КИПР

ТУСУР

Кафедра конструирования
и производства радиоаппаратуры

Разветвлённые программы с графическим интерфейсом в среде Lazarus

Томск 2017

Кобрин Юрий Павлович

Разветвлённые программы с графическим интерфейсом в среде Lazarus. Методические указания к лабораторной работе и по организации самостоятельной работы по дисциплинам «Информатика» и «Информационные технологии» для студентов очного и заочного обучения специальностей 211000.62 (бакалавриат) и 162107.65 «Информатика и информационные технологии» (специалитет). - Томск: Томский государственный университет систем управления и радиоэлектроники (ТУСУР), кафедра КИПР, 2017. – 36 с.

Lazarus — открытая бесплатная кроссплатформенная среда визуальной разработки программного обеспечения для компилятора *Free Pascal*, максимально приближённая к *Delphi*. Методические указания посвящены освоению основ объектно-ориентированного программирования (ООП) в интегрированной среде *Lazarus*, а также с особенностями технологии визуального программирования простейших разветвлённых программ с графическим интерфейсом.

Методические указания предназначены для помощи в подготовке бакалавров и магистрантов в Информатике, выполнения курсовых и дипломных проектов.

©Кафедра КИПР федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Томский государственный университет систем управления и радиоэлектроники (ТУСУР)», 2017**.**
© Кобрин Ю.П. 2017

Оглавление

1 Цели и задачи работы

Цель работы:

- Закрепление навыков работы в интегрированной среде Lazarus.
- Знакомство с операторами управления в программах в интегрированной среде Lazarus.
- Освоение приёмов программирования и отладки разветвлённых алгоритмов в интегрированной среде Lazarus

2 Порядок выполнения работы

Перед выполнением этой работы следует:

1) Ознакомиться с разделами настоящей работы «Операторы управления программой», «Составление и отладка разветвлённых программ». В качестве дополнительной литературы использовать [1,2,3,4,5,6,7,8].
2) Разработать программу в соответствии с индивидуальным заданием (Раздел 7).
3) Войти в свой личный каталог, загрузить и настроить систему программирования ***Lazarus***.
4) Ввести шаблон программы, сделанный на прошлом занятии. Сделать его копию, записав под новым именем (разветвлённой программы).
5) Ввести разработанную Вами программу, корректируя и дополняя свой шаблон.
6) Освоить методику поиска причин и исправления синтаксических ошибок, а также отладки программы «по шагам» по тестовому примеру (Раздел 6). Добиться, чтобы программа дала правильные результаты.
7) Ответить на контрольные вопросы.
8) Оформить отчёт и защитить его у преподавателя.

3 Отчётность

Отчёт должен быть выполнен в соответствии с [9] и состоять из следующих разделов:

1) Тема и цель работы.
2) Индивидуальное задание.
3) Блок-схема алгоритма.
4) Откомпилированный текст программы (в электронном виде).
5) Ответы на контрольные вопросы.
6) Результаты выполнения программы.
7) Выводы.

При защите отчёта по работе для получения зачёта студент должен:

- уметь отвечать на контрольные вопросы;
- обосновать структуру выбранного алгоритма и доказать, что разработанное приложение работает правильно;
- уметь пояснить назначение элементов разработанного приложения;
- продемонстрировать навыки работы в интегрированной среде *Lazarus*.

4 Контрольные вопросы

Ответьте на следующие контрольные вопросы:

1) Что такое *составной оператор*? Каковы причины его использования? В чем состоит особенность расстановки точек с запятой при записи составного оператора?

2) Что представляет собой оператор безусловного перехода? Укажите его назначение и особенности применения. Почему рекомендуется избегать оператора ***GoTo***?

3) Какая алгоритмическая конструкция называется ветвлением? Для чего необходимо ветвление в алгоритмах?

4) Какие формы ветвления различают? Сравните формы ветвления между собой. Приведите примеры её использования.

5) Что такое условие? Приведите примеры логических выражений. Из чего они состоят?

6) К какому типу данных всегда принадлежит результат проверки логического выражения? Приведите примеры его использования в инструкциях присваивания и вывода.

7) Перечислите основные операции отношения, используемые в *Lazarus*.

8) Перечислите основные логические операции, используемые в *Lazarus*.

9) Как определяется порядок выполнения операций в составе логических выражений?

10) Поясните, почему каждое простое условие в логическом выражении необходимо заключать в скобки. Приведите примеры.

11) Чем отличаются друг от друга *ветвление* и *обход*? Поясните с использованием блок-схем алгоритмов. Приведите примеры листингов, содержащих варианты использования инструкции ***if***.

12) Найдите значение Y при $X = 20$ (Рис. 4.1).

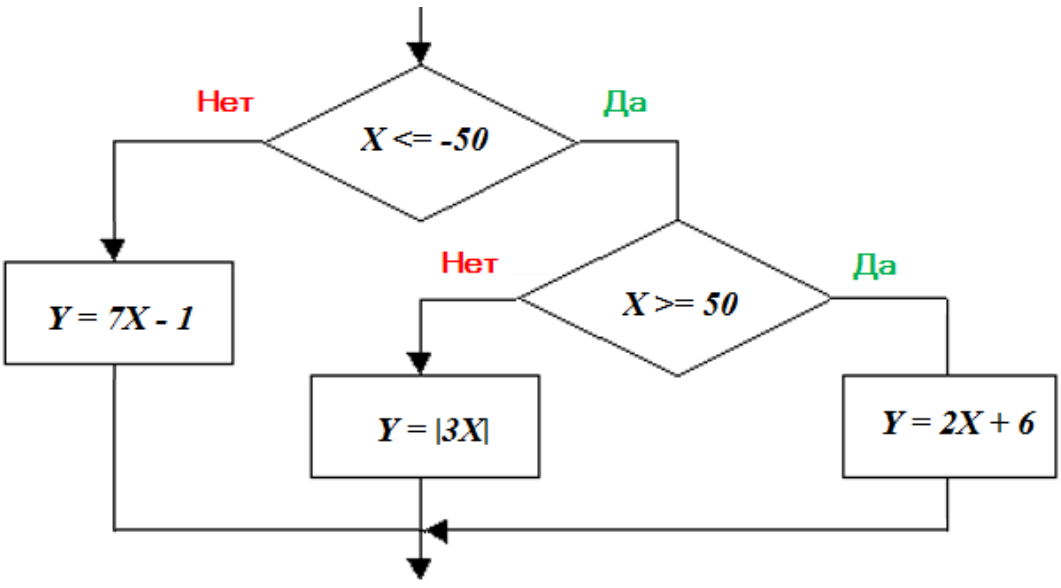


Рис. 4.1 – Анализ работы разветвлённого алгоритма

13) При каких начальных значениях переменных алгоритм на блок-схеме Рис. 4.2 закончит работу ($a \ mod \ 2$ = остаток от деления a на 2)?

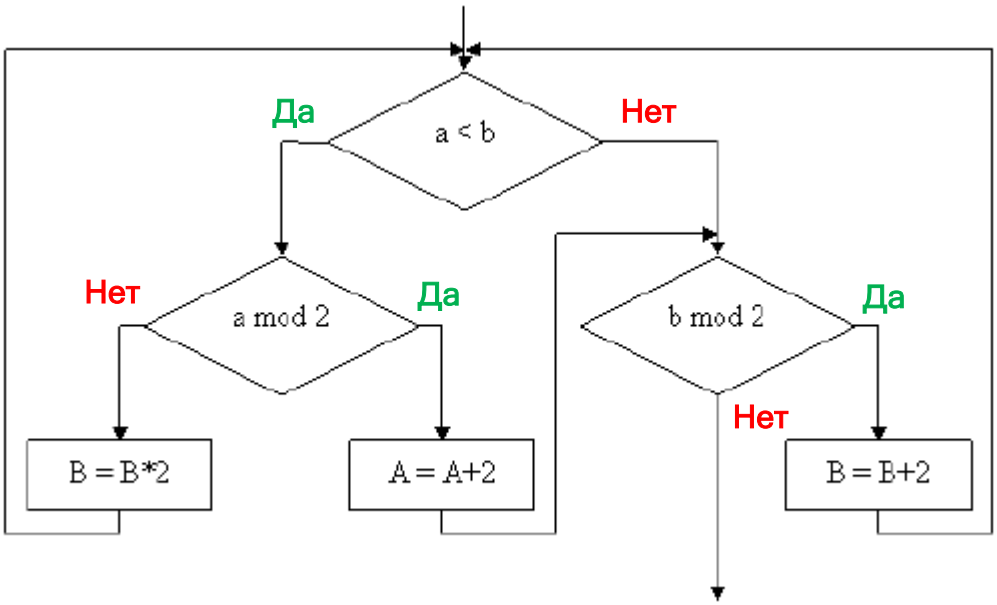


Рис. 4.2 – Анализ условий выхода разветвлённого алгоритма 13

14) При каких начальных значениях переменных алгоритм на блок-схеме Рис. 4.3 закончит работу?

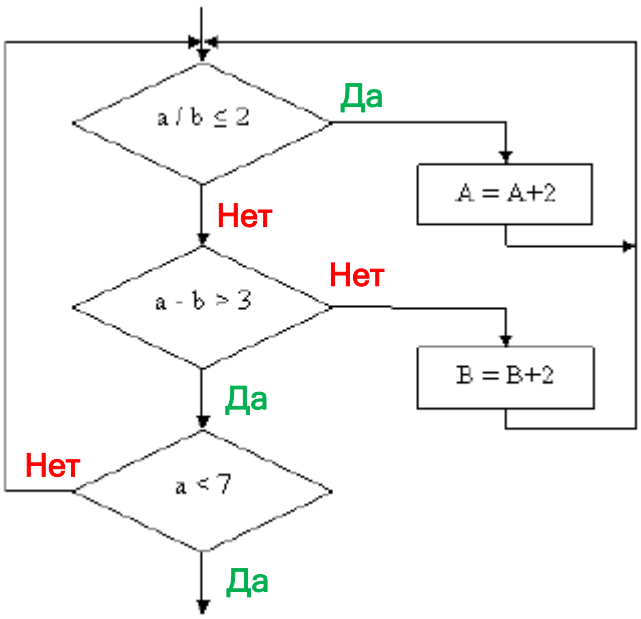


Рис. 4.3 – Анализ условий выхода разветвлённого алгоритма 14

15) В каких случаях возникает необходимость использования *оператора выбора* case? Перечислите основные особенности использования оператора case.

16) Можно ли вообще обойтись без оператора case?

17) Что произойдёт, если проверяемое условие — ложно, а часть оператора, стоящая после служебного слова else, отсутствует?

18) В чем сущность процесса отладки?

5 Операторы управления программой

5.1 Постановка задачи

Рис. 5.1 - Налево или направо?

А может прямо?

Нередко в зависимости от ситуации, возникшей в ходе решения задачи, нужно вы брать один из двух или более вариантов решения (Рис. 5.1). Выбор той или иной ветви вычислительного процесса в алгоритмах *с разветвляющейся структурой* осуществляется в зависимости от выполнения поставленного условия.

Для организации изменения *естественного порядка выполнения операторов* (так как выполняются операторы в линейной программе - один за другим) в языке ***Free Pascal*** предусмотрены следующие управляющие операторы:

- составной оператор ***begin . . end***;
- оператор *безусловной передачи управления* ***GoTo***;
- оператор *ветвления (условный)* ***if . . then . . else . .***;
- оператор *выбора (варианта)* ***case . . of***.

Как видим, в ***Free Pascal*** имеется два типа условных оператора ветвления: ***If*** и ***case***. Причём *условным оператором* чаще всего называют оператор ***If***, а оператор ***case*** именуют *оператором выбора* или *оператором варианта.*

5.2 Составные операторы

В большинстве структурных операторов правила построения конструкций языка ***Free Pascal*** допускают использование *только одного оператора*. В тоже время, нередко алгоритм в этом месте программы предусматривает выполнение целой группы операторов. Чтобы обойти это ограничение применяют *операторные скобки* - ***составной оператор.***

Составной оператор объединяет группу из произвольного числа операторов, после чего эта группа операторов воспринимается как единый оператор.

Синтаксическая диаграмма составного оператора приведена на (Рис. 5.2).

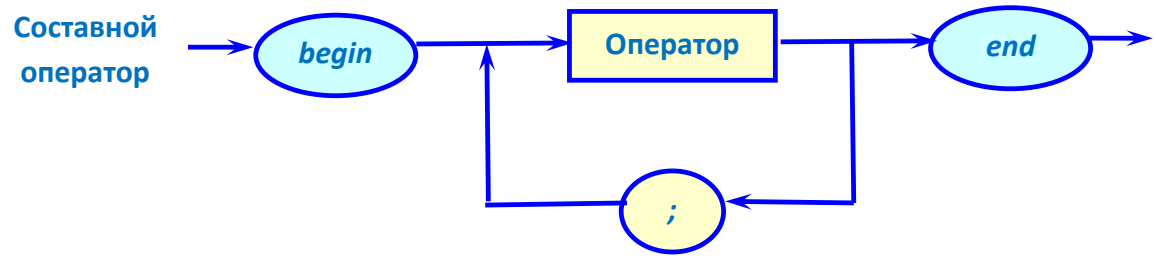


Рис. 5.2 - Синтаксическая диаграмма составного оператора

Во многих языках программирования (в том числе и *free Pascal*) условный оператор ***if***, циклы ***while*** и ***for*** требуют в качестве ветвей и тела один оператор.

Поэтому при необходимости разместить в ветвях условного оператора или теле цикла несколько команд используются *составные операторы:*

begin {начало составного оператора – открывающая операторная скобка}
оператор 1; {группа операторов *1, 2, ..., N* рассматриваемая как единое целое}
оператор 2;
…
оператор N
end; {конец составного оператора – закрывающая операторная скобка}

Таким образом, весь раздел операторов, обрамлённый словами ***begin .. end***, представляет собой *один составной оператор*[1].

Запомните!

Поскольку зарезервированное слово ***end*** является закрывающей операторной скобкой, оно одновременно указывает и конец предыдущего оператора. Поэтому ставить перед ***end*** символ «;» **необязательно**, так как оператор ***end*** является разделителем такого же уровня, как и «***;***».

5.3 Логические выражения

Логические выражения - это выражения, результат вычисления которых «истина» или «ложь». В логических выражениях могут использоваться операции отношения, логические и арифметические операции. Часто переменные, которые хранят логические значения, называются *булевыми переменными*[2], а операторы, которые производят над ними действия называют *булевыми операторами* или просто *логическими операторами.*

5.3.1 Логический тип данных

Данные логического типа в могут принимать только два значения: истина (***true***) или ложь (***false***).

Таблица 5.1 представляет допустимые логические типы данных, определённые

Таблица 5.1 - Возможные типы логических данных в среде Lazarus

Тип	Занимает память, байт	Внутреннее представление истины
Boolean	1	1
ByteBool	1	Любое ненулевое значение
WordBool	2	Любое ненулевое значение
LongBool	4	Любое ненулевое значение

[1] С такой точки зрения любая программа или подпрограмма на *Free Pascal* состоят из единственного оператора – составного.

[2] **Булевы переменные и функции** названы по фамилии английского математика и логика Джорджа Буля.

Обычно используют тип ***boolean***, остальные типы применяют при необходимости обеспечения совместимости с другими языками.

Данные логического типа относятся к порядковым данным языка Free Pascal. Значение функции ***Ord*** позволяет определить последовательность значений ***False*** и ***True*** (обычно $False < True$, соответственно значения функции ***Ord*** равны 0 и $\mathbf{1}$).

Переменные логического типа определяют в разделе объявления переменных:

```
Var
  Error, Flag: Boolean;
```

Присваивание значения для переменных логического типа:

```
Error:= True; Flag := False;
```

В правой части от знака присваивания могут также находиться выражения, результат которых логическая величина.

Определить значение логической переменной с помощью ввода информации во время работы программы *нельзя*.

5.3.2 Операции отношения

Операции отношения и эквивалентности являются самыми распространёнными в логических выражениях и используются при сравнении двух операндов. Они возвращают ***true*** - истина, если указанное соотношение операндов выполняется, и ***false***$(\mathbf{0})$ - ложь, если соотношение не выполняется (Таблица 5.2).

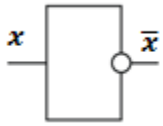
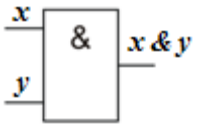
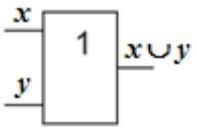
Таблица 5.2 - Операции отношения

Математическая операция	Запись операции на Free Pascal	Выражение на Free Pascal	Результат
= (равно)	**=**	***A = B***	***True***, если ***A*** равно ***B***
≠ (не равно)	**<>**	***A <> B***	***True***, если ***A*** не равно ***B***
> (больше)	**>**	***A > B***	***True***, если ***A*** больше ***B***
≥ (больше или равно)	**>=**	***A >= B***	***True***, если ***A*** больше или равно ***B***
< (меньше)	**<**	***A < B***	***True***, если ***A*** меньше ***B***
≤ (меньше или равно)	**<=**	***A <= B***	***True***, если ***A*** меньше или равно ***B***

5.3.3 Логические операции

При помощи знаков *логических операций* (Таблица 5.3) из простых составляются более сложные логические выражения. Заметим, что в отличие от операций отношения, выполняемых над данными численного типа (***Integer***, ***Real*** и т.п.), логические операции выполняются *только* над данными типа ***boolean***.

Таблица 5.3 - Основные логические операции

Математическая операция	Запись операции в Free Pascal	Таблица истинности	Логический элемент
$\neg x$ или $\overline{x}$ (логическое отрицание, логическое «НЕ», инверсия)	**not x**	x | $\overline{x}$ 0 | 1 1 | 0	 Инвертор
&, « · », $\wedge$, $\cap$ (конъюнкция, пересечение, операция «И», логическое умножение)	**x and y**	x | y | $x \& y$ 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1	 Логический элемент 2И
$\vee$, $\cup$ (дизъюнкция, объединение, операция «ИЛИ», логическое сложение)	**x or y**	x | y | $x \cup y$ 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 1	 Логический элемент 2ИЛИ

5.3.4 Логические выражения

При вычислении логических выражений операции выполняются в следующем порядке: ***not***, ***and***, ***or***, операции отношения, арифметические операции. Если порядок выполнения операций нужно изменить, то применяют скобки.

Запомните!

В Pascal логические операции имеют более высокий приоритет, чем операции отношения. Поэтому каждая операция отношения в логическом выражении обязательно заключается в скобки.

Примеры записи логических выражений:

Математическая операция	Запись выражения на Pascal
$5 > 3$	$5 > 3$
$2 \leq 6$	$2 <= 6$
$2x + 5 \neq 0$	$(2 * x + 5) <> 0$
$a = b = c$	$(a = b)\ and\ (b = c)$
$0 < x < 100$	$(x > 0)\ and\ (x < 100)$
$-100 \geq x \geq 100$	$(x <= -100)\ or\ (x >= 100)$
$\overline{a}\ \&\ (b \cup c)$	$not\ a\ and\ (b\ or\ c)$

5.4 Условный оператор *if*

При решении многих задач порядок вычислений зависит от определённых условий, например, от исходных данных или от промежуточных результатов, полученных в предыдущих вычислениях.

Условный оператор if (Рис. 5.3) позволяет во время выполнения программы проверить некоторое *логическое условие* и в зависимости от результатов проверки выбрать одно из двух направлений дальнейших вычислений (последовательности инструкций, которые *должны быть выполнены*).

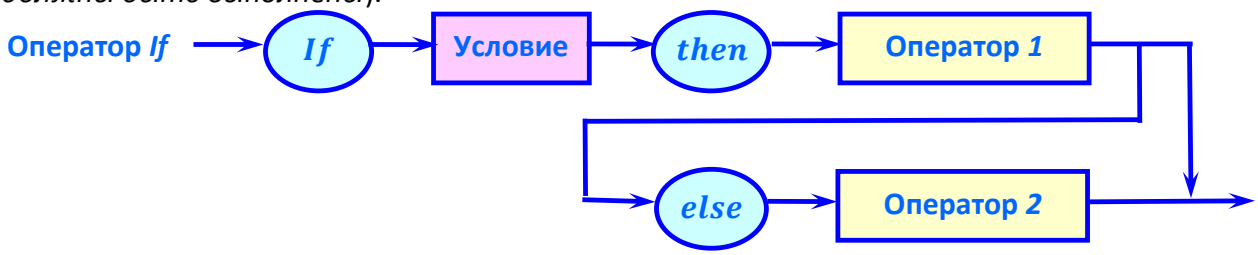


Рис. 5.3 - Синтаксическая диаграмма оператора ***If***

Зарезервированные слова $\boldsymbol{if}, \boldsymbol{then}, \boldsymbol{else}$ соответственно обозначают «***если***», «***то***», «***иначе***».

Выполнение условного оператора начинается с вычисления логического выражения, записанного в условии[3]. Если оно *истинно* ($\boldsymbol{true}$, «да», $\mathbf{1}$), то выполняется *единственный* **Оператор 1***,* стоящий *после* служебного слова $\boldsymbol{then}$. Если условие *ложно* (***false***, "***нет***", ***0***), то выполняется *единственный* **Оператор 2**, стоящий после служебного слова $\boldsymbol{else}$.

Проверяемое условие записывается в виде *логического выражения,* которое может состоять из одного или нескольких *отношений* (см. Таблица 5.2), выполняющих сравнение двух операндов и определяющих, *истинно* значение отношения - $\boldsymbol{true}$ или *ложно* - $\boldsymbol{false}$.

Оператор ***if*** можно записать в полной и сокращённой формах. определённой

5.4.1 Полная форма оператора ***If***

Оператор ***if*** в полной форме записывается так[4]:

```
if Условие          //если результат вычисления логического условия - True (истина)
  then Оператор1    //то выполняется Оператор 1
  else Оператор2;   //иначе, если Условие – False (ложь), выполняется Оператор 2
```

[3] Результатом проверки логического условия может быть только значение булевского типа (***True*** или ***False***)!

[4] Структуру программы можно сделать максимально ясной и понятной, если использовать **отступы**. **Отступ** (отклонение от края колонки одной или нескольких строк, идущих подряд на два – три пробела) используют, чтобы показать вложенность или подчинённость определённой группы операторов некоторому зарезервированному слову ($\boldsymbol{begin}, \boldsymbol{then}, \boldsymbol{else}, \boldsymbol{Type}, \boldsymbol{Var}$ и др). **Вертикальный отступ** (пустые строки) делают, чтобы разделить логические связанные группы операторов внутри одного компонента.

На (Рис. 5.4) приведена блок-схема оператора ***if***, позволяющего осуществить ветвление на два направления

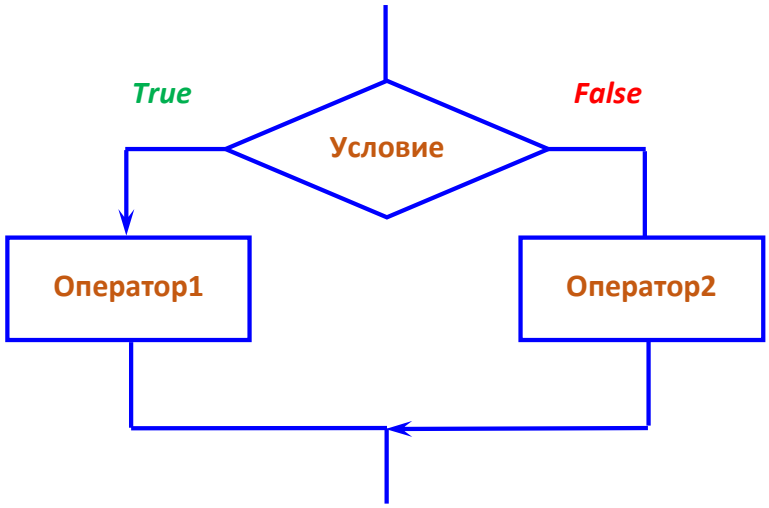


Рис. 5.4 – Блок-схема оператора ветвления ***if*** в полной форме

Пример 5.1.

Выбрать наименьшее из чисел a и b и присвоить его значение переменной Min.

Решение.

Блок-схема алгоритма решения этого примера приведена на Рис. 5.5.

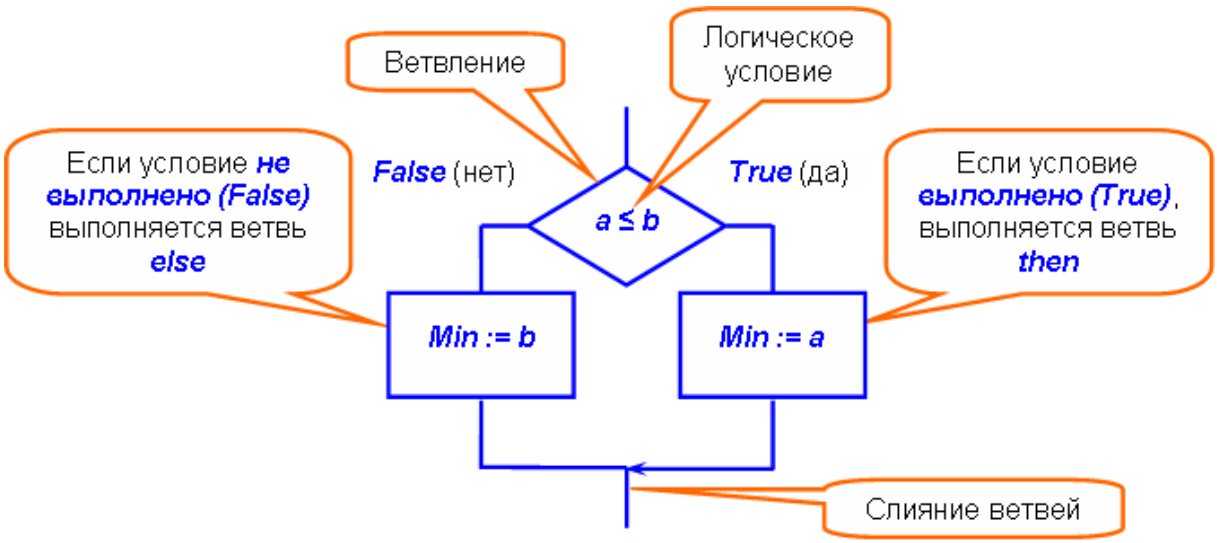


Рис. 5.5 - Блок-схема решения задачи выбора наименьшего из двух чисел ***a*** и ***b***

Фрагмент программы определения наименьшего из двух чисел на ***Free Pascal*** может выглядеть так:

```
...
If  a <=  b        //если выполняется условие  a ≤ b,
  then Min := a          //то переменной Min присваивается значение переменной a}
  else Min := b;//иначе переменной Min присваивается значение b
...
```

5.4.2 *Ошибки в записи оператора If в полной форме*

Обратите внимание на следующие особенности записи оператора ***If***.

Запомните!

Перед ключевым словом ***else*** символ «;» никогда не ставится, так как он заканчивает любой оператор, а оператора, который начинался бы с ***else*** в Free Pascal нет!

Пример 5.2 (с синтаксической ошибкой)

```
...
If  a <=  b
  then Min := a ;  ⇐ ошибка!
  else Min := b;
...
```

5.4.3 *Неполная форма условного оператора If*

Альтернативная ветвь ***else*** в условном операторе может отсутствовать, если в ней нет необходимости. Как видно из синтаксической диаграммы оператора ***If*** (см. Рис. 5.3), допускается *неполная форма* условного оператора ***If***:

```
if Условие          //если результат вычисления логического условия - True (истина)
  then Оператор;    //то выполняется Оператор, иначе действие пропускается
```

Подобную форму усечённого ***If*** (без альтернативной ветви ***else***) называют **Обход**, а её блок-схема приведена на Рис. 5.6.

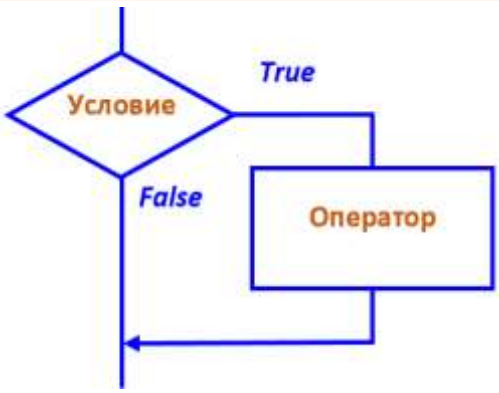


Рис. 5.6 - Блок-схема неполного оператора ***if*** (Обход)

5.4.4 *Вложенные операторы if*

В качестве оператора в каждой ветви условного оператора может применяться любой исполняемый оператор ***Free Pascal***, в том числе и сам оператор ***if***. В этом случае говорят о *вложенности* операторов ***if***.

Нередко (особенно, если применяются неполные формы ***if***), возникает неоднозначность – какому из вложенных ***if*** соответствует ветвь ***else.***

Пример 5.3.

Как работает следующий фрагмент программы?

```
...
If Условие1 then If Условие2 then Оператор1 else Оператор2;
...
```

Синтаксически всё верно, но какой же из вложенных операторов ***if*** управляет оператором в ветви ***else***? Разобраться сложно, если не знать простое правило:

Запомните правило!

Ключевое слово ***else*** связывается с **ближайшим** стоящим перед ним ключевым словом ***if***, которое ещё не было связано с каким-либо ключевым словом ***else***

Если проанализировать с помощью этого правила предложенный в *примере 5.3* фрагмент программы и на этой основе более чётко обозначить структуру вложенности, записав после ***if*** с условием ***else*** и ***then*** в отдельных строках с отступами на одном уровне, то получим достаточно понятный фрагмент программы:

```
...
If  Условие1
  then if Условие2
          then Оператор 1;
          else Оператор 2;
...
```

Синтаксически в операторе ***if*** после ключевых слов ***then*** и ***else*** может стоять *всего лишь один оператор*. Конечно, применить составной оператор ***begin*** и ***end*** в этом случае не имеет смысла. Составные конструкции с ***begin..end*** употребляют только для смыслового объединения группы операторов, для использования в структурных операторах.

Существенно повышает облегчает понимание структуры программы и уменьшает число ошибок запись управляющих операторов с отступами.

Опыт программистов показывает, что при использовании оператора ***If*** целесообразно использовать формы записи, приведённые ниже (Таблица 5.4).

Таблица 5.4 - Оптимальные варианты записи оператора ***If***

Количество операторов в ветви *then*	*else*	Обобщённая форма оператора *If*
один	один	***If Условие*** ***then* Оператор 1** ***else* Оператор 2**
несколько	один	***If Условие*** ***then begin*** **Оператор 1**; **Оператор 2**; **…** **Оператор N** ***end*** ***else* Оператор**
один	несколько	***If Условие*** ***then* Оператор** ***else begin*** **Оператор 1;** **Оператор 2;** **…** **Оператор N** ***end***
несколько	несколько	***If Условие*** ***then begin*** **Оператор 1;** **Оператор 2;** **…** **Оператор N** ***end*** ***else begin*** **Оператор 1;** **Оператор 2;** **…** **Оператор N** ***end***

5.5 Оператор варианта (*case*)

Условный оператор ***if*** даёт возможность выбирать лишь одно из двух возможных действий в зависимости от результата вычисления логического выражения. Если же при написании программы используются многократно вложенные друг в друга условные операторы ***if***, то программа становится громоздкой и сложно воспринимаемой.

Оператор выбора (варианта, переключателя) ***case*** (Рис. 5.7) даёт возможность выполнять одно из нескольких возможных направлений дальнейшего хода программы в зависимости от значения *выражения-переключателя (селектора)*.

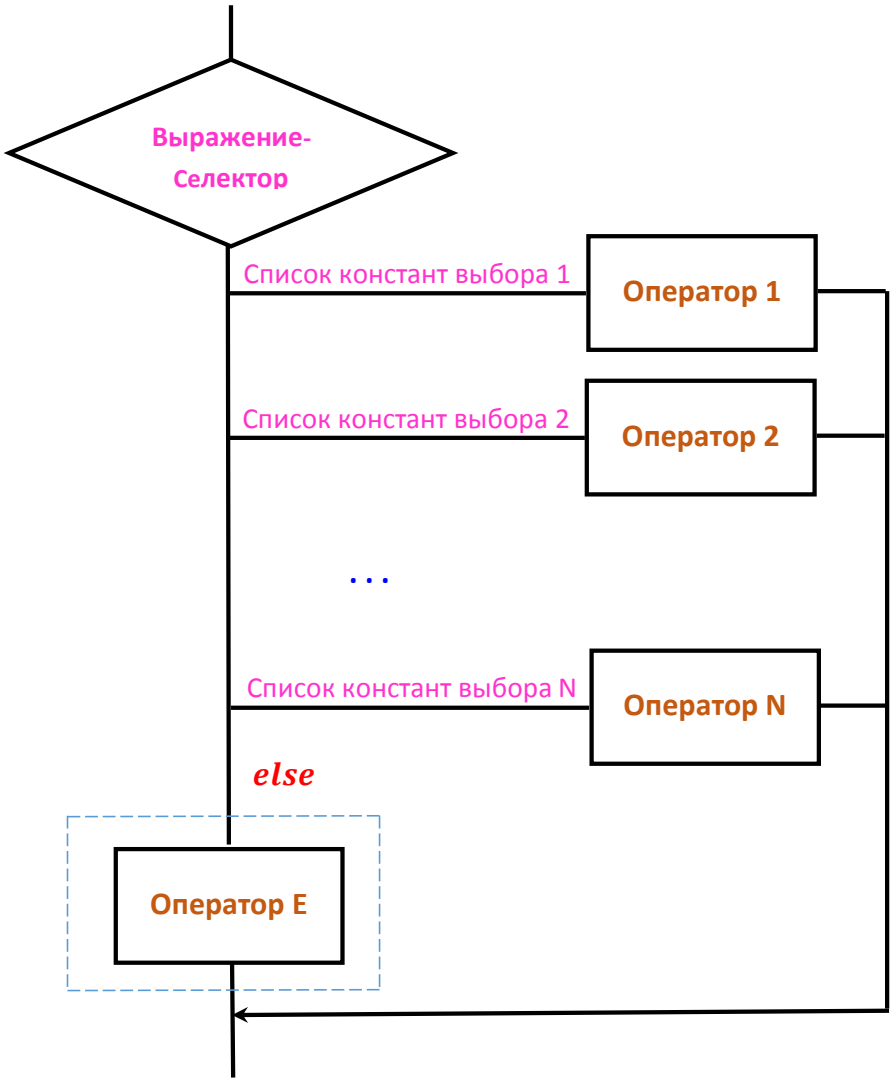


Рис. 5.7 - Блок-схема оператора ***case*** (варианта)

Как видно из Рис. 5.7, оператор варианта ***case*** состоит из выражения-переключателя (*англ*. *Selector* - переключатель) и списка операторов, каждому из которых предшествует одна или более констант (они называются константами выбора), и (необязательно) ветви ***else***.

Переключатель обязательно должен иметь порядковый тип (***Shortint***, ***Integer***, ***Word***, ***Byte***, ***Char***, а также перечисляемые и интервальные типы) размером в байт или слово.

Списки констант выбора могут состоять из произвольного числа *значений* или *диапазонов*, отделённых друг от друга запятыми. Границы диапазона указывают наименьшим и наибольшим значениями, разделёнными двумя точками «**..**» (например, ***-128 .. 127, 'a' .. 'z', 1 .. 100***).

Все константы выбора должны быть уникальными и иметь порядковый тип, совместимый с типом переключателя и не превышать допустимых для этого типа значений.

Записывается оператор ***case*** так:

```
...
case <Переключатель> of //<Переключатель> – выражение порядкового типа
  <Список констант выбора 1>: <Оператор 1>;
  <Список констант выбора 2>: <Оператор 2>;
  ...
  <Список констант выбора N>: <Оператор N>
else  <Оператор E>
end;
...
```

Оператор варианта ***case*** приводит к выполнению оператора, которому предшествует константа выбора, равная *вычисленному значению переключателя,* или *диапазону выбора*, в котором находится значение переключателя. Если такой константы выбора или такого диапазона выбора не существует, но присутствует ветвь ***else***, то выполнятся оператор, следующий за ключевым словом ***else*** (**Оператор E**). Если же такой ветви не существует, то никакой оператор не выполняется.

Приведём характерные примеры использования оператора ***case***.

Пример 5.5. Решаем задачу создания калькулятора. Для выполнения определённой операции над операндами ***A*** и ***B***, вид которой задан значением переключателя ***Symbol*** (тип ***Char*** – это номера кодов символов по таблице) можно использовать фрагмент программы, приведённой ниже:

```
...
case Symbol of // Symbol переменная символьного типа Char
  '+':    C := A + B; //выполняется сложение, если Symbol = '+'
  '-':    C := A - B;  //выполняется вычитание, если Symbol = '-'
  '*':    C:= A * B;  //выполняется умножение, если Symbol = '*'
  '/':    C:= A / B  //выполняется деление, если Symbol = '/'
end;
...
```

Пример 5.6. Используем оператор выбора ***Case*** для определения дня недели***.***

Пусть переменная ***Num_Day*** (тип ***Integer***) хранит номер дня недели (от 1 до 7), а переменная ***Day*** типа ***String –*** строку символов с названием дня.

Возможные варианты решения примера 5.6 приведены ниже.

Вариант 1.

```
...
case Num_Day of // Num_Day  переменная порядкового типа
   1, 2, 3, 4, 5: Day := 'Рабочий день. '; //Перечисляем рабочие дни
  6: Day := 'Суббота!';
  7: Day := 'Воскресенье!'
end;
...
```

Вариант 2.

```
...
case Num_Day of // Num_Day  переменная порядкового типа
   1 .. 5: Day := 'Рабочий день. '; //Используем диапазон рабочих дней
  6: Day := 'Суббота!';
  7: Day := 'Воскресенье!'
end;
...
```

Вариант 3.

```
...
case Num_Day of // Num_Day  переменная порядкового типа
  6: Day := 'Суббота!';
  7: Day := 'Воскресенье!'
else
  Day := 'Рабочий день' //кроме Субботы и Воскресенья все дни рабочие
end;
...
```

5.6 Безусловное изменение естественного порядка выполнения операторов

Можно теоретически показать, что рассмотренных операторов вполне достаточно для написания программ любой сложности. В этом отношении наличие в языке операторов перехода кажется излишним. Более того, современная технология структурного программирования основана на принципе «программировать без ***GoTo***».

Правила структурного программирования требуют *минимального использования*, которые делают программу запутанной, малопонятной и сложной в отладке. Кроме того,

при употреблении *операторов безусловного перехода* резко возрастает вероятность появления ошибок в логике программы.

Запомните!

Богатый выбор условных и циклических операторов *Free Pascal* позволяет писать логически чёткие программы практически без меток и операторов

Организацию безусловного перехода к оператору с меткой ***N*** осуществляют с помощью оператора безусловного перехода ***GoTo***.

Метка в ***Free Pascal*** - это произвольный идентификатор, позволяющий именовать некоторый оператор программы и таким образом ссылаться на него. В целях совместимости со стандартным языком Паскаль в языке ***Free Pascal*** допускается в качестве меток использование также целых чисел без знака в диапазоне от 0 до 9999.

Метка располагается непосредственно перед помечаемым оператором и отделяется от него двоеточием. Оператор можно помечать несколькими метками, которые в этом случае отделяются друг от друга двоеточием. Перед тем как появиться в программе, метка должна быть описана с помощью предложения ***Label N,*** где ***N*** - метка.

Обязательным является требование, чтобы метка, указанная в операторе перехода, располагалась в том же блоке или модуле, что и сам оператор ***GoTo***. Следовательно, запрещены как передача управления внутрь процедуры или функции, так и передача управления из подпрограммы в другую подпрограмму или вызывающую программу.

Запомните!

Оператор, следующий непосредственно за оператором безусловной передачи управления ***GoTo N***, обязательно должен иметь метку, так как иначе он **никогда не сможет выполниться!**

Пример 5.7. Программа, использующая операторы безусловного перехода

```
Program DemoGoTo; {Демонстрация работы операторов безусловного перехода}
Label  {Раздел объявления меток}
  Metka1, Metka2;
begin {Начало основного блока программы DemoGoTo}
  writeln('Демонстрация применения операторов безусловного перехода');
  GoTo Metka2; {Изменение естественного порядка выполнения операторов
программы - безусловный переход к оператору с меткой Metka2}
  Metka1: writeln('Работает оператор с меткой Metka1');
  Halt; {Завершение работы программы DemoGoTo}
  Metka2: writeln('Работает оператор с меткой Metka2');
  GoTo Metka1 {Безусловный переход к оператору с меткой Metka1}
end. {Конец основного блока программы DemoGoTo}
```

5.7 Организация разветвления вычислительного процесса более чем на два направления

В общем случае количество ветвей в алгоритме разветвляющейся структуры *не обязательно равно двум*.

5.7.1 *Использование оператора* if

Пример 6.1. Вычислить значение функции, заданной формулой:

$$z = \begin{cases} \sin x, \text{если } x \le a; \\ \cos x. \text{ если } a < x < b; \\ \operatorname{tg} x, \text{если } x \ge b; \end{cases}$$

В этом примере функция z задана различными формулами на разных интервалах. В зависимости от значения проверяемого условия, нужно выбрать одну их трёх ветвей для продолжения процесс вычисления по «своей» формуле.

Для того, чтобы получить количество ветвей большее двух, можно каждую из ветвей оператора If «расщепить» ещё *на два направления* с помощью добавленных условных операторов $if\ldots then\ldots else$. Заметим, что после любого такого разветвления расщепленные ветви обязательно должны *снова слиться*.

При выполнении условия $x \le a$ в условном блоке «развилка» (Рис. 5.8) можно попасть в первую из ветвей расчёта, где вычисляется $z = sin(x)$. Если это условие не выполняется (т.е. $x > a$), то мы , очевидно, попадём во второй или третий интервал. Чтобы определить – в какой именно, следует использовать ещё один условный блок (обведён пунктиром), позволяющий расщепить ветвь $else$ ещё на два направления.

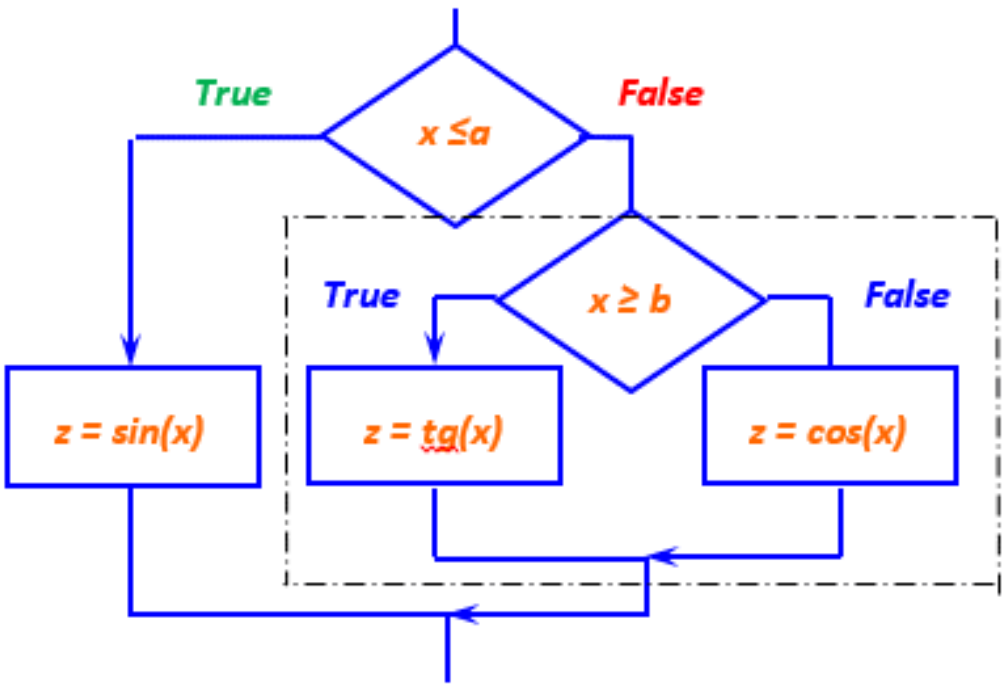


Рис. 5.8 - Разделение вычислительного процесса на три направления

После вычислений по любой из формул осуществляется слияние всех ветвей к общему продолжению.

Фрагмент программы для расчёта функции из примера 6.1 на ***Free Pascal*** (без ввода-вывода и объявления переменных) может выглядеть так:

```
…
if  x <= a   //проверяем первое условие x <= a
  then z := sin(x)  //если  первое условие выполняется, то z := sin(x)
  else if  x >= b      //иначе проверяем третье условие
        then z  :=  tg(x)  //если третье условие выполняется, то z  :=  tg(x)
        else z := cos(x); //иначе, если третье условие не выполняется, z := cos(x)
…
```

Пример 6.2. Вычислить значение зависимости напряжения от времени ***U(t)***, меняющегося в соответствии с функцией, кусочно-заданной графически (все кривые – фрагменты окружности) (**Ошибка! Источник ссылки не найден.**).

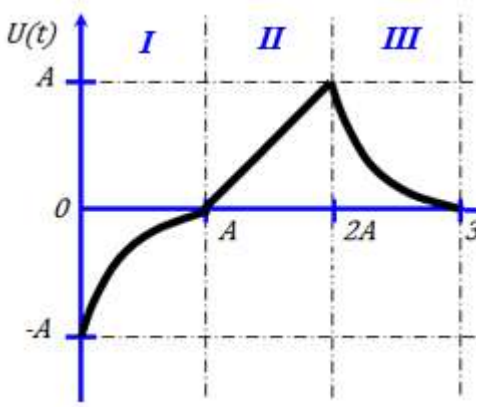


Рис. 5.9 – Кусочная функция напряжения U(t) задана графичес

В нашем примере для первого интервала $\boldsymbol{0 \le t \le A}$ *используется верхний фрагмент окружности* с координатами её центра в точке $(\boldsymbol{A}, -\boldsymbol{A})$, поэтому скорректируем нашу формулу:

$$\boldsymbol{U = +\sqrt{A^2 - (t - A^2)} - A}$$

Из математики также известно, что уравнение прямой, проходящей через начало координат (Рис. 5.10)

$$\boldsymbol{U = kt},$$

где *k* – угловой коэффициент наклона.

Если значения катетов численно равны, то $\boldsymbol{k = 1}$, и

$$\boldsymbol{U = t}.$$

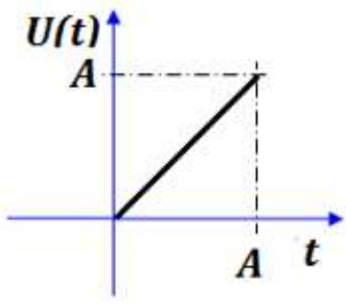


Рис. 5.10 - Прямая с центром в начале координат

Учитывая смещение относительно начала координат, получим для второго интервала $\boldsymbol{A \le t \le 2 \cdot A}$

$$\boldsymbol{U = t - A}.$$

Координаты центра окружности, нижний фрагмент которой используется на третьем интервале $2 * A \leq t \leq 3 \cdot A$ в точке $(3A, A)$, поэтому

$$U = -\sqrt{A^2 - (t - 3 \cdot A^2)} + A$$

Блок-схема решения поставленной задачи представлена на Рис. 5.11.

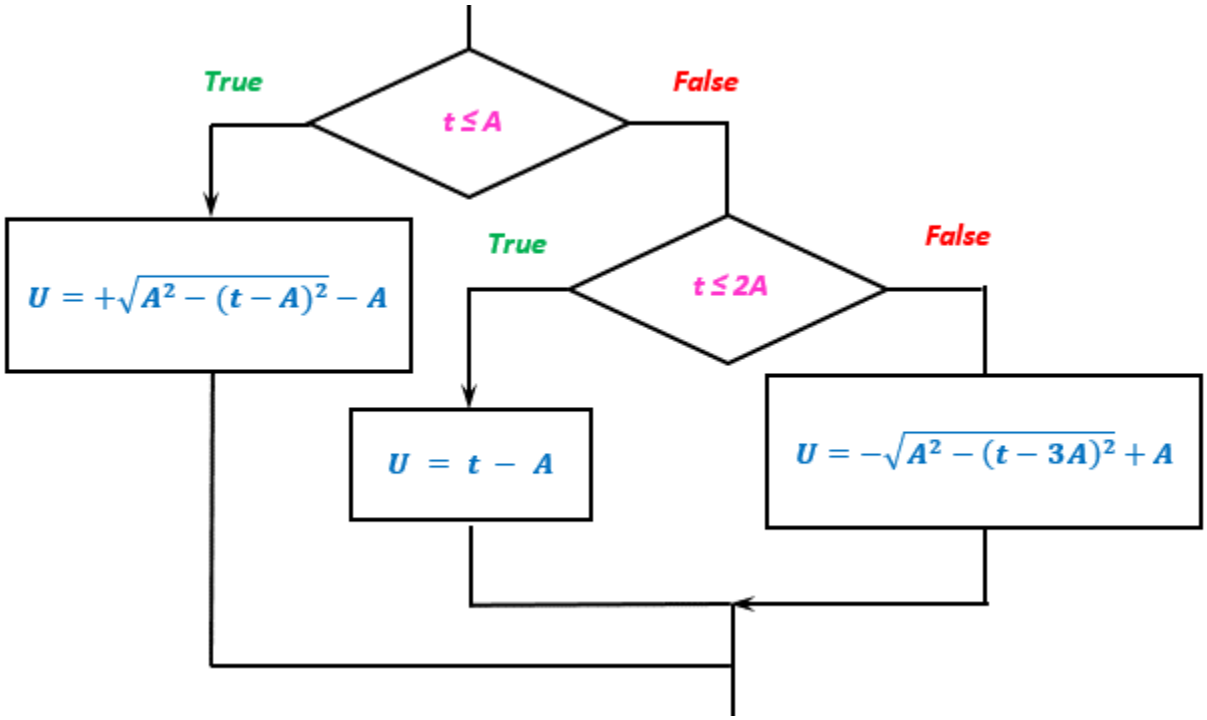


Рис. 5.11 – Использование вложенных операторов If в примере 6.2

Фрагмент программы на ***Free Pascal*** (без ввода-вывода и объявления переменных) может выглядеть так:

```
…
if  t <= A   //проверяем – t в первом интервале?
  then U := sqrt(A * A – sqr(t - A)) - A  //используем формулу для 1-го интервала
  else if  t <= 2 * A      //проверяем – t во 2-м интервале?
          then U  :=  t - A  { используем формулу для 2-го интервала}
          else U := -sqrt(A * A – sqr(t – 3 * A)) + A; /используем 3-ю формулу
…
```

5.7.2 Использование оператора варианта Case.

Чтобы воспользоваться оператором варианта ***case*** разумно получить выражение-селектор, определяющий порядковый номер расчётного интервала ***N***. Учитывая, что длины расчётных интервалов одинаковы и равны ***A,*** получим

$$N = trunc\left(\frac{t}{A}\right) + 1,$$

где встроенная функция $\boldsymbol{trunc}$ алгоритмического языка Free Pascal отбрасывает дробную часть вещественного числа и преобразует его к порядковому типу.

Фрагмент программы на *Free Pascal* (без ввода-вывода и объявления переменных) может выглядеть так:

```
…
case trunc(t / A) + 1 of //определяем порядковый номер расчётного интервала
  1: U := sqrt(A * A – sqr(t - A)) – A;        //используем формулу для 1-го интервала
  2: U  :=  t – A;                             //используем формулу для 2-го интервала
  3: U := -sqrt(A * A – sqr(t – 3 * A)) + A    //используем формулу для 3-го интервала
end;
…
```

5.8 Отладка программы

5.8.1 Этапы обработки программы компилятором

Чтобы программа на Free Pascal смогла выполняться на компьютере, её нужно перевести (оттранслировать) в двоичный машинный код, чтобы создать исполняемый файл.

Известны трансляторы (англ. *translator* - переводчик) двух видов:

1. **Трансляторы-интерпретаторы** (англ. *interpreter* — истолкователь, устный переводчик) переводят каждый оператор программы в машинный код и по мере перевода выполняют программу строка за строкой. При каждом очередном запуске программы интерпретатор должен заново переводить исходную программу на машинный язык

Интерпретируемые программы проще исправлять и изменять, однако они не могут работать вне транслятора-интерпретатора.

2. **Трансляторы-компиляторы** (англ. *compiler* - составитель, собиратель) читают всю программу целиком, делают её перевод и создают законченный вариант программы на машинном языке. И если этот перевод прошёл без ошибок, программа затем и выполняется. После того, как программа откомпилирована, ни сама исходная программа, ни компилятор больше не нужны.

Обычно откомпилированные программы работают быстрее, хотя сам процесс компиляции протекает дольше. При малейшем внесении изменений в исходный код требуется повторная компиляция.

Free Pascal использует транслятор-компилятор.

Вначале компилятор анализирует, имеются ли все необходимые функции и процедуры (такие, как $\boldsymbol{sin}(\boldsymbol{x}), \boldsymbol{cos}(\boldsymbol{x})$ и т.п.) в подключённых с помощью $uses$ внешних библиотеках-модулях, подвергает рассмотрению текст программы, проверяет синтаксические ошибки и в случае их отсутствия формирует *двоичный объектный файл* с расширением $\boldsymbol{.obj}$. Объектный файл (англ. *object file*) - это промежуточный файл отдельного модуля программы с неопределёнными адресами ссылок на данные и процедуры в других объектных модулях. Он может быть объединён с другими объектными файлами при помощи редактора связей (компоновщика) для получения готового исполнимого модуля, либо библиотеки.

Компоновщик собирает код и данные каждого объектного модуля в итоговую программу, вычисляет и заполняет адреса перекрёстных ссылок между модулями и генерирует исполняемый код программы. Связывание с операционной системой и динамическими библиотеками выполняется при исполнении программы после её загрузки в память.

5.8.2 Нахождение ошибок в программе

Отладка - это процесс поиска и исправления ошибок в программе, препятствующих её корректной работе.

> **Запомните !!**
>
> Текст новой или изменённой программы необходимо обязательно сохранить до (!) первого запуска программы на выполнение, так как из-за ошибок в программе или сбоя компьютера набранный текст может быть безвозвратно утрачен…

После набора текста программы и его сохранения можно приступить к отладке (тестированию) программы, во время которого приходится многократно запускать программу на выполнение.

5.8.3 Виды ошибок

Существует три основных типа ошибок:

1) ошибки этапа компиляции;
2) ошибки этапа выполнения;
3) логические (алгоритмические) ошибки.

Ошибки этапа компиляции (*Compile-time error*) (*синтаксические ошибки*) случаются, если Ваша программа записана с нарушением правил синтаксиса *Free Pascal*. Когда компилятор встречает оператор, который он не может распознать, процесс компиляции приостанавливается, соответствующий файл выводится в окне редактирования, курсор позиционируется на то место, которое не понял компилятор, и в окне сообщений выводится сообщение об ошибке. Такие ошибки могут возникнуть при неправильном написании оператора или имени подпрограммы, при попытке обратиться к переменной или константе, которую не объявляли (Рис. 5.12), при попытке вызвать подпрограмму, переменную или константу из модуля, который не был подключён в разделе ***uses***.

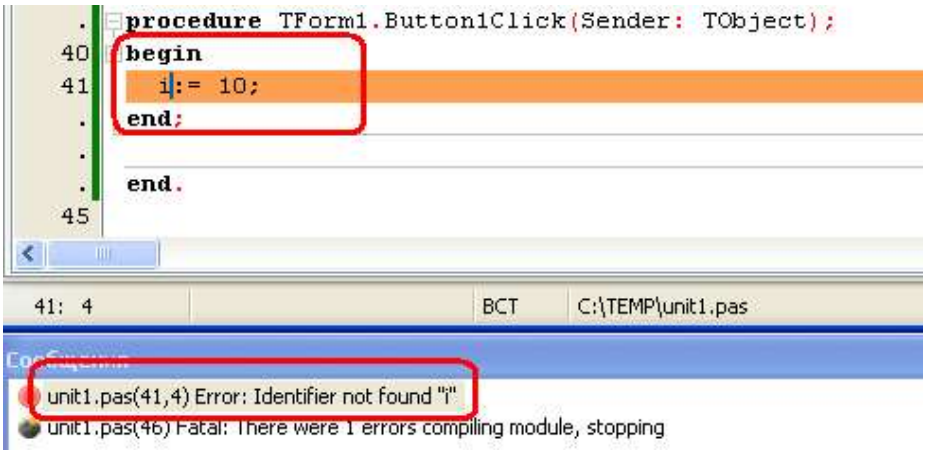


Рис. 5.12 - Синтаксическая ошибка - нет объявления переменной i

Весьма часто на этапе компиляции выявляются ошибки набора (опечатки), пропущенные точки с запятой, ссылки на неописанные переменные, передача неверного числа (или типа) параметров подпрограммы и присваивание переменной значений неверного типа.

Неочевидны ошибки, связанные с неверной записью *составного оператора*. Каждой открывающей скобке составного оператора ***begin*** должна соответствовать и закрывающая скобка ***end***. Если её нет, транслятор может позиционировать курсор на место, где с Вашей точки зрения ошибок нет. Во избежание таких ошибок разумно вносить в программу обе операторные скобки сразу, непосредственно друг за другом, а затем вписывать между ними содержание составного оператора.

После внесения в программу исправлений необходимо компиляцию повторить.

1. ***Ошибки этапа выполнения*** (***run – time errors***) происходят, когда Ваша синтаксически правильная откомпилированная программа при попытке загрузки или в момент совершения ошибки (происходит что-то неверно при выполнении операторов), аварийно приостанавливает свою работу и выводит на экран соответствующее сообщение. Например, программа пытается открыть для ввода несуществующий файл (Рис. 5.13), выполнить деление на ноль, не может выполнить какие-либо преобразования (например, строки символов в число) и т.п.

Рис. 5.13 - Пример сообщения Lazarus об ошибке времени выполнения

2. ***Логические (алгоритмические) ошибки*** - это ошибки проектирования и реализации программы. Случается, что компиляция программы завершается успешно и при пробных запусках программа ведёт себя нормально (нет ошибок времени выполнения), однако при анализе результата выясняется, что он неверный. Другими словами, Ваши операторы допустимы и что-то делают, но не то, что Вы предполагали. Следовательно, в программе имеются *алгоритмические ошибки.* Найти их можно только в процессе *тестирования программы*, позволяющего убедиться в том, что программа выполняет своё назначение.

Алгоритмические ошибки часто трудно отследить, поскольку IDE не может найти их автоматически, как синтаксические и семантические ошибки. Для того чтобы устранить алгоритмическую ошибку, приходится анализировать алгоритм, вручную «прокручивать» его выполнение. К счастью, IDE включает в себя средства отладки, помогающие Вам найти логические ошибки.

5.8.4 Методы отладки программ в интегрированной среде Lazarus

Часто причина, по которой программа делает что-то непредвиденное, достаточно очевидна, и Вы можете быстро внести исправления в исходный текст программы. Однако многие ошибки более трудноуловимы и зависят от работы различных частей программы.

В меню **Запуск** (Рис. 5.14) имеются команды выполнения которые дают возможность *построчного выполнения программы* (по шагам):

1) **Шаг со входом (F7)** - режим пошагового отладочного построчного выполнения программы со входом в вызываемые подпрограммы;

2) **Шаг в обход (F8)** - режим пошагового отладочного выполнения программы без входа в вызываемые подпрограммы;

3) **Шаг с выходом (Shift+F8)** – режим пошагового отладочного выполнения программы без входа в вызываемые подпрограммы, после чего переход к следующему оператору в текущей процедуре. Следовательно, отображаемый следующий оператор является следующим оператором в текущей процедуре, независимо от того является ли текущий оператор вызовом другой процедуры. Доступно только в режиме приостановки выполнения.

Выполнение программы по шагам или её трассировка могут помочь Вам найти ошибки в алгоритме программы. Например, операторы, где происходит деление на ноль, переполнение и т.п. Тем не менее, кроме этого обычно хотелось бы знать, что происходит на каждом шаге со значениями некоторых переменных, сравнивая их со значениями этих же переменных в заранее рассчитанном тестовом примере.

Подсвечивая строку выделенным цветом, встроенный отладчик всегда сообщает Вам, какую строку Вы выполняете следующей (строка выполнения).

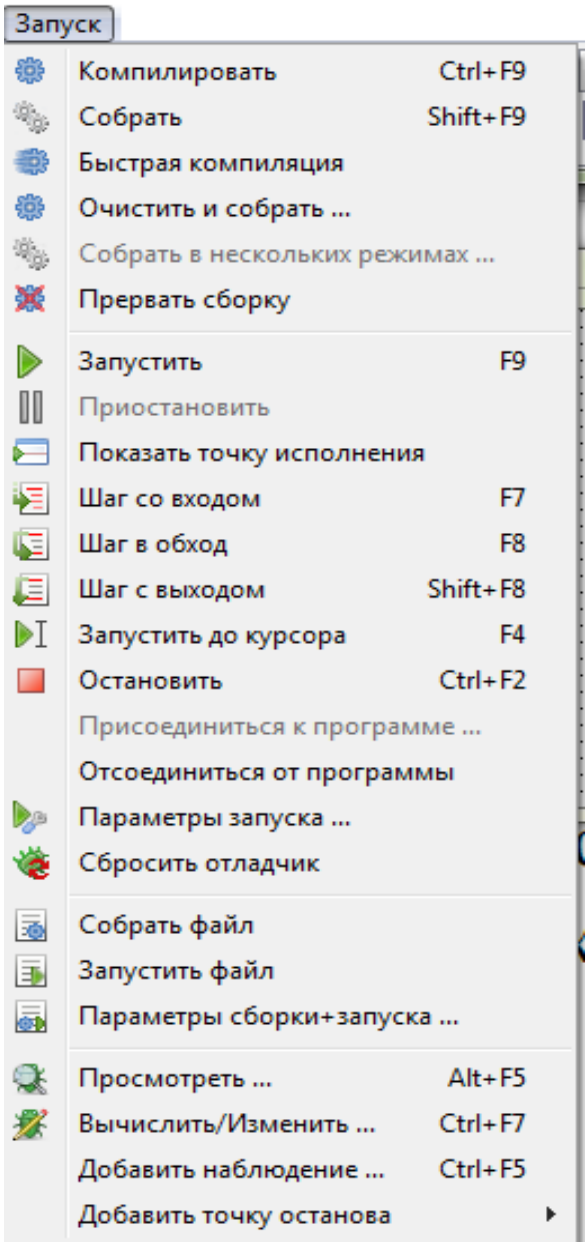


Рис. 5.14 - Команды меню **Запуск** позволяют запускать проект на выполнение и отладку

Запомните !!

Наименьшим выполняемым элементом при отладке является *строка*. Поэтому, если на одной строке программы содержится несколько операторов, Вы не сможете отладить эти операторы индивидуально.

С другой стороны, с целью отладки Вы можете разбить оператор на несколько строк, которые будут выполняться за один шаг.

Часть программы может быть выполнена *автоматически*, до определённой строки (*точки останова*[5]), а затем возможна её трассировка или выполнение по шагам. Существует два способа указания, что программу нужно выполнить автоматически до определённой точки, а затем остановить.

Первый способ состоит в том, чтобы установить курсор на строку в программе, где Вы хотите остановиться, затем выбрать в меню **Запуск** команду *Запустить до курсора* (***F4***). В точке останова Вы можете проверить значения и далее продолжать выполнение непрерывно или по шагам.

Второй способ состоит в том, чтобы установить в определённой строке Вашей программы точку останова с помощью команды *Добавить точку останова в меню* **Запуск**. Вы можете установить одну такую точку или несколько в разных местах программы. Очевидно, точки останова следует устанавливать *до тестируемого участка программы, а не после него*. При этом каждая строка с точкой останова будет выделена **красным цветом** (Рис. 5.15), а строка, которая будет выполнена далее – серым (Рис. 5.16).

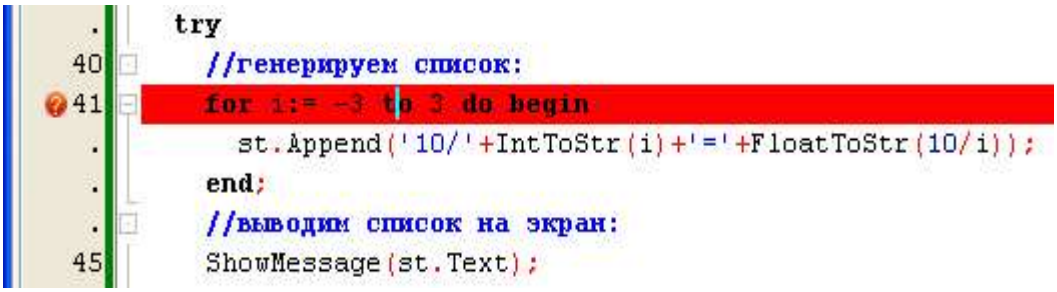


Рис. 5.15 - Строка с точкой останова

Рис. 5.16 - Строка, которая будет выполнена далее

Установить точку останова можно разными способами:

1) Командой главного меню «**Запуск** ⇒ **Добавить точку останова** ⇒ **Точка останова в исходном коде**». В открывшемся окне «Параметры точки останова» нажать "ОК".
2) Щёлкнуть по строке правой кнопкой, и в всплывающем меню выбрать «**Отладка** ⇒ **Переключить точку останова**».
3) Нажать «горячую клавишу» <**F5**>.
4) Щёлкнуть по нужной строке в левой части **Редактора кода**, где указаны номера строк.

Убрать точку останова можно последними двумя способами.

---

[5] **Точки останова** - это строки, перед выполнением которых отладчик приостанавливает выполнение программы, и ждёт ваших дальнейших действий. При запуске программы автоматически выполняются все операторы до точки останова и остановит выполнение программы. Далее вы можете продолжить выполнение программы по шагам.

Ещё один полезный инструмент **Встроенного отладчика** - возможность *просматривать значения* переменных, выражений и структур данных.

Нажмите кнопку «**Запустить**» (**F9**). После того, как программа приостановит выполнение программы на строке останова, добавьте наблюдение над текущими значениями отслеживаемых элементов с помощью команды «**Запуск ⇒ Добавить наблюдение**» (**Ctrl+F5**). Эта команда в меню (см. Рис. 5.14) становится доступной, после того, как программа оттранслирована и стала выполняться. В строке «**Выражение**» укажите имя проверяемой переменной (Рис. 5.17), и нажмите «**ОК**». Чтобы добавить новую переменную щёлкните правой клавишей в окне наблюдений, в открывшемся диалоговом окне введите имя переменной. Вместо переменной в строку «**Выражение**» можно ввести выражение (из существующих в тестируемой программной единице переменных), за значением которого мы желаем наблюдать.

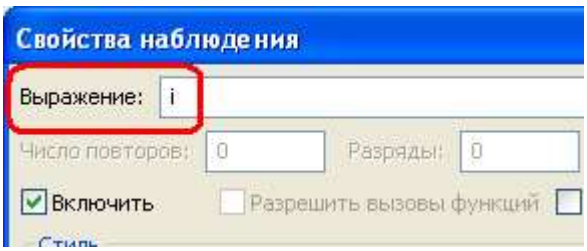


Рис. 5.17 - Установка наблюдения за переменной

Чтобы просматривать текущие значения отмеченных переменных, нужно вывести на экран окно **Список наблюдений** (Рис. 5.18) с помощью команды «**Вид ⇒ Окна отладки ⇒ Окно наблюдений**» или «горячими клавишами» <**Ctrl+Alt+W**>..

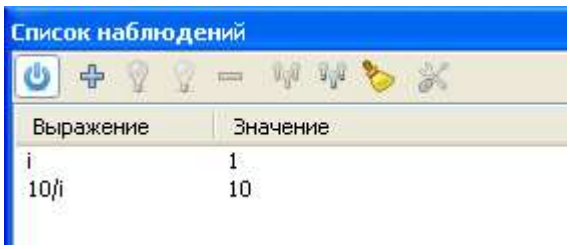


Рис. 5.18 - Окно

Увидеть значение переменной можно также просто наведя мышь на эту переменную в редакторе исходного кода.

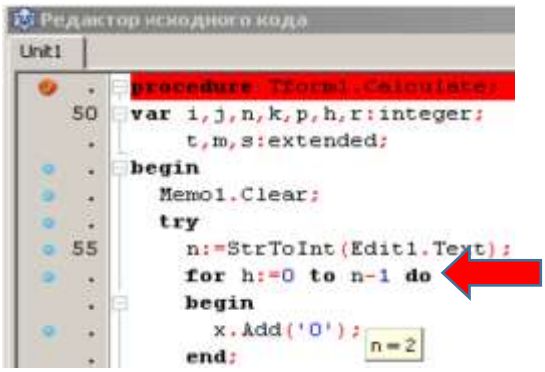


Рис. 5.19 - Мышь наведена на переменную n в 56 строке редактора

6 Варианты заданий для составления разветвлённых программ

6.1 Условие задания и рекомендации по его выполнению

Импульсный источник напряжения ***U(t)*** с внутренним сопротивлением R_U, заданный на периоде $0 .. T_U$, подключён к сопротивлению нагрузки R_H (Рис. 6.1).

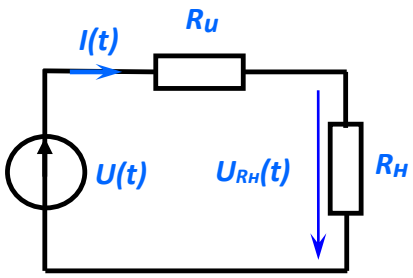


Рис. 6.1 – Расчётная электрическая схема

По формуле

$$U_{RH}(t) = U(t) * R_H / (R_H + R_U)$$

рассчитать напряжение на нагрузке $U_{RH}(t)$ для любого ведённого времени t ($0 < t < T_U$). Варианты функции напряжения ***U(t)*** **импульсного** источника заданы графически и приведены ниже.

Для быстрейшего установления вариантов формул расчёта напряжения на разных временных интервалах разумно воспользоваться приёмами, рассмотренными в *примере 6.1.*

Элементы, из которых состоят варианты функций источника напряжений - отрезки прямых и части окружностей. Напомним, что уравнение прямой, проходящей через начало координат $y = k * x$. В нашем случае угловой коэффициент может принимать значения $k = \{-1, +1\}$.

Уравнение окружности с центром в начале координат $y = \pm\sqrt{R^2 - x^2}$, где R – радиус, x, y – координаты окружности. Знак "+" соответствует верхней полуокружности, знак "-" - нижней.

Значение параметра A следует ввести с клавиатуры (удобно использовать $A = 10$). Для всех вариантов период импульса $T_U = 6 * A$.

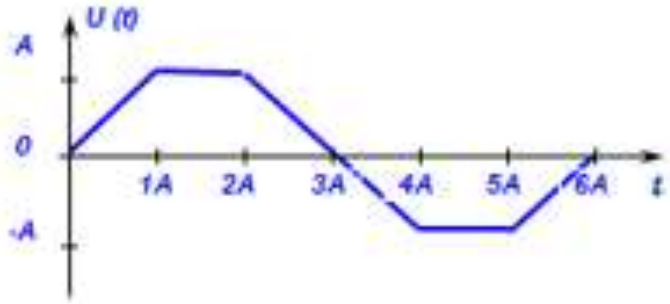
Для сдвига кривых по осям координат используйте формулы переноса координат.

Обязательно проверяйте правильность расчёта напряжения предварительной подстановкой соответствующих значений t в середине и на границах интервалов определения составляющих расчётной функции.

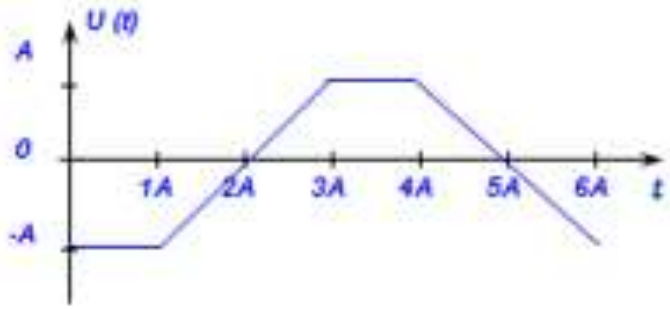
Для вывода на поверхность формы иллюстраций, представленных в bmp–формате. можно использовать компонент $TImage$.

6.2 Варианты функций источника напряжения

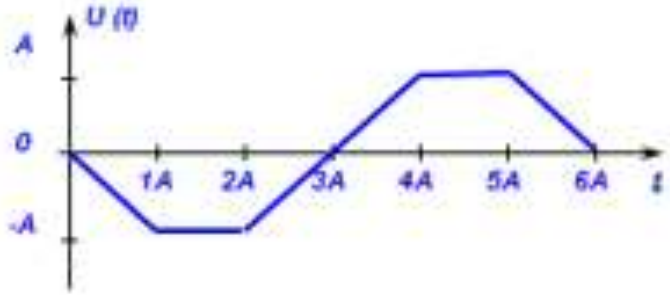
1



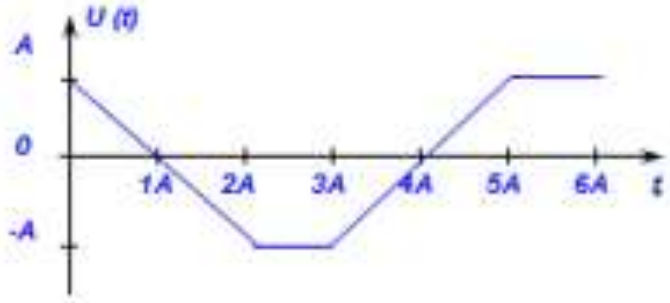
2



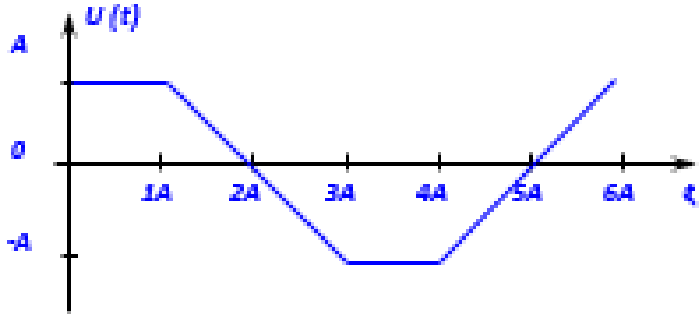
3



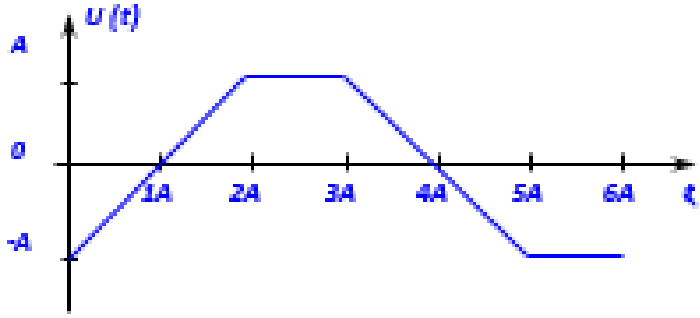
4



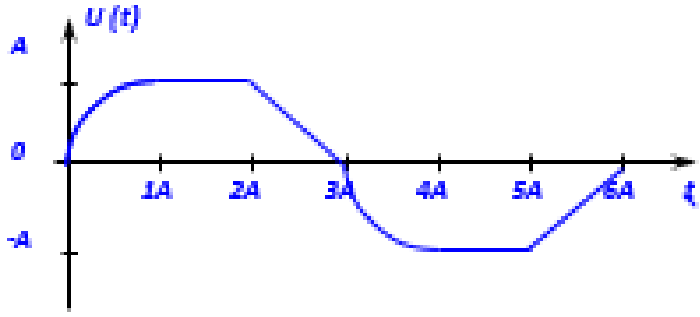
5


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

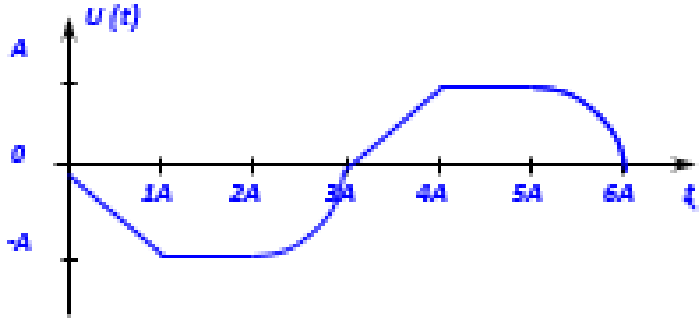
6


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

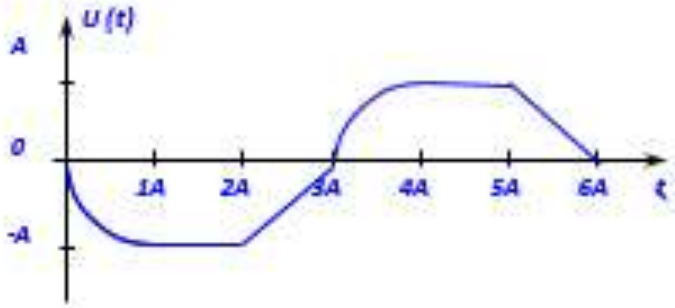
7


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

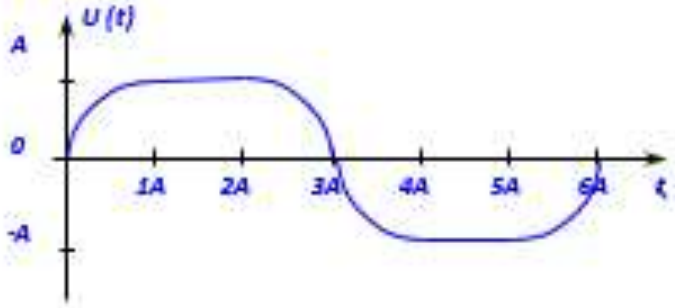
8


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

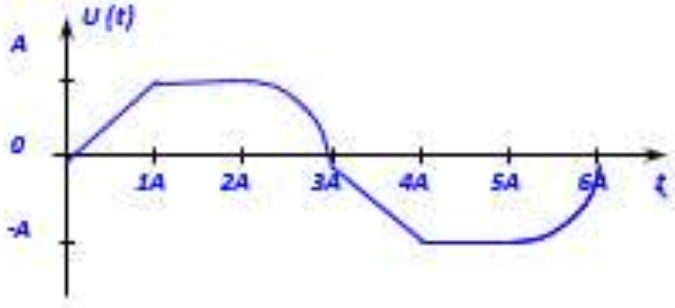
9


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

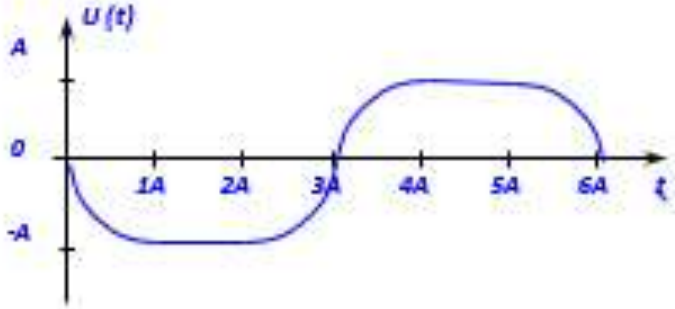
10


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

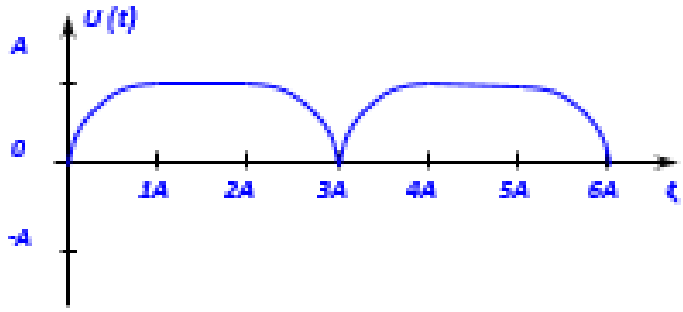
11


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

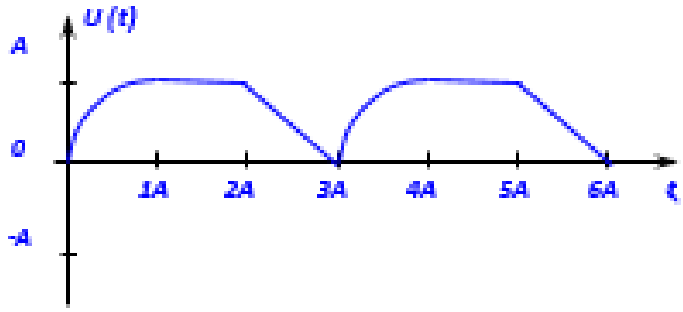
12


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

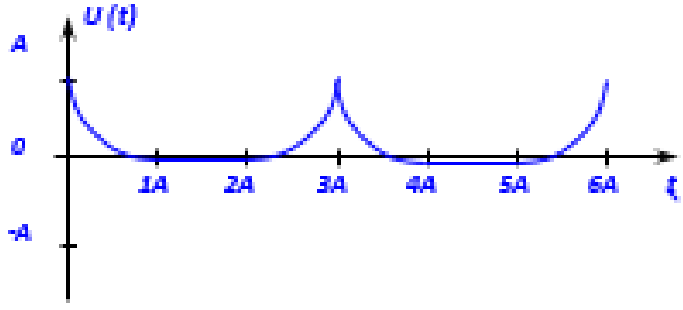
13


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

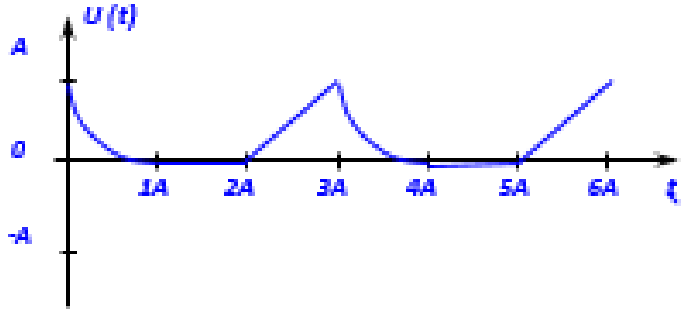
14


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

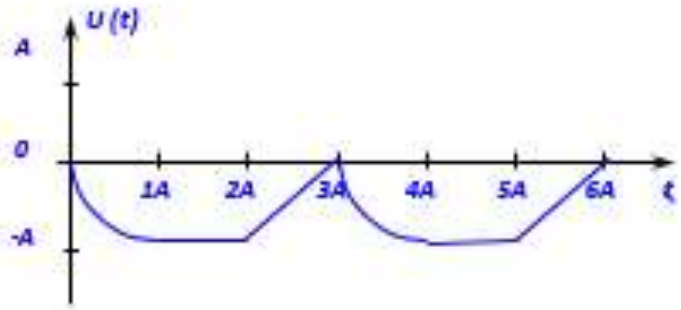
15


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

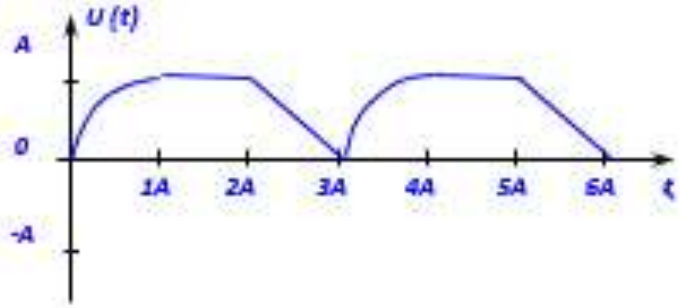
16


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

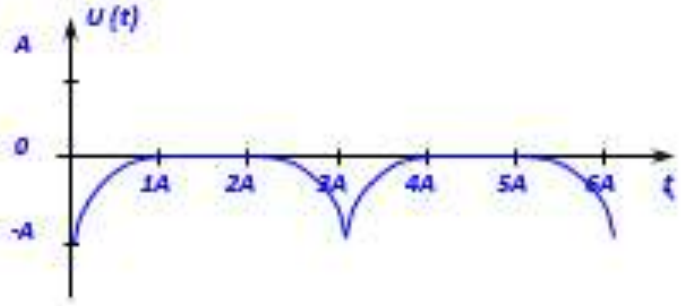
17


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

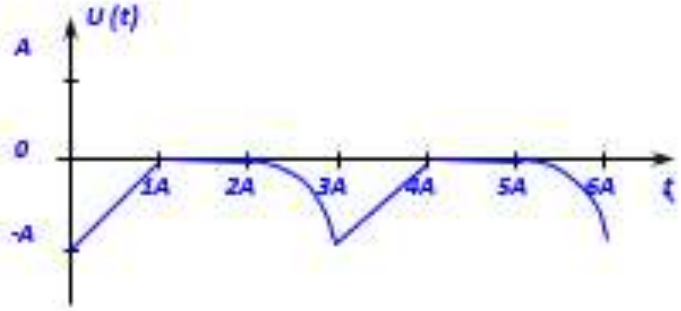
18


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

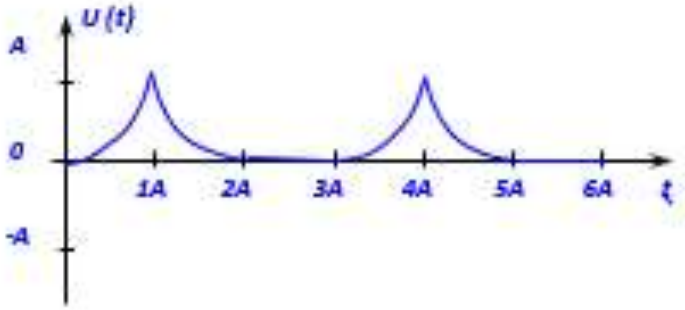
19


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

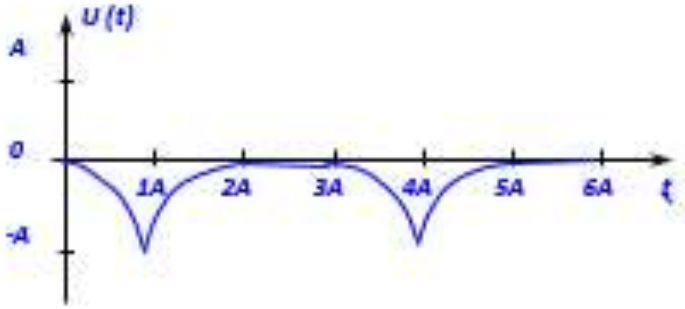
20


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

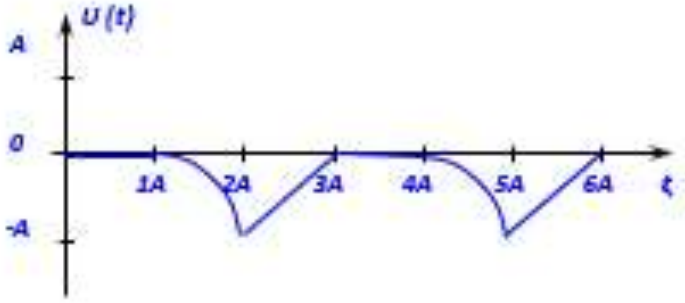
21


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

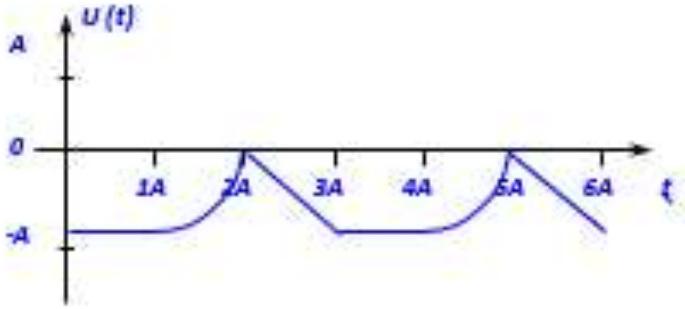
22


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

23


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

24


U (t)
A
0
-A
1A
2A
3A
4A
5A
6A
t

7 Список литературы

1. Алексеев Е.Р., Чеснокова О.В., Кучер Т.В. Free Pascal и Lazarus: Учебник по программированию / Е.Р. Алексеев, О.В. Чеснокова, Т.В. Кучер. - М.: Издательский дом ДМК-пресс, 2010. - 440 с.
2. Алексеев Е.Р., Чеснокова О.В., Кучер Т.В.. Самоучитель по программированию на Free Pascal и Lazarus.. - Донецк: ДонНТУ, Технопарк ДонНТУ УНИТЕХ, 2011. - 503 с.
3. Кетков Ю.Л. Свободное программное обеспечение. FREE PASCAL для студентов и школьников / Ю.Л. Кетков, А.Ю. Кетков. — СПб.: БХВ-Петербург, 2011. — 384 с.
4. Мансуров К.Т. Основы программирования в среде Lazarus. - М.: Нобель пресс, 2013. – 772 с.
5. Фаронов В.В. Turbo Pascal. Наиболее полное руководство (в подлиннике). — СПб.: БХВ-Петербург, 2004. — 1056 с.
6. Фленов М.Е. Библия Delphi. — 2-е изд., перераб. и доп. - СПб.: БХВ-Петербург, 2008. - 800 с.
7. Lazarus Tutorial/ru [Электронный ресурс] // База знаний о Free Pascal, Lazarus и родственных проектах: [сайт]. URL: http://wiki.freepascal.org/Lazarus_Tutorial/ru
8. Программирование на Lazarus [Электронный ресурс] // «ИНТУИТ» Национальный открытый университет: [сайт]. URL: http://www.intuit.ru/studies/courses/13745/1221/lecture/23276?page=1
9. ОС ТУСУР 01-2013 (СТО 02069326.1.01-2013). Работы студенческие по направлениям подготовки и специальностям технического профиля. Общие требования и правила оформления. - Томск: ТУСУР, 2013. – 57 с.
10. Кобрин Ю.П. Основные понятия языка Free Pascal. - Томск: ТУСУР, кафедра КИПР, 1916. - 37 с/.