минимальный набор математических функций и констант.

Краткие названия типов:

- `complexi` – `complex<int>`;
- `complexf` – `complex<float>`;
- `complexd` – `complex<double>`;
- `complexld` – `complex<ldouble>`.

Константы:

- M_PI – π;
- M_2PI – 2π;
- M_PI_3 – $\pi/3$;
- `complexld complexld_i(0., 1.);`
- `complexld complexld_0(0.);`
- `complexld complexld_1(1.);`
- `complexd complexd_i(0., 1.);`
- `complexd complexd_0(0.);`
- `complexd complexd_1(1.);`
- `double deg2rad` = $\pi/180$ – для перевода из градусов в радианы;
- `double rad2deg` = $180/\pi$ – для перевода из радиан в градусы;

Функции:

- `int pow2(const int p)` – возведение 2 в целочисленную положительную степень;

- `double sqr(const double & v)` – квадрат числа;
- `double sinc(const double & v)` – $\sin(\pi \cdot v)/(\pi \cdot v)$;
- `complexd atanc(const complexd & c)` – арктангенс от комплексного числа;
- `complexd asinc(const complexd & c)` – арксинус от комплексного числа;
- `complexd acosc(const complexd & c)` – арккосинус от комплексного числа;

Векторный и матричный аппараты реализованы в двух формах: шаблонной и динамической.

Шаблонные классы вектора и матрицы:

- `template<uint Size, typename Type = double> class` **`PIMathVectorT`**,
- `template<uint Cols, uint Rows = Cols, typename Type = double> class` **`PIMathMatrixT`**.

По умолчанию вектор и матрица имеют тип double, матрица квадратная.

Размеры шаблонных векторов и матриц нельзя менять после их объявления.

Динамические классы вектора и матрицы:

- `template<typename Type = double> class` **`PIMathVector`**,
- `template<typename Type = double> class` **`PIMathMatrix.`**

Размеры динамических векторов и матриц можно менять после их объявления, однако шаблонные типы работают быстрее.

Векторно-матричный аппарат поддерживает следующие операции:

- умножение/деление вектора/матрицы на число;
- сумма/вычитание векторов/матриц;
- вектор * вектор – векторное произведение;
- вектор ^ вектор – скалярное произведение;
- умножение матрицы на матрицу;
- умножение матрицы на вектор;
- транспонирование и инвертирование матриц.

Для получения единичной матрицы нужного формата существует статичная функция identity(), например: «PIMathMatrixT<3u, 3u>::identity()». Вектора и матрицы имеют потоковые операторы вывода в std::ostream. Для удобства введены краткие именования типов:

- PIMathVectorT2i – PIMathVectorT<2u, int>;
- PIMathVectorT3i – PIMathVectorT<3u, int>;
- PIMathVectorT4i – PIMathVectorT<4u, int>;
- PIMathVectorT2d – PIMathVectorT<2u, double>;
- PIMathVectorT3d – PIMathVectorT<3u, double>;
- PIMathVectorT4d – PIMathVectorT<4u, double>;

- PIMathMatrixT22i – PIMathMatrixT<2u, 2u, int>;
- PIMathMatrixT33i – PIMathMatrixT<3u, 3u, int>;
- PIMathMatrixT44i – PIMathMatrixT<4u, 4u, int>;
- PIMathMatrixT22d – PIMathMatrixT<2u, 2u, double>;
- PIMathMatrixT33d – PIMathMatrixT<3u, 3u, double>;
- PIMathMatrixT44d – PIMathMatrixT<4u, 4u, double>;
- PIMathVectori – PIMathVector<int>;
- PIMathVectord – PIMathVector<double>;
- PIMathMatrixi – PIMathMatrix<int>;
- PIMathMatrixd – PIMathMatrix<double>.

Вычислитель

Для вычислений по формуле, заданной текстовой строкой и значениями переменных, существует класс **PIEvaluator**. Он оперирует с комплексными числами с точность double. Вычисление происходит в 3 этапа:

1. задание переменных (необязательно);
2. проверка формулы;
3. вычисление значения.

Функции для работы с переменными:

- int setVariable(const PIString & name, complexd value = 0.) – задает переменной name значение value, если переменной с таким именем нету, она создается. Возвращает индекс переменной для быстрого доступа в ней;
- void setVariable(int index, complexd value = 0.) – задает переменной с индексом index значение value;
- int variableIndex(const PIString & name) – получить индекс переменной с именем name;
- void removeVariable(const PIString & name) – удаляет переменную с именем name;
- void clearCustomVariables() – удаляет все заданные переменные.

Для задания формулы используется функция bool check(const PIString & string), которая возвращает истину если формула корректна и ложь, если некорректна. Преобразованную формулу можно получить функцией const PIString & expression(). Описание ошибки можно получить функцией const PIString & error(). Значение формулы вычисляется функцией complexd evaluate().

При задании функции создается список низкоуровневых инструкций, привязанных к переменным. Это позволяет использовать вычислитель для потоковых вычислений, когда по фиксированной формуле можно получать значения при различных значениях переменных. Т. е. после задания функции можно менять значения переменных и вновь вычислять значение выражения. При этом вычислитель выполняет низкоуровневые последовательные инструкции, а не парсит текстовую строку. Однако, такой метод негативно

сказывается на производительности при постоянно меняющихся формулах, т. к. каждый раз при изменении формулы необходимо вызывать функцию `bool check(const PIString & string)`.

Вычислитель сам по возможности расставляет скобки и знаки умножения, что можно увидеть по преобразованной формуле.

Вычислитель по умолчанию имеет 3 константы: `i`, `e`, `pi`.

Операторы, поддерживаемые вычислителем:

- `+` – сумма;
- `–` – вычитание;
- `*` – умножение;
- `/` – деление;
- `^` – возведение в степень;
- `%` – остаток от деления;
- `==` – равно;
- `!=` – не равно;
- `>` – больше;
- `<` – меньше;
- `>=` – не меньше;
- `<=` – не больше.

Функции, поддерживаемые вычислителем:

- `arcsin(x)` – арксинус;
- `arccos(x)` – арккосинус;
- `arctg(x)` – арктангенс;
- `arcctg(x)` – арккотангенс;
- `random(f, t)` – случайное число от `f` до `t`;
- `sin(x)` – синус;
- `cos(x)` – косинус;
- `tg(x)` – тангенс;
- `ctg(x)` – котангенс ;
- `exp(x)` – экспонента;
- `sh(x)` – гиперсинус;
- `ch(x)` – гиперкосинус;
- `th(x)` – гипертангенс;
- `cth(x)` – гиперкотангенс;
- `sqrt(x)` – квадратный корень;
- `sqr(x)` – квадрат;
- `pow(x, p)` – возведение `x` в степень `p`;
- `abs(x)` – модуль;
- `ln(x)` – натуральный логарифм;
- `lg(x)` – десятичный логарифм;
- `log(x, b)` – логарифм `x` по основанию `b`;
- `im(x)` – мнимая часть комплексного числа;
- `re(x)` – действительная часть комплексного числа;

- `arg(x)` – аргумент комплексного числа;
- `len(x)` – длина комплексного числа;
- `conj(x)` – комплексно-сопряженное число;
- `sign(x)` – знак числа (действительной части);
- `rad(x)` – градусы в радианы;
- `deg(x)` – радианы в градусы.

Например:

```
int main() {
     PIEvaluator eval;
     eval.check("elogeee");
     cout << eval.expression() << " = " << eval.evaluate()
<< endl;
};
```

В консоли будет «`e*log(e*e*e) = (8.15485,0)`».

```
int main() {
     PIEvaluator eval;
     int ind = eval.setVariable("x", 2.);
     eval.check("powx,2) + icosxpi");
     cout << eval.expression() << endl;
     cout << eval.content.variable(ind).value << ": " <<
eval.evaluate() << endl;
     eval.setVariable(ind, 3.);
     cout << eval.content.variable(ind).value << ": " <<
eval.evaluate() << endl;
};
```

В консоли будет «`pow(x,2)+i*cos(x*pi)`
`(2,0): (4,1)`
`(3,0): (9,-1)`».

Захват клавиатуры

Для захвата клавиатуры в консольных приложениях используется класс **PIKbdListener**. В конструкторе класса указывается функция возврата в формате `void function(char key, void * data)`, где `key` – нажатая клавиша, `data` – данные, переданные вторым аргументом в конструкторе или функцией `void setData(void * data)`. Для удобства существует статичная переменная `bool exiting`, которая управляется данным классом после вызова функции `void enableExitCapture(char key = 'Q')`. По указанной клавише статичная переменная сменится на истину. Для ожидания выхода существует макрос WAIT_FOR_EXIT.

Например:

```
void keyFunc(char key, void * ) {
     cout << "you press \'" << key << "\'" << endl;
```

```
};

PIKbdListener kl(keyFunc);

int main() {
     cout << "press \'Q\' for exit" << endl;
     kl.enableExitCapture();
     kl.start();
     WAIT_FOR_EXIT
};
```

Консоль

Для динамического автоматизированного форматированного отображения данных в консоль существует класс **PIConsole**. Этот класс позволяет выводить в консоль переменные различных типов по их указателям в несколько столбцов, группировать переменные по вкладкам, а также выполняет функции захвата и клавиатуры и ввода с нее строк.

Функция добавления, например, строки: addVariable(const PIString & name, PIString * ptr, int column = 1, PIFlags<PIConsole::Format> format = PIConsole::Normal), где name – имя, отображаемое слева от значения, ptr – указатель на переменную, column – столбец в текущей вкладке, format – формат вывода. По умолчанию переменная добавляется в первый столбец со стандартным форматом. Всего существует 14 функций addVariable для разных типов переменных:

- PIString;
- char;
- bool;
- short;
- int;
- long;
- llong;
- uchar;
- ushort;
- uint;
- ulong;
- ullong;
- float;
- double.

Для добавления строки без переменной используется функция addString(const PIString & name, int column = 1, PIFlags<PIConsole::Format> format = PIConsole::Normal).

Для добавления пустой строки используется функция addEmptyLine(int column = 1, uint count = 1), где count – количество добаляемых пустых строк, по умолчанию одна строка.

Для добавления битовых переменных используется функция addBitVariable(const PIString & name, void * ptr, int fromBit, int bitCount, int column = 1, PIFlags<PIConsole::Format> format = PIConsole::Normal), где ptr – указатель на переменную, содержащую битовое поле, fromBit – смещение битового поля относительно начала, bitCount – количество бит в поле.

Каждая вкладка содержит свой набор переменных, и переключение между ними происходит автоматически по нажатию соответствующей вкладке клавиши. Если консоль содержит несколько вкладок, то снизу автоматически формируется строка подсказки, которая содержит имена вкладок с клавишами переключения. Также есть возможность добавить собственную строку подсказки для текущей вкладки при помощи функции addCustomStatus(const PIString & str), удалить строку можно функцией clearCustomStatus(). После создания консоль имеет одну вкладку с именем «main» и клавишей переключения = 0.

Для добавления вкладки используется функция int addTab(const PIString & name, char bind_key = 0), где name – имя вкладки, bind_key – клавиша переключения, если = 0, то переключится на вкладку из консоли невозможно, а возможно только из кода. Функция возвращает индекс созданной вкладки.

Для переключения на вкладку из кода используются функции:

- bool setTab(uint index) – переключиться по индексу;
- bool setTab(const PIString & name) – переключиться по имени. Если такой вкладки нету, то функции вернут ложь.

Формат формируется из перечисления PIConsole::Format:

- Normal – стандартный формат;
- Bold – полужирный;
- Faint – затененный (почти нигде не работает);
- Italic – курсив (почти нигде не работает);
- Underline – подчеркнутый;
- Blink – мерцающий;
- Inverse – поменять местами цвета фона и текста;
- Black – черный текст;
- Red – красный текст;
- Green – зеленый текст;
- Yellow – желтый текст;
- Blue – синий текст;
- Magenta – пурпурный текст;
- Cyan – голубой текст;
- White – белый текст;
- BackBlack – черный фон;
- BackRed – красный фон;
- BackGreen – зеленый фон;
- BackYellow – желтый фон;
- BackBlue – синий фон;
- BackMagenta – пурпурный фон;

- BackCyan – голубой фон;
- BackWhite – белый фон;
- Dec – десятичный формат (целые числа);
- Hex – шестнадцатеричный формат (целые числа);
- Oct – восьмеричный формат (целые числа);
- Scientific – научный формат (*E±*, числа с плавающей точкой).

В конструкторе консоли ей можно передать указатель на функцию возврата, также как и классу захвата клавиатуры. Консоль предоставляет такой же функционал ожидания выхода, как и класс захвата клавиатуры. Функция запуска консоли `void start(bool wait = false)` имеет один аргумент `wait` – при истине эта функция становится блокирующей до тех пор, пока не будет нажата клавиша выхода.

Пример использования:

```
struct BitsStruct {
     uchar b1: 1;
     uchar b2: 2;
     uchar b4: 4;
};

int i = 1;
BitsStruct bits;

void keyFunc(char key, void * ) {
     switch (key) {
     case '-': i--; break;
     case '+': i++; break;
     case '[': bits.b1--; break;
     case ']': bits.b1++; break;
     case ';': bits.b2--; break;
     case '\'': bits.b2++; break;
     case ',': bits.b4--; break;
     case '.': bits.b4++; break;
     }
};

PIConsole c(false, keyFunc);

int main() {
     c.addString("PIConsole example");
     c.addEmptyLine();

     c.addTab("colors", '1');
     c.addString("Red string", 1, PIConsole::Red);
     c.addString("Blue on yellow", 1, PIConsole::Blue |
PIConsole::BackYellow);
     c.addEmptyLine();
```

```
    c.addTab("columns", '2');
    c.addString("column 1", 1, PIConsole::BackYellow);
    c.addString("column 2", 2, PIConsole::BackCyan);
    c.addString("column 3", 3, PIConsole::BackMagenta);
    c.addEmptyLine();

    c.addTab("bits", '3');
    c.addBitVariable("b1", &bits, 0, 1);
    c.addBitVariable("b2", &bits, 1, 2);
    c.addBitVariable("b4", &bits, 3, 4);
    c.addCustomStatus("[] - -/+ b1, ;\' - -/+ b2, ,. - -/+
b4");
    c.addEmptyLine();

    c.addTab("formats", '4');
    c.addVariable("our int", &i);
    c.addVariable("dec", &i);
    c.addVariable("oct", &i, 1, PIConsole::Oct);
    c.addVariable("hex", &i, 1, PIConsole::Hex);
    c.addCustomStatus("-+ - -/+ number");
    c.addEmptyLine();

    c.setTab(0);
    c.enableExitCapture();
    c.start(true);
};
```

Эта программа наглядно демонстрирует возможности отображения в консоль форматированных строк и данных, использования вкладок, обработки нажатия клавиш и ожидания выхода.

Конфигурационный файл

Для получения данных из файлов формата, близкого в ini, существует класс **PIConfig**. Этот класс читает и записывает файлы, состоящие из строк следующего формата: «[пробелы] имя = значение [#[тип(один символ)] комментарий]».

Все данные внутри класса хранятся в виде дерева. Пустое дерево содержит один узел «root». Каждый узел дерева имеет имя и может содержать значение, тип и комментарий, или массив дочерних узлов. В текстовом файле имя параметра может быть записано в виде последовательности имен узлов, разделенных символом или строкой разделителя, по умолчанию это точка.

Например, строка «n1.n2.n3 = 1» интерпретируется как узел n3, который является дочерним к узлу n2, который дочерний от узла n1. В этом случае узел n3 имеет значение, тип и комментарий, а узлы n1 и n2 имеют массивы дочерних узлов.

Для описания узла и массива узлов введены внутренние классы

PIConfig::**Entry** и PIConfig::**Branch**.

Класс PIConfig::Entry имеет имя, тип, комментарий, указатель на узел-родитель и массив дочерних узлов в виде экземпляра PIConfig::Branch. Также этот класс имеет перегруженный оператор неявного преобразования в следующие типы:

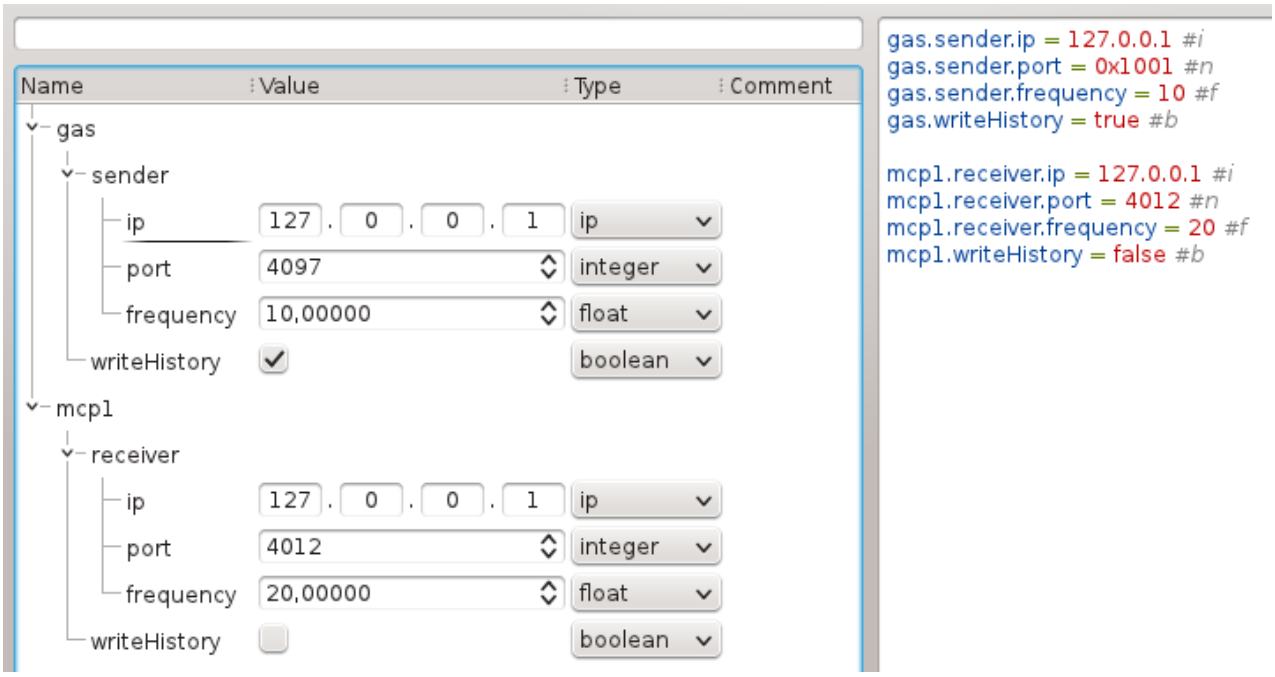
- bool;
- char;
- short;
- int;
- long;
- uchar;
- ushort;
- uint;
- ulong;
- float;
- double;
- PIString;
- PIStringList.

Класс PIConfig::Branch наследован от PIVector<Entry * > и имеет функции для поиска узлов по дочерним узлам содержимого.

Основные функции класса PIConfig и PIConfig::Entry:

- PIConfig::Entry & getValue(const PIString & vname, const PIString & def = PIString(), bool * exist = 0) и ещё 12 подобных функций с различными типами def. Функция возвращает узел с путем vname или пустой узел со значением def, если не находит узла по пути vname. В переменную exist записывается был ли найден узел.
- void setValue(const PIString & name, const PIString & value, const PIString & type = "s", bool write = true) общего вида и void setValue(const PIString & name, const char * value, bool write = true) и еще 10 функций с различными типами value. Первая функция записывает узел по пути name со значением value и типом type. Если write истина, то сразу же производится запись в файл. Второй вариант функций без аргумента type, он определяется в зависимости от типа value.
  Если указанного узла нету, он создается.

Лучше всего соответствие между текстовым содержимым файла и древовидной структурой можно отразить на рисунке:



Например:

```
PIConfig conf("file.conf");
cout << conf.allTree();
```

В консоли будет:

```
«gas
   sender
     ip = 127.0.0.1
     port = 0x1001
     frequency = 10
   writeHistory = true
 mcp1
   receiver
     ip = 127.0.0.1
     port = 4012
     frequency = 20
   writeHistory = false»
```

Еще:

```
PIConfig conf("file.conf");
PIConfig::Entry gas = conf.getValue("gas");
cout << gas.getValue("sender.ip") << endl;
cout << conf.getValue("gas.sender.ip") << endl;
cout << conf.getValue("gas").getValue("sender.ip") << endl;
cout << conf.getValue("gas.sender").getValue("ip") << endl;
```

В консоли будет:

```
«127.0.0.1
 127.0.0.1
 127.0.0.1
 127.0.0.1»
```

Еще:

```
PIConfig conf("file .conf");
PIConfig::Branch ports = conf.allLeaves().filter("port");
cout << ports << endl;
```

В консоли будет:
«port = 0x1001
 port = 4012»

Еще:

```
PIString ip;
int port;
PIConfig conf("protocols.conf~");
ip = conf.getValue("mcp1.receiver.ip");
port = conf.getValue("mcp1.receiver.port");
cout << ip << ":" << port << endl;
```

В консоли будет: «127.0.0.1:4012»

Последовательный порт

Работу с последовательным портом в PIP реализует класс **PISerial**. В конструкторе класса указывается имя устройства (например, на Windows – «COM1», на Linux – «/dev/ttyS1», на QNX – «/dev/ser1»), данные, передаваемые в функции возврата, и функция фозврата, вызываемая после приема данных через порт. Функция возврата имеет формат `bool function(void * data, char * rec, int size)`, где `data` – указанные данные, `rec` – принятые данные, `size` – размер принятых данных.

Например:

```
PIString our_data = "test string";

bool recFunc(void * data, char * recData, int recSize) {
     cout << "our data: " << *((PIString*)data) << endl;
     cout << "received " << recSize << " bytes" << endl;
};

PISerial serial("/dev/ttyS0", &our_data, recFunc);

int main() {
     serial.setSpeed(PISerial::S115200);
     serial.setParameters(PISerial::ParityControl |
PISerial::TwoStopBits);
     serial.start();
     FOREVER_WAIT
};
```

Эта программа инициализирует прием по устройству "/dev/ttyS0"

со скоростью 115200 бод, контролем четности и двумя стоповыми битами. Также в качестве данных указывается строка. После приема данных в консоль сообщаются данные (строка) и количество принятых байт.

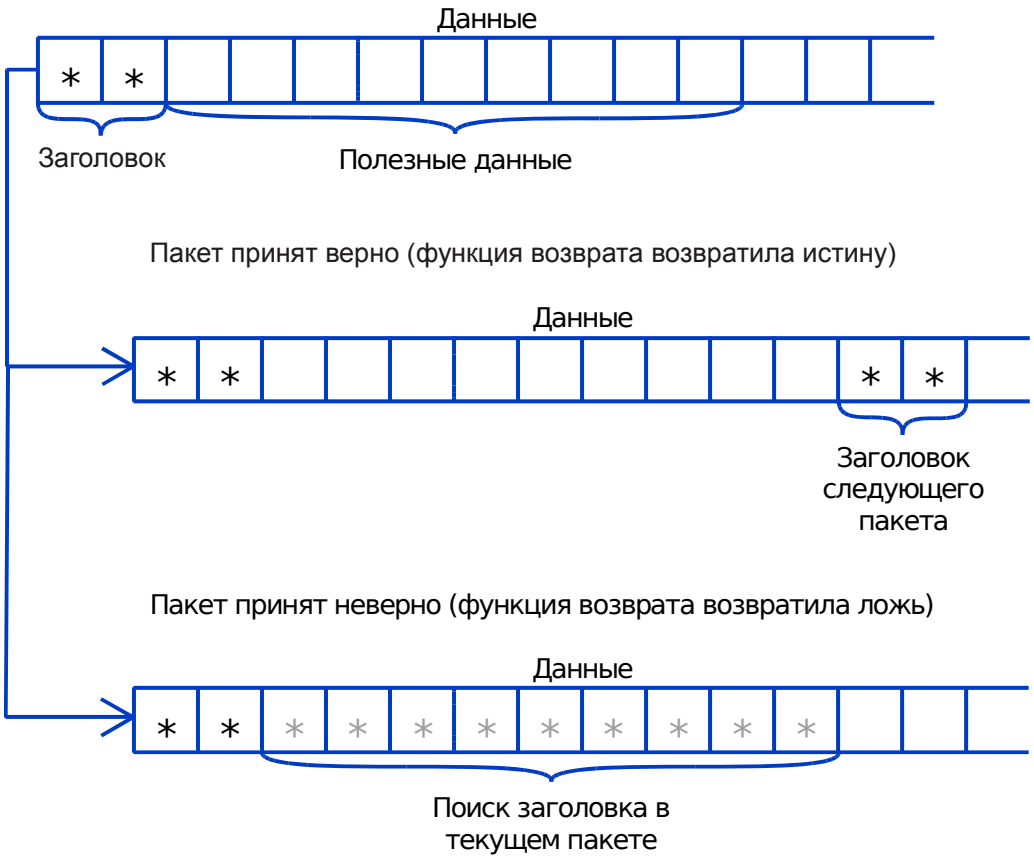
Класс имеет возможность приема данных с фиксированным заголовком. Обычно при передаче по последовательному порту первые несколько байт используют как признак начала пакета. Для включения анализа принимаемых данных используется функция

`void setReadData(void * headerPtr, int headerSize, int dataSize)`, где

- `headerPtr` – указатель на заголовок пакета;
- `headerSize` – размер заголовка пакета;
- `dataSize` – размер полезных данных.

Если после приема пакета функция возврата возвратила ложь, то класс считает, что неправильно определил начало пакета, и далее ищет в этом же пакете признак начала пакета.

Это хорошо иллюстрирует следующий рисунок:



Сетевой интерфейс

Работу с сетью в PIP реализует класс **PIEthernet**, который очень похож на PISerial. Только в конструкторе вместо устройства указывается IP и порт для прослушки.

Сеть работает по протоколу UDP.

Протокол

Базовый класс **PIProtocol** реализует прием-передачу данных по заданным каналам, отслеживает качество связи и имеет виртуальные функции для контроля информационного обмена. Он содержит экземпляры классов PISerial и PIEthernet и может осуществлять прием и передачу данных независимо друг от друга через последовательный порт или по сети.

Качество связи оценивается с помощью таймера, работающего по ожидаемой частоте приема, и имеет тип PIProtocol::Quality {Unknown = 1, Failure = 2, Bad = 3, Average = 4, Good = 5} – неизвестно, нету связи, плохая связь, средняя, хорошая.

В классе имеется 2 типа частоты: мгновенная и интегральная. Мгновенная частота вычисляется по задержке последнего принятого пакета, а интегральная вычисляется по определенному количеству ожидаемых пакетов. Это количество определяется из ожидаемой частоты и равно 3*частоту, но не менее 10. По этому же количеству ожидаемых пакетов определяется качество связи: до получения первого пакета это Unknown, при полностью потерянной связи это Failure, при ≤ 20% принятых пакетов от ожидаемых это Bad, при > 20% и ≤ 80% это Average, при ≥ 80% это Good.

Класс имеет и пустой конструктор, но лучше всего использовать второй: PIProtocol(const PIString & config, const PIString & name, void * recHeaderPtr = 0, int recHeaderSize = 0, void * recDataPtr = 0, int recDataSize = 0, void * sendDataPtr = 0, int sendDataSize = 0), где:

- config – путь к конфигурационному файлу;
- name – имя протокола;
- recHeaderPtr – указатель на заголовок принимаемой структуры;
- recHeaderSize – размер заголовка принимаемой структуры;
- recDataPtr – указатель на принимаемую структуру;
- recHeaderSize – размер принимаемой структуры;
- sendDataPtr – указатель на передаваемую структуру;
- sendDataSize – размер передаваемой структуры.

В файле config должен быть узел с именем name, в котором описан протокол:

- в узле «receiver» описывается приемник;
- в узле «sender» описывается передатчик.

Для использования последовательного порта описываются параметры:

- обязательные:
  - «device» – устройство COM-порта, строковый;
  - «speed» – скорость обмена, бод;
- необязательные:
  - «parity» – включить контроль четности, логический;
  - «twoStopBits» – два стоповых бита вместо одного, логический.

Для использования сети описываются параметры:

- обязательные:
  - «ip» – IP адрес;
  - «port» – порт;

Также в узлах «receiver» и «sender» можно задать значение «frequency»: для приемника это необходимый параметр для оценки качества связи, для передатчика это позволяет использовать собственный таймер для отправки пакетов.

В случае использования для приема и передачи последовательного порта можно описывать только узел «receiver», для передатчика будет использоваться это же устройство с теми же параметрами. Аргументы заголовка, заданные в конструкторе, передаются объекту PISerial для наилучшего приема данных.

Если заголовочные аргументы ненулевые и используется сетевой обмен, то считается, что указатель начала структуры для приема – это указатель на заголовок (recHeaderPtr), а размер принимаемой структуры – это сумма размеров заголовка и принимаемой структуры (recHeaderSize + recHeaderSize).

Пример описания протокола в конфигурационном файле:

```
mcp1.receiver.ip = 127.0.0.1 #i
mcp1.receiver.port = 4012 #n
mcp1.receiver.frequency = 20 #f
mcp1.sender.ip = 192.168.0.190 #i
mcp1.sender.port = 4013 #n
mcp1.sender.frequency = 20 #f
```

В этом случае протокол «mcp1» будет принимать и посылать пакеты по сети с частотой 20 Гц.

Другой пример:

```
ts.receiver.device = /dev/ttyS0 #s
ts.receiver.speed = 57600 #n
ts.receiver.parity = false #b
ts.receiver.twoStopBits = false #b
```

Здесь протокол «ts» будет использовать для приема и передачи пакетов COM-порт «/dev/ttyS0», настроенный на скорость 57600 бод, без контроля четности и с одним стоповым битом.

Для управления приемом и передачей данных в классе-наследнике от PIProtocol можно переопределить следующие виртуальные функции:

- bool receive(char * data, int size) – выполняется сразу после приема данных, до проверки на корректность. По умолчанию копирует данные data в заданный указатель на принимаемую структуру;
- bool validate() – выполняется после приема данных и функции receive, в ней можно проверить данные на корректность, например, проверить поля принятой структуры на правильность, сверить контрольную сумму. Если данные приняты корректно, необходимо вернуть истину, иначе ложь. От возвращаемого значения зависит также контроль качества связи. По умолчанию возвращается истина;
- bool aboutSend() – выполняется непосредственно перед передачей

данных. Если вернуть ложь, то передача не осуществится. По умолчанию возвращается истина;

- `uint checksum_i(void * data, int size)` – функция подсчета контрольной суммы, удобно использовать при проверки корректности приема. По умолчанию это побитно инвертированная побайтовая сумма данных `data + 1`;
- `uchar checksum_c(void * data, int size)` – то же, что и выше, только сумма считается в переменной типа `uchar`.

Пример класса-наследника:

```
class MV2: public PIProtocol {
public:
     MV2(const PIString & config): PIProtocol(config, "mv2",
0, 0, &from_mv2, sizeof(from_mv2), &to_mv2, sizeof(to_mv2))
{
          PIConfig conf(config, PIFile::Read);
          startReceive();
          startSend();
     }

     fromMV2 from_mv2;
     toMV2 to_mv2;

private:
     bool validate() {
          if (from_mv2.checksum != checksum_c(&from_mv2,
sizeof(from_mv2) - 1)) return false;
          return true;
     }
     bool aboutSend() {
          to_mv2.checksum = checksum_c(&to_mv2,
sizeof(to_mv2) - 1);
          return true;
     }

};
```

Это не рабочий пример, а демонстрация создания своего протокола переопределением `PIProtocol`. `fromMV2` и `toMV2` – это структуры приема и передачи данных, они здесь не объявлены. Для работы экземпляра данного класса необходимо всего лишь создать его экземпляр в рабочей программе. После создания объекта внутри него запустятся два таймера и один поток: таймеры качества связи и отправки пакетов и поток приема.

Для получения информации о протоколе существуют функции:

- `float immediateFrequency()` – мгновенная частота приема;
- `float integralFrequency()` – интегральная частота приема;
- `ullong receiveCount()` – количество корректно принятых пакетов;
- `ullong wrongCount()` – количество некорректно принятых пакетов;
- `ullong sendCount()` – количество отправленных пакетов;

- PIProtocol::Quality quality() – качество связи;
- PIString receiverDeviceState() – состояние приемника;
- PIString senderDeviceState() – состояние передатчика.

Перечисленные выше функции для удобного интегрирования с консолью PIConsole имеют аналоги с приставкой «_ptr», которые возвращают соответствующие указатели, которые можно передавать функциям addVariable. Также PIProtocol имеет функции PIString receiverDeviceName() и PIString senderDeviceName(), которые возвращают полные имена приемника и передатчика. Для COM-портов это имя формата «/dev/ttyS0», для сети – «127.0.0.1:1024».

Сигналы

Приложения могут принимать сигналы, например SIGINT, генерируемый терминалом при нажатии Ctrl+C. По умолчанию за реакцию на сигналы отвечает системная библиотека, но с помощью класса **PISignals** можно переопределить обработчик нужных сигналов на свой. Этот класс определяет перечисление сигналов enum Signal:

- Interrupt – прерывание с клавиатуры (Ctrl+C);
- Illegal – неверная инструкция;
- Abort – ;
- FPE – исключения плавающей точки;
- SegFault – ошибка сегментирования;
- Termination – завершение;

  Далее только для *NIX/QNX систем:
- Hangup – отсоединение от текущего терминала;
- Quit – выход (Ctrl+\);
- Kill – убить процесс;
- BrokenPipe – некорректный pipe;
- Timer – сигнал таймеры системы;
- UserDefined1 – определенный пользователем 1;
- UserDefined2 – определенный пользователем 2;
- ChildStopped – дочерний процесс остановлен или завершен;
- Continue – продолжить выполнение;
- StopProcess – остановить выполнение;
- StopTTY – остановить вывод в терминал;
- StopTTYInput – остановить ввод из терминала для дочерних процессов;
- StopTTYOutput – остановить вывод в терминал для дочерних процессов;

  Для всех систем:
- All – все сигналы.

Этот класс используется только как пространство имен, весь функционал реализован через статические функции:

- setSlot(SignalEvent slot) – установить свою функцию-обработчик сигналов, SignalEvent это указатель на функцию формата void function(PISignals::Signal);

- grabSignals(PIFlags<PISignals::Signal> signals) – переопределить обработчик указанных сигналов на свой;
- raiseSignal(PISignals::Signal signal) – вызвать сигнал signal.

Пример:

```
void signalFunc(PISignals::Signal signal) {
     if (signal == PISignals::Interrupt) cout << "Ctrl+C
pressed" << endl;
};

int main() {
     PISignals::setSlot(signalFunc);
     PISignals::grabSignals(PISignals::Interrupt);
     FOREVER_WAIT
};
```

Анализ командной строки

Приложению часто передают некоторые ключи и аргументы при запуске, из командной строки. Для разбора ключей можно использовать int argc, char * argv[] функции main, но для автоматизации этого процесса создан класс **PICLI** (Command Line Interface). В конструкторе ему передаются аргументы функции main – int argc и char * argv[].

Ключи аргумента могут быть полными и короткими, и начинаются с определенной приставки (по умолчанию «--» и «-» соответственно). Несколько коротких ключей могут идти слитно и иметь одну общую приставку в начале. Каждый аргумент может иметь значение, которое должно следовать сразу после ключа через пробел.

Для добавления аргумента используется функция addArgument. Первый аргумент функции – это имя аргумента, последний (необязательный) – это логическое значение есть ли у аргумента значение, по умолчанию ложь. Вторым и третьим аргументами могут идти короткий и полный ключи аргумента, по умолчанию полный ключ совпадает с именем аргумента, а короткий ключ – первая буква полного имени ключа. Например:

- addArgument("debug") – полный ключ = "debug", короткий = "d";
- addArgument("debug", "D") – полный ключ = "debug", короткий = "D";
- addArgument("debug", "D", "Dbg") – полный ключ = "Dbg", короткий = "D";

Для проверки вхождения аргумента в команду запуска приложения используется функция bool hasArgument(const PIString & name), где name – имя аргумента.

Для получения значения аргумента используется функция PIString argumentValue(const PIString & name), где name – имя аргумента.

Для изменения полной и короткой приставок используются функции setShortKeyPrefix и setFullKeyPrefix.

Пример:

```
int main(int argc, char * argv[]) {
     PICLI cli(argc, argv);
     cli.addArgument("debug");
     cli.addArgument("value", "V", "Val", true);
     if (cli.hasArgument("debug"))
          cout << "has debug" << endl;
     if (cli.hasArgument("value"))
          cout << "value = " << cli.argumentValue("value") <<
endl;
};
```

Результаты запуска данной программы:

```
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test -d
     has debug
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test --debug
     has debug
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test -V
     value =
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test -V 123
     value = 123
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test --Val 123
     value = 123
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test --Va 123
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test --Va 123 --debug
     has debug
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test -d --Val 123
     has debug
     value = 123
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test -dV 123
     has debug
     value = 123
peri4@peri4-gentoo ~/pprojects/pip $ ./pip_test -Vd 123
     has debug
     value =
```

Процесс

Для запуска и контроля процессов существует класс **PIProcess**. Класс может запускать процесс, ожидать его завершения, управлять процессом, перехватывать его ввод и вывод.

Процесс не завершается, если запустивший его экземпляр класса PIProcess перестал существовать.

Для запуска процесса с текущим окружением и в текущей рабочей директории используются функции:

- `void exec(const PIString & program)`, где program – имя запускаемого файла;
- `void exec(const PIString & program, const PIString & arg)`, где arg– аргумент командной строки;

- exec(const PIString & program, const PIString & arg1, const PIString & arg2);
- exec(const PIString & program, const PIString & arg1, const PIString & arg2, const PIString & arg3);
- exec(const PIString & program, const PIStringList & args), где args – список аргументов командной строки.

Для изменения окружения процесса используются функции:

- PIStringList environment() – вернуть текущее окружение дочернего процесса;
- static PIStringList currentEnvironment() – вернуть текущее окружения главного приложения;
- clearEnvironment() – очистить окружение;
- removeEnvironmentVariable(const PIString & variable) – удалить из окружения переменную с именем variable;
- setEnvironmentVariable(const PIString & variable, const PIString & value) – установить переменной окружения с именем variable значение value, если такой переменной нет, она создается.

Для изменения рабочей директории используются функции:

- PIString workingDirectory() – вернуть текущую рабочую директория или пустую строку, если используется рабочая директория главного приложения;
- void setWorkingDirectory(const PIString & path) – установить рабочую директория path;
- void resetWorkingDirectory() – сбросить рабочую директорию.

Для захвата ввода/вывода процесса используются функции:

- void setGrabInput(bool yes) – разрешить захват ввода;
- void setInputFile(const PIString & path) – установить ввод как файл path;
- void unsetInputFile() – установить ввод как внутренний буфер PIProcess.

Также есть функции аналогичные этим трем для управления потоками вывода и ошибок, вместо «Input» в их названиях «Output» и «Error».

Для ожидания завершения процесса используется функция bool waitForFinish(int timeout_msecs = 60000), где timeout_msecs – количество миллисекунд для ожидания завершения, -1 для бесконечного ожидания, по умолчанию 1 минута. Если функция дождалась завершения процесса, то возвращается истина, если истекло время ожидания, то возвращается ложь.

Для получения содержимого перехваченных потоков вывода и ошибок используются функции PIByteArray readOutput() и PIByteArray readError().

Описание классов

```
PIFlags<Type>

public:
    PIFlags()
    PIFlags(Enum e)
    PIFlags(const PIFlags & f)
    PIFlags(const int i)
    void           operator =(const PIFlags & f)
    void           operator =(const Enum & e)
    void           operator =(const int & i)
    void           operator |=(const PIFlags & f)
    void           operator |=(const Enum & e)
    void           operator |=(const int i)
    void           operator &=(const PIFlags & f)
    void           operator &=(const Enum & e)
    void           operator &=(const int i)
    PIFlags &      operator |(PIFlags f) const
    PIFlags &      operator |(Enum e) const
    PIFlags &      operator |(int i) const
    PIFlags &      operator &(PIFlags f) const
    PIFlags &      operator &(Enum e) const
    PIFlags &      operator &(int i) const
    bool           operator [](Enum e) – проверка на вхождение e
                   operator int() const – неявное преобразование в
         int
```

```
PIVector<Type>

наследован от std::vector<Type>
public:
    PIVector()
    PIVector(const Type & value)
    PIVector(const Type & v0, const Type & v1)
    PIVector(const Type & v0, const Type & v1, const Type & v2)
    PIVector(const Type & v0, const Type & v1, const Type & v2,
         const Type & v3)
    PIVector(uint size, const Type & value = Type())
    const Type &   at(uint index) const
    Type &         at(uint index)
    const Type *   data(uint index = 0) const
    Type *         data(uint index = 0)
    int            size_s() const
    bool           isEmpty() const
    PIVector &     fill(const Type & t)
    PIVector &     pop_front()
    PIVector &     push_front(const Type & t)
    PIVector &     remove(uint num)
    PIVector &     remove(uint num, uint count)
    PIVector &     remove(const Type & t) – удаление всех элементов
         равных t
    PIVector &     operator <<(const Type & t) – добавление
         элемента t
    PIVector &     operator <<(const PIVector & t) – добавление
         вектора t
    bool           contain(const Type & v) const – проверка на
         вхождение элемента v
```

PIList<Type>

наследован от std::list<Type>
public:
    PIList()
    PIList(const Type & value)
    PIList(const Type & v0, const Type & v1)
    PIList(const Type & v0, const Type & v1, const Type & v2)
    PIList(const Type & v0, const Type & v1, const Type & v2,
        const Type & v3)
    PIList(uint size, const Type & value = Type())
    const Type * ***data***(uint index = 0) const
    Type *   ***data***(uint index = 0)
    int   ***size_s***() const
    bool   ***isEmpty***() const
    PIList &   ***fill***(const Type & t)
    PIList &   ***remove***(uint num)
    PIList &   ***remove***(uint num, uint count)
    PIList &   ***insert***(uint pos, const Type & t)
    PIList &   ***operator <<***(const Type & t) – добавление
        элемента t
    PIVector   ***toVector***()

PISet<Type>

наследован от std::set<Type>
public:
    PISet()
    PISet(const Type & value)
    PISet(const Type & v0, const Type & v1)
    PISet(const Type & v0, const Type & v1, const Type & v2)
    PISet(const Type & v0, const Type & v1, const Type & v2, const
        Type & v3)
    int   ***size_s***() const
    bool   ***isEmpty***() const
    PISet &   ***remove***(uint num)
    PISet &   ***remove***(uint num, uint count)
    PISet &   ***operator <<***(const Type & t) – добавление элемента t
    bool   ***operator []***(const Type & t) – проверка на вхождение
        t
    PIVector ***toVector***()

PIStack<Type>

наследован от PIVector<Type>
public:
    PIStack()
    PIStack(const Type & value)
    PIStack(const Type & v0, const Type & v1)
    PIStack(const Type & v0, const Type & v1, const Type & v2)
    PIStack(const Type & v0, const Type & v1, const Type & v2,
         const Type & v3)
    PIStack &   ***push***(const Type & v) – положить в стек v
    Type   ***pop***() – взять из стека
    Type &   ***top***() – вершина стека
    PIVector   ***toVector***()

```
PIDeque<Type>

наследован от std::deque<Type>
public:
    PIDeque()
    PIDeque(const Type & value)
    PIDeque(const Type & v0, const Type & v1)
    PIDeque(const Type & v0, const Type & v1, const Type & v2)
    PIDeque(const Type & v0, const Type & v1, const Type & v2,
         const Type & v3)
    int                 size_s() const
    bool                isEmpty() const
    PIDeque &           operator <<(const Type & t) – добавление
         элемента t
    PIVector<Type>      toVector()
```

PIQueue<Type>

наследован от PIDeque<Type>
public:
    PIQueue()
    PIQueue(const Type & value)
    PIQueue(const Type & v0, const Type & v1)
    PIQueue(const Type & v0, const Type & v1, const Type & v2)
    PIQueue(const Type & v0, const Type & v1, const Type & v2,
        const Type & v3)
    PIQueue &   ***enqueue***(const Type & v) — поставить в
        очередь v
    Type   ***dequeue***() — взять из очереди
    const Type &   ***head***() const — голова очереди
    PIVector<Type>   ***toVector***()

```
PIHash<Type, Key>

наследован от PISet<PIPair<Key, Type> >
public:
    PIHash()
    PIHash(const Type & value, const Key & key)
    PIHash &  insert(const Type & value, const Key & key)
    Type      value(Key key)
    Type      operator [](Key key)
```

PIByteArray

наследован от PIVector<uchar>
public:
```
     PIByteArray()
     PIByteArray(const uint size)
     PIByteArray(const char * data, const uint size)
     PIByteArray(const uchar * data, const uint size)
     PIByteArray &  convertToBase64()
     PIByteArray &  convertFromBase64()
     PIByteArray &  compressRLE(uchar threshold = 192)
     PIByteArray &  decompressRLE(uchar threshold = 192)
     PIByteArray    toBase64()
     PIByteArray    fromBase64()
     PIByteArray    compressedRLE(uchar threshold = 192)
     PIByteArray    decompressedRLE(uchar threshold = 192)
```

PIBitArray

public:

```
     PIBitArray()
     PIBitArray(const int & size)
     PIBitArray(uchar val)
     PIBitArray(ushort val)
     PIBitArray(uint val)
     PIBitArray(ulong val)
     PIBitArray(ullong val)
     PIBitArray(uchar * bytes, uint size)
     uint           bitSize() const
     uint           byteSize() const
     PIBitArray &   resize(const uint & size)
     PIBitArray &   clearBit(const uint & index)
     PIBitArray &   setBit(const uint & index)
     PIBitArray &   writeBit(const uint & index, const bool &
value)
     PIBitArray &   writeBit(const uint & index, const uchar &
value)
     PIBitArray &   insert(const uint & index, const bool & value)
     PIBitArray &   insert(const uint & index, const uchar & value)
     PIBitArray &   push_back(const bool & value)
     PIBitArray &   push_back(const uchar & value)
     PIBitArray &   push_front(const bool & value)
     PIBitArray &   push_front(const uchar & value)
     PIBitArray &   pop_back()
     PIBitArray &   pop_front()
     PIBitArray &   append(const PIBitArray & ba)
     uchar *        data()
     uchar          toUChar()
     ushort         toUShort()
     uint           toUInt()
     ulong          toULong()
     ullong         toULLong()
     bool           at(const uint & index) const
     bool           operator [](const uint & index) const
     void           operator +=(const PIBitArray & ba)
     bool           operator ==(const PIBitArray & ba) const
     bool           operator !=(const PIBitArray & ba) const
     void           operator =(const uchar & val)
     void           operator =(const ushort & val)
     void           operator =(const uint & val)
     void           operator =(const ulong & val)
     void           operator =(const ullong & val)
```

PIChar

public:

```
     PIChar()
     PIChar(const char c)
     PIChar(const short c)
     PIChar(const int c)
     PIChar(const uchar c)
     PIChar(const ushort c)
     PIChar(const uint c)
     PIChar(const char * c)
     PIChar &            operator =(const char v)
     bool                operator ==(const PIChar & o) const
     bool                operator !=(const PIChar & o) const
     bool                operator >(const PIChar & o) const
     bool                operator <(const PIChar & o) const
     bool                operator >=(const PIChar & o) const
     bool                operator <=(const PIChar & o) const
     bool                isDigit() const
     bool                isGraphical() const
     bool                isControl() const
     bool                isLower() const
     bool                isUpper() const
     bool                isPrint() const
     bool                isSpace() const
     bool                isAlpha() const
     bool                isAscii() const
     int                 toInt() const
     const wchar_t *     toWCharPtr() const
     const char *        toCharPtr() const
     const wchar_t       toWChar() const
     char                toAscii() const
     int                 unicode16Code() const
     PIChar              toUpper() const
     PIChar              toLower() const
```

PIString

наследован от PIVector<PIChar>
public:
```
     PIString()
     PIString(const PIChar c)
     PIString(const char * str)
     PIString(const wchar_t * str)
     PIString(const string & str)
     PIString(const wstring & str)
     PIString(const PIByteArray & ba)
     PIString(const char * str, const int len)
     PIString(const int len, const char c)
     PIString(const int len, const PIChar & c)
     PIString(const PIString & str)
     PIString &     operator +=(const PIChar c)
     PIString &     operator +=(const char * str)
     PIString &     operator +=(const wchar_t * str)
     PIString &     operator +=(const string & str)
     PIString &     operator +=(const PIByteArray & ba)
     PIString &     operator +=(const PIString & str)
     PIString &     operator +=(const wstring & str)
     bool           operator ==(const PIString & str) const;
     bool           operator !=(const PIString & str) const
     bool           operator <(const PIString & str) const
     bool           operator >(const PIString & str) const;
     bool           operator <=(const PIString & str) const
     bool           operator >=(const PIString & str) const
     PIString &     operator <<(const PIString & str)
     PIString &     operator <<(const PIChar c)
     PIString &     operator <<(const char * str)
     PIString &     operator <<(const wchar_t * str)
     PIString &     operator <<(const string & str)
     PIString &     operator <<(const int & num)
     PIString &     operator <<(const short & num)
     PIString &     operator <<(const long & num)
     PIString &     operator <<(const float & num)
     PIString &     operator <<(const double & num)
     PIString       mid(const int start, const int len = -1) const
     PIString       left(const int len) const
     PIString       right(const int len) const
     PIString &     cutMid(const int start, const int len)
     PIString &     cutLeft(const int len)
     PIString &     cutRight(const int len)
     PIString &     trim()
     PIString       trimmed() const
     PIString &     replace(const int from, const int count, const
PIString & with)
     PIString       replaced(const int from, const int count, const
PIString & with) const
```

```
     PIString &     replace(const PIString & what, const PIString &
with, bool * ok = 0)
     PIString       replaced(const PIString & what, const PIString
& with, bool * ok = 0) const
     PIString &     replaceAll(const PIString & what, const
PIString & with)
     PIString       replaceAll(const PIString & what, const
PIString & with) const
     PIString &     insert(const int index, const PIChar & c)
     PIString &     insert(const int index, const char & c)
     PIString &     insert(const int index, const PIString & str)
     PIString &     insert(const int index, const char * c)
     PIString &     expandRightTo(const int len, const PIChar & c)
     PIString &     expandLeftTo(const int len, const PIChar & c)
     const char *   data()
     const string   stdString() const
     wstring        stdWString() const
     PIByteArray    toByteArray()
     PIStringList   split(const PIString & delim) const – разбить
строку по разделителю delim на массив строк
     PIString       toUpperCase() const
     PIString       toLowerCase() const
     PIString       toNativeDecimalPoints() const
     int            find(const char str, const int start = 0) const
     int            find(const PIString str, const int start = 0)
const
     int            find(const char * str, const int start = 0)
const
     int            find(const string str, const int start = 0)
const
     int            findLast(const char str, const int start = 0)
const
     int            findLast(const PIString str, const int start =
0) const
     int            findLast(const char * str, const int start = 0)
const
     int            findLast(const string str, const int start = 0)
const
     int            length() const
     bool           isEmpty() const
     bool           toBool() const
     char           toChar() const
     short          toShort() const
     int            toInt() const
     long           toLong() const
     llong          toLLong() const
     float          toFloat() const
     double         toDouble() const
     ldouble        toLDouble() const
     PIString &     setNumber(const int value)
```

PIString & ***setNumber***(const short value)
PIString & ***setNumber***(const long value)
PIString & ***setNumber***(const float value)
PIString & ***setNumber***(const double value)
PIString & ***setNumber***(const ldouble value)

статичные функции:
PIString ***fromNumber***(const int value)
PIString ***fromNumber***(const uint value)
PIString ***fromNumber***(const short value)
PIString ***fromNumber***(const long value)
PIString ***fromNumber***(const float value)
PIString ***fromNumber***(const double value)
PIString ***fromNumber***(const ldouble value)
PIString ***fromBool***(const bool value)

PIStringList

наследован от PIVector<PIString>
public:
     PIStringList()
     PIStringList(const PIString & str)
     PIStringList(const PIString & s0, const PIString & s1)
     PIStringList(const PIString & s0, const PIString & s1, const PIString & s2)
     PIStringList(const PIString & s0, const PIString & s1, const PIString & s2, const PIString & s3)
     PIString   ***join***(const PIString & delim) const – соединить все строки в одну с разделителем delim
     PIStringList & ***removeStrings***(const PIString & value)
     PIStringList & ***remove***(uint num)
     PIStringList & ***remove***(uint num, uint count)
     PIStringList & ***removeDuplicates***() – удалить повторяющиеся строки
     uint   ***contentSize***() – общая длина строк

PIFile

public:
PIFile()
PIFile(const PIString & path, PIFlags<Mode> mode = Read | Write)
bool ***open***(const PIString & path, PIFlags<Mode> mode)
bool ***open***(PIFlags<Mode> mode)
bool ***open***(const PIString & path)
bool ***open***()
void ***close***()
void ***clear***() – очистить файл
void ***seek***(llong pos) – перейти к позиции pos
void ***seekToBegin***() – перейти к началу файла
void ***seekToEnd***() – переецти к концу файла
void ***seekToLine***(llong line) – перейти к строке line
void ***resize***(llong new_size, char fill = 0)
void ***fill***(char c) – заполнить файл байтами c
void ***flush***() – обновить файл
PIString ***readLine***() – прочитать строку
llong ***readAll***(void * data) – прочитать всё содержимое в data
PIByteArray ***readAll***() – прочитать всё содержимое в байтовый массив
void ***remove***() – удалить файл
PIString ***path***() const
void ***setPath***(const PIString & path)
PIFlags<Mode> ***mode***() const
llong ***size***()
llong ***pos***()
bool ***isOpened***()
bool ***isEnd***()
bool ***isEmpty***()
PIFile & ***writeData***(const void * data, int size)
PIFile & ***readData***(void * data, int size)
PIFile & ***writeBinary***(const char v)
PIFile & ***writeBinary***(const short v)
PIFile & ***writeBinary***(const int v)
PIFile & ***writeBinary***(const long v)
PIFile & ***writeBinary***(const uchar v)
PIFile & ***writeBinary***(const ushort v)
PIFile & ***writeBinary***(const uint v)
PIFile & ***writeBinary***(const ulong v)
PIFile & ***writeBinary***(const float v)
PIFile & ***writeBinary***(const double v)
PIFile & ***operator <<***(const char & v)
PIFile & ***operator <<***(const PIString & v)
PIFile & ***operator <<***(const PIByteArray & v)
PIFile & ***operator <<***(const short & v)
PIFile & ***operator <<***(const int & v)

```
PIFile &      operator <<(const long & v)
PIFile &      operator <<(const uchar & v)
PIFile &      operator <<(const ushort & v)
PIFile &      operator <<(const uint & v)
PIFile &      operator <<(const ulong & v)
PIFile &      operator <<(const float & v)
PIFile &      operator <<(const double & v)
PIFile &      operator >>(char & v)
PIFile &      operator >>(short & v)
PIFile &      operator >>(int & v)
PIFile &      operator >>(long & v)
PIFile &      operator >>(uchar & v)
PIFile &      operator >>(ushort & v)
PIFile &      operator >>(uint & v)
PIFile &      operator >>(ulong & v)
PIFile &      operator >>(float & v)
PIFile &      operator >>(double & v)
```

```
PIThread

public:
     PIThread()
     PIThread(bool startNow, int run_delay)
     bool                 start(int run_delay = -1)
     bool                 startOnce()
     void                 stop(bool wait = false)
     void                 terminate()
     void                 terminate(bool hard = false)
     void                 setPriority(PIThread::Priority prior)
     PIThread::Priority   priority() const
     bool                 isRunning() const
     bool                 waitForFinish(int timeout_msecs)
     bool                 waitForStart(int timeout_msecs)
     void                 needLockRun(bool need)
     void                 lock()
     void                 unlock()
     PIMutex &            mutex()

private:
     virtual void   begin() – выполняется в начале потока
     virtual void   run() – рабочее тело цикла потока
     virtual void   end() – выполняется по завершении потока
```

PITimer

наследован от PIThread (только на Windows)
public:

```
     PITimer()
     PITimer(TimerEvent slot, void * data)
     void      setData(void * data)
     void      setSlot(TimerEvent slot)
     void      reset()
     void      start(double msecs)
     void      stop()
     bool      isRunning() const
     void      needLockRun(bool need)
     void      lock()
     void      unlock()
     void      addDelimiter(int delim, TimerEvent slot = 0)
     void      removeDelimiter(int delim)
     void      removeDelimiter(TimerEvent slot)
     void      removeDelimiter(int delim, TimerEvent slot)
     void      clearDelimiters()
     double    elapsed_n()
     double    elapsed_u()
     double    elapsed_m()
     double    elapsed_s()
```

```
PIEvaluator

public:
     PIEvaluator()
     bool           check(const PIString & string)
     int            setVariable(const PIString & name, complexd
value = 0.)
     void           setVariable(int index, complexd value = 0.)
     void           setCustomVariableValue(int index, complexd
value = 0.)
     complexd       evaluate()
     void           removeVariable(const PIString & name)
     void           clearCustomVariables()
     int            variableIndex(const PIString & name) const
     PIStringList   unknownVariables()
     PIString       expression()
     PIString       error()
     complexd       lastResult()
```

PIKbdListener

наследован от PIThread
public:
```
     PIKbdListener()
     PIKbdListener(KBFunc slot, void * data)
     void      setData(void * data)
     void      setSlot(KBFunc slot)
     void      enableExitCapture(char key = 'Q')
     void      disableExitCapture()
     bool      exitCaptured() const
     char      exitKey() const
     bool      isActive()
     void      setActive(bool yes)
```

PIConsole

наследован от PIThread
public:
     PIConsole()
     PIConsole(bool startNow, KBFunc slot)
     void   ***addString***(const PIString & name, int column,
PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, PIString * ptr,
int column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, char * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, bool * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, short * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, int * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, long * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, llong * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, uchar * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, ushort * ptr,
int column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, uint * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, ulong * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, ullong * ptr,
int column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, float * ptr, int
column, PIFlags<PIConsole::Format> format)
     void   ***addVariable***(const PIString & name, double * ptr,
int column, PIFlags<PIConsole::Format> format)
     void   ***addBitVariable***(const PIString & name, void * ptr,
int fromBit, int bitCount, int column, PIFlags<PIConsole::Format>
format)
     void a   ***ddEmptyLine***(int column, uint count)
     PIString ***getString***(int x, int y)
     short   ***getShort***(int x, int y)
     int   ***getInt***(int x, int y)
     float   ***getFloat***(int x, int y)
     double   ***getDouble***(int x, int y)
     PIString ***getString***(const PIString & name)
     short   ***getShort***(const PIString & name)
     int   ***getInt***(const PIString & name)
     float   ***getFloat***(const PIString & name)
     double   ***getDouble***(const PIString & name)

```
uint      tabsCount() const
PIString  currentTab() const
int       addTab(const PIString & name, char bind_key = 0)
void      removeTab(uint index)
void      removeTab(const PIString & name)
bool      setTab(uint index)
bool      setTab(const PIString & name)
bool      setTabBindKey(uint index, char bind_key)
bool      setTabBindKey(const PIString & name, char bind_key)
void      clearTabs(bool clearScreen = true)
void      addCustomStatus(const PIString & str)
void      clearCustomStatus()
void      clearVariables(bool clearScreen = true)
void      waitForFinish()
void      start(bool wait)
void      stop(bool clear)
PIString  fstr(PIFlags<PIConsole::Format> format)
void      enableExitCapture(char key = 'Q')
void      disableExitCapture()
bool      exitCaptured() const
char      exitKey() const
```

PIConfig

наследован от PIFile
public:
     PIConfig(const PIString & path, PIFlags<Mode> mode)
     Entry & ***getValue***(const PIString & vname, const PIString &
def, bool * exist = 0)

PISerial

наследован от PIThread
public:
     PISerial()
     PISerial(PIString name, void * data, SerialFunc slot)
     void   ***setSlot***(SerialFunc func)
     void   ***setSpeed***(PISerial::Speed speed)
     void   ***setOutSpeed***(PISerial::Speed speed)
     void   ***setInSpeed***(PISerial::Speed speed)
     void   ***setDevice***(const PIString & dev)
     void   ***setParameters***(PIFlags<PISerial::Parameters>
parameters)
     void   ***setData***(void * d)
     void   ***setReadData***(void * headerPtr, int headerSize, int
dataSize)
     bool   ***send***(char * data, int size)
     bool   ***init***()
     bool   ***initialized***() const
     void   ***terminate***()

PIEthernet

наследован от PIThread
public:
     PIEthernet()
     PIEthernet(PIString ip, int port, void * data, EthernetFunc slot)
     void   ***setSlot***(EthernetFunc func)
     void   ***setData***(void * d)
     void   ***setReadAddress***(PIString ip, int port)
     void   ***setSendAddress***(PIString ip, int port)
     bool   ***send***(PIString ip, int port, char * data, int size)
     bool   ***send***(char * data, int size)
     bool   ***init***()
     bool   ***initSend***()
     bool   ***receiverInitialized***() const
     bool   ***senderInitialized***() const
     void   ***terminate***()

PIProtocol

public:
    PIProtocol()
    PIProtocol(const PIString & config, const PIString & name, void * recHeaderPtr, int recHeaderSize, void * recDataPtr, int recDataSize, void * sendDataPtr, int sendDataSize)
    void ***startReceive***(float exp_frequency)
    void ***stopReceive***()
    void ***setExpectedFrequency***(float frequency)
    void ***setReceiverDevice***(const PIString & device, PISerial::Speed speed, bool force)
    void ***setReceiverData***(void * dataPtr, int dataSize)
    void ***setReceiverAddress***(const PIString & ip, int port, bool force)
    void ***setReceiverParameters***(PIFlags<PISerial::Parameters> parameters)
    void ***setReceiveSlot***(ReceiveFunc slot)
    void ***startSend***(float frequency)
    void ***stopSend***()
    void ***setSenderFrequency***(float frequency)
    void ***setSenderDevice***(const PIString & device, PISerial::Speed speed, bool force)
    void ***setSenderData***(void * dataPtr, int dataSize)
    void ***setSenderAddress***(const PIString & ip, int port, bool force)
    void ***setSenderParameters***(PIFlags<PISerial::Parameters> parameters)
    void ***start***()
    void ***stop***()
    void ***send***()
    void ***send***(const void * data, int size)
    void ***setName***(const PIString & name)
    PIString ***name***() const
    float ***immediateFrequency***() const
    float ***integralFrequency***() const
    float * ***immediateFrequency_ptr***()
    float * ***integralFrequency_ptr***()
    ullong ***receiveCount***() const
    ullong * ***receiveCount_ptr***()
    ullong ***wrongCount***() const
    ullong * ***wrongCount_ptr***()
    ullong ***sendCount***() const
    ullong * ***sendCount_ptr***()
    PIProtocol::Quality ***quality***() const
    int * ***quality_ptr***()
    PIString ***receiverDeviceName***() const
    PIString ***senderDeviceName***() const

```
     PIString           receiverDeviceState() const
     PIString *         receiverDeviceState_ptr()
     PIString           senderDeviceState() const
     PIString *         senderDeviceState_ptr()
     void *             receiveData()
     void *             sendData()

protected:
     virtual bool       receive(char * data, int size)
     virtual bool       validate()
     virtual uint       checksum_i(void * data, int size)
     virtual uchar      checksum_c(void * data, int size)
     virtual bool       aboutSend()
```

PISignals

статические функции:
     static void ***setSlot***(SignalEvent slot)
     static void ***grabSignals***(PIFlags<PISignals::Signal> signals)
     static void ***raiseSignal***(PISignals::Signal signal)

PICLI

```
public:
     PICLI(int argc, char * argv[])
     void           addArgument(const PIString & name, bool value)
     void           addArgument(const PIString & name, const PIChar
& shortKey, bool value)
     void           addArgument(const PIString & name, const char *
shortKey, bool value)
     void           addArgument(const PIString & name, const PIChar
& shortKey, const PIString & fullKey, bool value)
     void           addArgument(const PIString & name, const char *
shortKey, const PIString & fullKey, bool value)
     PIString       rawArgument(int index)
     PIString       mandatoryArgument(int index)
     PIString       optionalArgument(int index)
     PIStringList   rawArguments() const
     PIStringList   mandatoryArguments() const
     PIStringList   optionalArguments() const
     PIString       programCommand() const
     bool           hasArgument(const PIString & name)
     PIString       argumentValue(const PIString & name)
     PIString       argumentShortKey(const PIString & name)
     PIString       argumentFullKey(const PIString & name)
     PIString       shortKeyPrefix() const
     PIString       fullKeyPrefix() const
     int            mandatoryArgumentsCount() const
     int            optionalArgumentsCount() const
     void           setShortKeyPrefix(const PIString & prefix)
     void           setFullKeyPrefix(const PIString & prefix)
     void           setMandatoryArgumentsCount(const int count)
     void           setOptionalArgumentsCount(const int count)
```

PIProcess

наследован от PIThread (private)
public:
```
     PIProcess()
     int            exitCode() const
     int            pID() const
     void           setGrabInput(bool yes)
     void           setGrabOutput(bool yes)
     void           setGrabError(bool yes)
     void           setInputFile(const PIString & path)
     void           setOutputFile(const PIString & path)
     void           setErrorFile(const PIString & path)
     void           unsetInputFile()
     void           unsetOutputFile()
     void           unsetErrorFile()
     PIString       workingDirectory() const
     void           setWorkingDirectory(const PIString & path)
     void           resetWorkingDirectory()
     void           exec(const PIString & program)
     void           exec(const PIString & program, const PIString &
arg)
     void           exec(const PIString & program, const PIString &
arg1, const PIString & arg2)
     void           exec(const PIString & program, const PIString &
arg1, const PIString & arg2, const PIString & arg3)
     void           exec(const PIString & program, const
PIStringList & args)
     void           terminate()
     bool           waitForFinish(int timeout_msecs = 60000)
     PIByteArray    readOutput()
     PIByteArray    readError()
     PIStringList   environment()
     void           clearEnvironment()
     void           removeEnvironmentVariable(const PIString &
variable)
     void           setEnvironmentVariable(const PIString &
variable, const PIString & value)
```

статические:
```
     static PIStringList     currentEnvironment()
```