МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет

Кафедра радиофизики

Справочные материалы по практикуму ТСАНИ

Новосибирск, 2015

УДК 53:004.9(075.8)
ББК В185.505
С 325

Середняков С. С., Бехтенев Е. А. **Справочные материалы по практикуму ТСАНИ** ; Новосиб. гос. ун-т. – Новосибирск : РИЦ НГУ, 2015. – 52 с.

Данное методическое пособие содержит справочные данные необходимые для выполнения лабораторных работ практикума ТСАНИ. В нем собраны справочные материалы по языку С, возможностям и функциям среды LabWindows/CVI, а также по библиотекам доступа к оборудованию, используемому в данном практикуме.

Рецензент *Батраков А.М.*

Ответственная за выпуск *Тенекеджи О. А.*

Глава 1. Справочное руководство по языку С

1. Структура программы

Стандартная программа на языке С выглядит следующим образом:

```
#include “file1.h”
int Var1;
float Var2, Var3;
void Func(float Var)
{
 Char str[20];
 // some code
 //……
}
int main(int argc, char *argv[ ])
{
 // some code
 // ……..
 return 0;
}
```

Текст программы должен обязательно содержать функцию **main** с указанным выше списком параметров. Все используемые функции должны быть объявлены либо в данном файле перед их использованием выше по тексту программы, либо в файлах, подключенных с помощью директивы **#include** (см. пример выше). Все переменные, которые будут использоваться в программе, нужно объявить, указав при этом тип переменной. Переменные, объявленные вне какой либо функции (**Var1, Var2, Var3**) называются **глобальными**. Они доступны в любом месте программы и сохраняются в памяти до конца выполнения программы. Переменные, объявленные внутри функции (**str[20]**) называются **локальными**, и удаляются из памяти сразу после выхода из данной функции.

1.1. Объявление функций

Собственные функции можно объявлять в любом месте программы, не обязательно перед её использованием. Однако в случае объявления функции после ее использования по тексту программы, необходимо в самом начале файла, в котором будет осуществляться вызов данной функции, вставить её прототип (см. пример).

Объявление функций происходит следующим образом:

```
<тип возващаемого значения>  Имя_Функции ( список параметров )
{
   // тело функции
   return value;
}
```

Где:

<тип возващаемого значения> – тип переменной, значение которой вернет функция после завершения.

Имя_Функции – имя функции, по которому она будет вызываться.

(список параметров) – список типов переменных, с которыми данная функция вызывается. Может быть пустым – **()**.

return value – оператор выхода из функции и возвращения значения. Тип переменной **value** должен быть идентичен типу возвращаемого значения. Если тип возвращаемого значения – **void**, то в конце функции просто **return** можно не писать, однако такая конструкция может понадобиться, если нужно принудительно выйти из функции по какому-либо условию. Оператор **return** не обязательно помещать в конец функции, можно использовать его в произвольном месте.

Примеры:

```
float MyFunc1(int Arg);
int MyFunc2(int Arg1, float Arg2)
{
   int i = 0;
   //some code
   If(i < 0) return -1;
   // some code
   return i;
}
```

```
int main(int argc, char *argv[ ])
{
   int N1, N2, N3;
   float F1;
   N1 = 3;
   N2 = 19;
   // some code
   // ……..
   N3 = MyFunc2(N1, N2);
   F1 = MyFunc1(N3);
   return 0;
}

float MyFunc1(int Arg)
{
   float Var1;
   // some code
   return Var1 ;
}
```

2. Комментарии

Комментарии могут быть многострочными и однострочными. Многострочные комментарии заключаются в символы /* */. Однострочный комментарий начинается с символов // и заканчивается концом строки.

Пример:

```
/*
Многострочный
комментарий
*/
// Однострочный комментарий
```

3. Переменные

3.1. Основные типы переменных

Основные типы переменных, используемые в языке С следующие – булевы, целочисленные, численные с плавающей точкой, строковые (массивы переменных **`char`**). Некоторые из этих типов имеют «подтипы» – типы переменных сходные по типу данных, но отличающиеся по размерности, либо диапазону значений. Например, для целочисленных типов существует подтип, обозначаемый модификатором **`unsigned`**, который представляет только положительные числа. Для типов с плавающей точкой (**`float`**) есть подтип **`double`**, который занимает в 2 раза больше памяти, и имеет большие точность и диапазон значений. Основные типы языка С приведены в табл. 1.

Таблица 1

Типы переменных в языке С

Тип (псевдоним)	**Размер (байт)**	**Диапазон значениий**
`bool`	1	`true-false`
`char`	1	`–128…127`
`unsigned char`	1	`0…255`
`int (long)`	4	`–2147483648…2147483647`
`unsigned int`	4	`0…4294967295`
`float`	4	`+/–3.4 E+38`
`double`	8	`+/–1.7 E+308`

3.2. Структуры

В языке С есть возможность создавать составные типы (структуры). Структура – это тип переменных, который содержит произвольное число переменных разных типов. Обращение к элементам структуры происходит через точку (**.**). Ниже показан пример создания и использования структур:

```
struct Complex
{
 float Real;
 float Image;
};
struct Complex Number1;
Number1.Real = 10.0;
Number1.Image = 0.0;
```

3.3. Имена переменных, функций (Идентификаторы)

Имя переменной может представлять собой любую последовательность букв и цифр произвольной длины. Первый символ – обязательно буква, цифры не допускаются; символ подчеркивания "_" считается буквой; прописные и строчные буквы различаются.

Пример:

```
int IntVar; // верное имя
float 1Var; // ошибка, неверное имя
intVar = 1; // ошибка, переменная с данным именем не определена
IntVar = 1; // верно
```

Зарезервированные идентификаторы для нужд языка

Эти идентификаторы используются для обозначения различных операторов, и их нельзя использовать как названия переменных, функций:

```
typedef enum struct union for while do break
continue char int float double long void
switch default break include pragma define
return if ifdef
```

3.4. Запись данных в различных числовых системах

Как известно, кроме десятичной, существуют еще двоичная, восьмеричная и шестнадцатеричная системы записи числа. В программах на языке С можно вводить числа (константы, значения переменных при присвоении) во всех (кроме двоичной) системах. Для записи числа в шестнадцатеричной системе достаточно перед значением числа поставить приставку «**0x**», а для восьмеричной системы – «**0**» (см. пример ниже)

```
int Var;
Var = 12345;  // десятичный формат
Var = 0777;   // восьмеричный формат
Var = 0x11FF; // шестнадцатеричный формат
```

4. Операторы

Все действия над переменными осуществляются с помощью операторов. Операторы бывают унарными и бинарными. Унарным операторам нужна одна переменная (операнд), бинарным – две. В тексте программы операторы группируются в выражения. В выражениях с несколькими операторами в первую очередь выполняется оператор с более высоким приоритетом. Ниже приведена таблица с наиболее часто используемыми в языке С операторами, их приоритетами и типами переменных, с которыми они могут использоваться.

Обозначения типов переменных в табл. 2:

f – число с плавающей точкой

i – целое число

b – битовое число

L – логическое

P– указатель

all – все типы переменных.

Таблица 2

Операторы и приоритеты

Оператор	Описание	тип	приоритет	пример
`->`	выбор члена структуры по указателю	P	15	`pVar->Element`
`.`	выбор члена структуры из объекта	all	15	`Var.Element`
`[]`	Элемент массива по индексу	all	15	`Var[i]`
`()`	вызов функции	all	15	`func(Var1,Var2,…)`
`sizeof`	размер переменной в памяти	all	14	`sizeof (Type)`
`++`	Инкремент (увеличение значения на 1)	i f P	14	`Var++`
`++`	инкремент префикс (до)	i f P	14	`++ Var`
`--`	Декремент (уменьшение значения на 1) после	i f P	14	`Var --`
`--`	декремент до	i f P	14	`-- Var`
`!`	логическое не	L	14	`!Var`
`~`	не (побитовое)	L	14	`~ Var`
`-`	унарный минус	i f	14	`- Var`
`+`	унарный плюс	i f	14	`+ Var`
`&`	адрес объекта (переменной) в памяти	all	14	`&Var`
`*`	Значение переменной, на которую указывает указатель	P	14	`*pVar`
`()`	преобразование типа	all	14	`(тип) Var`
`()`	скобки	all	15	`(выр)`
`*`	умножение	i f	13	`Var1 * Var2`
`/`	деление	i f	13	`Var1 / Var2`
`%`	взятие по модулю (остаток)	i f	13	`Var1 % Var2`
`+`	сложение	i f P	12	`Var1 + Var2`

-	вычитание	i f P	12	Var1 - Var2
<<	Побитовый сдвиг влево	i b	11	Var << N
>>	Побитовый сдвиг вправо	i b	11	Var >> N
<	меньше	i f P	10	Var1 < Var2
<=	меньше или равно	i f P	10	Var1 <= Var2
>	больше	i f P	10	Var1 > Var2
>=	больше или равно	i f P	10	Var1 >= Var2
==	равно	i f P	9	Var1 == Var2
!=	не равно	i f P	9	Var1 != Var2
&	побитовое И	b	8	Var1 & Var 2
^	побитовое исключающее ИЛИ	b	7	Var1 ^ Var2
|	побитовое ИЛИ	b	6	Var1 | Var2
&&	логическое И	L	5	Var1 && Var2
||	логическое ИЛИ	L	4	Var1 || Var2
? :	Выбор по условию	L	3	Var1 ? Var2 : Var3
=	присваивание	all	2	Var1 = Var2
*=	умножить и присвоить	i f	2	Var1 *= Var2
/=	разделить и присвоить	i f	2	Var1 /= Var2
%=	взять остаток и присвоить	i f	2	Var1 %= Var2
+=	сложить и присвоить	i f	2	Var1 += Var2
-=	вычесть и присвоить	i f	2	Var1 -= Var2
<<=	сдвинуть влево и присвоить	i b	2	Var1 <<= Var2
>>=	сдвинуть вправо и присвоить	i b	2	Var1 >>= Var2
&=	И и присвоить	i b	2	Var1 &= Var2
|=	ИЛИ и присвоить	i b	2	Var1 |= Var2
^=	исключающее ИЛИ и присвоить	i b	2	Var1 ^= Var2
,	запятая (следование)	all	1	Var1, Var2

5. Побитовые и логические операции

5.1. Побитовые операции над целыми числами

Как известно, каждое целое число можно записать в двоичной системе, а в компьютере все данные хранятся именно в двоичном виде (в виде нулей и единиц). Для написания программ, управляющих аппаратурой, часто бывает необходимо работать с двоичным представлением числа, модифицируя его отдельные двоичные разряды, или их группы. Для этого в языке С существует ряд битовых операций:

1. **Операции побитового сравнения (& | ^ ~)**. Данные операции используются для различных действий с одним или двумя операндами на уровне их двоичных разрядов (см. таблицу ниже). Результат зависит от значений битов операндов в соответствующих разрядах.

Операция	Синтаксис	Описание
`&`	`C = A&B`	Поразрядное И. Возвращает `1` в соответствующем разряде, если у обоих операндов в этом разряде `1`
`|`	`C = A|B`	Поразрядное ИЛИ. Возвращает `1` в соответствующем разряде, если хотя бы у одного операнда в этом разряде `1`
`~`	`C = ~A`	Поразрядное НЕ. Возвращает `1` если в соответствующем разряде операнда стоит `0`. В противном случае возвращает `1`
`^`	`C = A^B`	Поразрядное исключающее ИЛИ. Возвращает `1` только если в соответствующем разряде операндов разные значения (`0` и `1`)

Примеры:

```
int N, M;
bool A, B, C;
N = 0xEA12;
// оставляем только младшие 8 бит (0xFF) :
M = N & 0xFF; // M = 0x12
// старшие 8 бит устанавливаем в 1, младшие оставляем как есть :
M = N | 0xFF00;  // M = 0xFF12
N = 0xAA;
// Инвертируем число N :
M = ~N;        // M = 0x55
// выделяем только те разряды, которые равны 0 :
M = N ^ 0xFF ;// M = 0x55
```

2. **Операции побитового сдвига (>> <<).** Данная операция приводит к поразрядному сдвигу числа в двоичном представлении на заданное число двоичных разрядов (**N** или **M**) влево (**<<**) или вправо (**>>**). Каждый сдвиг влево эквивалентен умножению на 2 , сдвиг вправо – делению на 2. При сдвиге влево, появляющиеся справа биты заполняются нулями.

Использование:

```
Var2 = Var1<<N;   Var2 = Var1>>M;
```

Сдвиг значений переменной **Var1** на **N** разрядов влево и на **M** разрядов вправо

Примеры:

```
int i, k, n;
i = 1;   //    (bin = 1)
k = i<<2;      //    k=4 ( bin = 100)
k = k<<3;      //    k =32 (bin = 100000)
n = k>>4;      //    n = 2 (bin = 10)
```

5.2. Логические операции

Логические операции обычно используются для получения булевого значения (true-false), в зависимости от значений операндов. Применяются в основном в конструкциях **if/else, while**. Среди логических операций есть как унарные, так и бинарные (см. таблицу).

Логические операции		
!	`!A`	Логическое НЕ. Возвращает `1` (true) если операнд равен `0` (false) и `0` при любых других значениях операнда.
&&	`(A && B)`	Логическое И. Возвращает true, если оба операнда имеют значение true.
||	`(A || B)`	Логическое ИЛИ. Возвращает true, если хотя бы один из операндов имеет значение true.

Примеры применения логических операций рассматриваются ниже в главе «Условные переходы».

6. Указатели, массивы, строки

6.1. Указатели

Указатель – это переменная, содержащая адрес другой переменной в оперативной памяти. Переменная-указатель объявляется с помощью знака «*****» перед именем переменной. При этом задается и тип переменной, на которую этот указатель может ссылаться. После объявления указателя ему нужно присвоить некоторое значение – сообщить адрес переменной, на которую он будет указывать. В процессе работы программы указатель можно «перенаправлять» на другие переменные.

Для работы с указателями также используются следующие операторы:

***** – получить значение переменной, на которую ссылается данный указатель.

& – получить адрес данной переменной в памяти.

Пример работы с указателями:

```
float VarX, VarY, VarZ;
float *pfvar;  // переменная-указатель на переменную типа float
VarX = 5.1;
pfvar = &VarX; // pfvar теперь указывает на (содержит адрес
               // переменной) VarX
VarY = *pfvar; // VarY присвоено значение переменной, на которую
               // указывает указатель pfvar, то есть переменной VarX
VarZ = 1.23;
pfvar = &VarZ; // pfvar теперь содержит адрес переменной VarZ
VarY = *pfvar; // теперь VarY = VarZ
```

В случае простых типов переменных (`float`, `int`, `char`) указатели часто используются, если нужно из вызываемой функции передать больше чем одно значение. В этом случае просто вернуть (return) все значения нельзя, а изменять значения нелокальных переменных функция не может. Здесь можно обойтись только указателями (см. пример).

Пример:

```
void GetDecartCoords(float Mod, float Phase, float *Real,
float *Img)
{
   float  Re, Im;
   Re = Mod * cos(Phase);
   Im = Mod * sin(Phase);
   // Передача значений переменных :
   *Real = Re;
   *Img = Im;
}
```

```
void main(int argc, char** argv)
{
   float Mod, Angle, XReal, YImg;
   Mod = 10.0;
   Angle = 1,8;
   GetDecartCoords(Mod, Angle, &XReal, &YImg);
   // после выполнения функции в переменных XReal и YImg
   // появятся новые значения
}
```

Типичный пример такого применения – функция форматированного ввода **scanf()**:

```
scanf(“ %d %f %d ”, &iVar, &fVar, &iVar1);
```

В программе переменные, являющиеся указателями, лучше называть, начиная с буквы **p** (pointer – указатель) для наглядности (**pVar**).

6.2. Массивы

Массив – это имеющая имя последовательность переменных одного типа, доступ к элементам которой осуществляется по номеру этого элемента (**[i]**).

Объявление массива:

```
float Farr[20];
```

В программе имя массива содержит его адрес, и его можно использовать как указатель на первый элемент массива. Увеличение или уменьшение данного указателя приводит к перемещению по элементам массива.

Пример:

```
float Farr[20], fvar;
float *pF;
Farr[0] = 0.6; // присвоение значения элементу массива
fvar = Farr[0];     // получение значения элемента массива
pF = Farr;          // тоже что и pF = &Farr[0];
*(pF + 2) = 2.34;   // Farr[2] = 2.34
// вывод на печать всех элементов массива Farr :
for( int i=0; i < 20; i++ )  printf(“ Farr[%d]  = %f  \n ”, I ,
*( pF + i ) );
pF += 2;
fvar = *pF;   // теперь fvar = Farr[2];
```

Приведенный выше пример описывает статическое объявление массива, когда размер массива задается при его объявлении, и является постоянным числом.

Массивы можно объявлять также и динамически. Это бывает необходимо, когда размер массива не известен на этапе компиляции программы. При динамическом объявлении массива используется указатель на тип переменных, хранящихся в массиве. Для динамического создания массива нужно воспользоваться функцией **`malloc(N)`**, которая выделяет блок **`N`** байт памяти и возвращает указатель на этот блок типа **`void*`**. Для согласования этот указатель нужно привести к указателю нужного типа. Если блок памяти нужного размера не удалось выделить, функция возвращает **`NULL`**.

После того как массив стал не нужен, нужно освободить занимаемую им память, воспользовавшись функций **`free()`**.Для использования этих функций в программе, нужно подключить файл **`malloc.h`**.

Пример:

```
#include <malloc.h>
float *Farr;  //   указатель (имя будущего массива)
int i, N, ArrSize;
N = 30;
ArrSize = N + 4;
Farr = (float *) malloc(ArrSize * sizeof(float)); // создание массива
If( Farr == NULL )
{
   printf("не удалось выделить память нужного размера \n");
   return;
}
for(i=0; i < ArrSize; i++)
   Farr[i] = i * 0.2;
// Работа с массивом…….
// ……
free(Farr);   //    удаление массива
```

6.3. Строки

В языке С переменные типа **char** могут содержать символьные константы, а массивы таких переменных образуют строки. Каждая строка заканчивается символом «**\0**» – конец строки, после которого все символы при всех операциях над строкой игнорируются. Объявление и инициализация строк производится, как и для обычных массивов.

Пример:

```
char symbol = “A”;
char str[ ] = “строка на С”;
char fixedstr[20];
sprintf(fixedstr, ” Строка ”);
char *str1;
int StrSize = 40;
str1 = (char *) malloc(StrSize * sizeof(char));
```

7. Ввод-вывод данных

7.1. Функции ввода-вывода

Для перевода информации из числового вида в текстовый и обратно в языке С существует ряд функций. Основные из них – функции форматированного ввода-вывода – **printf** и **scanf**. Существует несколько модификаций этих функций, различающихся по типу устройства ввода-вывода:

printf, scanf – ввод-вывод на главное устройство ввода-вывода.

fprintf, fscanf – ввод-вывод в файл. Первый аргумент функции – указатель на дескриптор файла (см. ниже), в который осуществляется ввод.

sprintf, sscanf – ввод-вывод в строку. Первый аргумент – указатель на строку (`char *`), в которую осуществляется ввод.

Функции имеют следующие прототипы:

```
int printf(const char *format, arg1, arg 2, …);
int scanf(const char *format, arg1, arg 2, …);
int fprintf(FILE *stream, const char *format, arg1, arg 2, …);
int fscanf(FILE *stream, const char *format, arg1, arg 2, …);
int sprintf(char *buf, const char *format, arg1, arg 2, …);
int sscanf(char *buf, const char *format, arg1, arg 2, …);
```

здесь:

***format** – форматирующая строка, задающая количество и формат ввода-вывода переменных **arg1**, **arg2**... Каждой выводимой переменной должен соответствовать символ **%** в форматирующей строке с параметрами вывода:

%[flags] [width] [.precision] [type]

Где:

поле **[flags]**. Значения:

- выравнивание по левому краю

+ выводить знак + для положительных чисел

поле **[width]**:

минимальное количество выводимых символов

поле **[precision]**:

количество знаков поле запятой для чисел с плавающей точкой

поле **[type]** (обязательное !)

задает тип выводимой переменной:

i – целое

x – целое шестнадцатеричное

d – целое десятичное

f – с плавающей точкой в формате `[-]dddd.dddd`

e – с плавающей точкой в формате `[-]d.ddde[sign]Edd`

g – с плавающей точкой в формате e или g, в зависимости в каком формате компактнее

c – символ

s – строка

Помимо форматирующих вывод переменных символов, при составлении строки часто используются т. н. **спецсимволы**, которые применяются для различных манипуляций со строкой, не связанных с вводимыми данными. Любой спецсимвол начинается с символа «слэш» «****». Ниже приведены самые часто применяемые спецсимволы:

\n – новая строка (перевод строки)

\t – горизонтальная табуляция

\r – возврат каретки

\’ – апостроф

\” – двойная кавычка

**** – обратный слэш

Примеры форматирования строк:

```
int I = 5, N = 0xAAFF;
double d1 = 8.465, d2 = 4.786;
char text[ ] = ”Data1 : ”;
char str1[80];

printf(“вывод : %s i=%i  d1=%.2f d2=%.4f  N = 0x%X \n”, text,
i, d1, d2, N);
// На экран печатаются значения переменных text, I, d1, d2, N
// (в шестнадцатеричном формате), затем производится
// перевод строки

sprintf(str1, “строка : i=%d,  N=%d,  d=%f”, i+2, N, d1);
// в строку str1 печатаются значения переменных i+2, N, d1

printf(“%s\n”, str1);
// содержимое строки str1 выводится на экран,
// после чего следует перевод строки
```

Результат:

```
вывод : Data1 :   i=5  d1=8.46  d2=4.7860  N = 0xAAFF
строка : i=7, N=43775, d=8.465
```

В случае использования функций ввода (**scanf**, **sscanf**, **fscanf**) в полях переменных нужно указывать адреса переменных, принимающих вводимые значения:

```
scanf(“ %d %d %f ”, &N1, &N2, &F1);
// с экрана считываются 3 числовых значения и сохраняются
// в переменных  N1, N2, F1

char Str1[80],str2[20];
// инициализация строки Str1
// ……..

sscanf(Str1,” %d  %f  %s ”,&N1,&F1,str2);
// из строки Str1 считываются значения 3-х переменных и
сохраняются
// в переменные N1, F1,str2
```

Функции ввода (**scanf**, **sscanf**, **fscanf**) возвращают число успешно прочитанных переменных, функции вывода (**printf**, **sprint**, **fprintf**) – число выведенных (напечатанных) символов.

7.2. Работа с файлами

В программах на языке С работу с файлами можно осуществлять через переменную типа **FILE** (дескриптор файла). Для этого нужно создать указатель на переменную этого типа, после чего открыть файл функцией **fopen**, указав имя файла и параметры работы с ним. После успешного открытия файла, чтение и запись в него осуществляется стандартными функциями **fprintf**, **fscanf** аналогично функциям **printf**, **scanf**. После работы с файлом его обязательно нужно обязательно закрыть функцией **fclose**.

Функция **fopen** имеет следующий прототип:

```
FILE *fopen( const char *filename, const char *mode );
```

Где:

filename – имя открываемого файла (строка).

mode – символ, задающий режим открытия файла (чтение, запись и т. д.), может принимать следующие значения:

"r" – Открывает существующий файл для чтения. **"w"** – Создает новый файл для записи. Если файл с таким именем существует, он будет предварительно удален.

"a" – Открывает существующий файл для добавления данных. Если файл с таким именем не найден, он будет создан.

"r+" – Открывает существующий файл для чтения и записи.

"w+" – Создает новый файл для чтения и записи. Если файл с таким именем существует, он будет предварительно удален.

"a+" – Открывает файл для чтения и для добавления данных. Если файл с таким именем не найден, он будет создан.

Функция **fopen** при успешном выполнении возвращает указатель на дескриптор файла, и **NULL** в случае ошибки.

Пример:

```
FILE *pF;
pF = fopen("MyFile.txt", "w"); // открытие файла "MyFile.txt"
                               // для записи
if( pF == NULL ) printf("Can't open file");
float Var1 = 1.2;
int Var2 = 3;
fprintf(pF," Data1=%f \n Data2=%d ", Var1, Var2); // запись в файл
fclose(pF);     //   закрытие файла
```

8. Операторы логического ветвления `if/else`, `switch/case`

Данные операторы служат для управления ходом программы в зависимости от значений управляющей переменной.

1. **`if/else`** – выбор выполнения блока программы в зависимости от булевого результата:

```
if( булевское_выражение )
{
   // выполняется если выражение истинно
}
else
{
   // выполняется если выражение ложно
}
```

2. **`switch/case`** – выбор выполнения блока программы, в зависимости от значения выражения внутри оператора **`switch`**. Выполнение передается блоку, который ассоциирован с конкретным значением данного выражения заголовком [**`case значениеN :`**], либо, если значение выражения не совпало ни с одним значением, блоку с заголовком **`default`**, (если он существует). Если внутри одного из блоков стоит оператор **`break`**, то его выполнение приведет к немедленному выходу из конструкции **`switch/case`**.

   Синтаксис блока **`switch/case`**:

```
switch( выражение )
{
   case  значение1 : op1;
   case  значение2 :
   {
     // some code
     break;
   }
   case значение3 :
   {
     // some code
     break;
   }
   default :
   {
     // some code
   }
}
```

Пример:

```
char C;
int N, M, K;
// some code
//……..
switch( C )
{
    case 'A' :  N++;  // выполняется, если C = ‘A’
    case 'B' :        // выполняется, если C = ‘B’
    {
        M++;
        N++;
        break;
    }
   default :          // выполняется во всех остальных случаях
       K++;
}
```

9. Циклы

Циклы используются, если нужно многократно выполнить определенный блок программы.

Цикл **for** выполняет заданный блок кода указанное число раз.

Цикл **while** выполняет заданный блок до тех пор, пока аргумент оператора **while** сохраняет значение **true**.

Оператор **break** – приводит к немедленному выходу из циклов **for**, **while** а также из конструкции **switch/case**.

Оператор **continue** – приводит к немедленному завершению текущей итерации циклов **for, while** и к началу следующей.

Примеры:

```
int i, k, n;
float F1, F2, Fmax;
Fmax = 10.0;
for( i = 0 ; i < 20 ; i ++ )
{
 F1 = I * 0.3;
 F2= F1 * F1;
 if( F2 > Fmax ) break;  //  выход из цикла
}
```

```
float F1;
bool flag;
int MaxIter, Nhits;
flag = true;
MaxIter = 100000;
Nhits = 0;
while( flag )
{
 F1 = 10.0 * rand();
 If( F1 == 7,34 )
 {
    Nhits++;
    continue;  //  немедленный переход к выполнению
               // следующей итерации
 }
 If( I >  MaxIter )  flag = false;
             // значение аргумента while стало false –
             // цикл прервется
 i++;
}
```

10. Функции стандартной библиотеки С

Среда CVI содержит реализацию языка С в достаточном объеме для решения задач в рамках существующей модели. Все декларации С-функций собраны в едином файле «**ansi_c.h**», который должен быть подключен к проекту:

```
#include <ansi_c.h>
```

Ниже приведены функции данной библиотеки, которые могут быть полезны в данном практикуме.

10.1. Тригонометрические функции

Имя	Декларация	Описание
cos	***double*** *cos(****double*** *Input)*	Вычисляет косинус величины ***Input***, заданной в радианах
sin	***double*** *sin(****double*** *Input)*	Вычисляет косинус величины ***Input***, заданной в радианах
tan	***double*** *tan(****double*** *Input)*	Вычисляет тангенс величины ***Input***, заданной в радианах
acos	***double*** *acos(****double*** *Input)*	Вычисляет арккосинус величины ***Input*** в радианах. Значение величины ***Input*** должно быть в пределах -1…1
asin	***double*** *asin(****double*** *Input)*	Вычисляет арксинус величины ***Input*** в радианах. Значение величины ***Input*** должно быть в пределах -1…1
atan	***double*** *atan(****double*** *Input)*	Вычисляет арктангенс величины ***Input*** в радианах. Значение величины ***Input*** должно быть в пределах -1…1
atan2	***double*** *atan2(****double*** *Input1,* ***double*** *Input2)*	Вычисляет арктангенс величины ***Input1/Input2*** в радианах

10.2. Гиперболические функции

Имя	Декларация	Описание
cosh	***double*** *cosh(****double*** *Input)*	Вычисляет косинус гиперболический величины ***Input***
sinh	***double*** *sinh(****double*** *Input)*	Вычисляет синус гиперболический величины ***Input***
tanh	***double*** *tanh(****double*** *Input)*	Вычисляет тангенс гиперболический величины ***Input***

10.3. Экспоненциальные и логарифмические функции

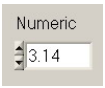
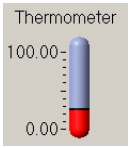
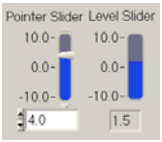
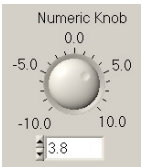
Имя	Декларация	Описание
exp	***double** exp(**double** Input)*	Вычисляет экспоненту от величины ***Input***
log	***double** log(**double** Input)*	Вычисляет натуральный логарифм от величины ***Input***
log10	***double** log10(**double** Input)*	Вычисляет десятичный логарифм от величины ***Input***
pow	***double** pow(**double** Input, **double** Power)*	Вычисляет величину, равную значению *Input*, возведенному в степень *Power*
sqrt	***double** sqrt(**double** Input)*	Вычисляет корень квадратный из величины *Input*

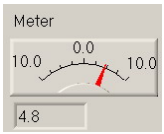
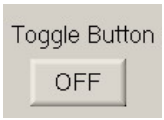
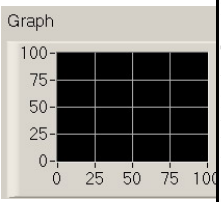
10.4. Функции округления

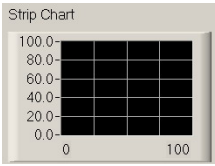
Имя	Декларация	Описание
ceil	***double** ceil(**double** Input)*	Округляет величину ***Input*** до следующего за ним целого числа
floor	***double** floor(**double** Input)*	Округляет величину ***Input*** до стоящего перед ним целого числа
fabs	***double** fabs(**double** Input)*	Возвращает модуль от величины ***Input***
fmod	***double** fmod(**double** Dividend, **double** Divisor)*	Вычисляет остаток, полученный при делении *Dividend/Divisor*

Глава 2. Справочные материалы по среде программирования LabWindows/CVI

1. Основные элементы управления

Название элемента управления	Основное применение	Свойства, константы, описывающие его поведение
1. Цифровое поле 	Служит для ввода-вывода числовых значений с клавиатуры. Расположение: **Numeric->**	**Свойства:** 1. **Minumum, Maximum**– минимальное, максимальное возможное значение. 2. **Default Value** – начальное значение. 3. **Data Type** – тип вводимых данных. 4. **Control Mode** – режим ввода-вывода элемента. Возможные значения: ***Indicator*** – возможет только вывод значений из программы. ***Hot, Normal, Validate*** – возможен вывод и ввод значений пользователем. 5. **Range Checking** – Настройка режима проверки введенного значения на предмет попадания в допустимый диапзон (Minimum…Maximum). Возможные значения: ***Coerce*** – в случае непопадания в диапазон, введенное значение уменьшается до одного из пределов (какой ближе). ***Notify*** – в случае непопадания
2. Термометр 	Служит для визуального отображения величины, хотя можно использовать и для ввода. Расположение: **Numeric->**	
3. Слайдеры 	Аналогично предыдущему. Можно использовать как для ввода, так и для отображения значений (в зависимости от значения поля **Control Mode**). Присутствуют как в вертикальном, так и в горизонтальном варианте. Расположение: **Numeric->**	
4. Ручка 	Служит для визуальной регулировки вводимого пользователем значения. Расположение: **Numeric->**	

5. Стрелка 	Служит для визуального отображения выводимого программой значения. Расположение: **Numeric->**	
6. Светодиод 	Служит для отображения двоичного состояния (On/Off) в виде цвета лампочки (красный/зеленый). Расположение: **LED->**	
7. Кнопка 	Стандартная кнопка. При нажатии срабатывает функция-обработчик. Расположение – **Command Button->**	
8. Кнопка-состояние 	Служит для задания двоичного состояния. Имеет 2 состояния – нажатое и ненажатое. Расположение – **Toggle Button->**	**Свойства:** 1. **OnText/OffText –** надписи на кнопке, соответствующие состояниям **On** и **Off**. 2. **Initial State –** Состояние кнопки при запуске программы
9. График 	Служит для отображения массивов, либо единичных данных в виде графиков зависимости одной координаты от другой. Графики можно добавлять, удалять, менять масштаб, добавлять описание, и т. д. Расположение – **Graph->**	**Свойства:** 1. **Data Mode** – задает, сохранять данные графика в памяти после прорисовки или нет. ***Retain*** – сохранять. ***Discard*** – не сохранять. 2. Кнопки **Top/Bottom/Left/Right X/Y-axis** – задают свойства осей со всех 4-х сторон графика. Можно менять диапазон, плотность сетки, название шкалы и т. д. Основные параметры: ***AutoScale*** – автоматический

		выбор диапазона. При выключении можно задавать диапазон в полях Minimum и Maximum. ***Show Grid/Show Minor Grid*** – отображать или нет основную и дополнительную сетку данной шкалы на графике. ***AutoDivisions*** – автоматический выбор количества делений на шкале. При выключении число делении можно задать в поле ***Divisions***. ***Log Scale*** – включает логарифмическую шкалу. ***Axis Names –*** задается имя шкалы
10. Осциллограмма 	Подходит для отображения данных в реальном времени. Данные добавляются на график слева направо. Можно отображать несколько величин одновременно. Расположение – **Graph->**	**Свойства:** 1. **Points per screen** – число точек на графике по оси X. 2. **Scroll Mode** – Задает режим перерисовки осциллограммы, если она достигла правого края графической области. Значения: ***Sweep*** – График начинает рисоваться слева, пририсовывая старые данные. ***Continuous –*** График начинает как целое, сдвигаться влево, новые данные при этом появляются справа. ***Block*** – График полностью удаляется, и начинает строиться заново. 3. Кнопка **Traces** – задает макс. количество графиков, их цвет, тип и толщину линии

11. Таймер 	Невидимый во время выполнения программы элемент. Генерирует событие **EVENT_TIMER_TICK** с заданным периодом, что приводит к периодическому выполнению функции-обработчика. Расположение – **Timer->**	**Свойства:** 1. **Interval(Seconds) –** период между срабатыванием таймера (секунды)

Все свойства, приведенные для одного типа, или группы элементов управления, сохраняют свое значение и для других элементов, если в них присутствуют свойства с теми же именами.

2. Функции для работы с элементами управления LabWindows/CVI

Для вызова большинства нижеперечисленных функций необходимо указывать 2 переменные : **`panelHandle`** и **`ControlID`**.

Переменная **`panelHandle`** (идентификатор панели) принимает свое значение после создания панели (вызова функции **`LoadPanel`** в функции **`main`**).

Переменная **`controlID`** является «полным» именем данного элемента управления и формируется следующим образом:

```
controlID = PanelName_ControlName.
```

Где:

`PanelName` – Имя панели

`ControlName` – Имя данного элемента управления

Данные переменные задаются в окне свойств соответствующего элемента, в поле **`Constant Name`**.

Пример:

```
int panelHandle;
float F1=7.81;
panelHandle = LoadPanel (0, "MainPanel.uir",PANEL));
//…….
SetCtrlVal(panelHandle,PANEL_NUMERIC1,F1);
// В элемент с именем NUMERIC1, расположенный в панели PANEL,
// с идентификатором panelHandle выводится значение переменной F1.
```

Список функций

Прототип функции	Основное применение	Описание аргументов
Элементы для цифрового ввода + Общие функции		
SetCtrlVal (int panelHandle, int controlID, value); **GetCtrlVal** (int panelHandle, int controlID, *value);	Устанавливает значение в данный элемент, либо считывает из него значение. Удобно применять для элементов цифрового ввода-вывода – Слайдеры, ручки, индикаторы, поля ввода (см. таблицу выше) **Данные функции не применимы к графическим элементам (График, Осцилограмма)**	**Параметры:** 1. **panelHandle** – идентификатор панели. 2. **controlID** – полное имя данного элемента управления. 3. **value** – переменная, для чтения/записи
SetCtrlAttribute (int panelHandle, int controlID, int controlAttribute, attributeValue);	Устанавливает значение конкретного атрибута (параметра) данного элемента управления. Список атрибутов элементов, и возможные константы-значения элемента данного типа нужно смотреть в справочном руководстве	**Параметры:** 1. **controlAttribute –** название атрибута, значение которого хочется поменять. 2. **attributeValue** – новое значение данного атрибута
Элементы графического вывода (График, Осциллограмма)		
SetAxisScalingMode (int panelHandle, int controlID, int axis, int axisScaling, double min, double max);	Настраивает режим, диапазон шкал со всех четырех сторон графика. Каждая шкала имеет название: **VAL_BOTTOM_XAXIS,** **VAL_TOP_XAXIS,** **VAL_LEFT_YAXIS,** **VAL_RIGHT_YAXIS**	**Параметры:** 1. **axis** – название оси, для которой производится настройка. 2. **axisScaling** – способ выбора диапазона: **VAL_MANUAL** – с помощью переменных min,max. **VAL_AUTOSCALE** –

		автоматический выбор шкалы в зависимости от значений переменных, загружаемых в график. **VAL_LOCK –** загружает заданные изначально значения диапазона и запрещает их изменять в дальнейшем. 3. **min, max** – значения диапазона для данной шкалы
PlotY (int panelHandle, int controlID, void *yArray, int numberOfPoints, int yDataType, int plotStyle, int pointStyle, int lineStyle, int pointFrequency, int color);	Рисует график массива переменных по Y, в зависимости от индекса массива, откладывая его по оси X. Только для элемента **Graph**	**Параметры:** 1. **yArray, xArray –** указатель на массив данных для отображения. 2. **numberOfPoints** – число точек для графика. Может быть меньше, чем размерность массива. 3. **yDataType –** тип данных в массиве. 4. **plotStyle** – тип графика (линии, точки, столбики, ...) 5. **pointStyle –** тип точки, если выбрано рисование точками. 6. **lineStyle** – тип линии графика (сплошная, прерывистая…) 7. **pointFrequency –** частота рисования точек. 8. **color** – цвет линии графика. Большинство этих параметров могут принимать фиксированный набор значений (констант), которые приведены в справочном руководстве к данной функции.
PlotX (int panelHandle, int controlID, void *xArray, int numberOfPoints, int xDataType, int plotStyle, int pointStyle, int lineStyle, int pointFrequency, int color);	Аналогично PlotY, только значения осей X и Y меняются местами. Только для элемента **Graph**.	
PlotXY (int panelHandle, int controlID, void *xArray, void *yArray, int numberOfPoints, int xDataType, int yDataType, int plotStyle, int pointStyle,	Строит график двух массивов переменных **Y(X)**: X-координата – элемент из первого массива (**xArray**), Y-координата – элемент из второго массива (**yArray**)	

int lineStyle, int pointFrequency, int color);		Возвращаемое значение: int plotHandle – идентификатор построенного графика
PlotPoint (int panelHandle, int controlID, double xCoordinate, double yCoordinate, int pointStyle, int color);	Ставит точку на график	**Параметры:** 1. **xCoordinate, yCoordinate** – координаты точки. 2. **pointStyle, color** – см. PlotY
PlotLine (int panelHandle, int controlID, double x1, double y1, double x2, double y2, int color);	Рисует линию с заданными координатами.	**Параметры:** 1. **x1, y1, x2, y2 –** координаты концов линии
DeleteGraphPlot (int panelHandle, int controlID, int plotHandle, int refresh);	Удаляет график из графической области	**Параметры:** 1. **plotHandle** – идентификатор графика, либо `-1` – удалить все графики, точки, линии. 2. **refresh** – задает, когда перерисовывать графическую область. Значения: **VAL_DELAYED_DRAW –** после какого-либо действия с графической областью (добавления нового графика, изменение масштаба…) **VAL_IMMEDIATE_DRAW –** немедленная перерисовка **VAL_NO_DRAW –** не делать перерисовку
RefreshGraph (int panelHandle, int controlID);	Перерисовывает всю графическую область	

PlotStripChart (int panelHandle, int controlID, void *yArray, int numberOfPoints, int startingIndex, int skipCount, int yDataType);	Добавляет (дорисовывает) массив точек к каждому графику в осциллограмме. Удобно использовать, если осциллограмма содержит несколько графиков. В этом случае можно создать массив размерностью равной числу графиков, проинициализировать его новыми значениями, и передать его в эту функцию.	**Параметры:** 1. **yArray** – массив данных для вывода на график. 2. **numberOfPoints** – число точек для графика. Может быть меньше, чем размерность массива. 3. **startingIndex** – индекс массива данных, с которого нужно начинать брать значения. Должно быть кратно числу графиков на осциллограмме. Чаще всего равно `0`. 4. **skipCount –** число элементов массива, которое нужно пропускать после того, как такое же число элементов было добавлено на график. 5. **yDataType –** тип данных в массиве.
PlotStripChartPoint (int panelHandle, int controlID, double y);	Добавляет точку к графику на осциллограмме. Можно использовать **ТОЛЬКО** если осциллограмма содержит один график.	**Параметры:** 1. **y** – значение, которое нужно добавить.
Дополнительные функции		
Random (double minimum, double maximum);	Генерирует случайное число в заданном диапазоне.	**Параметры:** **minimum, maximum** – диапазон, в пределах которого будет сгенерированно случайное число

FFT (double arrayXReal[], double arrayXImg[], int numberOfElements);	Выполняет быстрое преобразование Фурье от массива комплексных чисел.	**Параметры:** 1. **arrayXReal, arrayXImg –** массивы реальных и мнимых частей массива комплексных чисел. После выполнения функции массивы содержат соответственно реальные и мнимые части получившегося результата. 2. **numberOfElements** – число элементов в массиве.
KeyHit();	Возвращает **TRUE,** если была нажата клавиша на клавиатуре.	Не имеет параметров
GetKey();	Ожидает нажатия на клавиши клавиатуре и затем возвращает код нажатой клавиши.	Не имеет параметров

3. Инструменты для отладки приложений в среде LabWindows/CVI

Очень часто в процессе разработки программного обеспечения возникает необходимость проследить, как выполняется программа буквально «по шагам», посмотреть как меняются значения переменных, и т. д. Среда программирования LabWindows/CVI предлагает целый набор средств для отладки приложения во время исполнения. Условно эти средства можно разделить на 2 группы:

1. Инструменты для управления ходом программы:
   - точки останова (**Breakpoints**) – места в тексте программы, на которых прекращается её исполнение для дальнейшего анализа или пошагового выполнения.
   - пошаговое исполнение (**Step Over, Step Into**) – возможность исполнять инструкции программы последовательно, заходя внутрь вызываемых функции, либо выполнять их, переходя к следующей строке. Также есть возможность наоборот выйти из функции, выполнив ее до конца (**Finish Function**).
   - Возможность выполнить программу до любого места (положения курсора) и остановиться (**Go to Cursor**), либо просто про-

должить выполнение программы до следующей точки останова либо до конца (**Continue**).

2. Инструменты для наблюдения и изменения значений переменных и целых выражений в тексте программы. Сюда входят:

- Окно для наблюдения и изменения всех значений локальных и глобальных переменных программы (**Variables**).
- Окно для наблюдения и изменения отдельных переменных, либо целых выражений из текста программы (**Watch**).

Все инструменты 1-й группы находятся в меню **Run**. Сюда входят следующие команды:

- **Toggle BreakPoints (F9)** – установить либо снять (если уже установлена) точку разрыва выполнения программы.
- **Continue (F5)** – продолжить выполнение программы.
- **Go to Cursor (F7) –** продолжить выполнение программы о остановленного места, до положения курсора в окне редактирования.
- **Step Over (F10) –** выполнить следующее выражение в тексте программы. Выполняемая конструкция при этом будет выделяться красным цветом.
- **Step Into (F8) –** Если текущая конструкция в программе является вызовом функции, то зайти в тело этой функции.
- **Finish Function (Ctrl+F10)** – выполнить до конца функцию, на которой в данный момент остановлена программа, выйти из неё и остановиться.

Почти все эти команды становятся доступными только в режиме отладки программы (кроме самой первой). Следовательно, чтобы ими воспользоваться, нужно поставить хотя бы одну точку останова (**BreakPoint**) (клавиша **F9**) в тексте программы и запустить её на выполнение. После остановки программы можно будет воспользоваться всеми вышеперечисленными командами.

Инструменты 2-й группы (окна **Variables** и **Watch**) вызываются командами меню **Window (**соответственно **Variables** и **Watch**). Окна появляются в нижней части рабочего окна среды LabWindows/CVI.

- Окно **Variables** содержит значения всех переменных программы, сгруппированных по функциям. Обычно это окно разделено на 2 вертикальные части (см. Рис. 1): Окно **Stack Trace** которое содержит список функций, вызванных в данный момент справа, и непосредственно окно переменных слева. Окно переменных в свою очередь разделено на две горизонтальных области: верхняя отображает глобальные и статические переменные программы, нижняя – локальные переменные функции, активной в данный момент. При выборе другой функции в окне **Stack Trace**, содержимое нижней части меняется – отображаются локальные переменные выбранной функции. Обе горизонтальные части

окна переменных имеют 3 колонки : **Name** (имя переменной), **Value** (её значение), **Type** (тип). По правому щелчку мыши на переменной появляется контекстное меню, с помощью которого, кроме всего прочего можно изменить её значение (команда **Edit Value** контекстного меню переменной).

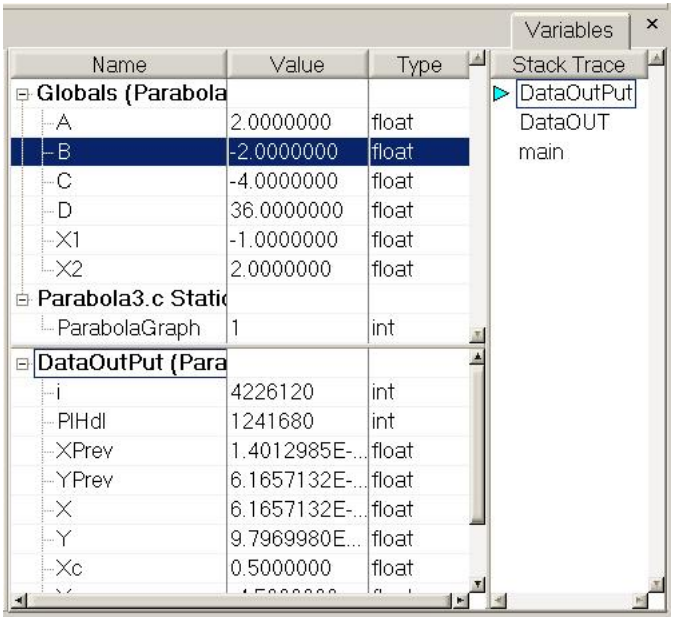


Рис. 1. Окно Variables

- Окно **Watch** во многом аналогично окну **Variables**, за исключением того, что в него нужно добавлять переменные вручную. Также в это окно можно добавлять выражения как из текста программы, так и создавать свои выражения с использованием программных переменных. В столбце **Values** отобразится результат. Для добавления переменной либо выражения в это окно, нужно выделить текст переменной либо выражения (без символа "**;**" в конце), далее щелкнуть правой кнопкой мыши на выделенном тексте и выбрать команду «**Add Watch Expression**». Также можно вызвать это же меню, щелкнув на свободное место окна **Watch**, и вписать текст переменной/выражения в поле **Variable/Expression**.

Следует отметить, что переменные в окне **Watch** останутся там даже после того, как вы закроете это окно, и будут выводиться при следующем запуске отладки программы. Также нужно иметь ввиду, что вывод значе-

ний переменных в это окно может сильно замедлить выполнение программы, если например, в это окно добавлены локальные переменные часто вызываемых функций. Для удаления переменных из окна **Watch** нужно выделить соответствующую строку в окне **Watch** и нажать **Delete** на клавиатуре.

4. Инструменты автодополнения в среде LabWindows/CVI

При работе с большим количеством новых функций с большим количеством параметров очень удобным является механизм **автодополнения**. Данный механизм заключается в том, что при введении первых 3-4-х букв функции, система выводит список всех доступных названий функций, и пользователю остается только выбрать их них нужную. Кроме того данный механизм сильно упрощает ввод аргументов функций.

Для использования данного механизма используются следующие комбинации клавиш:

1. **<Ctrl-Пробел>** – автодополнение имени функции. Если написать начало имени функции и нажать **<Ctrl-Пробел>**, среда предложит список подходящих имен функций.

2. **<Ctrl-Shift-Пробел>** – показ декларации функции и автодополнение аргументов функции. При нахождении курсора на имени функции показывается декларация функции с типами аргументами. При нахождении курсора на одном из аргументов показывается список возможных значений аргумента (см. рис. 2).

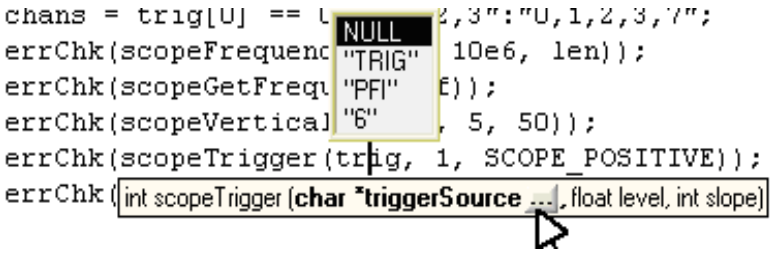


Рис. 2. Автодополнение аргумента функции

Глава 3. Функции библиотеки *tsanilib*

1. Функции для работы с модулем NI-PXI6251 (универсальный модуль ввода-вывода)

Доступ к байтовым портам	
Функция	**portMask** – запись маски порта
декларация	#include<tsani.h> **int** portMask(**int** *portN*, **unsigned char** *mask*);
назначение	Запись маски порта. Размер маски 8 бит. Значение i-го бита равное `0` означает, что в байте соответствующий бит назначается для ввода данных. Значение бита `1` означает, что бит назначается для вывода данных. При вызове функции линии ввода/вывода битов, которые должны читаться переходят в высокоимпеданстное состояние, линии, назначенные на запись, не меняют своего значения
параметры	**int** *portN* – номер порта ввода вывода, принимает значение `0, 1, 2` **unsigned char** *mask* – маска порта ввода вывода
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Функция	**portOut** – вывод данных в порт
декларация	#include<tsani.h> **int** portOut(**int** *portN*, **unsigned char** *data*);
назначение	Вывод данных в порт, биты для которых маска установлена на запись выводятся на выходные линии, биты для которых маска установлена на чтение игнорируются
параметры	**int** *portN* – номер порта ввода вывода, принимает значение `0, 1, 2` **unsigned char** *data* – выводимые данные

возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Функция	**portIn** – чтение данных с линий порта
декларация	#include<tsani.h> **int** portIn(**int** *portN*, **unsigned char*** *data*);
назначение	Чтение данных из порта, биты для которых установлено направление чтения читаются с внешних линий, биты для которых маска установлена на запись принимают ранее записанное значение
параметры	**int** *portN* – номер порта ввода вывода, принимает значение `0, 1, 2` **unsigned char*** *data* – указатель на переменную в которой сохраняется прочитанное значение
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Доступ к аналоговым портам	
Функция	**analogOut** – вывод напряжения на ЦАП NI PXI-6251
декларация	#include<tsani.h> **int** analogOut(**int** *idx*, **double** *val*);
назначение	Вывод скалярных данных на соответствующий выход ЦАПа
параметры	**int** *idx* – номер выхода ЦАП-а, `0` или `1` **double** *val* – выводимое напряжение в вольтах
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение

Функция	**analogIn** – измерение напряжения с помощью АЦП NI PXI-6251
декларация	#include<tsani.h> **int** analogIn(**int** *idx*, **double*** *val*);
назначение	Измерение напряжения на входе АЦП
параметры	**int** *idx* – номер входа АЦП, `0` или `1` **double*** *val* – измеренное напряжение в вольтах
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Функция	**analogOutClk** – выбор источника и частоты измерения ЦАП
декларация	#include<tsani.h> **int** analogOutClk(**const char** *source*[], **double** *rate*);
назначение	Выбор источника синхронизации ЦАП, по умолчанию используется внутренний источник
параметры	**const char** *source*[] источник синхронизации "OnboardClock" – внутренний источник измерений "ai/SampleClock" – ЦАП измеряет синхронно с АЦП **double** *rate* – частота измерений в случае внутреннего источника Гц. максимальная частота 1e6 Гц
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Функция	**analogInClk** – выбор источника и частоты измерения ЦАП
декларация	#include<tsani.h> **int** analogInClk(**const char** *source*[], **double** *rate*);
назначение	Выбор источника синхронизации АЦП, по умолчанию используется внутренний источник

параметры	**const char** *source*[] – выбор источника синхронизации "OnboardClock" – внутренний источник измерений "ao/SampleClock" – АЦП измеряет синхронно с ЦАП **double** *rate* – частота измерений в случае внутреннего источника Гц. максимальная частота 1e6 Гц
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Функция	**analogWrite** – вывод массива данных на ЦАП
декларация	#include<tsani.h> **int** analogWrite(**int** *idx*, **double*** *warray*, **int** *size*);
назначение	Вывод массива данных *warray* размером *size* на ЦАП
параметры	**int** *idx* – номер выхода ЦАП, `0` или `1`. **double*** *warray* – указатель на массив данных, напряжение в вольтах. **int** *size* – размер массива данных
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Функция	**analogRead** – чтение массива данных с АЦП
декларация	#include<tsani.h> **int** analogRead(**int** *idx*, **double*** *rarray*, **int** *size*, **int*** *readed*);
назначение	Чтение с АЦП массива *rarray* размером *size*, в параметре *readed* записывается реально прочитанное количество чисел
параметры	**int** *idx* – номер входа АЦП, `0` или `1` **double*** *rarray* – массив для записи прочитанных данных, напряжение в вольтах **int** *size* – размер массива **int*** *readed* – реально прочитанное количество данных

возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Необязательные сервисные функции	
Функция	**ni6251Slot** – установка номер слота NI PXI-6251
декларация	#include<tsani.h> **int** ni6251Slot(**int** *slot*);
назначение	Инициализация библиотеки работы с NI PXI-6251, если блок в стандартной позиции вызов не обязателен
параметры	**int** *slot* – номер слота где установлен блок
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение
Функция	**ni6251Close** – окончание работы с блоком NI PXI-6251
декларация	#include<tsani.h> **int** ni6251Close(void);
назначение	Освобождение и закрытие всех ресурсов библиотеки
параметры	Нет
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение

2. Функции для работы с модулем NI-PXI5402 (Генератор сигналов)

Функция	**fgenStartStandartWaveForm** – начать генерацию сигнала
декларация	**ViStatus** fgenStartStandartWaveForm(**float** Amplitude, **float** Frequency, **int** WaveFormType);

назначение	Начать генерацию сигнала заданной формы, амплитуды и частоты
параметры	**float** Amplitude – амплитуда сигнала в вольтах ПИК-ПИК **float** Frequency – частота генерируемого сигнала **int** WaveFormType – форма сигнала : FGEN_SINE – синосуидальный сигнал FGEN_SQUARE – меандр FGEN_TRIANGLE – треугольный FGEN_RAMP_UP – линейновозрастающий FGEN_RAMP_DOWN – линейноубывающий FGEN_VAL_WFM_DC – постоянный FGEN_NOISE – белый шум
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**fgenGenerateFreqList –** подготовка списка частот
декларация	**ViStatus** fgenGenerateFreqList(**int** FreqSteps, **float** FreqMin, **float** FreqMax, **float** StepDuration, **double** *pFreqList, **double** *pTimeIntervalList);
назначение	Подготовка списка частот и длительности их генерации с изменением частоты по линейно возрастающему загону
параметры	**int** FreqSteps – количество точек частоты, число больше 2-х **float** FreqMin – минимальная частота, Гц **float** FreqMax – максимальная частота, Гц **float** StepDuration – длительность генерации одного шага, сек **double** *pFreqList – массив для сохранения частот, размер больше или равен FreqSteps **double** *pTimeIntervalList – массив для сохранения

	длительностей, размер больше или равен FreqSteps
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
функция	**fgenStart** – начать генерацию сигнала
декларация	**ViStatus** fgenStart(**int** WaveFormType, **double** Amplitude, **int** ListSize, **double** *pFreqList, **double** *pTimeIntervalList);
назначение	Начать генерацию сигнала по таблице и длительности частот
параметры	**int** WaveFormType – форма сигнала, см. fgenStartStandartWaveForm **double** Amplitude – амплитуда сигнала ПИК-ПИК, вольт **int** ListSize – размер таблицы **double** *pFreqList – таблица частот **double** *pTimeIntervalList – таблица длительностей
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки

Функция	**fgenStop** – прекращение генерации
декларация	**ViStatus** fgenStop(**void**);
назначение	Прекращение генерации
параметры	нет
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки.
Функция	**fgenExportTrig** – выбор выхода импульса синхронизации
декларация	**ViStatus** fgenExportTrig(**ViConstString** outputTerminal);
назначение	Выбор выхода импульса синхронизации, импульс генерируется при старте генерации
параметры	**ViConstString** outputTerminal NULL, “” – отключить выдачу импульса “PFI0“, “PFI1“ – использовать соответствующий выход
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки.
Функция	**fgenFrequency** – установить частоту генерации
декларация	**ViStatus** fgenFrequency(**float** NewFrequency);
назначение	Установить частоту генерации в режиме монохроматической генерации, можно менять во время генерации
параметры	**float** NewFrequency – частота, Гц
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки.

Функция	**fgenAmplitude** – установить амплитуду генерации
декларация	**ViStatus** fgenAmplitude(**float** NewAmplitude);
назначение	Установить амплитуду генерации в режиме монохроматической генерации, можно менять во время генерации
параметры	**float** NewAmplitude – амплитуда сигнал ПИК-ПИК, вольт
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки.
Функция	**fgenSlot** – номер слота ni5402
декларация	**ViStatus** fgenSlot(**int** slot);
назначение	Установить номер слота генератора ni5402
параметры	**int** slot – номер слота, 5-8
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки.
Функция	**fgenClose** – закрыть устройство
декларация	**ViStatus** fgenClose(**void**);
назначение	Закрыть устройство, освободить ресурсы.
параметры	нет
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки.

3. Функции для работы с модулями NI-PXI5114 и NI-PXI5105 (Цифровой осциллограф)

Функция	**scopeSlot** – номер слота ni5114, ni5105
декларация	**ViStatus** scopeSlot(**int** slot);
назначение	Выбор номера слота АЦП
параметры	**int** slot – номер слота, 5-8
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**scopeClose** – закрыть устройство
декларация	**ViStatus** scopeClose(**void**);
назначение	Закрыть устройство, освободить ресурсы.
параметры	Нет
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**scopeFrequency** – установка частоты оцифровки
декларация	**ViStatus** scopeFrequency(**const char*** sourceCLK, **double** frequency, **int** numSamples);
назначение	Настройка временной шакалы АЦП
параметры	**const char*** sourceCLK – источник опорной частоты отцифровки NULL , “” – внутренний источник частоты PFI1 – ввод сигнала через разъем на передней панели

	double frequency – частота, Гц. В случае внутреннего источника будет установлена указанная частота дискретизации, в случае внешнего источника следует указывать частоту близкую к частоте внешнего источника для правильной настройки схемы фазовой подстройки частоты **int** numSamples – размер буфера, должен совпадать с размером буфера функций семейства scopeStart*
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**scopeGetFrequency** – получить частоту отцифровки
декларация	**ViStatus** scopeGetFrequency(**double*** frequency);
назначение	Получить фактическую частоту отцифровки
параметры	**double*** frequency – адрес для сохранения частоты
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**scopeVertical** – настройка вертикальной шкалы
декларация	**ViStatus** scopeVertical(**const char*** ChanList, **float** range, **int** impedance);
назначение	Настройка вертикальной шкалы АЦП
параметры	**const char*** ChanList – список настраиваемых каналов "`0`" – одиночный канал "`0,1,5`" – список каналов **float** range – входной диапазон ПИК-ПИК, вольт **int** impedance – сопротивление входов SCOPE_1_MEG_OHM – 1 МОм SCOPE_50_OHM – 50 Ом

возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**scopeTrigger** – настройка триггера запуска
декларация	**ViStatus** scopeTrigger(**const char*** triggerSource, **float** level, **int** slope);
назначение	настройка триггера внешнего запуска
параметры	**const char*** triggerSource – вход триггера NULL, "" – программный запуск "TRIG" – вход TRIG "PFI" – вход PFI "6" – вход CH6 **float** level – уровень срабатывания триггера, вольт **int** slope – выбор фронта SCOPE_POSITIVE – положительный фронт SCOPE_NEGATIVE – отрицательный фронт
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**scopeStart** – старт измерений
декларация	**ViStatus** scopeStart(**int** numSamples);
назначение	Начать измерение
параметры	**int** numSamples – размер буфера
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки

Функция	**scopeStartContinius –** начать измерение в непрерывном режиме
декларация	**ViStatus** scopeStartContinius(**int** numSamples, **int** divisor);
назначение	Начать измерение в непрерывном режиме
параметры	**int** numSamples – размер буфера **int** divisor – делитель частоты отцифровки
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**scopeStatus** – статус работы АЦП
декларация	**ViStatus** scopeStatus(**void**);
назначение	Определить состояние АЦП
параметры	нет
возвращаемое значение	Возвращаемое значение SCOPE_COMPLETE – измерение законченно SCOPE_IN_PROGRESS – измерение продолжается SCOPE_UNKNOWN – ошибка
Функция	**scopeStop** – остановить измерение
декларация	**ViStatus** scopeStop(**void**);
назначение	Остановка измерение
параметры	Нет
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки

Функция	**scopeStartRead** – провести измерение
декларация	**ViStatus** scopeStartRead(**const char*** ChanList, **double*** wfm, **int** numSamples);
назначение	Проведение измерения, Функция производит старт измерения, ожидание окончания измерения и чтение данных
параметры	**const char*** ChanList – список измеряемых каналов "6" – измерение одиного канала "4,6" – измерение нескольких каналов **double*** wfm – массив для сохранения данных, размер массива количество_каналов* numSamples **int** numSamples – количество данных
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки
Функция	**scopeFetch** – чтение данных
декларация	**ViStatus** scopeFetch(**const char*** ChanList, **double*** wfm, **int** numSamples, **int*** numSamplesReaded);
назначение	Чтение измеренных данных
параметры	**const char*** ChanList – список запрашиваемых каналов "6" – измерение одиного канала "4,6" – измерение нескольких каналов **double*** wfm – массив для прочитанных данных, размер массива количество_каналов* numSamples **int** numSamples – количество данных **int*** numSamplesReaded – количество реально прочитанных данных
возвращаемое значение	В случае успеха функция возвращает значение `0`, в случае ошибки функция возвращает отрицательное значение ошибки

Литература

1. Керниган, Ричи. Язык С.
2. National Instruments LabWindows/CVI Help.

Оглавление