Содержание

1. Введение

В ходе комплексного тестирования программных средств возникает необходимость интерпретации результатов множества тестов, написанных по различным правилам и для различных целей. Для решения задачи автоматизации запуска, сбора информации и интерпретации результатов тестирования необходимо привести интерфейсную часть всех тестирующих программ к единообразному виду позволяющему с наименьшими затратами решать поставленную задачу. Для данных целей предлагается использовать единую библиотеку (далее именуемую библиотекой *libtio*) с небольшим прикладным программным интерфейсом (*API*), исключающую возможность административного взаимонепонимания при реализации правил для обработки входных параметров и систематизации выходных данных в средствах тестирования и диагностики.

В рамках дипломной работы будет проведен анализ требований, а также разработаны соглашения об использовании основных функций данной библиотеки. После эго будут представлены блок-схемы некоторых функций.

В технологической части с целью повышения производительности будет проведена профилировка и, проанализировав наиболее узкие места, по возможности, будут внесены изменения в отдельные функции для оптимизации работы библиотеки в целом. Следующим пунктом будет проверка работоспособности функций библиотеки в различных условиях и с различными параметрами с целью обнаружения неявных ошибок. В случае если при проверке возникнут ошибки, поведется их локализация и устранение.

В части охраны труда и окружающей среды будет представлен анализ помещения, в котором проводилась разработка библиотеки *libtio*, а именно: анализ естественного освещения, анализ и расчет искусственного, анализ микроклимата.

Экономическая часть будет отведена под обоснование экономической эффективности разработки с подсчетом годового экономического эффекта и сроков окупаемости проекта.

2. Специальная часть

2.1. Анализ исходных требований для разрабатываемой библиотеки обработки входных параметров и систематизации выходных данных

Так как в исходных требованиях к разрабатываемой библиотеке указана необходимость совместимости с архитектурами *SPARC V8*, *SPARC V9*, *i386*, *x86_64*, то следует обеспечить независимость данного программного продукта от архитектуры процессора. Это достигается путем использования при разработке языка программирования высокого уровня (Си), обеспечивающего создание кросс-платформенного приложения.

Си является стандартизированным языком программирования. Это дает гарантию того, что однажды написанная, программа может быть использована на разных архитектурах. Ответственность за адаптацию высокоуровневых конструкций языка программирования к особенностям конкретной архитектуры берет на себя компилятор с этого языка для данной конкретной архитектуры. В данной работе использовался компилятор *GNU Compiler Collection* (обычно используется сокращение *gcc*), поддерживающий большое количество архитектур, в том числе и требуемые. К тому же данный компилятор обеспечивает возможность кросс-компиляции, то есть создание исполняемого файла (в данном случае – библиотеки) для платформы отличной от той, на которой запускается компиляция.

Для осуществления кросс-компиляции в *gcc* обычно применяется команда <*архитектура*>-*gcc* (например: *SPARC-gcc*). Существует также и второй вариант: использование команды *gcc* с ключом *–b* <*архитектура*>.

Для каждой архитектуры в *gcc* имеется свой список опций. В частности для компиляции под процессоры архитектуры *SPARC V8* необходимо указать ключ *–mcpu=v8*, а для 9 версии – ключ *–mcpu=v9*.

Основное различие 8 и 9 версий архитектуры *SPARC* заключается в том, что в 9 версии добавлена поддержка 64–битной адресации. Также все целочисленные регистры *SPARC V8* были расширены из 32-битных в 64-битные. Кроме того появились новые инструкции для работы с 64-битными операндами.

Аналогично для семейств архитектур *i386* и *x86-64* у *gcc* имеется свой набор опций.

Если заранее известно на процессоре какой архитектуры будет использоваться данная библиотека, то для оптимизации её работы можно при кросс-компиляции использовать ключ *–mtune=<cpu-type>*, где *<cpu-type>* - это тип конкретной архитектуры процессора.

Основной отличительной особенностью архитектуры x86-64 от i386 является поддержка 64-битных регистров общего назначения, 64-битных арифметических и логических операций над целыми числами и 64-битных виртуальных адресов.

Процессоры архитектуры *x86-64* поддерживают два режима работы: *Long mode* («длинный» режим) и *Legacy mode* («наследственный», режим совместимости с 32-битным *x86*), которые можно явно задать при компиляции, используя ключи *–m64* и *–m32* соответственно. Следовательно, если нужно чтобы библиотека запускалась и на архитектуре *i386* и на архитектуре *x86-64* нужно использовать ключ *–m32*.

Важным отличием *SPARC* архитектур от архитектур *i386* и *x86-64* является порядок записи байт. *SPARC* использует *big-endian* (порядок байт от старшего к младшему), а *i386* и *x86_64* – *little-endian* (от младшего к старшему). Запись многобайтового числа из памяти компьютера в файл или передача по сети требует соблюдения соглашений о том, какой из байтов является старшим, а какой младшим, так как прямая запись ячеек памяти

приводит к возможным проблемам при переносе приложения с платформы на платформу. Для разрабатываемой библиотеки был принят порядок байт от старшего к младшему (*big-endian*), так как он является стандартным для протокола *TCP*/*IP* (протокола управления передачей по сети).

В исходных требованиях указано, что разрабатываемая библиотека будет использоваться в операционных системах использующих стандарты *POSIX*.

POSIX (англ. *Portable Operating System Interface for Unix* — Переносимый интерфейс операционных систем *Unix*) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой. Стандарт создан для обеспечения совместимости различных *UNIX*-подобных операционных систем и переносимости прикладных программ на уровне исходного кода.

С точки зрения программиста-разработчика следование стандарту *POSIX* заключается в использовании заголовочных файлов и системных вызовов языка Си, которые должны быть предоставлены соответствующей стандарту системой.

Библиотека должна быть написана на языке высокого уровня «Си» в соответствии со спецификацией *C99*.

C99 — современный стандарт языка программирования Си. Определен в ISO/IEC 9899:1999. Является развитием стандарта *C89*.

В *C99* было добавлено несколько новых возможностей, а также удалены лишние.

Добавленные средства:

- Встраиваемые функции (объявленные с ключевым словом *inline*)
- Место, в котором возможно объявление переменных, больше не ограничено глобальной областью видимости и началом составного оператора (блока)
- Несколько новых типов данных, включая *long long int*, дополнительные расширенные целые типы, явный логический тип

данных, а также комплексный тип (*complex*) для представления комплексных чисел

- Массивы переменной длины (*variable-length arrays*)
- Поддержка однострочных комментариев, начинающихся с //, как в *C++*
- Новые библиотечные функции, как, например, *snprintf*
- Новые заголовочные файлы, такие как *stdbool.h* и *inttypes.h*
- Типовые математические функции (*tgmath.h*)
- Составные константы
- Поддержка вариативных макросов (макросов переменной арности)

Некоторые удаленные средства:

Самым заметным "излишеством", удаленным при создании *C99*, было правило "неявного *int*". В *C89* во многих случаях, когда не было явного указания типа данных, подразумевался тип *int*. А в *C99* такое не допускается. Также удалено неявное объявление функций. В *C89*, если функция перед использованием не объявлялась, то подразумевалось неявное объявление. А в *C99* такое не поддерживается. Если программа должна быть совместима с *C99*, то из-за двух этих изменений, возможно, придется подправить код.

C99 является большей частью обратно совместимым с *C90*, но вместе с тем в некоторых случаях является более жёстким. В частности, объявление без указания типа больше не подразумевает неявное задание типа *int*. Комитет по стандартизации языка Си решил, что для компиляторов будет более важным определять пропуск по невнимательности указания типа, чем «тихая» обработка старого кода, полагавшаяся на неявное указание *int*.

GCC и другие компиляторы языка Си поддерживают многие нововведения стандарта *C99*. Тем не менее, ощущается недостаточная поддержка стандарта со стороны крупных производителей средств разработки, таких как *Microsoft* и *Borland*, которые сосредоточились, в

основном, на языке *C++*, так как *C++* обеспечивает функциональность, схожую с предоставляемой нововведениями стандарта.

Согласно *Sun Microsystems*, *Sun Studio* полностью поддерживает стандарт *C99*.

Интерпретатор языка Си *Ch* поддерживает основные особенности *C99* и свободно доступен в версиях для *Windows*, *Linux*, *Mac OS X Solaris*, *QNX* и *FreeBSD*.

Другие компиляторы с полной или частичной поддержкой стандарта *C99*

Digital Mars	*Intel C Compiler*	*Oracle Solaris Studio*
VPF	*LCC*	*Pelles C*
Open Watcom C		

2.2. Разработка соглашений о вызовах функций библиотеки

2.2.1. Разработка соглашений о вызовах функций обработки ошибок работы библиотеки

Все функции библиотеки, которые предназначенные для пользователей программистов и являющиеся экспортируемыми должны возвращать значение.

Функции, возвращающие указатель на некий тип данных, в случае ошибки должны возвращать нулевой указатель (NULL).

Функции, возвращающие числовое значение некого типа данных, при аварийном завершении возвращают максимально допустимое значение своего возвращаемого типа. Функции, возвращающий параметр которых имеет символьный тип, также относятся к возвращающим числовое значение. Стоит отметить, что в таком случае возвращаемый параметр является беззнаковым.

Код ошибки для последней вызванной функции библиотеки можно получить используя функцию *tioGetError*(), возвращаемым значением которой и будет код ошибки.

При возникновении ситуации из-за которой не может продолжаться нормальная работа функций библиотеки необходимо вызвать функцию

```
int tioDie  (  int status,
               const char* buff,
            )
```

Вследствие её работы ресурсы памяти, занятые библиотекой будут освобождены.

Аргументы:

status - статус завершения приложения (TOFAIL, TOTESTNOTSTART)
msg - сообщение размещаемое в потоке ошибок

Сигнал TOFAIL означает, что программа тестирования завершилась с неудовлетворительным результатом.

Если передать сигнал TOTESTNOTSTART, это будет означать, что ошибка произошла ещё на стадии инициализации библиотеки.

2.2.2. Разработка соглашения о вызове функции инициализации библиотеки

Функцией инициализации библиотеки является функция *tioInit*. До её вызова запрещается вызов любой другой функции библиотеки, за исключением функции *tioGetVersion*. В задачи *tioInit* входит не только выделения памяти и задание начальных значений для переменных, массивов и структур, без которых невозможно использовать другие функции разрабатываемой библиотеки, но и производит разбор входных параметров для программы тестирования. Функция принимает как "длинные" так и "короткие" параметры. Все параметры, ключи которых содержат больше одного символа за исключением символа двоеточия на конце, являются

длинными, все прочие называются короткими. Ключ из одного символа так же может быть длинным.

Прототип функции tioInit:

```
int tioInit  (  const char* version,
                const char* help,
                const _param[],
                int argc,
                char *argv[]
             )
```

Как видно из прототипа функция принимает 5 параметров:

1. *version* - версия теста, для которого инициализируется библиотека;

2. *help* - короткое описание назначения теста;

3. *_param[]* - список параметров принимаемых приложением и тех ключей для параметров, что используются в данном приложении. Признаком конца списка параметров является структура tio_param, у которой все поля имеют значение NULL. Поля структуры tio_param приводятся ниже;

4. argc - количество аргументов командной строки;

5. argv[] - список аргументов командной строки;

tio_param представляет собой структуру вида:

```
typedef struct _tio_param
{
      char *key;
      char *name;
      char* description;
} tio_param;
```

Где *key* — ключ, используемый при вызове из командной строки, *name* - имя параметра, используемое при взаимодействии приложения с библиотекой, а *description* - короткое пояснение для каких целей используется параметр.

В качестве имени параметра разрешается использовать любую последовательность символов, состоящую из букв, цифр, символов подчеркивания и знака минус длиной до 126 символов.

В качестве ключа разрешено использовать последовательность символов, начинающуюся с буквы или с цифры. В теле последовательности могут содержаться буквы, цифры и знак минус. Так же строка не должна совпадать со словами «*help*», «*version*».

Если при запуске программы тестирования используется ключ --*help* , то вместо выполнения теста в стандартный поток вывода будет представлена информация о списке ключей, доступных при вызове.

Если при запуске использовать ключ --*version*, то в стандартный поток вывода будет представлена информация о версии теста.

2.2.3. Разработка соглашений о вызовах функций получения входных параметров программ тестирования

Для того чтобы автоматизировать получение параметров командной строки предлагается использовать семейство функций *tioGet** и *tioGetDef**, где вместо знака «*» должна быть подставлена одна из следующих букв, означающих какого типа будет возвращаемое значение:

- L – long
- D – double
- C – char
- S – char* (string)

Коды ошибок в результате работы функций приведены ниже (Таблица 2.1).

TENOPAR	Параметр не зарегистрирован при инициализации библиотеки
TEINCTYPE	Параметр не может быть приведен к запрошенному типу
TENOTSET	Параметр не передан при вызове приложения.

TENES	Размер буфера недостаточно велик для помещения параметра
TEFAILL	Отказ по непонятным причинам

Таблица 2.1

Функции tioGetS и tioGetDefS

```
int tioGetS  (  const char* name,
                char* buff,
                size_t buff_len
             )
```

Функция получения параметра командной строки в форме последовательности символов. *name* – указатель на имя параметра, значение которого необходимо получить. *buff* – указатель на адрес памяти, куда функция поместит значение искомого параметра в виде последовательности символов. *buff_len* – переменная, содержащая значение максимальной длины строки.

Возвращает 0 в случае успешного выполнения. В противном случае возвращаемое значение примет вид кода ошибки из таблицы 2.1. При возникновении любой из ошибок функция *tioGetS* заносит в *buff* нулевой символ.

```
int tioGetDefS  (  const char* name,
                   const char* default,
                   char* buff,
                   size_t buff_len
                )
```

Функция получения параметра командной строки в форме последовательности символов. *name* – указатель на имя параметра, значение которого необходимо получить. *default* – значение параметра, связанного с именем *name* по умолчанию. *buff* – указатель на адрес памяти, куда функция поместит значение искомого параметра в виде последовательности символов. *buff_len* – переменная, содержащая значение максимальной длины строки.

В случае если значение, связанное с именем *name* получить не удалось, то в буфер *buff* присваивается значение параметра *default*.

Возвращает 0 в случае успешного выполнения. В противном случае возвращаемое значение примет вид кода ошибки из таблицы 2.1. При возникновении любой из ошибок функция *tioGetDefS* заносит в *buff* нулевой символ.

Функции tioGetL и tioGetDefL

```
long tioGetL  (  const char* name  )
```

Функция возвращает значение параметра командной строки, связанного с именем name. Значение должно быть расположено в промежутке от минимально допустимого для типа long до предшествующего максимально допустимому значению для типа long (от LONG_MIN до LONG_MAX-1). В случае если такого параметра нет, или значения параметра не находятся в указанном промежутке, или не могут быть приведены к типу данных long, возвращается максимально допустимое значение для типа long. Код ошибки в этом случае может быть получен с помощью функции tioGetError().

Возможные ошибки: TENOTSET и TEINCTYPE.

```
long tioGetDefL  (  const char* name,
                    const long default
                 )
```

Функция возвращает значение параметра командной строки, связанного с именем name. Значение должно быть расположено в промежутке от минимально допустимого для типа long до предшествующего максимально допустимому значению для типа long (от LONG_MIN до LONG_MAX-1). В случае если такого параметра нет, или значения параметра не находятся в указанном промежутке, или не могут быть приведены к типу данных long, возвращается значение по умолчанию присвоенное при вызове функции параметру default. Код ошибки в этом случае может быть получен с помощью функции tioGetError().

Возможные ошибки: TENOTSET и TEINCTYPE.

Функции tioGetC И tioGetDefC

```
unsigned char tioGetC  (  const char* name  )
```

Функция возвращает значение символа, переданного из командной строки и связанного с именем *name*. В случае если возвращаемое значение не может быть приведено к типу *unsigned char*, возвращаемое значение будет иметь вид максимально допустимого числа для этого типа данных.

Код ошибки может быть получен при помощи вызова *tioGetError*. Возможные ошибки: TENOTSET и TEINCTYPE.

```
unsigned char tioGetDefC  (  const char* name,
                             const unsigned char default
                          )
```

В случае успешного завершения функции, возвращаемое значение будет равно значению переданному из командной строки и связанному с именем *name*. В случае, если получить значение, связанное с именем *name* не удалось, то возвращаемое значение будет взято из параметра *default*.

Код ошибки может быть получен при помощи вызова *tioGetError*. Возможные ошибки: TENOTSET и TEINCTYPE.

Функции tioGetD И tioGetDefD

```
double tioGetD  (  const char* name  )
```

Функция возвращает число с плавающей запятой, переданное в программу с параметром *name*. Значение числа может быть любым допустимым для переменной в формате *double*, за исключением значения максимально допустимого для данного типа данных. В случае неуспешного выполнения, возвращаемое значение принимает вид максимально возможного значения для типа *double*.

Код ошибки может быть получен при помощи вызова *tioGetError*. Возможные ошибки: TENOTSET и TEINCTYPE.

```
double tioGetDefD  (  const char* name,
                      const double default
                   )
```

Функция получения параметра, связанного с именем *name* в форме числа с плавающей точкой, со значением по умолчанию. Значение числа может быть любым допустимым для переменной в формате *double*, за исключением значения максимально допустимого для данного типа данных. В случае если

по каким либо причинам получить значение параметра по его имени не удалось, функция возвращает значение по умолчанию, определенное в параметре *default*.

Код ошибки может быть получен при помощи вызова *tioGetError*. Возможные ошибки: TENOTSET и TEINCTYPE.

2.2.4. Разработка соглашений о вызовах функций обработки выходных данных программ тестирования

Функции вывода делятся на два типа: функции строчного вывода и функции табличного вывода.

Функции табличного вывода.

Для предоставления данных в табличной форме определено следующее семейство функций:

```
void* tioTableBegin ( const char* format, … );
void* tioTableRecord ( void *td, … );
int tioTableEnd( void *td ).
```

Первая функция предназначена для инициализации таблицы, а так же для задания количества столбцов и их заголовков. В том числе в функции *tioTableBegin* происходит определение для каждого столбца типа данных, которые он будет содержать в себе.

Параметр *format* содержит строку символов, которая содержит в себе список имен столбцов таблицы, разделенных знаком амперсанд (&). В случае если знак амперсанд является частью имени столбца, необходимо использовать последовательность символов, состоящих из двух амперсандов подряд. Далее в прототипе функции идет переменный список параметров, количество параметров которого зависит от количества столбцов таблицы. Значения этих параметров определяют типы значений соответствующих столбцов. В случае успеха возвращаемое значение является указателем на таблицу.

Функция *tioTableRecord* предназначена для добавления новой строки в таблицу, передаваемую с параметром *td*. Далее идет переменный список параметров, в каждом из которых содержатся значение соответствующей ячейки таблицы. В случае успеха возвращаемым значение, также как и в предыдущей функции, является указатель на таблицу.

Функция *tioTableEnd* является функцией, которая выводит в виде таблицы сформированные данные, полученные от вызовов предыдущих функций семейства *tioTable*. В том случае, если какие либо значения не могут быть представлены в одной строке ячейки таблицы, то функция добавляет столько строк в таблицу, сколько нужно для полного представления данного значения.

Между вызовами функций *tioTable** разрешен вызов любых других функций библиотеки.

Функции строчного вывода

Все функции для форматирования строки вывода используют формат широко применяемый в системных функция. Последовательности символов начинающихся с символа % и продолжающихся символами из приведенной далее таблицы:

Символ	Описание типа
c	Символ (char)
d | i	Целое число в десятичной форма (long)
e	Число с мантиссой для чисел с плавающей запятой (double)
f	Число с плавающей точкой (double)
o	Целое число в восьмеричном представлении (long)
s	Строка завещающаяся нулем (char*)
x	Беззнаковое шеснадцатиричное представления (long)
X	Беззнаковое шестнадцатеричное представления с буквами в верхнем регистре (long)

Таблица 2.2

Для вывода символа % используется последовательность %%.

```
int tioPrint(const char * message)
```

Выводит префикс «(II):» и строку с на которую указывает параметр message в стандартный поток вывода.

```
int tioPrintF(const char* template, ... )
```

Выводит префикс «(II):» и форматируемую строку в стандартный поток вывода.

Префикс «(II):» говорит о том что строка имеет информационный характер. Обычно сообщения с таким префиксом выводятся для пояснения чего либо при работе программы.

```
int tioWarning( const char * message)
```

Выводит префикс «(WW):» и строку на которую указывает параметр message в поток ошибок.

```
int tioWarningF(const char* template, ... )
```

Выводит префикс «(WW):» и форматируемую строку в поток ошибок.

Префикс «(WW):» говорит о том что строка является предупреждением. Обычно сообщения с таким префиксом выводятся чтобы предупредить о каком либо событии, которое может повлечь за собой ошибки.

```
int tioError( const char * message)
```

Выводит префикс «(EE):» и строку на которую указывает параметр message в поток ошибок.

```
int tioErrorF(const char* template, ... )
```

Ввыводит префикс «(EE):» и форматируемую строку в поток ошибок.

Префикс «(EE):» говорит о том что строка является сообщением об ошибке. Обычно сообщения с таким префиксом выводятся для того чтобы сообщить какого рода произошла ошибка, с целью облегчения поиска места ее возникновения.

Следующие две функции выводят сообщения только если программа, использующая библиотеку, была запущена с ключом --tio-debug

```
int tioDebug( const char * message)
```

Выводит префикс «(DD):» и строку на которую указывает параметр message в стандартный поток вывода.

```
int tioDebugF(const char* template, ... )
```

Выводит префикс «(DD):» и форматируемую строку в стандартный поток вывода.

Префикс «(DD):» говорит о том что строка является отладочной информацией. Обычно сообщения с таким префиксом выводятся на этапе реализации кода программы.

2.3.Реализация функций разрабатываемой библиотеки

Набор кода функций библиотеки *libtio* проводился при помощи средств текстового редактора *Vim*.

Vim - это свободный режимный текстовый редактор. Одна из главных особенностей редактора — применение двух основных, вручную переключаемых, режимов ввода: командного (позволяет использовать клавиши клавиатуры не для печатанья, а для различных команд) и текстового (режим непосредственного редактирования текста, аналогичный большинству «обычных» редакторов).

Эффективная работа с редактором требует предварительного обучения, так как интерфейс этого редактора нельзя назвать интуитивно понятным.

Vim обладает возможностью, позволяющей разбивать рабочую поверхность редактора на множество окон как по вертикали, так и по горизонтали. В нем присутствует: поддержка *Unicode* символов, неограниченная глубина отмены (*undo*) и возврата (*redo*) действий, режим

сравнения двух файлов, подсветка синтаксиса, автоматическое определение величины отступа для каждой строки в зависимости от языка программирования, автоматическое продолжение команд, слов, строк целиком и имён файлов, сворачивание (*folding*) текста для лучшего обзора. поддержка цикла разработки «редактирование — компиляция — исправление» программ.

При реализации функций библиотеки *libtio* использовалась распределённая система управления версиями файлов *Git.*

Все настройки *Git* хранятся в текстовых файлах конфигурации. Такая реализация делает эту систему легко портируемой на любую платформу и даёт возможность легко интегрировать *Git* в другие системы (в частности, создавать графические *git*-клиенты с любым желаемым интерфейсом). Репозиторий *Git* представляет собой каталог файловой системы, в котором находятся файлы конфигурации репозитория, файлы журналов, хранящие операции, выполняемые над репозиторием, индекс, описывающий расположение файлов и хранилище, содержащее собственно файлы. Структура хранилища файлов не отражает реальную структуру хранящегося в репозитории файлового дерева, она ориентирована на повышение скорости выполнения операций с репозиторием. Когда ядро обрабатывает команду изменения (неважно, при локальных изменениях или при получении патча от другого узла), оно создаёт в хранилище новые файлы, соответствующие новым состояниям изменённых файлов. Следует отметить, что никакие операции не изменяют содержимого уже существующих в хранилище файлов.

По умолчанию репозиторий хранится в подкаталоге с названием «.git» в корневом каталоге рабочей копии дерева файлов, хранящегося в репозитории. Любое файловое дерево в системе можно превратить в репозиторий git, отдав команду создания репозитория из корневого каталога этого дерева (или указав корневой каталог в параметрах программы). Репозиторий может быть импортирован с другого узла, доступного по сети.

При импорте нового репозитория автоматически создаётся рабочая копия,, соответствующая последнему зафиксированному состоянию импортируемого репозитория (то есть не копируются изменения в рабочей копии исходного узла, для которых на том узле не была выполнена команда *commit*).

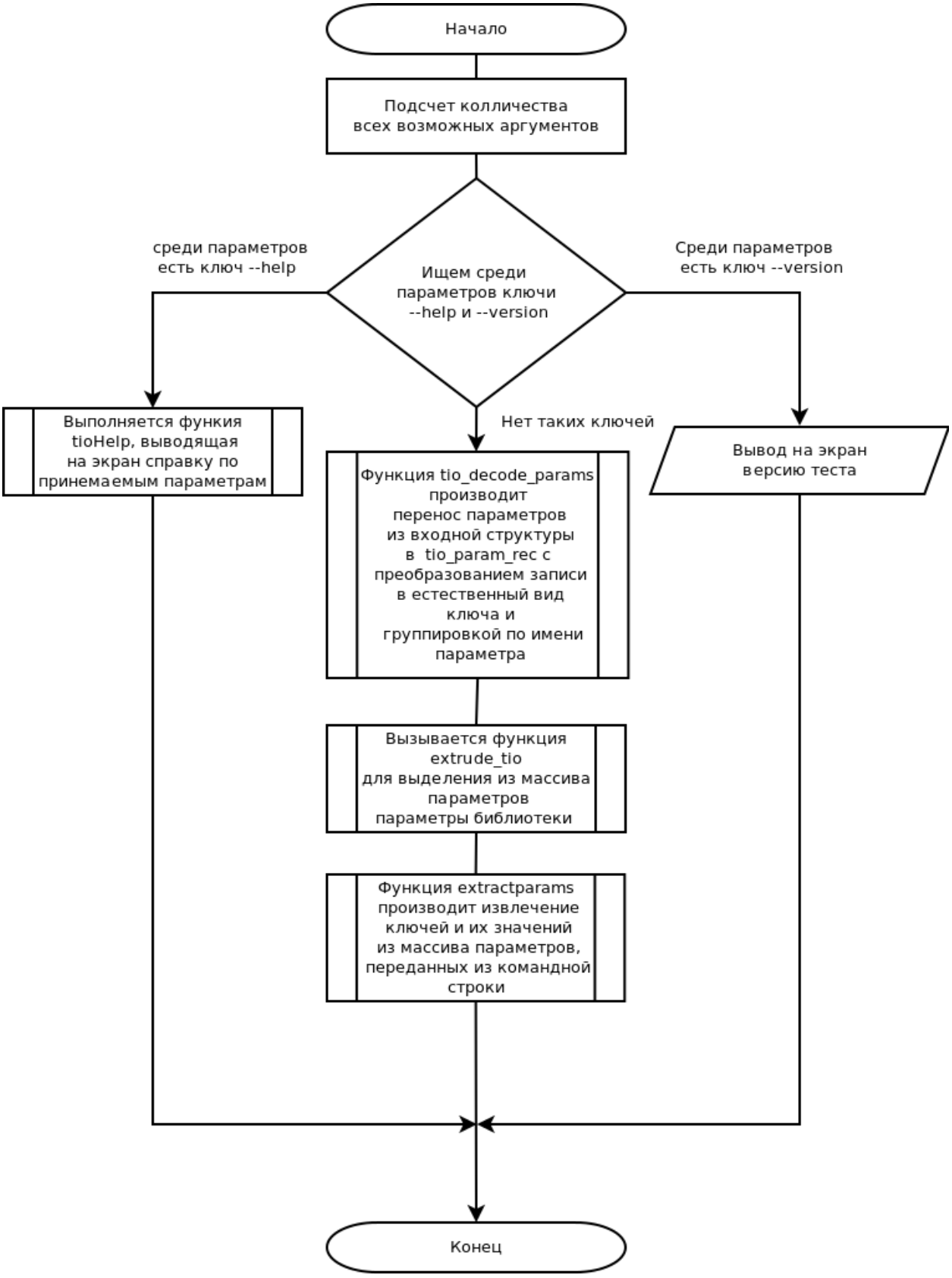
Git изначально идеологически ориентирована на работу с изменениями, а не с файлами, «единицей обработки» для нее является набор изменений, или патч.

Преимуществами *git* перед другими системами контроля версиями:

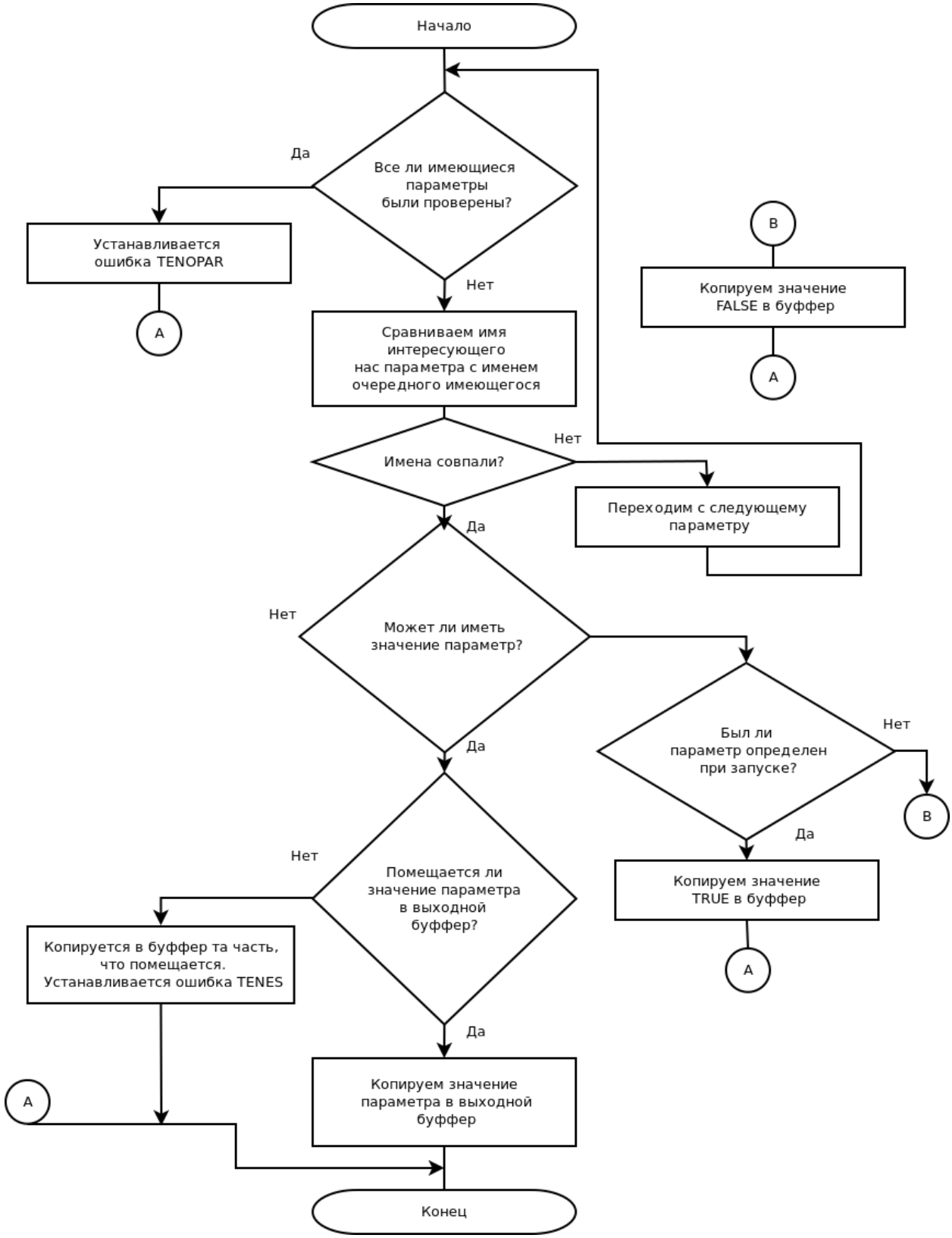
- Высокая производительность.
- Продуманная система команд, позволяющая удобно встраивать git в скрипты.
- Репозитории *git* могут распространяться и обновляться общесистемными файловыми утилитами архивации и обновления благодаря тому, что фиксации изменений и синхронизации не меняют существующие файлы с данными, а только добавляют новые (за исключением некоторых служебных файлов, которые могут быть автоматически обновлены с помощью имеющихся в составе системы утилит). Для раздачи репозитория по сети достаточно любого веб-сервера.

Блок-схемы реализаций некоторых функций.

Функция *tioInit()*.

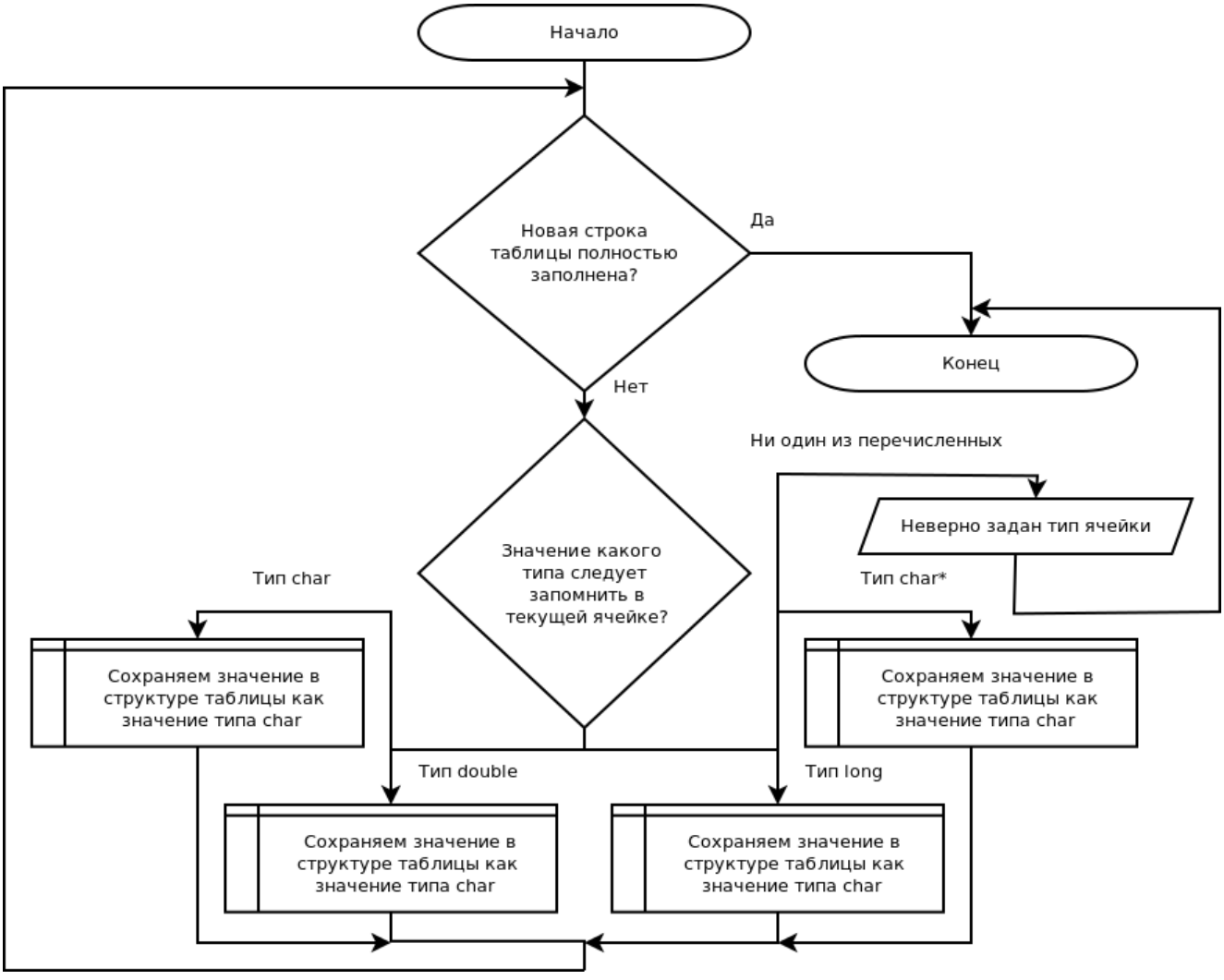


Функция *tioGetS()*

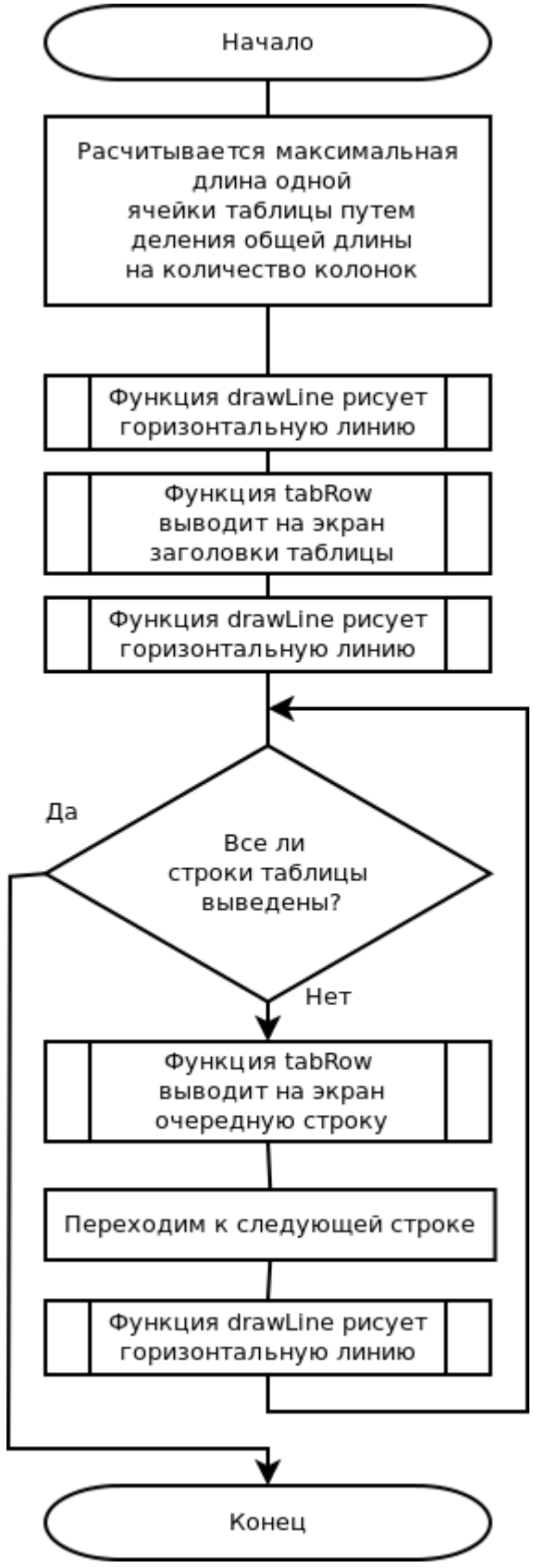


Функция *tioTableBegin()*

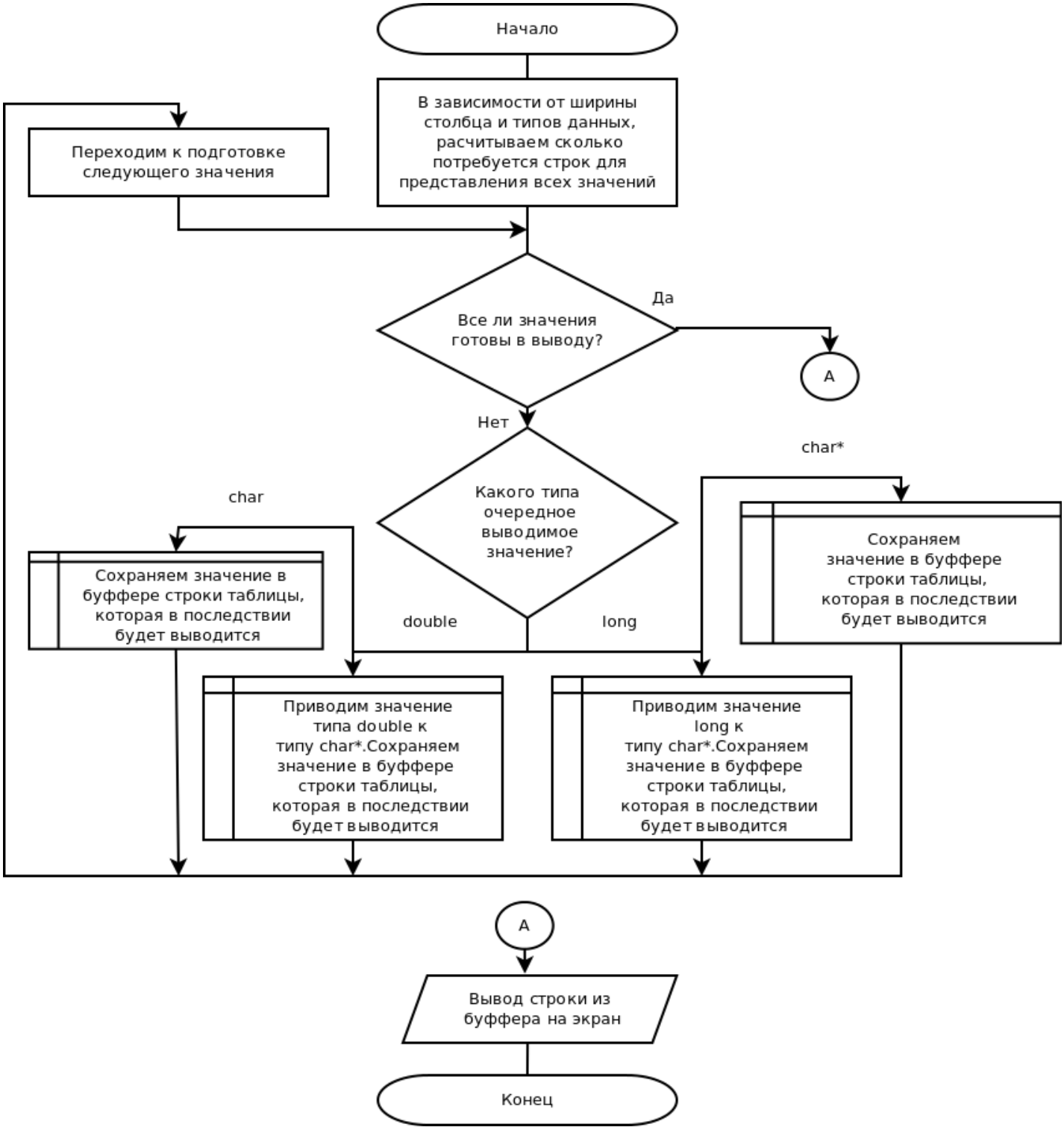
Функция *tioTableRecord()*



Функция *tioTableEnd()*



Функция *tabRow()*



2.4.Прототипирование среды исполнения подпрограмм библиотеки

Базовые возможности библиотеки рассмотрим на примере программы, которая тестирует функцию подсчета корней квадратного уравнения.

Есть функция, решающая квадратное уравнение.

Задача: протестировать являются ли корни, полученные на выходе функции, верными для уравнения заданного вида.

Для тестирования возьмем уравнение вида $x^2 + 2x - 3 = 0$. Значит параметр «*a*» равен 1, параметр «*b*» равен 2 и параметр «*c*» равен -3. Известно, что корнями данного уравнения являются 1 и -3. Для того чтобы что бы убедиться в корректности работы функции решения квадратных уравнений, напишем тест, использующий функции разработанной библиотеки.

Параметры a, b, c, первый эталонный корень и второй эталонный корень передаются при вызове теста как параметры командной строки. Согласно данному уравнению, строка, запускающая тест, должна выглядеть так:

./quadratic-equation -a 1 -b 2 -c -3 --root1=1 --root2= -3

Программа теста считывает входные параметры, запускает тестируемую функцию с параметрами *a, b, c*. Получившиеся результаты работы функции решения квадратного уравнения выводит на экран вместе с эталонными значениями *root1* и *root2*.

Выполнение теста показано на рис. 2.1.

```
nwcfang@nf-lrti:~/current-task/prototypes/quadratic-equation$ ./quadratic-equation -a 1 -b 2 -c -3 --root1=1 --root2=-3
Тест написал: Гусев Михаил
Короткое описание теста:
Тестирование функции решения квадратного уровнения.
[RUN]: Запуск ./quadratic-equation
+-----------------------------------------------------------+-----------------------------------------------------------+
|Name                                                       |Value                                                      |
+-----------------------------------------------------------+-----------------------------------------------------------+
|Argument A                                                 |1                                                          |
+-----------------------------------------------------------+-----------------------------------------------------------+
|Argument B                                                 |2                                                          |
+-----------------------------------------------------------+-----------------------------------------------------------+
|Argument C                                                 |-3                                                         |
+-----------------------------------------------------------+-----------------------------------------------------------+
Сравнение эталонных и возвращаемых функцией корней
+-----------------------------------------------------------+-----------------------------------------------------------+
|Roots etalon                                               |Roots from function                                        |
+-----------------------------------------------------------+-----------------------------------------------------------+
|1                                                          |1                                                          |
+-----------------------------------------------------------+-----------------------------------------------------------+
|-3                                                         |-3                                                         |
+-----------------------------------------------------------+-----------------------------------------------------------+
```

Рис. 2.1

Для разбора параметров командной строки, а также инициализации разработанной библиотеки использовалась функция *tioInit*. После вызова этой функции можно использовать функции библиотеки семейства *tioGet* для доступа к интересующим нас параметрам командной строки.

Наглядное представление выходных данных обеспечивается функциями семейства *tioTable*, позволяющих рисовать динамическую таблицу, в которой можно изменять заголовки и количество колонок, а также количество строк и тип данных в каждой ячейке строки.

С помощью ключа *--help*, переданному при вызове тестирующей программы, использующей библиотеку *libtio*, вместо выполнения теста на экран выведет список аргументов, которые можно передать из командной строки (рис. 2.2).

```
nwcfang@nwcfang-Z68AP-D3:~/current_task/prototypes/quadratic-equation$ ./quadratic-equation --help
Тест написал: Гусев Михаил
Короткое описание теста:
Тестирование функции решения квадратного уровнения.
Использование: ./quadratic-equation [КЛЮЧ]... [ФАЙЛ]...
This is test
 -a <ПАРАМЕТР>                  Параметр A
 -b <ПАРАМЕТР>                  Параметр B
 -c <ПАРАМЕТР>                  Параметр C
 --root1 <ПАРАМЕТР>             Первый корень
 --root2 <ПАРАМЕТР>             Второй корень
nwcfang@nwcfang-Z68AP-D3:~/current_task/prototypes/quadratic-equation$
```

Рис. 2.2

При использовании ключа *--version* после команды, запускающей исполняемый файл программы тестирования, будет выведена справка о версии запускаемого теста (рис. 2.3).

```
nwcfang@nwcfang-Z68AP-D3:~/current_task/prototypes/quadratic-equation$ ./quadratic-equation --version
Программа ./quadratic-equation версия v0.9 alpha
```

Рис. 2.3

Теперь рассмотрим программу тестирующую работоспособность COM-порта.

Программа может работать в трех режимах:

- режим «Клиент»;
- режим «Сервер»;
- режим «Клиент/Сервер».

Если выбран режим «Клиент», то программа работает по следующему алгоритму:

В течении двадцати секунд ожидает сообщение от программы «Сервер» о готовности к передаче данных. Если по истечению данного периода сообщение не получено, то тест завершается провалом. В случае если сообщение о готовности «Сервера» пришло, отправляется сообщение о готовности принимать данные. После чего принимаем пакеты.

Если выбран режим «Сервер», то программа работает так:

Вначале отправляется сообщение, что «Сервер» готов к передаче данных. Получив, ответ от «Клиента», что он готов к передаче, «Сервер» начинает передавать пакеты.

Режим «Клиент/Сервер» отличается от предыдущих тем, что создается процесс потомок, который берет на себя роль «Сервера», а родитель будет работать как «Клиент».

В качестве входных параметров для тестирующей программы принимаются следующие ключи:

- «-*D*» – ключ имеет числовое значение. Продолжительность передачи пакетов;
- « -*m*» - ключ имеет числовое значение. Скорость передачи пакетов;
- «-*s*» - ключ имеет числовое значение. Размер передаваемого пакета;
- «-*d*» - программа будет работать в режиме Сервера, то есть отправлять пакеты Клиенту;

- «*-l*» - программа будет работать в режиме Клиента, то есть принимать пакеты от Сервера;
- «*-L*» - программа работает в режиме Клиент/Сервер, то есть пакеты будут отправляться и приниматься на одной и той же ЭВМ.

Эта программа была написана без использования библиотеки *libtio*. Метод обработки входных параметров описывался в отдельном файле и выглядел так:

```
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <unistd.h>

#include "config.h"

Configuration config = {
    115200, // speed
    -1,     // device fd
    "/dev/ttyUSB0", // device path
    1000,       // minimu m transfered data count
    CLIENTMODE, // mode of test
    0, // duration (nowtime is unused)
    1 // work mode
};

static int
calculate_configuration(Configuration *cfg)
{
    if (!cfg)
        return EINVAL;

    if (cfg->duration)
    {
        cfg->sendPacksLength = (cfg->duration * cfg-
>portSpeed / 8);
        cfg->duration = 0;
    }
    return 0;
```

```
}


int
write_configuration(Configuration *cfg, char **argv, int
argc)
{
    int opt;
    int already_typed = 0;

    if (!cfg || !argv ||  argc < 1)
        return EINVAL;

    while (-1 != (opt = getopt(argc, argv, "D:m:s:dlLh")))
    {
        switch(opt)
        {
        case 'D':
            cfg->duration = atol(optarg);
            if (cfg->duration <= 0)
                return EAGAIN;
            break;
        case 'm':
            cfg->portSpeed = atol(optarg);
            if (cfg->portSpeed <= 0)
                return EAGAIN;
            break;
        case 's':
            cfg->sendPacksLength = atol(optarg);
            if (cfg->sendPacksLength <= 0)
                return EAGAIN;
            break;
        case 'd':
            if (already_typed)
                return EAGAIN;
            cfg->serverClientMode = SERVERMODE;
            already_typed = 1;
            break;
        case 'l':
            if (already_typed)
                return EAGAIN;
            cfg->serverClientMode = CLIENTMODE;
            already_typed = 1;
            break;
        case 'L':
            if (already_typed)
                return EAGAIN;
```

```
            cfg->serverClientMode = CLIENTSERVERMODE;
            already_typed = 1;
            break;
        default:
            return EAGAIN;
        }
    }
    calculate_configuration(cfg);

    if (optind < argc)
        strcpy(config.DeviceName, argv[optind]);

    return 0;
}
```

Использование библиотеки libtio, а в частности функции *tioInit* позволяет сократить данный файл до вида:

```
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <unistd.h>

#include <tio.h>

#include "config.h"

Configuration config = {
    -1,         //device fd
    1000,       //minimu m transfered data count
    1           //work mode
};
static int
calculate_configuration(Configuration *cfg)
{
    if (!cfg)
        return EINVAL;

    if( tioGetDefL( "DURATION", 0 ) )
        cfg->sendPacksLength = ( tioGetL( "DURATION" ),
tioGetDefL( "PORTSPEED", 115200 ) / 8);
    return 0;
}
int
```

```
write_configuration(Configuration *cfg )
{
    cfg->sendPacksLength = tioGetDefL( "SENDPACKSLENGTH",
1000 );
    calculate_configuration(cfg);
    return 0;
}
```

Причем, чем больше ассортимент параметров командной строки, тем больше преимущество по времени у программиста, использующего библиотеку *libtio* пред программистом, пишущим код разбора входных параметров самостоятельно.

Стандартный поток вывода после выполнения тестирующей программы в режиме «Клиент/Сервер» показан на рис. 2.4

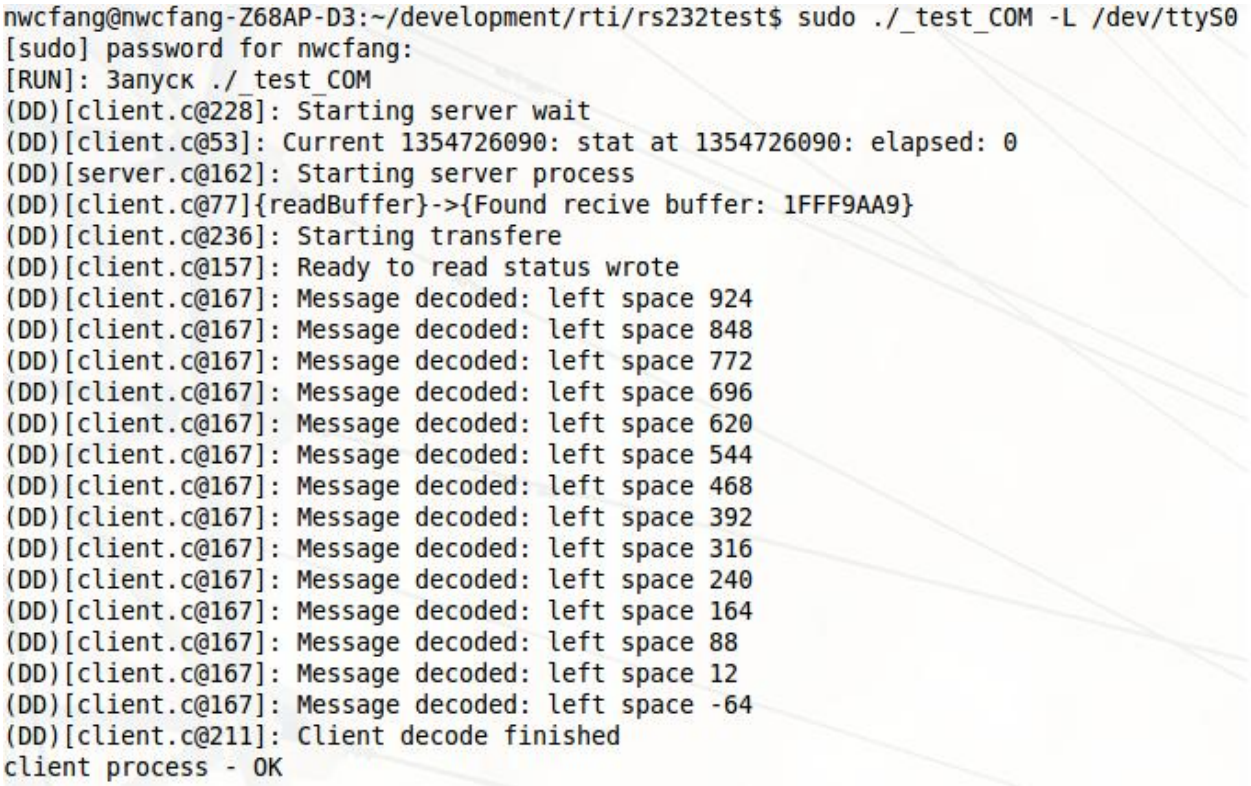


Рис. 2.4

3. Технологическая часть

3.1. Профилирование разрабатываемого программного обеспечения

Профилирование – это сбор характеристик программного обеспечения, таких как данные о продолжительности и частоте выполнения каждой из функций программы, поиск утечек памяти, и прочих ошибок, связанных с неправильной работой с областями памяти – чтением или записью за пределами выделенных регионов и тому подобное.

В качестве профилировщика используется *Valgrind*, предоставляющий множество инструментов для поиска узких мест в коде. Инструмент по умолчанию, а также наиболее используемый – *Memcheck.*

Memcheck вставляет дополнительный код в программное обеспечение, который отслеживает любые манипуляции и перемещения данных в памяти. Более того, *Memcheck* заменяет стандартное выделение памяти языка Си собственной реализацией, которая помимо прочего включает в себя защиту памяти (*memory guards*) вокруг всех выделенных блоков. Данная возможность позволяет *Memcheck* обнаруживать ошибки несоответствия (*off-by-one errors*), при которых программа считывает или записывает вне выделенного блока памяти. Проблемы, которые может обнаруживать *Memcheck* и предупреждать о них, включают в себя:

- чтение или запись по неправильным адресам памяти — за границами выделенных блоков памяти и т.п.;
- использование не инициализированных значений, в том числе и для переменных, выделяемых в стеке;
- ошибки освобождения памяти, например, когда блок памяти уже был освобожден в другом месте;
- передача некорректных параметров системным вызовам, например указание неправильных указателей для операций чтения из буфера, указанного пользователем;

- пересечение границ блоков памяти при использовании операций копирования/перемещения данных между двумя блоками памяти.

Однако, использование *Memcheck* способствует снижению производительности в 5-12 раз, а также использованию большего объёма памяти, из чего следует, что использование *Memcheck*-реализации выделения памяти не приветствуется и должно использоваться только при профилировании программного обеспечения.

На рис. 3.1 показан вывод профилировщика *Valgrind*, использовавшего инструмент *Memcheck*. В качестве объекта для исследования используется программа тестирования функции решения квадратного уравнения (см. п. 2.4).

```
nwcfang@nwcfang-Z68AP-D3:~/current_task/prototypes/quadratic-equation$ valgrind --tool=memcheck --leak-check=full
--show-reachable=yes --track-origins=yes ./quadratic-equation -a 1 -b 2 -c -3 --root1=1 --root2=-3
==3409== Memcheck, a memory error detector
==3409== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==3409== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==3409== Command: ./quadratic-equation -a 1 -b 2 -c -3 --root1=1 --root2=-3
==3409== Parent PID: 2454
==3409==
==3409== HEAP SUMMARY:
==3409==     in use at exit: 1,712 bytes in 16 blocks
==3409==   total heap usage: 234 allocs, 218 frees, 14,102 bytes allocated
==3409==
==3409== 16 bytes in 1 blocks are still reachable in loss record 1 of 6
==3409==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3409==    by 0x4009C6: main (source.c:24)
==3409==
==3409== 16 bytes in 1 blocks are still reachable in loss record 2 of 6
==3409==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3409==    by 0x4009D4: main (source.c:25)
==3409==
==3409== 240 bytes in 2 blocks are definitely lost in loss record 3 of 6
==3409==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3409==    by 0x4E39581: tabRow (tioTableBegin.c:371)
==3409==    by 0x4E39253: tioTableEnd (tioTableBegin.c:277)
==3409==    by 0x400B72: main (source.c:66)
==3409==
==3409== 240 bytes in 2 blocks are definitely lost in loss record 4 of 6
==3409==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3409==    by 0x4E39581: tabRow (tioTableBegin.c:371)
==3409==    by 0x4E39253: tioTableEnd (tioTableBegin.c:277)
==3409==    by 0x400AE8: main (source.c:60)
==3409==
==3409== 480 bytes in 4 blocks are definitely lost in loss record 5 of 6
==3409==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3409==    by 0x4E39581: tabRow (tioTableBegin.c:371)
==3409==    by 0x4E39284: tioTableEnd (tioTableBegin.c:286)
==3409==    by 0x400B72: main (source.c:66)
==3409==
==3409== 720 bytes in 6 blocks are definitely lost in loss record 6 of 6
==3409==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3409==    by 0x4E39581: tabRow (tioTableBegin.c:371)
==3409==    by 0x4E39284: tioTableEnd (tioTableBegin.c:286)
==3409==    by 0x400AE8: main (source.c:60)
==3409==
==3409== LEAK SUMMARY:
==3409==    definitely lost: 1,680 bytes in 14 blocks
==3409==    indirectly lost: 0 bytes in 0 blocks
==3409==      possibly lost: 0 bytes in 0 blocks
==3409==    still reachable: 32 bytes in 2 blocks
==3409==         suppressed: 0 bytes in 0 blocks
==3409==
==3409== For counts of detected and suppressed errors, rerun with: -v
==3409== ERROR SUMMARY: 18 errors from 6 contexts (suppressed: 2 from 2)
```

Рис. 3.1

В данной проверке поведение *Memcheck* настраивается следующими ключами:

--leak-check=full – функция обнаружения утечек памяти, будет выводить не только сводную информацию, но и информацию о месте, в котором происходит утечка памяти.

--show-reachable=yes – будет показана информация о блоках, не освобожденной памяти, указатель на которые все ещё не потерян.

--track-origins=yes – при задании этого ключа, будет выводиться информация о неинициализированных переменных и об их происхождении.

Следующий инструмент, который использовался в дипломной работе для профилирования основных функций библиотеки, - *Callgrind*.

Callgrind анализирует вызовы функций. На основе полученных данных при использовании этого инструмента можно построить дерево вызовов функций, и соответственно, проанализировать узкие места в работе программы. По умолчанию он собирает данные о количестве выполненных инструкций, зависимостях между вызывающей и вызываемой функциями и количество вызовов конкретных функций.

Для визуализации данных, полученных в результате работы *Callgrind*, использовалась программа *Kcachegrind*.

На рис. 3.2 показана схема вызовов функций для программы тестирования функции решения квадратного уравнения (см. п. 2.4). Данная схема не содержит в себе все вызываемые функции, так как нет смысла анализировать функции, выполнение которых занимает незначительную часть процессорного времени.

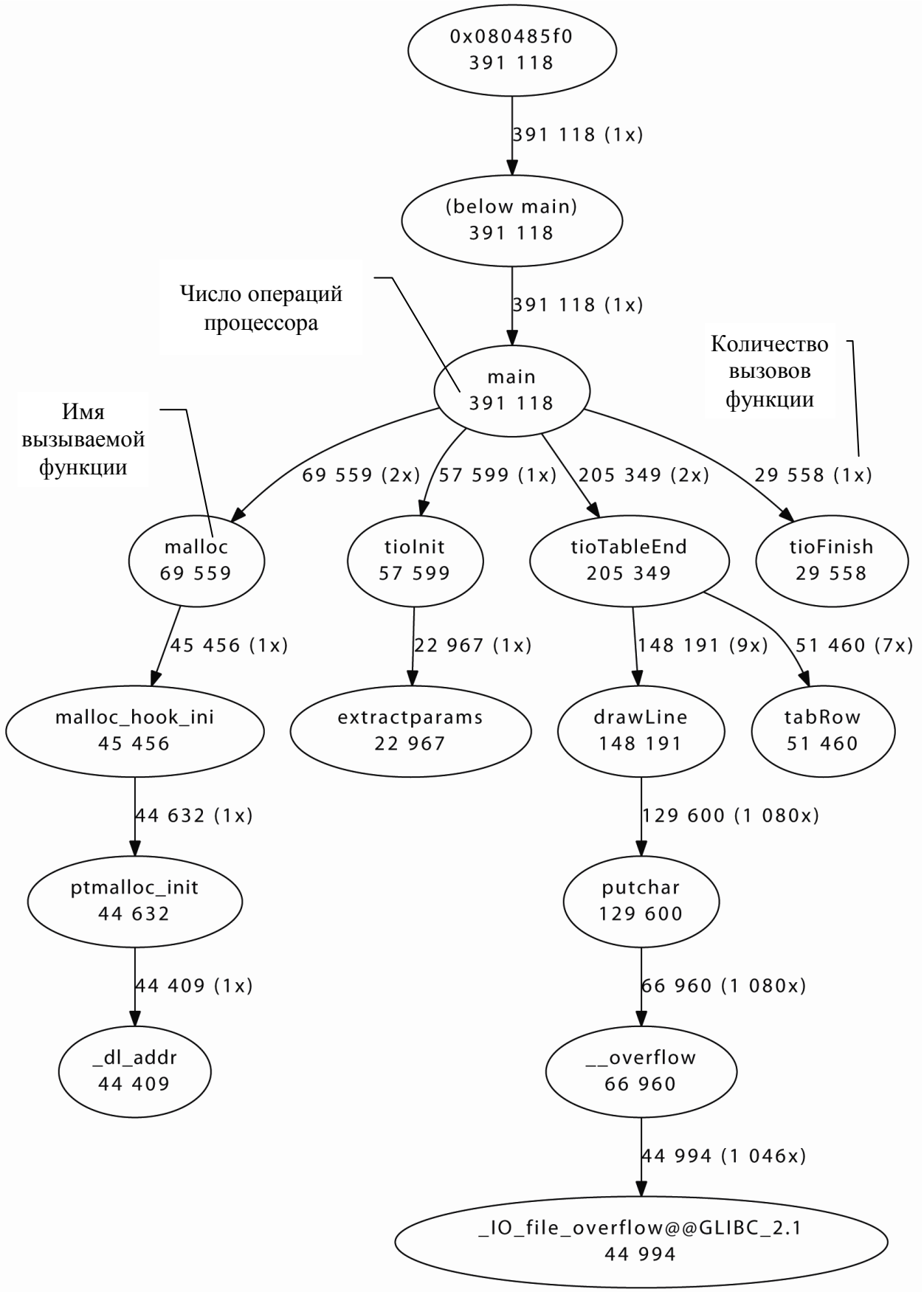

0x080485f0
391 118
391 118 (1x)
(below main)
391 118
391 118 (1x)
Число операций процессора
main
391 118
Количество вызовов функции
Имя вызываемой функции
69 559 (2x)
57 599 (1x)
205 349 (2x)
29 558 (1x)
malloc
69 559
tioInit
57 599
tioTableEnd
205 349
tioFinish
29 558
45 456 (1x)
22 967 (1x)
148 191 (9x)
51 460 (7x)
malloc_hook_ini
45 456
extractparams
22 967
drawLine
148 191
tabRow
51 460
44 632 (1x)
129 600 (1 080x)
ptmalloc_init
44 632
putchar
129 600
44 409 (1x)
66 960 (1 080x)
_dl_addr
44 409
__overflow
66 960
44 994 (1 046x)
_IO_file_overflow@@GLIBC_2.1
44 994

Рис. 3.2

3.2.Анализ производительности библиотеки интерфейсов

По результатам работы профилировщика *Valgrind*(инструмент профилировки *Memcheck*) из рис. 3.1 видно, что приложением выделяется 234 блока памяти, весящие 14,102 байта, а освобождается только 218. Следовательно, 16 блоков весом 1,712 байт не освобождены должным образом. Произошла утечка памяти. Так же определено, что на 14 блоков не указывает не один указатель, что говорит о том, что управление над этими блоками памяти потеряно. Оставшиеся 2 блока, на момент выхода из программы были все ещё достигаемыми, то есть существовали указатели, хранившие в себе адреса этих блоков. Так же из рис. 3.1 видны конкретные строки кода в которых выделяется память для этих блоков, что существенно упрощает поиск тех самых блоков памяти.

Устранение утечки памяти связанной с двумя блоками памяти, которые на момент выхода из программы были все ещё достигаемыми:

```
diff --git a/prototypes/quadratic-equation/source.c
b/prototypes/quadratic-equation/source.c
index 14b83a7..a0aa898 100644
--- a/prototypes/quadratic-equation/source.c
+++ b/prototypes/quadratic-equation/source.c
@@ -22,7 +22,7 @@ int quad( long a, long b, long c, SRoots*
Roots ) {

 int main( int argc, const char* argv[] ) {
     SRoots *Roots = malloc( sizeof(SRoots) );

-    SRoots *RootsEtalon = malloc( sizeof(SRoots) );


     //int myargc = 6;

@@ -65,8 +65,8 @@ int main( int argc, const char* argv[] ) {
     tioTableRecord( td, tioGetL( "ROOT2" ), Roots->root2 );
     tioTableEnd( td );


-    tioFinish( 0 );
     free(Roots);
+    tioFinish( 0 );
```

```
     return 0;

 }
```

Изменения в исходном коде функции отображения строки таблицы, помогающие исправить утечку памяти (14 потерянных блоков).

```
diff --git a/src/tioTableBegin.c b/src/tioTableBegin.c
index 030406d..422cf06 100644
--- a/src/tioTableBegin.c
+++ b/src/tioTableBegin.c
@@ +459,11 @@ int tabRow( void **strings, int *bufType, int
countColum, int lenColCon )
             }
             /*Insert spaces*/
             for(extraCounter = colStr[i] + 1; extraCounter <
(max + 1); ++ extraCounter )
+            {
                 for( offset = 0; offset < (lenColCon - 1);
++ offset )
                     data[i][extraCounter][offset] = ' ';
+                data[i][extraCounter][offset] = '\0';
+            }
             break;
         default:
             printf("ERROR!");
@@ -481,7 +484,7 @@ int tabRow( void **strings, int *bufType,
int countColum, int lenColCon )
     /*FREE */
     for( i = 0; i < countColum; ++ i )
     {
-        for( j = 0; j < colStr[i]; ++ j )
+        for( j = 0; j <= colStr[i]; ++ j )
         {
             free(data[i][j]);
         }
```

В результате внесенных изменений *Valgrind*, запущенный с теми же ключами и для той же программы, что и в п. 3.1 выводит сообщение показанное на рис. 3.5.

```
nwcfang@nwcfang-Z68AP-D3:~/current_task/prototypes/quadratic-equation$ valgrind --tool=memcheck --leak-check=full
--show-reachable=yes --track-origins=yes ./quadratic-equation -a 1 -b 2 -c -3 --root1=1 --root2=-3
==4116== Memcheck, a memory error detector
==4116== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==4116== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==4116== Command: ./quadratic-equation
==4116== Parent PID: 1716
==4116==
==4116== HEAP SUMMARY:
==4116==     in use at exit: 0 bytes in 0 blocks
==4116==   total heap usage: 212 allocs, 212 frees, 13,845 bytes allocated
==4116==
==4116== All heap blocks were freed -- no leaks are possible
```

Рис. 3.3

Теперь проанализируем данные о числе вызовов функция, полученных с помощью инструмента *Callgrind*. Из рис. 3.2 видно, что выполнение функции *drawLine* занимает примерно 25% процессорного времени, затрачиваемого на работу всей программы.

Для оптимизации работы данной функции были сделаны следующие изменения:

```
diff --git a/src/tioTableBegin.c b/src/tioTableBegin.c
index 030406d..60bdcd9 100644
--- a/src/tioTableBegin.c
+++ b/src/tioTableBegin.c
@@ -516,15 +516,20 @@ int capMap( int countColum, char **cap,
int lenColCon )
 /*Draw line function*/
 int drawLine( int lenColCon )
 {
+
+    char *pLine = malloc( WIDTH * sizeof( char ) );
     int i;
     for( i = 0; i < WIDTH; ++ i )
     {
         if((i % lenColCon) == 0)
-            printf("+");
+            pLine[i] = '+';
         else
-            printf("-");
```

```
+            pLine[i] = '-';
     }

-    printf( "+\n" );
+    pLine[i] = '+';
+    pLine[++i] = '\n';
+    fputs( pLine, stdout );

     return 0;
 }
```

Данные изменения помогли уменьшить процессорное время необходимое для выполнения функции *drawLine* (рис. 3.6). Теперь оно составляет около 5% от общего времени выполнения программы.

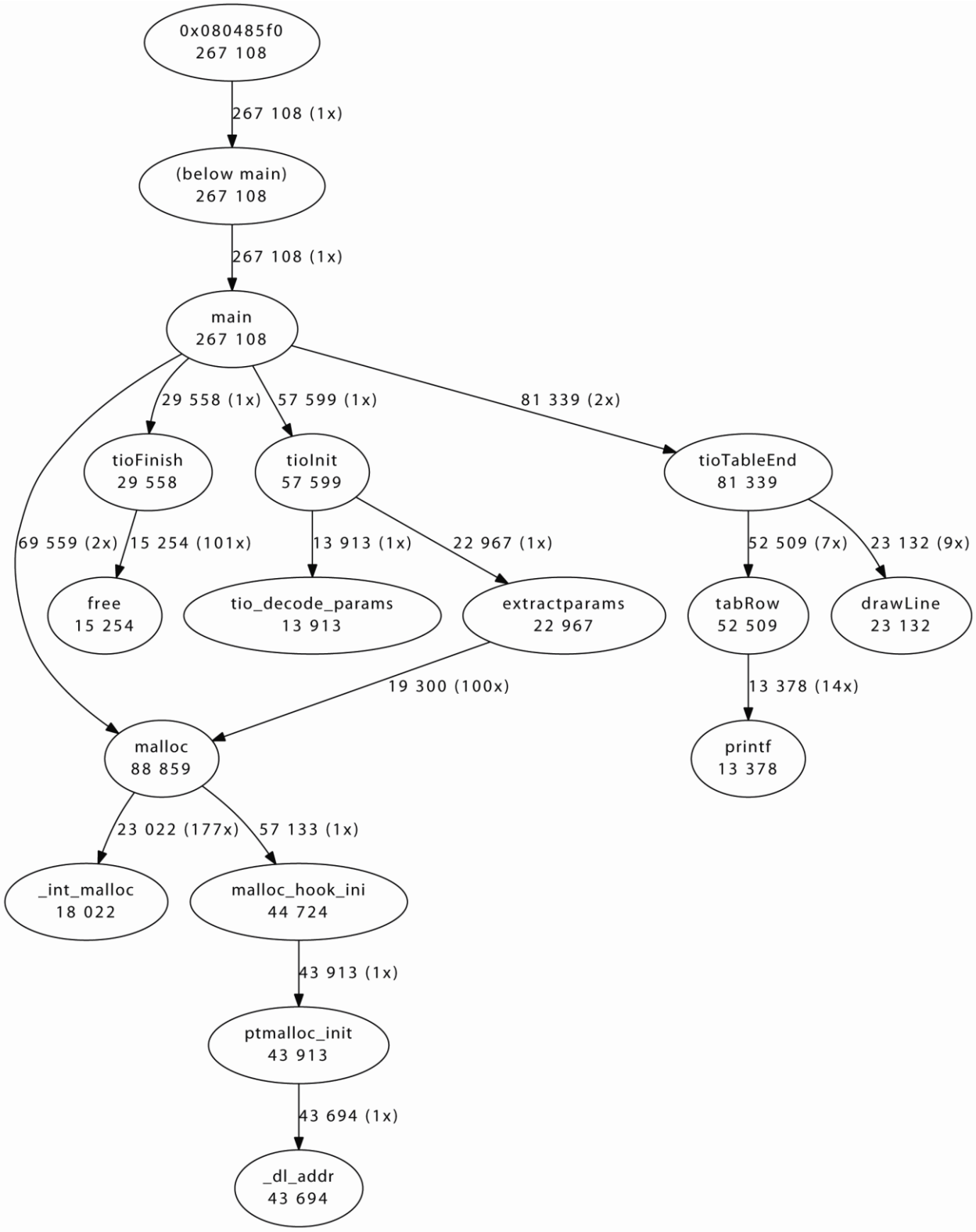


Рис. 3.4

3.3.Отладка и тестирование разрабатываемой библиотеки

Отладка разработанной библиотеки производилась с помощью программы *GDB* (*GNU Debugger*), первоначально написанной Ричардом

Столлмэном в 1988 году и являющейся свободным программным обеспечением.

GDB работает на многих *UNIX*-подобных системах и умеет производить отладку многих языков программирования, включая Си, *C++*, *Free Pascal*, *FreeBASIC*, *Ada* и Фортран.

Отладчик имеет средства для слежения и контроля за выполнением компьютерных программ. Пользователь может изменять внутренние переменные программ и вызывать функции независимо от обычного поведения программы.

С версии 7.0 добавлена поддержка «обратимой отладки», позволяющей отмотать назад процесс выполнения, чтобы посмотреть, что произошло.

При разработке библиотеки *libtio* после добавления новой функции, проводилось автоматическое модульное тестирование по принципу черного ящика, что позволяло быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчало обнаружение и устранение таких ошибок.

Поток вывода после запуска модульных тестов изображен на рисунках рис. 3.5 – рис. 3.8.

```
bin/runtests.sh bin
I have color
(TS): Run test: bin/test_all_out
[RUN]: Запуск ./bin/test_all_out
(II): Hello my darling
(II):This programm is 1-st test for me
(WW): I think It's rather prety
(WW):I programm it well be usefull
(EE): But i think it weel be great
(EE):It's over 2328
[PASS]: ./bin/test_all_out : Тест пройден успешно
(TS): Test  bin/test_all_out [PASS]
(TS): Run test: bin/test_basetioGetC
[RUN]: Запуск test_basetioGetC.c
[PASS]: test_basetioGetC.c : Тест пройден успешно
(TS): Test  bin/test_basetioGetC [PASS]
(TS): Run test: bin/test_basetioGetD
[RUN]: Запуск test_basetioGetD.c
[PASS]: test_basetioGetD.c : Тест пройден успешно
(TS): Test  bin/test_basetioGetD [PASS]
(TS): Run test: bin/test_basetioGetDefC
[RUN]: Запуск test_basetioGetDefC.c
[PASS]: test_basetioGetDefC.c : Тест пройден успешно
(TS): Test  bin/test_basetioGetDefC [PASS]
(TS): Run test: bin/test_basetioGetDefD
[RUN]: Запуск test_basetioGetDefD.c
[PASS]: test_basetioGetDefD.c : Тест пройден успешно
(TS): Test  bin/test_basetioGetDefD [PASS]
(TS): Run test: bin/test_basetioGetDefL
[RUN]: Запуск test_basetioGetDefL.c
[PASS]: test_basetioGetDefL.c : Тест пройден успешно
(TS): Test  bin/test_basetioGetDefL [PASS]
(TS): Run test: bin/test_basetioGetDefS
[RUN]: Запуск self
TRUE
[PASS]: self : Тест пройден успешно
(TS): Test  bin/test_basetioGetDefS [PASS]
(TS): Run test: bin/test_basetioGetL
[RUN]: Запуск self
[PASS]: self : Тест пройден успешно
(TS): Test  bin/test_basetioGetL [PASS]
(TS): Run test: bin/test_basetioGetS
[RUN]: Запуск self
TRUE
[PASS]: self : Тест пройден успешно
(TS): Test  bin/test_basetioGetS [PASS]
(TS): Run test: bin/test_error
Attempt to set error befor error initialization or after error free
(TS): Test  bin/test_error [PASS]
```

Рис. 3.5

```
(TS): Run test: bin/test_error2
(TS): Test  bin/test_error2 [PASS]
(TS): Run test: bin/test_ErrorF
(EE):char = X long = 123 le = 1.154340e+01  float = 11.543400. 100 in oct = 144. This is string! and h = 7b, H = 7B. Happy% end!(TS): Test  bin/test_ErrorF [PASS]
(TS): Run test: bin/test_help
Использование: Test [КЛЮЧ]... [ФАЙЛ]...
Проверка описания программы.
 -k, -d, --list, --ls, --lst    List information about the FILEs (the current dire
                                ctory by default).  Sort entries alphabetically if
                                 none of -cftuvSUX nor --sort. Mandatory arguments
                                 to long options are mandatory for short options t
                                oo.
 --ip, --adress <ПАРАМЕТР>      show / manipulate routing, devices, policy routing
                                 and tunnels
 -s, -S, --sort <ПАРАМЕТР>      sort lines of text files
 --file, --fl <ПАРАМЕТР>        determine file type
 -t <ПАРАМЕТР>                  table view
(TS): Test  bin/test_help [PASS]
(TS): Run test: bin/test_longto
(TS): Test  bin/test_longto [PASS]
(TS): Run test: bin/test_output
(EE):267676836876843768763487686387683726863876876387638746873648763847687643 87jddhkjhfkjdhkjherw;jhl;kjelskjlkjelkjlkjefl;sj;lkej;lkj;lej;
ojlkhjlkewyp9ub oiuro;icnj8p2[oicnpoi2poi2p3ipoi2309878yrhjewhiouyfipdhlu3rpo8u7093kpojkfd09iupjrepwu09ufeoij09fueoijoifeu980j32oiy87y9jddh
lkjlkjesflkjlke
(EE):Символ = Y число 128 с плавающей: 1.154340e+01, ещё раз: 11.543400, 100 в окт: 144, строка: 8048700 в хексе: 7b, в ХЕКСЕ: 7B %THe end.
(EE):Символ = Z число 129 с плавающей: 1.154340e+01, ещё раз: 11.543400, 100 в окт: 144, строка: 8048700 в хексе: 7b, в ХЕКСЕ: 7B %THe end.
(TS): Test  bin/test_output [PASS]
(TS): Run test: bin/test_strcmp
(TS): Test  bin/test_strcmp [PASS]
```

Рис. 3.6

```
(TS): Run test: bin/test_table
Cap string
Call "tioTableBegin".

Call "tioTableRecord".

Call "tioTableEnd".
+----------------------------+----------------------------+----------------------------+----------------------------+
|char                        |st&ring                     |double                      |string                      |
+----------------------------+----------------------------+----------------------------+----------------------------+
|r                           |An advantage of COM+        |23.70                       |Animated by Ryan Woodward   |
+----------------------------+----------------------------+----------------------------+----------------------------+
|e                           |An advantage of COM+        |43.90                       |The essence of COM is a langu|
|                            |                            |                            |age-neutral way of implementi|
|                            |                            |                            |ng objects that can be used i|
|                            |                            |                            |n environments              |
+----------------------------+----------------------------+----------------------------+----------------------------+
(TS): Test  bin/test_table [PASS]
(TS): Run test: bin/test_tioInit
Test start
[RUN]: Запуск self
[PASS]: self : Тест пройден успешно
(TS): Test  bin/test_tioInit [PASS]
```

Рис. 3.7

```
(TS): Run test: bin/test_true
This test is allwayse good:)
(TS): Test  bin/test_true [PASS]
(TS): Run test: bin/test_utf
Амбивалентный
мбивалентный
бивалентный
ивалентный
валентный
алентный
лентный
ентный
нтный
тный
ный
ый
й
(TS): Test  bin/test_utf [PASS]
(TS): Run test: bin/test_version
Version 0.4.8, revision 209
(TS): Test  bin/test_version [PASS]
(TS): Run test: bin/fail_test2
test2
(TS): Test  bin/fail_test2 [PASS]
(TS): Run test: bin/fail_true
(TS): Test  bin/fail_true [PASS]
(TS): Run test: bin/fail_true2
(TS): Test  bin/fail_true2 [PASS]
(TS): All tests PASS
```

Рис. 3.8

Сценарий командной оболочки, позволяющий автоматизировать процесс запуска тестов, представлен в приложении

Вначале сценария приписываем в переменную окружения LD_LIBRERY_PATH папку *./lib*. Это делается для того, чтобы модульные тесты искали скомпилированную библиотеку *libtio* в директории lib, находящуюся в корневой директории проекта. Далее проводим настройку цветов некоторых элементов вывода, таких как *TS, Test, Run test, PASS, FAIL* и т. д. Далее все исполняемые файлы, начинающиеся с *test_*, поочередно запускаются и если завершаются без ошибок, то в поток вывода печатается сообщение о том, что тест пройден. В противном случае – печатается сообщение о том, что тест провален.

Так же среди прочих тестов, имеются тесты, успешным выполнением которых является их завершение с определенным кодом ошибки. Название таких тестов начинается с *fail_*. Такие тесты считаются успешно завершенными, если они возвращают значение равное значению из одноименного файла с типом *.result.*

В конце сценария переменная LD_LIBRERY_PATH возвращается в первоначальное значение.

Если все модульные тесты завершились успехом, то выводится сообщение, символизирующее отсутствие ошибок при автоматическом тестировании. Если хотя бы один тест провалился, то по сценарию выводится сообщение, что модульное тестирование не прошло успешно.

В ходе функционального тестирования было выявлено, что функция *tioTableEnd* отображает таблицу некорректно, если в полях таблицы используются символы кириллицы (рис. 3.9).

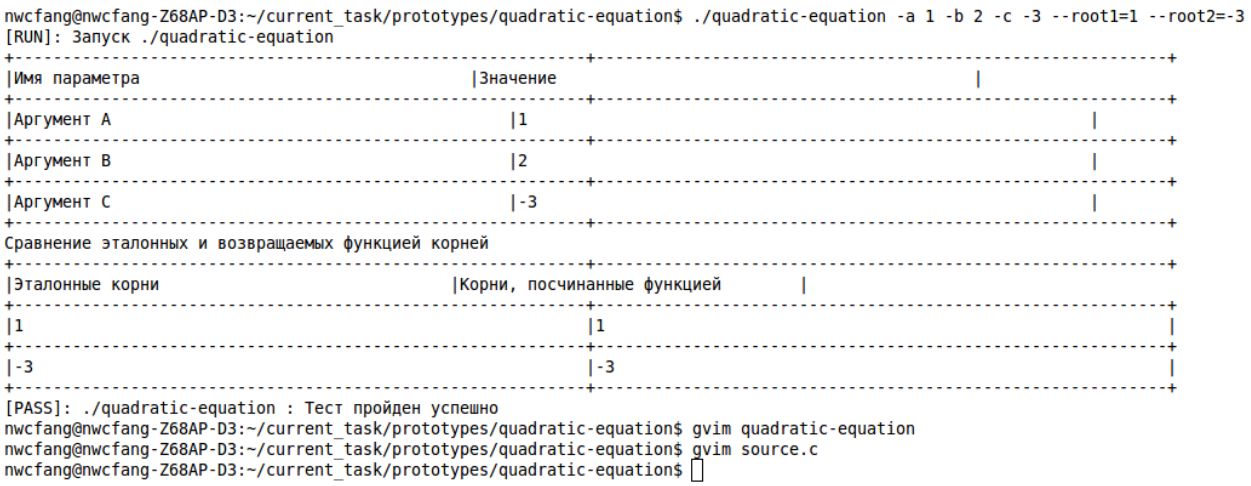


Рис. 3.9

При отладке было установлено что, так как при работе с символьными данными (строками) по умолчанию используется кодировка *UTF-8*, а это значит, что для представления латинских символов требуется 1 байт, тогда как для представления символов кириллицы необходимо отводить под каждую букву 2 байта. В *tioTableEnd* это не было учтено.

```
@@ -345,7 +345,7 @@ int tabRow( void **strings, int *bufType,
int countColum, int lenColCon )
```

```
     /*Calculation number of extra lines of the array*/
     for( i = 0; i < countColum; ++ i )
-    {
+    {
         colStr[i] = strlen( (char *)strings[i] ) /
lenColCon;
         if(max < colStr[i])
             max = colStr[i];
@@ -368,7 +368,7 @@ int tabRow( void **strings, int *bufType,
int countColum, int lenColCon )
         {
             for (j = 0; j < (max + 1); ++ j)
             {
-                if((data[i][j] = (char *) malloc (lenColCon
* sizeof(char))) == NULL)
+                if((data[i][j] = (char *) malloc ( 2 *
lenColCon * sizeof(char))) == NULL)
                 {
                     printf("ERROR!\n");
                     exit(EXIT_FAILURE);
@@ -419,14 +419,42 @@ int tabRow( void **strings, int
*bufType, int countColum, int lenColCon )
         case 4:
             for( extraCounter = 0; extraCounter <=
colStr[i]; ++ extraCounter )
             {
+                int index = 0;
                 j = extraCounter * (lenColCon - 1);
-                for( offset = 0;
+                offset = 0;
+
+                while( ( ( lenColCon - 1 ) != index ) && ( (
(char*)strings[i])[j] != '\0' ) )
+                {
+                    if( ( (char*)strings[i])[j] & 0x80 )
+                    {
+
+                        data[i][extraCounter][offset] =
((char *)strings[i])[j];
+                        ++ offset;
+                        ++ j;
+                        data[i][extraCounter][offset] =
((char *)strings[i])[j];
```

```
+                           ++ j;
+                           ++ index;
+                           ++ offset;
+                       }
+                       else
+                       {
+                           data[i][extraCounter][offset] =
((char *)strings[i])[j];
+                           ++ j;
+                           ++ offset;
+                           ++ index;
+                       }
+
+                   }
+
+
-                   for( offset = 0;
-                           ((offset != (lenColCon - 1)) &&
(((char *)strings[i])[j] != '\0' )); ++ offset, ++ j)
                    {
-                       data[i][extraCounter][offset] = ((char
*)strings[i])[j];
-                   }
                    /*Insert spaces*/

-                   for( offset; offset < (lenColCon - 1); ++
offset)
+                   for( index; index < (lenColCon - 1); ++
index, ++ offset)
                        data[i][extraCounter][offset] = ' ';
                }
                /*Insert spaces*/
```

После исправлений поля с символами кириллицы стали отображаться правильно (рис. 3.10).

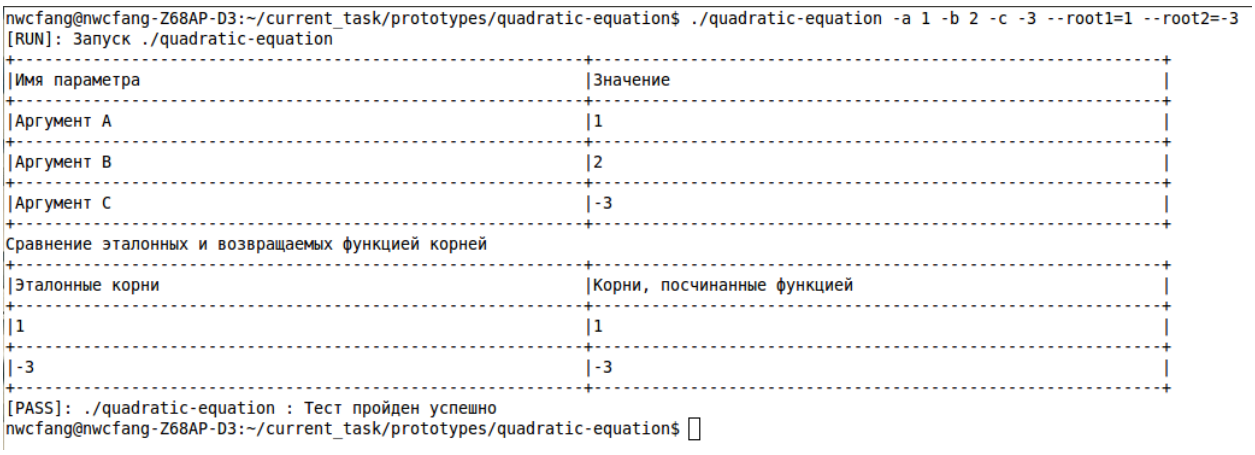


Рис. 3.10

Также было выявлено что функция *tioHelp* не выводит (см. рис. 3.11) название длинных ключей, если у них не существует аналогичных коротких.

```
nwcfang@nf-lrti:~/current-task/prototypes/quadratic-equation$ ./quadratic-equation --help
Использование: ./quadratic-equation [КЛЮЧ]... [ФАЙЛ]...
This is test
 -a <ПАРАМЕТР>                  Параметр A
 -b <ПАРАМЕТР>                  Параметр B
 -c <ПАРАМЕТР>                  Параметр C
 <ПАРАМЕТР>                     Первый корень
 <ПАРАМЕТР>                     Второй корень
nwcfang@nf-lrti:~/current-task/prototypes/quadratic-equation$
```

Рис. 3.11

Исправление, которое помогло решить эту проблему.

```
diff --git a/src/help.c b/src/help.c
index 9e7123a..f630ccd 100644
--- a/src/help.c
+++ b/src/help.c
@@ -29,6 +29,7 @@ int tioHelp( const char* help_msg, const
char* progName,
     int counter;
     for( idx = 0 ; idx < sz ; ++idx )
     {
+        nolong = 0;
         counter = MAX_TAB;
         // Вывод короткого ключа
         if( par[idx].skeys )
```

Теперь ключи отображаются правильно (рис. 3.12).

```
nwcfang@nwcfang-Z68AP-D3:~/current_task/prototypes/quadratic-equation$ ./quadratic-equation --help
Использование: ./quadratic-equation [КЛЮЧ]... [ФАЙЛ]...
This is test
 -a <ПАРАМЕТР>                 Параметр A
 -b <ПАРАМЕТР>                 Параметр B
 -c <ПАРАМЕТР>                 Параметр C
 --root1 <ПАРАМЕТР>            Первый корень
 --root2 <ПАРАМЕТР>            Второй корень
nwcfang@nwcfang-Z68AP-D3:~/current_task/prototypes/quadratic-equation$
```

Рис. 3.12

Еще одна ошибка в логике была выявлена в функции *tioInit*. В случае, когда в программу вместе с командной строкой передавались неименованные параметры, они неправильным образом сохранялись в памяти кучи, что приводило к потерям данных (рис. 3.13).

```
(TS): Test  bin/test_table [PASS]
(TS): Run test: bin/test_tioInit
Test start
[RUN]: Запуск self
unnamedKey is (null)
bin/runtests.sh: строка 41: 11171 Ошибка сегментирования
(TS): Test  bin/test_tioInit [FAIL]
```

Рис. 3.13

После исправлений исходный код функций выглядит так:

```
diff --git a/src/finish.c b/src/finish.c
index f4fb5ed..beeb71c 100644
--- a/src/finish.c
+++ b/src/finish.c
@@ -18,7 +18,7 @@
 #include <tioinit.h>

 #include <finish_msg.h>

-
+#define MAXARGS 100


 extern char *selfname;

@@ -30,6 +30,8 @@ void tioFinish(size_t num)
     {
         num = finish_count;
     }

+    for(int i = 0; i < MAXARGS; ++i)
+        free(tio_argv[i]);
```

```
     tioFree();
     fprintf(stdout, finish_messages[num], selfname);
     free(selfname);

diff --git a/src/init.c b/src/init.c
index bc16c10..da4fed0 100644
--- a/src/init.c
+++ b/src/init.c
@@ -573,6 +573,9 @@ static int extractparams(int start, int
argc, char** argv)
     tio_simple_chain *pt = NULL;
     tio_key_string *p;


+    for( int nfi = 0; nfi < MAXARGS; ++ nfi)
+        tio_argv[nfi] = malloc( sizeof(char) * 100);
+
     for (i=start; i < argc; i++)
     {
         if (argv[i][0]=='-')
@@ -681,7 +684,9 @@ static int extractparams(int start, int
argc, char** argv)
     }
     for (i = cnt; i>0;)
     {
-        tio_argv[--i]=pt->val;
+
+        strcpy( tio_argv[--i], pt->val  );
+        /*tio_argv[--i]=pt->val;*/
         pt=pt->next;
         free(ptr);
         ptr=pt;
```

Корректная работа функции *tioInit* показана на рис. 3.14.

```
(TS): Run test: bin/test_tioInit
Test start
[RUN]: Запуск self
unnamedKey is unnamedKey
[PASS]: self : Тест пройден успешно
(TS): Test  bin/test_tioInit [PASS]
```

Рис. 3.14

4. **Охрана труда и окружающей среды. Разработка мероприятий по обеспечению благоприятных санитарно-гигиенических условий труда инженера**

5. **Экономическая часть. Обоснование экономической эффективности разработки библиотеки функций унификации процессов обработки входных параметров и систематизации выходных данных**

6. **Заключение**

В ходе создания библиотеки унификации получения входных параметров и систематизации выходных данных были рассмотрены опции компилятора *Gcc* для компиляции программ под разные архитектуры процессоров. Также согласно требованиям из исходных данных было проанализированы основные отличия языка Си со спецификацией C99 от предшествующей спецификации C89.

Функции работы с ошибками библиотеки, функция инициализации, функции получения входных параметров, функции обработки выходных данных были спроектированы, реализованы и интегрированы в библиотеку. Для редактирования текста программы использовался редактор *Vim*. Также при разработке применялась система контроля версий *Git*.

Для разработанного набора средств получения входных параметров и систематизации выходных данных была проведена демонстрация основных возможностей на примере программы тестирования функции нахождения корней квадратного уравнения и программы, тестирующей работоспособность COM-порта.

Было проведено профилирование разработанной библиотеки, целью которого было не только увеличение общей производительности, но и выявление утечек памяти и ошибок, связанных с манипуляцией данными в памяти. В качестве профилировщика использовалась программа *Valgrind*.

Каждый раз при добавлении новых возможностей, для всех нетривиальных функции библиотеки проводилось автоматическое модульное

тестирование по принципу черного ящика с целью обнаружения ситуаций, в которых новый функционал препятствовал работе функций, добавленных в библиотеку ранее. Все выявленные ошибки, связанные с логикой работы функций были отлажены.

В разделе по охране труда и окружающей среды был проведен анализ освещения (естественного и искусственного) и микроклимат помещения в котором происходила разработка программного обеспечения. А также проанализированы визуальные параметры устройства отображения информации (монитора Samsung SyncMaster S27A550H).

В разделе обоснования экономической эффективности разработки программного обеспечения был рассчитан показатель годового экономического эффекта при разработке данного программного продукта который равен 5,2 млн. руб. Срок окупаемости проекта составляет 0,66 года (≈ 8 месяцев).

Данный вариант библиотеки libtio не является окончательным. В будущем планируется расширение функциональных возможностей этого программного продукта, а так же доработка уже имеющихся функций, которая может потребоваться при дальнейшем использовании библиотеки в реальных условиях.