Министерство образования и науки РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

Кафедра компьютерных систем в управлении и проектировании (КСУП)

**К ЗАЩИТЕ ДОПУСТИТЬ**

Заведующий кафедрой КСУП

д-р техн. наук, проф.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ю.А. Шурыгин

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**МОДУЛЬ РАСЧЕТА ХАРАКТЕРИСТИК ПРИНЦИПИАЛЬНЫХ СХЕМ СВЧ ДИАПАЗОНА**

Бакалаврская работа (дипломный проект)

по направлению 09.03.01 – Информатика и вычислительная техника

Пояснительная записка

КСУП.

Студент гр. 583-1

\_\_\_\_\_\_\_\_\_\_\_\_\_ Т.С. Скрябина

Руководитель

к.т.н., доцент каф. КСУП

\_\_\_\_\_\_\_\_\_\_\_\_\_ А.А. Калентьев

Томск 2017

**РЕФЕРАТ**

Отчет с., рис., табл., источников, прилож.

~~ПРИНЦИПИАЛЬНАЯ СХЕМА, РЭС, САПР, СВЧ, РАСЧЕТ, МЕТОД, МОДУЛЬ, БИБЛИОТЕКА, ЛИНЕЙНЫЙ АНАЛИЗ.~~

~~Объектами исследования являются методы и модули для расчета характеристик принципиальных схем.~~

~~Цель работы – изучить существующие модули расчета характеристик принципиальных схем СВЧ диапазона.~~

~~В процессе работы проводились исследования методов ручного и автоматизированного расчета схем, работающих в линейном режиме. В результате исследования выбрана программная библиотека, на базе которой будет разрабатываться модуль для анализа принципиальных схем СВЧ-устройств.~~

**ABSTARCT**

[реферат на иностранном языке]

Министерство образования и науки РФ

Федеральное государственное бюджетное образовательное

учреждение высшего образования

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

Кафедра компьютерных систем в управлении и проектировании (КСУП)

УТВЕРЖДАЮ

Заведующий кафедрой КСУП

д-р техн. наук, проф.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ю.А.Шурыгин

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_г.

**ЗАДАНИЕ**

на бакалаврскую работу студенту\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ группа \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ факультет \_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Тема работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (утверждена приказом по вузу от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ № \_\_\_\_\_\_\_\_\_\_\_\_ )

2. Срок сдачи студентом законченного проекта \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. Назначение и область применения системы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Требования к работе \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. Перечень вопросов, подлежащих разработке \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6. Перечень графического материала (с точным указанием обязательных чертежей): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ЗАДАНИЕ СОГЛАСОВАНО:

Консультант по нормам и требованиям ЕСКД \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель проектирования \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ф.И.О. должность, место работы

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г. Подпись\_\_\_\_\_\_\_\_

Задание принято к исполнению

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г. Студент\_\_\_\_\_\_\_\_

подпись

Оглавление

Изм.

Лист

№ докум.

Подпись

Дата

Лист

Шифр с титульного листа

Разраб.

ФИО диплом

Провер.

ФИО руковод

Н. Контр.

Хабибулина

Утверд.

Шурыгин

*Тема дипломного проекта (работы)*

Лит.

Листов

Общее число

ТУСУР, ФВС, каф.КСУП, гр.511-1

[Введение 7](#_Toc484162367)

[1 Выбор библиотеки 8](#_Toc484162368)

[1.1 Методы расчета сложных электрических цепей 9](#_Toc484162369)

[1.2 Описание алгоритмов в коммерческих САПР 12](#_Toc484162370)

[1.3 Сравнение существующих библиотек 16](#_Toc484162371)

[2 Описание функциональности выбранной библиотеки 25](#_Toc484162372)

[3 Подключение выбранной библиотеки к проекту .NET С# 26](#_Toc484162373)

[3.1 Методы интеграции машинного кода в .NET C# 27](#_Toc484162374)

[3.2 Интеграция ngspice.dll в .NET C# 50](#_Toc484162375)

[4 Разработка архитектуры модуля 54](#_Toc484162376)

[Заключение 55](#_Toc484162377)

[Сокращения, обозначения, термины и определения 56](#_Toc484162378)

[Список использованных источников 57](#_Toc484162379)

[Приложение А 59](#_Toc484162380)

# Введение

Изм.

Лист

№ докум.

Подпись

Дата

Лист

Шифр с титульного листа

Человечество вступило в XXI век, когда необходимо решать ряд сложных проблем. Определяющая роль в решении проблем отводится информационным технологиям. Среди информационных технологий автоматизация проектирования занимает особое место [1]. Развитие систем автоматизированного проектирования (САПР) – решение научно-технического прогресса общества. САПР опирается на прочную научно техническую базу: современные средства вычислительной техники, новые способы представления и обработки информации, создание новых численных методов решение инженерных задач и оптимизации [2].

Если говорить об устройствах сверхвысоких частот (СВЧ), то это направление, в котором системы автоматизированного проектирования развиваются очень интенсивно. Современным проектировщикам представлен большой выбор программных средств для моделирования [3]. Но на данный момент коммерческие САПР для СВЧ устройств решают широкий спектр задач, которые могут быть ненужными для пользователя. Высокая функциональность влечет высокие цены. Разработка модулей для расчета характеристик устройств с высокой точностью сложная задача, требующая много времени и затрат. В связи с этим российские производители создают свои программные обеспечения на основе существующих библиотек для анализа принципиальных схем.

# Выбор библиотеки

Изм.

Лист

№ докум.

Подпись

Дата

Лист

Шифр с титульного листа

Большинство проблем, связанных с анализом схем, решаются в два этапа [4]. Первый этап заключается в составлении уравнений электрического равновесия в форме, позволяющей использовать законы Кирхгофа и характеристики элементов, входящих в схему. Второй этап заключается в решении этих уравнений путем применения подходящих аналитических или численных методов. До появления ЭВМ эти уравнения, как правило, решались аналитическим путем; такой подход накладывал жесткие ограничения на размер и тип схем, которые могли быть подвергнуты анализу. Большие линейные схемы (содержащие, скажем, более 50 элементов) или даже небольшие нелинейные схемы редко поддавались точному анализу. Поэтому инженеры-проектировщики полагались в основном на интуицию и усидчивость, выполняя приближенный анализ таких схем. Во всех случаях процесс анализа завершался макетированием схемы и измерениями представляющих интерес переменных параметров.

Однако даже такой подход, предполагающий макетирование, становится непригодным для анализа интегральных схем, потому что оказывается невозможным дублировать интегральную схему дискретными компонентами. Очевидно, что путем макетирования с помощью дискретных элементов невозможно точно воспроизвести паразитные эффекты и характеристики согласования элементов, входящих в интегральную схему. Невозможно также выполнить с помощью макетирования анализ допусков или анализ наихудшего случая, так как практически невозможно осуществить при макетировании изменение параметров приборов. Вместо имитации схемы посредством макетирования были разработаны САПР для разработки электронных устройств, создания микросхем и печатных плат.

В данной работе будут рассмотрены основные методы расчета, существующие коммерческие САПР и программные библиотеки, позволяющие анализировать электрические цепи.

## Методы расчета сложных электрических цепей

В случае, когда схема электрической цепи достаточно сложна и не приводится к схеме одноконтурной цепи или цепи с двумя узлами, пользуются общими методами расчета [5].

Описываемые методы применимы для цепей постоянного и переменного тока.

### Метод токов ветвей

В данном методе сначала нужно определить направления токов в цепи, а затем написать уравнения, описывающие их отношения друг с другом через законы Кирхгофа и Ома [6]. Число уравнений равно количеству неизвестных токов ветвей и определяется как

NМТВ = NB – NJ, (2.1)

где NB – число ветвей электрической цепи;

NJ – число ветвей с источниками тока.

Количество уравнений, составляемых по Закону токов Кирхгофа (ЗТК) равно

N1 = NУЗЛОВ – 1, (2.2)

где NУЗЛОВ – количество узлов электрической цепи.

Количество уравнений, составляемых по Закону напряжений Кирхгофа (ЗНК) равно

N2 = NМТВ – N1. (2.3)

При составлении уравнений по II Закону Кирхгофа следует выбирать независимые контуры, не содержащие источников тока. Как только будут получены уравнения для каждого из неизвестных токов, можно будет решить систему уравнений, рассчитав тем самым все токи, а затем и все напряжения в цепи.

Последовательность расчета:

а) проводится топологический анализ схемы:

1) обозначаются токи во всех ветвях, произвольно выбираются их положительное направление и обозначаются на схеме стрелками;

2) подсчитывается общее число узлов и определяется число независимых узлов, и обозначаются на схеме;

3) подсчитывается число независимых контуров и обозначаются на схеме дугой;

б) по ЗТК для независимых узлов и по ЗНК для независимых контуров относительно токов ветвей записывают уравнения. После привидения подобных членов они сводятся к системе линейных алгебраический уравнений (ЛАУ);

в) Решив систему уравнений относительно токов по методу Крамера находят токи во всех ветвях схемы. Если значения токов отрицательные, то действительные направления их будут противоположны первоначально выбранным направлениям.

### Метод узловых напряжений

Метод заключается в том, что на основании ЗТК определяются потенциалы в узлах электрической цепи относительно некоторого базисного узла. Эти разницы потенциалов называются узловыми напряжениями, причем положительное направление их указывается стрелкой от рассматриваемого узла к базисному.

Напряжение на какой-либо ветви равно разности узловых напряжений концов данной ветви; произведение же этого напряжения на комплексную проводимость данной ветви равно току в этой ветви. Таким образом, зная узловые напряжения в электрической цепи, можно найти токи в ветвях.

Если принять потенциал базовой ветви равным нулю, то напряжения между остальными узлами и базисным узлом будут равны также потенциалам этих узлов. Поэтому данный метод называется также методом узловых потенциалов (МУП).

Если электрическая схема содержит q узлов, то на основании первого закона Кирхгофа получается система из q-1 уравнений.

Порядок расчета:

а) обозначить все токи ветвей и их положительное направление;

б) произвольно выбрать опорный узел и пронумеровать все остальные узлы;

в) определить собственные и общие проводимости узлов, а также узловые токи, т.е. рассчитать коэффициенты в системе уравнений;

г) записывается система уравнений по ЗТК;

д) полученную систему уравнений решить относительно неизвестных потенциалов при помощи метода Крамера;

е) с помощью обобщенного закона Ома рассчитать неизвестные токи;

ж) проверить баланс мощности.

### Метод контурных токов

Метод контурных токов (МКТ) при расчете токов цепи позволяет уменьшить количество уравнений, составляемых по ЗНК:

NМКТ = N2 = NМТВ – N1 (2.4)

Ток в любой ветви цепи модно представить в виде алгебраической суммы контурных токов, протекающих по этой ветви. Выбирают и обозначают известные и неизвестные контурные токи. Известные контурные токи можно считать совпадающими с соответствующими токами источников тока. Неизвестные контурные токи определяются по ЗНК и для них составляется система уравнений метода контурных токов. Решив данную систему находятся неизвестные контурные токи. И находят токи ветвей как сумма контурных токов, протекающих в этой ветви.

Последовательность расчета:

а) топологический анализ схемы:

1) определяют число ветвей;

2) определяют число узлов;

3) посчитывают число независимых контуров;

б) все независимые контура показывают дугой со стрелкой на них, которая показывает положительное направление обхода контура. Все контуры нумеруют и контуру присваивают свой контурный ток. За положительное направление контура принимают положительное направление обхода контура;

в) по ЗНК относительно контурных токов записывают уравнения, которые после приведения подобных членов образуют системы линейных уравнений;

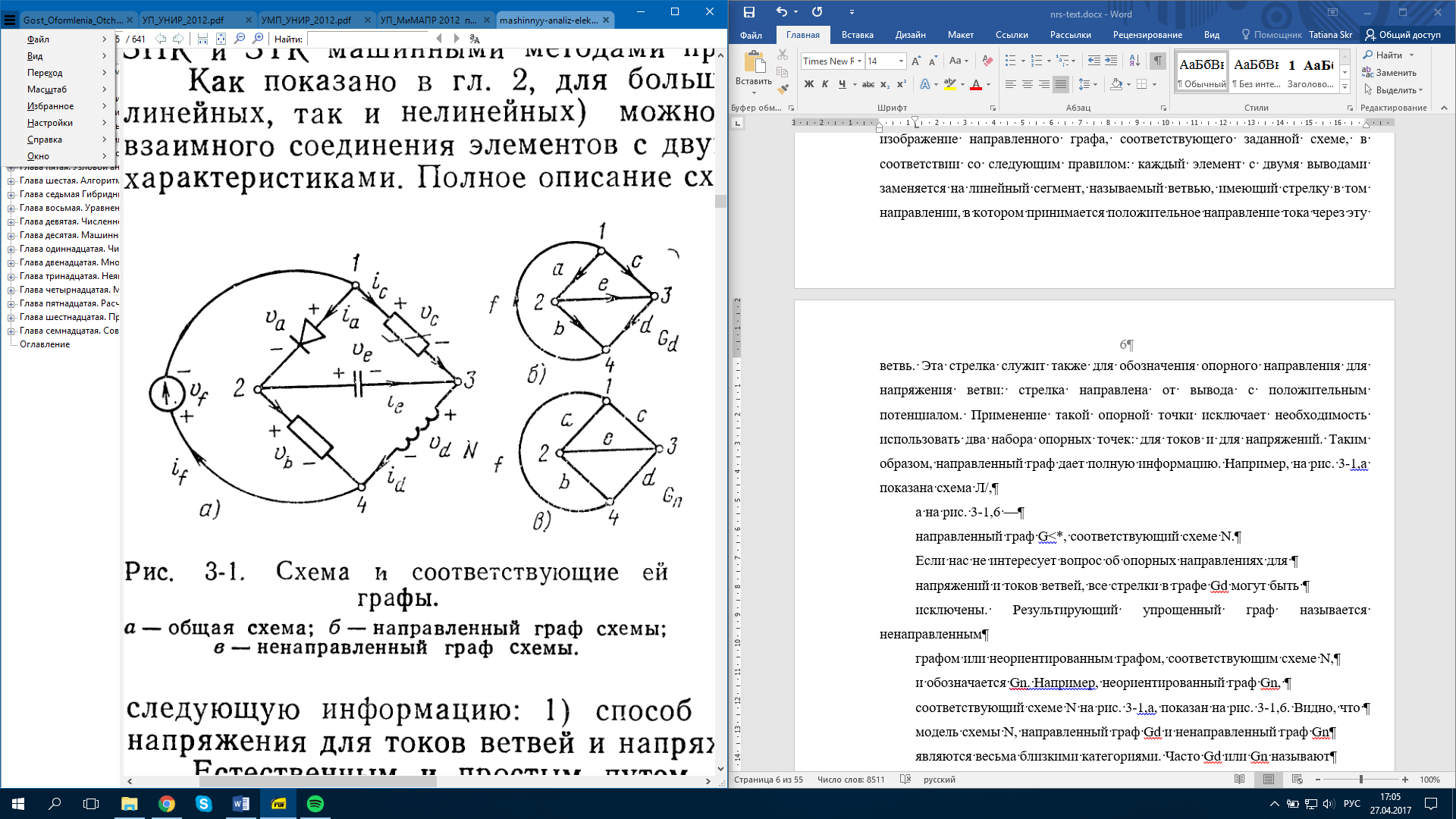
г) по правилу Крамера находят контурные токи;

д) токи в ветвях находят как алгебраическую сумму контурных токов, протекающих через данную ветвь. В алгебраической сумме контурные токи берутся с положительным знаком, если ток ветви совпадает с контурным и отрицательным, если не совпадает. Если токи ветви оказались положительными, то выбранное направление тока совпадает с истинным и наоборот.

## Описание алгоритмов в коммерческих САПР

Любая схема подчиняется трем основным законам: ЗНК, ЗТК и закону элементов (характеристики ветви) [1]. Топология схем рассматривает такие свойства сложных схем, которые связаны только с соединениями ветвей. Она является одним из направлений математики, называемой теорией графов.

Естественным и простым путем для описания информации является изображение направленного графа, соответствующего заданной схеме, в соответствии со следующим правилом: каждый элемент с двумя выводами заменяется на линейный сегмент, называемый ветвью, имеющий стрелку в том направлении, в котором принимается положительное направление тока через эту ветвь. Эта стрелка служит также для обозначения опорного направления для напряжения ветви: стрелка направлена от вывода с положительным потенциалом. Применение такой опорной точки исключает необходимость использовать два набора опорных точек: для токов и для напряжений. Таким образом, направленный граф дает полную информацию. Например, на Рисунок 2.1 показана схема N, а на Рисунок 2.2 – направленный граф G, соответствующий схеме N.

Рисунок 2.1 – Схема N

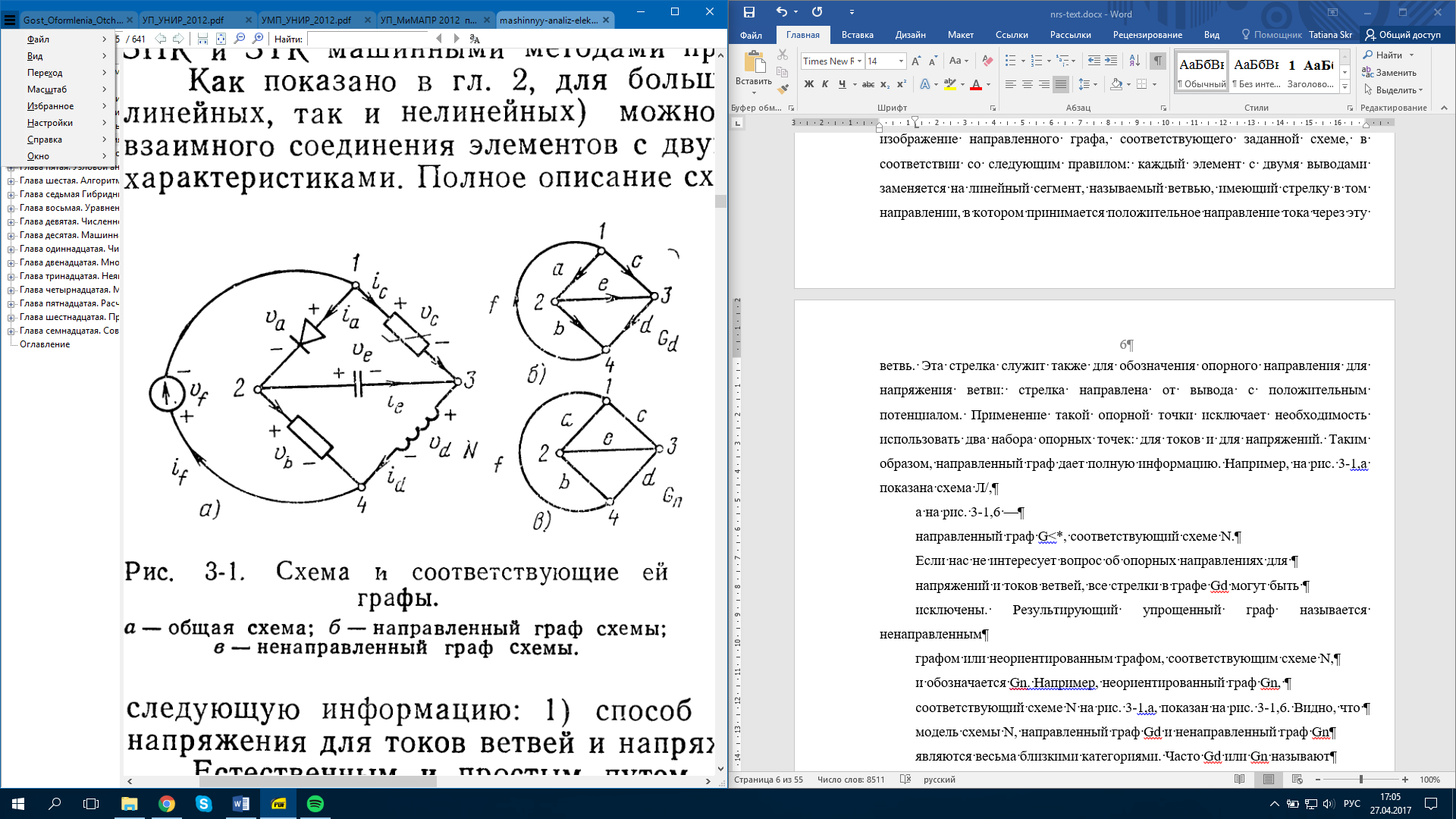


Рисунок 2.2 – Граф G

Хотя направленный граф G полностью описывается соединениями и опорными направлениями ветвей сложной схемы, эта форма выражения неудобна для записи в памяти ЭВМ. Для записи в цифровом виде наиболее удобно представление в виде матриц. Содержащаяся в направленном графе информация может быть полностью представлена матрицей, называемой матрицей инциденций.

Матрица инциденций (инцидентности) – это одна из форм представления графа, в которой указываются связи между инцидентными элементами графа (ребро(дуга) и вершина). Столбцы матрицы соответствуют ребрам, строки — вершинам.

Для того чтобы компактно выразить уравнения ЗНК в виде единственного матричного уравнения, необходимо ввести матрицу, называемую матрицей контуров (матрицей схемы), соответствующую направленному графу.

Чтобы выразить уравнения обобщенного ЗТК в виде единственного матричного уравнения, введем матрицу, называемую матрицей сечений (применительно к направленному графу). Законы Кирхгофа для тока и напряжения накладывают определенные условия на токи и напряжения ветвей в схеме. Вследствие этого лишь небольшая часть токов и напряжений ветвей независимы, а остальные переменные ветвей могут быть выражены через независимые величины.

Независимая система ЗНК и ЗТК уравнений для любого схемного графа может быть получена автоматически с помощью ЭВМ в виде топологических матриц. Результирующие уравнения относятся только к соединениям ветвей в схеме и содержат удвоенное количество переменных относительно количества независимых уравнений. Остальные уравнения можно получить из характеристик элементов. В этой работе рассматриваются только линейные схемы, содержащие RLC-элементы, независимые источники напряжения или тока и управляемые напряжением источники тока.

Для анализа линейных электрических цепей в САПР используют модифицированный МУП. Были рассмотрены следующие САПР: Microwave Office [7] (National Instruments Applied Wave Research (NI AWR), США) и Advanced Design System [8] (Keysight Technologies, США).

Идея модифицированного МУП заключается в разбиении элементов цепи (ветвей) на группы [9]:

1. ветви, которые можно описать через проводимости (ток через них не будет определен);
2. ветви, которые нельзя описать через проводимости, либо те элементы, ток в которых необходимо определить;
3. ветви независимых источников тока.

В результате решения будут найдены напряжения узлов Un и токи ветвей второй группы I2. Напряжение ветвей можно определить позже по уравнении связи напряжений ветвей и узлов Ub = At∙Un, а токи ветвей первой группы – на основании компонентных уравнений I1 = Y1∙U1. При расчете во временной области начальные токи в катушках и напряжения на конденсаторах учитываются с помощью эквивалентных источников, следующих из преобразования Лапласа.

Матричная форма уравнения модифицированного метода узловых потенциалов имеет вид:

(2.5)

YR – сокращенная подматрица узловых проводимостей, не учитывающая управляемые током элементы, G – подвектор независимых источников тока, подматрица B содержит частные производные от полученных по закону Кирхгофа для токов уравнений по дополнительным переменным. Вольтамперные связи, дифференцированные по дополнительным переменным, представлены подматрицами C и D. Подвектор F содержит вклад реактивных элементов в полный вектор независимых источников тока. Реактивные элементы рассматриваются только во временной области с учетом их конечно-разностного представления.

Таким образом, модифицированная узловая система представляет собой обычную узловую матрицу, построенную из ветвей первой группы, дополненную по принципу модифицированных табличных уравнений уравнениями ветвей второй группы. Вектор свободных членов соответственно представляет собой вектор эквивалентных узловых источников тока, дополненный источниками напряжениями ветвей второй группы. Вектор неизвестных содержит узловые напряжения и токи ветвей второй группы.

Реализация модифицированного метода узловых потенциалов достаточно проста и состоит в анализе признака ветви и внесении определенных коэффициентов либо в узловую матрицу, либо в ее дополнение. Источники тока вносятся в первую часть вектора свободных членов, а источники напряжения – во вторую.

В Microwave Office интегрированы библиотеки HSPICE (Synopsys), APLAC (NI AWR) и Spectre (Cadence). В Advanced Design System за линейный анализ отвечает элемент линейного моделирования – W2306EP Linear Simulator.

## Сравнение существующих библиотек

В ходе исследовательской работы были рассмотрены следующие симуляторы электронных схем: SPICE, SIMetrix/SIMPLIS, Ngspice, LTspice, TINA TI, Qucs, HSPICE. Ниже приведены их описания.

**SPICE** (Simulation Program with Integrated Circuit Emphasis) — симулятор электронных схем общего назначения с открытым исходным кодом. Является мощной программой, используемой в разработке как интегральных схем, так и печатных плат для проверки целостности схемы и для анализа её поведения [10].

SPICE был разработан в Electronics Research Laboratory в Калифорнийском университете в Беркли Лоуренсом Нагелем. SPICE стала индустриальным стандартом симуляции электрических схем. SPICE послужил основой для разработки множества других программ симуляции схем, как в академической, так и в промышленной среде. Первая коммерческая версия — ISPICE, (National CSS). Наиболее выдающиеся коммерческие версии SPICE: HSPICE (изначально Meta Software, ныне Synopsys) и PSPICE (ныне Cadence Design Systems). Академические версии программы: XSPICE (Georgia Tech, с поддержкой смешанных аналого-цифровых моделей) и Cider (ранее CODECS, UC Berkeley и Oregon State Univ.; с поддержкой полупроводниковых устройств). Индустрия проектирования интегральных схем достаточно рано начала пользоваться SPICE, и, до развития коммерческих реализаций, многие компании-разработчики микросхем имели собственные версии SPICE. В настоящее время крупные производители микросхем развивают собственные программы симулирования на базе SPICE. Например, ADICE у компании Analog Devices, LTspice у Linear Technology, Mica у Freescale Semiconductor, TISPICE у Texas Instruments.

Впервые SPICE1 был представлен на конференции в 1973. Программа была написана на языке FORTRAN и использовала анализ цепи методом узловых потенциалов для построения уравнений схемы. Метод узловых потенциалов имел ограничения в работе с индуктивностями, источниками переменного напряжения и с различными вариантами управляемых генераторов тока и напряжения. В SPICE1 было доступно небольшое количество элементов, программа использовала анализ переходных процессов (transient analysis) с фиксированным шагом по времени. Популярность пришла ко второй версии программы, SPICE2, в 1975 году. Она тоже была написана на FORTRAN, но имела больше элементов, позволяла изменять шаг по времени при анализе переходных процессов, уравнения цепей формулировались при помощи модифицированного метода узловых потенциалов (modified nodal analysis), устраняя тем самым ограничения метода узловых потенциалов. Последняя версия SPICE, написанная на языке FORTRAN, - 2G.6 (1983 год). Следующую версию, SPICE3 разработал Томас Кворлс (Thomas Quarles) в 1989. Она написана на языке Си, использует тот же формат данных об электрических цепях (netlist) и поддерживает визуализацию в среде X Window System.

В 2011 году появление SPICE было отнесено к IEEE Milestone, было отмечено, что SPICE и его производные стали неотъемлемой частью разработки практически любой интегральной схемы.

SPICE стал популярен, поскольку поддерживал анализ и содержал модели, необходимые для разработки интегральных схем того времени, и при этом был достаточно быстрым для практического использования. Предшественники SPICE часто имели лишь одно предназначение, например, BIAS позволял рассчитывать режимы биполярных транзисторов; SLIC производил анализ малых сигналов. SPICE комбинировал в себе несколько режимов анализа и достаточно обширную библиотеку моделей устройств.

SPICE2 включает в себя:

- AC анализ (анализ по переменному току);

- DC анализ (анализ по постоянному току) для слабых сигналов;

- анализ DC transfer curve;

- анализ шумов;

- анализ передаточной функции (входное и выходное усиление малых сигналов и вычисление импеданса);

- анализ переходных процессов.

Все SPICE симуляторы работают с текстовым файлом списка соединений (netlist). Список соединений сдержит перечень схемных элементов, узлов, с которыми эти элементы связаны, определения моделей и различные SPICE директивы.

Далее представлено очень краткое описание SPICE языка:

\* КОММЕНТАРИЙ

[ТИП\_ИМЯ\_КОМПОНЕНТА] [УЗЕЛ1] .. [УЗЕЛ\_N] [ПАРАМЕТР\_1] ... [ПАРАМЕТР\_N]

.END ставится в конце файла

Типы компонентов:

Rxx resistor резистор

Lxx inductor катушка индуктивности

Cxx capacitor конденсатор

Vxx voltage source источник напряжения

Ixx current source источник тока

Exx voltage controlled voltage source

Hxx current controlled voltage source

Fxx current controlled current source

Gxx voltage controlled current source

Dxx diode диод

Jxx junction FET JFET транзистор

Mxx MOSFET МОС транзистор

Bxx GaAsFET транзистор на арсениде галлия GaAs

Qxx BJT биполярный транзистор

**SIMetrix/SIMPLIS –** уникальный симулятор, созданный для высокоскоростного анализа линейных и смешанных цепей [11].

Приложение SIMPLIS (аббревиатура от «SIMulation for Piecewise LInear System») было создано с целью быстрого моделирования источников питания и импульсных стабилизаторов, а также исследования ключевых схем преобразователей напряжения. Оно также, как и программа SPICE, функционирует на уровне компонентов радиосхемы, однако анализы переходных процессов выполняет в 10-50 раз быстрее. Подобной скорости моделирования модуль SIMPLIS достигает благодаря применению кусочно-линейной аппроксимации взамен решения систем нелинейных уравнений, используемых в программе SPICE. Приложение выполняет следующие виды анализов: переходных процессов, ФЧХ/АЧХ с замкнутой петлей авторегулирования и в рабочей точке. Последний анализ представляет собой ускоренный расчет установившегося режима работы импульсного блока питания при постоянных нагрузке и питании. Для особо сложных в плане симуляции случаев предусмотрен еще один режим анализа – Multi-Tone AC, являющийся подобием двухчастотного метода определения интермодуляционных искажений. Кроме того, в программе имеется функция автоматического преобразования SPICE-моделей в модели формата SIMPLIS.

Вторая программа SIMetrix является полноценным симулятором аналогово-цифровых схем. В ее основе лежат два распространенных продукта – XSPICE и SPIСE. Приложение имеет схемный редактор, симулятор и графический постпроцессор. В качестве основного метода итерации применяется решение Ньютона-Рэфсона, обеспечивающего быстроту расчетов режимов переходных процессов и по постоянному току. Для особо «каверзных» схем имеется алгоритм псевдопереходного анализа с изоляцией/фиксацией не вычисляемых узловых потенциалов и адаптируемым шагом итерации. Программа SIMetrix выполняет поддержку моделей элементов спецификации MOS9, BSIM3/4, Mextram, VBIC и совместима с файлами HSPICE. Кроме того, приложение имеет пробник Бодэ и калькулятор КПД, а также поддерживает создание произвольного логического блока, имитирующего работу цифрового устройства.

Программы, входящие в данный комплект, были разработаны благодаря усилиям двух компаний-партнеров – SIMetrix Technologies (Великобритания, графство Беркшир, город Тэтчем) и SIMPLIS Technologies (США, штат Орегон, город Портленд).

**Ngspice** – симулятор электронных схем общего назначения с открытым исходным кодом, обеспечивающий моделирование в режиме смешанных сигналов и на смешанном уровне [12]. Является расширением Spice3f5, включившим в себя код проекта Cider, обеспечивающий моделирование на смешанном уровне, и проекта XSPICE, обеспечивающий моделирование смешанных сигналов. Является мощной программой, используемой в разработке как интегральных схем, так и печатных плат для проверки целостности схемы и для анализа её поведения.

**LTspice** (он же SwitcherCAD) представляет собой универсальную среду для проектирования и создания электрических схем с интегрированным симулятором смешанного моделирования [11]. Программа позволяет быстро менять компоненты и параметры электронных схем, испытывать работоспособность новых вариантов, находить оптимальные решения. Возможна загрузка списка соединений, сгенерированного другими инструментами для рисования схем или созданного вручную. От аналогичных программ (Microcap, OrCAD) рассматриваемое ПО отличается малым объемом необходимого дискового пространства и более высокой скоростью моделирования процессов.

LTspice содержит полную библиотеку компонентов компании Linear Technology Corporation (пассивные элементы и интегральные схемы, включая редкие модели импульсных контроллеров и регуляторов). Поскольку программа использует стандартные SPICE-модели электронных деталей, к имеющейся базе можно добавлять библиотеки сторонних производителей, а также создавать свои собственные модели.

**TINA-TI** представляет собой обычный SPICE-симулятор с простым, интуитивно понятным графическим интерфейсом, позволяющим освоить программу в кратчайшие сроки [11]. Все компоненты, представленные в TINA-TI, распределены по шести группам: основные пассивные радиодетали, ключи, полупроводники, измерительные приборы, макромодели сложных устройств и источники. Также данная программа включает в себя несколько десятков разнообразных примеров.

TINA-TI предоставляет широкие возможности по рисованию и редактированию электронных схем. Имеются следующие виды анализов: по постоянному и переменному току (сюда входит: вычисление узловых напряжений, создание таблицы результатов, построение переходных характеристик и температурный анализ), переходных процессов, шумов, преобразование Фурье и некоторые другие. В зависимости от вида выполняемого анализа программа генерирует результаты в виде графиков или таблиц. В программе TINA-TI также доступны возможности тестирования и измерения сигналов. Для этого существуют следующие виртуальные приборы: осциллограф, анализатор сигналов, цифровой тестер (с измерителем частоты), генератор функций и записывающее устройство. Виртуальные приборы программного комплекса максимально приближены по использованию к реальным устройствам. «Подключить» их можно к любой точке рассматриваемой схемы. Вся информация, снятая виртуальными приборами, может сохраняться в памяти компьютера.

**Qucs** – свободная программа, предназначенная для моделирования электронных цепей. Распространяется по лицензии GPL. Позволяет моделировать электронную аппаратуру в режиме малого и большого сигнала, а также шумовые характеристики. Пользовательский интерфейс — графический. Цифровая аппаратура моделируется с использованием VHDL и/или Verilog [13].

Отличительной особенность данного симулятора является расширенные функции для моделирования схем СВЧ-устройств: моделирование S-параметров, специальные модели СВЧ-компонентов и постпроцессор с расширенными функциями анализа комплексных сопротивлений в частотной области. По возможностям анализа СВЧ-устройств Qucs приближается к таким проеприетарным аналогам как AWR MicrowaveOffice. Но симулятор Qucsator несовместим со SPICE и поддерживает SPICE только через слой совместимости, что значительно осложняет использование существующих библиотек моделей электронных компонентов. Несмотря на отличные характеристики движка Qucsator для моделирования СВЧ-схем в частотной, во временной области имеются многочисленные ошибки.

**HSPICE** – система моделирования, а также сами модели, основанные на SPICE [14]. Изначально был создан Ashawna Hailey, позднее был куплен фирмой Synopsys. В настоящее время HSPICE является стандартом де-факто ("золотой" стандарт индустрии) для производителей микросхем, т.к. обеспечивает максимально правдоподобное моделирование узлов на транзисторном уровне. В настоящее время пакет сильно расширен за счет внедрения возможностей анализа в частотной области, анализа кабелей и полосковых линий, а также возможности импорта результата моделирования в других САПР (например, импорт S-параметров, с последующим обратным преобразованием Фурье). Стоит отметить, что эти возможности уже не могут обеспечить точности, которую дают специализированные САПР (например, HFSS (AnSys), или ADS (Agilent Techologies) для анализа кабелей, или печатных проводников), но позволяют качественно оценить работу системы.

В таблице 2.1 представлено сравнение существующих библиотек для анализа принципиальных схем СВЧ диапазона.

Таблица 2.1 – Сравнительная таблица

|  |  |  |
| --- | --- | --- |
| Библиотека | Лицензия | Анализ |
| SPICE | BSD | По переменному току, по постоянному току, передаточной функции, шумов, переходных процессов |
| SIMetrix/ SIMPLIS | платная | Переходного процесса, периодический анализ рабочих точек, по переменному току |
| Ngspice | BSD | Нелинейный по постоянному току, нелинейный переходных процессов, линейный по переменному току |
| LTSpice | freeware | Переходных процессов, по переменному току, шумов и по постоянному току, преобразование Фурье |
| TINA | freeware | По переменному току, переходных процессов, по постоянному току, преобразование Фурье |
| qucsator | GPL | По постоянному току, по переменному току, гармонический баланс, по цифровому сигналу, переходных процессов, S-параметров, развертка по параметру, оптимизация |
| HSPICE | платная  от $1000 в год (учебная версия) | По переменному току, по постоянному току, по цифровому сигналу, переходных процессов, спектральный, линейный, статистический, Монте-Карло, оптимизация |

Информация по типам лицензий представлен в приложении А.

Сделав обзор существующих библиотек можно сделать следующие выводы:

- возможности бесплатных библиотеки не сильно уступают платным;

- qucsator имеет расширенные функции для анализа СВЧ-устройств. Однако он несовместим со SPICE и поддерживает SPICE только через конвертер, что значительно осложняет использование существующих библиотек моделей электронных компонентов;

- библиотеки SPICE и Ngspice являются хорошим вариантом для разработки своего программного продукта.

# Описание функциональности выбранной библиотеки

[входные-выходные данные библиотеки, описания импортируемых методов]

Входные данные для библиотеки передаются в виде файла требуемого формата. Выходными данными является массив строк.

Структура входного файла

# Подключение выбранной библиотеки к проекту .NET С#

На сегодняшний день стала очень популярна платформа .Net Framework и язык C#. На ней пишутся приложения, игры, инструменты [?]. Язык программирования C# является наиболее гибким языком, реализующий быструю и удобную работу с формами. Однако существует много кода, написанного на Cи или C++, который часто нужно использовать с C# проектах.

Это могут быть математические библиотеки, код оставшийся от прошлых проектов или другой проект, для которого нужно сделать графический интерфейс.

В данном разделе рассмотрены методы, как интегрировать машинный код в C# проект и успешно его развивать. В качестве интегрированной среды разработки используется Microsoft Visual Studio.

## Методы интеграции машинного кода в .NET C#

Для начала необходимо определить понятия управляемого и неуправляемого кода. Управляемый код (англ. managed code) – термин, введённый фирмой Microsoft, для обозначения кода программы, исполняемой под «управлением» виртуальной машины .NET – Common Language Runtime (CLR) или Mono. При этом машинный код называется неуправляемым кодом (англ. unmanaged code) [wikipedia]. В общем случае при написании С#-программы создается управляемый код.

Основная часть выбранного симулятора электронных схем – Ngspice реализована на языке Си, в том числе и динамически подключаемая библиотека, которая является интерфейсом прикладного программирования (API) симулятора. Соответственно, API у Ngspice является неуправляемым кодом.

Существуют два способа интеграции неуправляемого кода в C# проект.

1. Сделать обертку на языке C++/CLI.
2. Использовать механизм Platform Invoke.

### Язык С++/CLI

C++/CLI – промежуточный язык для склеивания кода на C/C++ и .NET.

Это идеальный язык для генерации связующего кода между управляемым и неуправляемыми средами исполнения, поскольку он позволяет генерировать код для обоих сред и генерирует код перехода, избавляя разработчиков от необходимости склеивать что-то вручную [3]. Суффикс CLI – обозначает то, что язык реализует спецификацию Common Language Infrastructure, т.е. является полноправным членом семейства языков платформы .NET.

Есть множество ситуаций, требующих написания обёртки над библиотекой. Можно создавать обёртку для библиотеки, чей код можно изменять [4]. Можно оборачивать часть Win32 API, обёртка для которой отсутствует в FCL. Возможно, создаётся обёртка для сторонней библиотеки. Библиотека может быть как статической, так и динамической (DLL). Более того, это может быть как C, так и C++ библиотека. Эта глава содержит практические рекомендации, общие советы и решения для нескольких конкретных проблем.

Одной из основных целей при создании .NET было обеспечение взаимодействия разных языков. Если вы создаёте обёртку для нативной библиотеки, то возможность взаимодействия с разными языками приобретает особенную важность, поскольку разработчики, использующие библиотеку, скорее всего, будут использовать C# или другие языки .NET. Common Language Infrastructure (CLI) – это основа спецификации. Важной частью этой спецификации является Common Type System (CTS). Хотя все языки .NET основываются на общей системе типов, не все языки поддерживают все возможности этой системы.

Чтобы чётко определить, смогут ли языки взаимодействовать между собой, CLI содержит Common Language Specification (CLS). CLS — это контракт между разработчиками, использующими языки .NET, и разработчиками библиотек, которые можно использовать из разных языков. CLS задаёт минимальный набор возможностей, который обязан поддерживать каждый язык .NET. Чтобы библиотеку можно было использовать из любого языка .NET, соответствующего CLS, возможности языка, используемые в публичном интерфейсе библиотеки, должны быть ограничены возможностями CLS. Под публичным интерфейсом библиотеки понимаются все типы с видимостью public, определённые в сборке, и все члены таких типов с видимостью public, public protected или protected.

Можно использовать атрибут CLSCompliantAttribute, чтобы обозначить тип или его член как соответствующий CLS. По умолчанию, не помеченные этим атрибутом типы считаются не соответствующими CLS. Если вы примените этот атрибут на уровне сборки, то по умолчанию все типы будут считаться соответствующими CLS. Следующий пример показывает, как применять этот атрибут к сборкам и типам:

[assembly: CLSCompliant(true)]; // типы с видимостью public соответствуют CLS, если не указано иное

namespace ManagedWrapper

{

public ref class SampleCipher sealed

{

// ...

};

[CLSCompliant(false)] // этот класс явно помечен как не соответствующий CLS

public ref class AnotherCipherAlgorithm sealed

{

// ...

};

}

Согласно правилу 11 CLS все типы, присутствующие в сигнатурах членов класса (методов, свойств, полей и событий), видимых снаружи сборки, должны соответствовать CLS. Чтобы правильно применять атрибут [CLSCompliant], вы должны знать, соответствуют ли типы параметров метода CLS. Чтобы определить соответствие CLS, надо проверить атрибуты сборки, в которой объявлен тип, а также атрибуты самого типа.

В Framework Class Library (FCL) также используется атрибут CLSCompliant. mscorlib и большинство других библиотек FCL применяют атрибут [CLSCompliant(true)] на уровне сборки и помечают типы, не соответствующие CLS, атрибутом [CLSCompliant(false)].

Учтите, что следующие примитивные типы в mscorlib помечены как несоответствующие CLS: System::SByte, System::UInt16, System::UInt32 и System::UInt64. Эти типы (или эквивалентные им имена типов char, unsigned short, unsigned int, unsigned long и unsigned long long в C++) нельзя использовать в сигнатурах членов типов, которые считаются соответствующими CLS.

Если тип считается соответствующим CLS, то все его члены также считаются таковыми, если явно не указано обратного. Пример:

using namespace System;

[assembly: CLSCompliant(true)]; // типы с видимостью public соответствуют CLS, если не указано иное

namespace ManagedWrapper

{

public ref class SampleCipher sealed // SampleCipher соответствует CLS из-за атрибута сборки

{

public:

void M1(int);

// M2 помечен как несоответствующий CLS, потому что тип одного из его аргументов не соответствует CLS

[CLSCompliant(false)]

void M2(unsigned int);

};

}

К сожалению, компилятор C++/CLI не показывает предупреждений, когда тип, помеченный как соответствующий CLS, нарушает правила CLS. Чтобы понять, помечать тип как соответствующий CLS или нет, надо знать следующие важные правила CLS:

Имена типов и их членов должны быть различимы в языках с регистро-независимыми идентификаторами (правило 4).

Глобальные статические (static) поля и методы не совместимы с CLS (правило 36).

Пользовательские атрибуты должны содержать поля только следующих типов: System::Type, System::String, System::Char, System::Boolean, System::Int[16|32|64], System::Single и System::Double (правило 34).

Объекты исключений должны иметь тип System::Exception или унаследованный от него тип (правило 40).

Все аксессоры свойств должны быть либо виртуальными, либо не виртуальными одновременно, то есть не допускается смешение виртуальных и не виртуальных аксессоров (правило 26).

Упакованные значения не соответствуют CLS (правило 3).

Неуправляемые указатели не соответствуют CLS (правило 17). Под это правило также подпадают и ссылки в C++. Доступ к нативным классам, структурам и объединениям также осуществляется по указателю. Из этого следует, что данные нативные типы также не соответствуют CLS.

Прежде чем создавать обёртку для некоторой библиотеки, убедитесь, что FCL не содержит готового класса с требуемыми методами. FCL может предложить больше, чем кажется на первый взгляд. Например, BCL уже содержит довольно много алгоритмов шифрования. Они находятся в пространстве имён System::Security::Cryptography. Если нужным вам алгоритм шифрования уже есть в FCL, вам не нужно заново создавать для него обёртку. Если FCL не содержит реализации алгоритма, для которого вы хотите создать обёртку, но приложение не завязано на алгоритм, реализованный в нативном API, то обычно предпочтительней использовать один из стандартных алгоритмов из FCL.

Ваши типы должны не только соблюдать правила CLS, но и, по возможности, вписываться в философию .NET.

### Механизм Platform Invoke

Механизм Platform Invoke, более известный как P/Invoke, позволяет вызывать из управляемого кода функции в стиле языка C, экспортируемые библиотеками DLL [5]. Чтобы воспользоваться механизмом P/Invoke, управляемый код должен объявить статический внешний (static extern) метод с сигнатурой (типы параметров и тип возвращаемого значения), эквивалентной функции на языке C. Сам метод должен быть снабжен атрибутом DllImport, определяющим, как минимум, библиотеку DLL, экспортирующую требуемую функцию. Чтобы создать прототип, который допускает вызов неуправляемого кода для правильного маршалинга данных, необходимо выполнить указанные ниже действия:

- применить атрибут DllImportArtibutr к статической функци или методу в управляемом коде;

- заменить неуправляемые типы данных на управляемые.

Вызов неуправляемого кода – это служба, позволяющая управляемому коду вызывать неуправляемые функции, реализованные в библиотеках динамической компоновки (DLL), например, функции библиотек Win32 API. Он обнаруживает и вызывает экспортированную функцию и при необходимости маршалирует ее аргументы (целые числа, строки, массивы, структуры и так далее) через границы взаимодействия.

Использование экспортированных функций DLL включает в себя следующие пункты [6].

1. Идентификация функций в библиотеках DLL.

Как минимум, необходимо указать имя функции и имя библиотеки DLL, содержащей ее.

1. Создание класса, содержащего функции DLL.

Можно использовать существующий класс, создать отдельный класс для каждой неуправляемой функции или создать один класс, содержащий набор связанных неуправляемых функций.

1. Создание прототипов в управляемом коде.

[Visual Basic] Используйте оператор Declare с ключевыми словами Function и Lib. В редких случаях можно использовать атрибут DllImportAttribute с ключевыми словами Shared Function. Такие случаи рассматриваются далее.

[C#] Используйте атрибут DllImportAttribute для идентификации библиотеки DLL и функции. Пометьте метод модификаторами static и extern.

[C++] Используйте атрибут DllImportAttribute для идентификации библиотеки DLL и функции. Пометьте метод или функцию оболочки модификатором extern "C".

1. Вызов функции DLL

Вызовите метод управляемого класса так же, как и любой другой управляемый метод. Передача структур и реализация функций обратного вызова представляют собой особые случаи.

Вызов неуправляемого кода использует метаданные для нахождения экспортированных функций и маршалинга их аргументов во время выполнения. Этот процесс показан на рисунке 1.1.

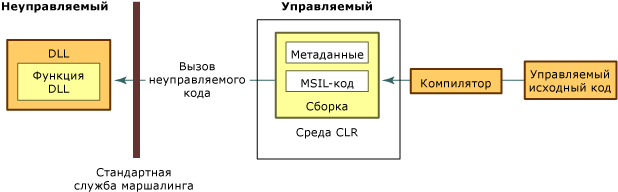


Рисунок 1.1 – Вызов неуправляемой функции DLL с помощью вызова неуправляемого кода

Когда неуправляемый код вызывает неуправляемую функцию, то он выполняет следующую последовательность действий:

1. Определяет библиотеку DLL, содержащую функцию.
2. Загружает библиотеку DLL в память.
3. Находит адрес функции в памяти и помещает ее аргументы в стек, выполнив по необходимости маршалинг данных. Поиск и загрузка библиотеки DLL, а также определение адреса функции в памяти выполняются только при первом вызове функции.
4. Передает управление неуправляемой функции.

Вызов неуправляемого кода вызывает исключения, создаваемые неуправляемой функцией для управляемого вызывающего объекта.

Ниже в таблице 1.1 перечислены типы данных, используемые в функциях Win32 API (перечислены в файле Wtypes.h) и в функциях в стиле C. Многие неуправляемые библиотеки содержат функции, передающие эти типы данных в качестве параметров и возвращаемых значений. В третьем столбце представлены соответствующие встроенные типы значений .NET Framework или классы, используемые в управляемом коде. В некоторых случаях типы, перечисленные в таблице, можно заменить на типы того же размера.

Таблица 1.1 – Соответствие типов данных

|  |  |  |  |
| --- | --- | --- | --- |
| Неуправляемый тип в Wtypes.h | Неуправляемый тип языка C | Имя управляемого класса | Описание |
| HANDLE | void\* | System.IntPtr | 32 бита в 32-разрядных операционных системах Windows, 64 бита в 64-разрядных операционных системах Windows. |
| BYTE | unsigned char | System.Byte | 8 бит |
| Short | Short | System.Int16 | 16 бит |
| WORD | unsigned short | System.UInt16 | 16 бит |
| int | int | System.Int32 | 32 бита |
| UINT | unsigned int | System.UInt32 | 32 бита |
| LONG | long | System.Int32 | 32 бита |
| BOOL | long | System.Byte | 32 бита |
| DWORD | unsigned long | System.UInt32 | 32 бита |
| ULONG | unsigned long | System.UInt32 | 32 бита |
| CHAR | char | System.Char | В кодировке ANSI. |
| WCHAR | wchar\_t | System.Char | В кодировке Юникод. |
| LPSTR | char\* | System.String или System.Text.StringBuilder | В кодировке ANSI. |
| LPCSTR | Const char\* | System.String или System.Text.StringBuilder | В кодировке ANSI. |
| Продолжение таблицы 1.1 | | | |
| Неуправляемый тип в Wtypes.h | Неуправляемый тип языка C | Имя управляемого класса | Описание |
| LPWSTR | wchar\_t\* | System.String или System.Text.StringBuilder | В кодировке Юникод. |
| LPCWSTR | Const wchar\_t\* | System.String или System.Text.StringBuilder | В кодировке Юникод. |
| float | float | System.Single | 32 бита |
| DOUBLE | Double | System.Double | 64 бита |

Ниже представлен фрагмент с объявлением функции – фактического и на языке C#.

// Фактическое объявление в WinBase.h:

HMODULE WINAPI LoadLibraryW(LPCWSTR lpLibFileName);

// Объявление на C#

class MyInteropFunctions

{

[DllImport("kernel32.dll", SetLastError = true)]

public static extern IntPtr LoadLibrary(string fileName);

}

В предыдущем фрагменте метод LoadLibrary определяется как функция, принимающая значение типа string и возвращающая значение типа IntPtr - указатель, который не может быть разыменован непосредственно, то есть, он не делает код небезопасным. Атрибут DllImport указывает, что функция экспортируется библиотекой kernel32.dll (главная библиотека Win32 API) и что последняя ошибка Win32 должна сохраняться в локальной памяти потока, чтобы она не была затерта неявными вызовами функций Win32 (например, выполняемыми средой выполнения CLR). В атрибуте DllImport можно также указать соглашение о вызове функции, кодировку для строк, параметры разрешения экспортируемых имен, и так далее.

Если сигнатура библиотечной функции содержит составные типы, такие как структуры языка C, в управляемом коде необходимо определить эквивалентные структуры или классы, используя эквивалентные типы для каждого поля. Порядок следования полей, типы полей и применяемые к ним правила выравнивания должны совпадать с ожидаемыми в коде на языке C. В некоторых случаях может потребоваться применить атрибут MarshalAs к полям в параметрах функции или в возвращаемом значении, чтобы изменить поведение механизма маршалинга по умолчанию.

Например, управляемый тип System.Boolean (bool) может иметь разные представления в низкоуровневом коде: тип Win32 bool имеет размер четыре байта, а значению true соответствует любое ненулевое значение, тогда как в C++ значение типа bool занимает один байт, а значению true соответствует целое число 1.

В следующем фрагменте демонстрируется применение атрибута StructLayout к структуре WIN32\_FIND\_DATA, определяющего расположение полей в памяти; без этого среда выполнения CLR может переупорядочить поля для большей эффективности. Атрибут MarshalAs применяется к полям cFileName и cAlternativeFileName, чтобы указать, что строки должны передаваться как строки фиксированного размера, встроенные в структуру, а не как простые указатели на внешние строки:

// Фактическое объявление в WinBase.h:

typedef struct \_WIN32\_FIND\_DATAW

{

DWORD dwFileAttributes;

FILETIME ftCreationTime;

FILETIME ftLastAccessTime;

FILETIME ftLastWriteTime;

DWORD nFileSizeHigh;

DWORD nFileSizeLow;

DWORD dwReserved0;

DWORD dwReserved1;

WCHAR cFileName[MAX\_PATH];

WCHAR cAlternateFileName[14];

}

WIN32\_FIND\_DATAW;

HANDLE WINAPI FindFirstFileW(\_\_in LPCWSTR lpFileName,

\_\_out LPWIN32\_FIND\_DATAW lpFindFileData);

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]

struct WIN32\_FIND\_DATA

{

public uint dwFileAttributes;

public FILETIME ftCreationTime;

public FILETIME ftLastAccessTime;

public FILETIME ftLastWriteTime;

public uint nFileSizeHigh;

public uint nFileSizeLow;

public uint dwReserved0;

public uint dwReserved1;

[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]

public string cFileName;

[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 14)]

public string cAlternateFileName;

}

[DllImport("kernel32.dll", CharSet = CharSet.Auto)]

static extern IntPtr FindFirstFile(string lpFileName, out WIN32\_FIND\_DATA lpFindFileData);

Когда программа вызовет метод FindFirstFile из предыдущего примера, среда выполнения CLR загрузит библиотеку DLL (kernel32.dll), найдет в ней требуемую функцию (FindFirstFile) и выполнит преобразование параметров из управляемого представления в низкоуровневое (и обратно). В данном примере, входной строковый параметр lpFileName будет преобразован в низкоуровневую строку, а низкоуровневую структуру win32\_ find\_dataw, на которую ссылается параметр lpFindFileData, - в управляемую структуру WIN32\_FIND\_DATAW. Все эти этапы более подробно будут описаны в следующих разделах.

При первом вызове функции с помощью механизма P/Invoke, обращением к функции LoadLibrary производится загрузка библиотеки DLL и всех ее зависимостей (если они еще не были загружены) в адресное пространство процесса. Затем выполняется поиск требуемой экспортируемой функции, при этом сначала предпринимается попытка найти управляемую версию функции. Порядок поиска зависит от значений полей CharSet и ExactSpelling в атрибуте DllImport:

а) если поле ExactSpelling имеет значение true, P/Invoke пытается найти функцию точно соответствующую указанному имени, учитывая только соглашения о вызове. В случае неудачи P/Invoke не выполняет поиск других версий с тем же именем и возбуждает исключение EntryPointNotFoundException;

б) если поле ExactSpelling имеет значение false, поведение механизма определяется значением свойства CharSet:

- для значения CharSet.Ansi (по умолчанию) P/Invoke сначала пытается найти точное соответствие указанному имени, а затем - измененного имени (с окончанием «A»);

- для значения CharSet.Unicode P/Invoke сначала пытается найти функцию с измененным именем (с окончанием «W»), а затем - точное соответствие.

По умолчанию поле ExactSpelling имеет значение false в C# и true в VB.NET. Во всех современных версиях Windows (выпущенных после Windows ME), значение CharSet.Auto соответствует значению CharSet.Unicode.

Используйте версии функций Win32 API для работы с Юникодом. Версии Windows NT и выше имеют встроенную поддержку Юникода (UTF16). Если пользоваться ANSI-версиями функций Win32 API, строки автоматически будут преобразовываться в Юникод и передаваться версиям функций для работы с Юникодом, что повлечет снижение производительности. Для внутреннего представления строк в .NET используется кодировка UTF16, поэтому маршалинг строковых параметров будет выполняться быстрее, если они уже будут в кодировке UTF16. Предусматривайте в своем коде и особенно в интерфейсах совместимость с Юникодом, что даст вам дополнительные преимущества с точки зрения глобализации. Устанавливайте ExactSpelling в значение true, это повысит производительность на начальном этапе за счет устранения поиска ненужных функций.

При первом вызове функции с помощью механизма P/Invoke, сразу после загрузки библиотеки DLL, заглушки (stubs) маршалера P/Invoke генерируются по требованию, и повторно используются в последующих вызовах. При вызове маршалер выполняет следующие действия.

1. Проверяет наличие у вызывающего процесса права на выполнение неуправляемого кода.
2. Преобразует управляемые аргументы в их неуправляемые аналоги, возможно с выделением памяти.
3. Устанавливает вытесняющий режим работы для потока выполнения сборщика мусора, чтобы сборка мусора могла производиться, не дожидаясь, пока поток выполнения достигнет безопасной точки.
4. Вызывает библиотечную функцию.
5. Восстанавливает кооперативный режим работы потока выполнения сборщика мусора.
6. При необходимости сохраняет код ошибки Win32 в локальной переменной потока для последующего доступа с помощью метода Marshal.GetLastWin32Error().
7. При необходимости преобразует значение типа HRESULT в исключение и возбуждает его.
8. Преобразует низкоуровневое исключение, если оно было возбуждено, в управляемое исключение.
9. Преобразует возвращаемое значение и выходные параметры обратно в их управляемые аналоги.
10. Освобождает любые временные блоки динамической памяти, выделенные при вызове.

Механизм P/Invoke может также использоваться низкоуровневым кодом для вызова управляемых методов. Обратная заглушка маршалера может быть сгенерирована для делегата (через Marshal.GetFunctionPointerForDelegate), если передается механизму P/Invoke как параметр неуправляемой функцией. В ответ неуправляемая функция получит указатель на функцию, которую можно вызвать, чтобы обратиться к управляемому методу. Этот указатель на функцию будет ссылаться на заглушку, которая помимо параметров маршалинга знает также адрес целевого объекта (указатель this).

В версии .NET Framework 1.x заглушки маршалера представляли собой либо код сгенерированной сборки (для простых сигнатур), либо код на языке ML (Marshaling Language) (для сложных сигнатур). Язык ML - это внутренний байт-код, выполняемый внутренним интерпретатором.

С появлением в .NET Framework 2.0 поддержки процессоров AMD64 и Itanium, в Microsoft обнаружили, что реализация параллельной инфраструктуры ML для каждого типа процессоров - слишком обременительное занятие. Поэтому заглушки для 64-разрядных версий .NET Framework 2.0 были реализованы исключительно на языке IL. Хотя заглушки на языке IL действуют значительно быстрее заглушек на языке ML, они все равно оказываются медленнее заглушек на языке ассемблера x86, поэтому в Microsoft было решено сохранить реализацию для архитектуры x86.

В .NET Framework 4.0, инфраструктура генерирования заглушек на языке IL была значительно оптимизирована, что сделало заглушки на языке IL даже быстрее, чем заглушки на языке ассемблера x86. Это позволило Microsoft полностью удалить реализацию заглушек для аппаратной архитектуры x86 и унифицировать их для всех поддерживаемых платформ.

Вызов функции, пересекающий границу между управляемым и неуправляемым кодом, как минимум на порядок медленнее непосредственного вызова в том же окружении. Если и управляемый, и неуправляемый код находятся в вашем ведении, конструируйте интерфейсы так, чтобы уменьшить количество взаимодействий между управляемым и неуправляемым кодом. Попробуйте объединить несколько «заданий» в один вызов. Аналогично, старайтесь объединять несколько вызов простых функций (таких как функции Get/Set) в один вызов, выполняющий всю необходимую работу.

Корпорация Microsoft предоставляет бесплатный инструмент IL Stub Diagnostics, который можно получить на сайте проекта CodePlex вместе с исходными кодами. Он подписывается на события CLR ETW IL создания заглушек и отображает сгенерированный код заглушек на языке IL в графическом интерфейсе.

Ниже приводится пример заглушки маршалера на языке IL с комментариями, состоящий из пяти разделов: инициализация, маршалинг входных параметров, вызов, маршалинг обратно возвращаемого значения и/или выходных параметров, и завершение. Заглушка маршалера генерируется для следующей сигнатуры:

// Управляемая сигнатура

[DllImport("Server.dll")]static extern int Marshal\_String\_In(string s);

// Неуправляемая сигнатура

unmanaged int \_\_stdcall Marshal\_String\_In(char \*s)

В фрагменте инициализации заглушка объявляет локальные переменные (на стеке), получает контекст заглушки и проверяет право на выполнение неуправляемого кода:

// Заглушка IL:

// Объем кода 153 (0x0099)

.maxstack 3

// Локальные переменные:

// IsSuccessful, pNativeStrPtr, SizeInBytes, pStackAllocPtr, result, result, result

.locals (int32,native int,int32,native int,int32,int32,int32)

call native int [mscorlib] System.StubHelpers.StubHelpers::GetStubContext()

// Проверка права на выполнение неуправляемого кода

call void [mscorlib] System.StubHelpers.StubHelpers::DemandPermission(native int)

В фрагменте маршалинга заглушка передает входные параметры неуправляемой функции. В данном примере выполняется маршалинг единственного входного строкового параметра. Для преобразования конкретных типов и их категорий из управляемого представления в неуправляемое, и обратно, маршалер может использовать вспомогательные типы в пространстве имен System.StubHelpersnamespace или обращаться к классу System.Runtime.InteropServices.Marshal. В данном примере для маршалинга строки вызывается метод CSTRMarshaler::ConvertToNative.

Здесь используется небольшая оптимизация: если управляемая строка достаточно короткая, память для нее выделяется на стеке (что намного быстрее). В противном случае для нее выделяется блок в динамической памяти:

ldc.i4 0x0 // IsSuccessful = 0 [положить 0 на стек]

stloc.0 // [сохранить в IsSuccessful]

IL\_0010:

nop // аргумент {

ldc.i4 0x0 // pNativeStrPtr = null [положить 0 на стек]

conv.i // [преобразовать int32 в "неуправляемый int"

// (указатель)]

stloc.3 // [сохранить результат в pNativeStrPtr]

ldarg.0 // if (managedString == null)

brfalse IL\_0042 // goto IL\_0042

ldarg.0 // [экземпляр managedString положить на стек]

// обратиться к свойству Length (получить число

// символов)

call instance int32 [mscorlib] System.String::get\_Length()

ldc.i4 0x2 // Прибавить 2 к длине, 1 - для нулевого символа

// в managedString, и 1 - для дополнительного

// нулевого символа

// [положить константу 2 на стек]

add // [фактическое сложение, результат - на стеке]

// загрузить статическое поле, значение зависит

// от lang. для приложений, не поддерживающих

// Юникод

ldsfld System.Runtime.InteropServices.Marshal::SystemMaxDBCSCharSize

mul // Умножить длину на SystemMaxDBCSCharSize чтобы

// получить количество байтов

stloc.2 // Сохранить в SizeInBytes

ldc.i4 0x105 // Сравнить SizeInBytes и 0x105, чтобы исключить

// возможность выделения слишком большого

// количества памяти на стеке [константу 0x105

// на стек] CSTRMarshaler::ConvertToNative для

// случая pStackAllocPtr == null и вызовет

// CoTaskMemAlloc для выделения большего объема

// памяти

ldloc.2 // [Положить SizeInBytes]

clt // [If SizeInBytes > 0x105, положить 1 иначе 0]

brtrue IL\_0042 // [If 1 goto IL\_0042]

ldloc.2 // Положить SizeInBytes (аргумент для localloc)

localloc // Выделить память на стеке, указатель оставить

// на стеке

stloc.3 // Сохранить в pStackAllocPtr

IL\_0042:

ldc.i4 0x1 // константу 1 на стек (параметр flags)

ldarg.0 // аргумент managedString на стек

ldloc.3 // pStackAllocPtr на стек (this может быть null)

// Вызвать функцию преобразования Unicode в ANSI

call native int [mscorlib]System.StubHelpers.

CSTRMarshaler::ConvertToNative(int32,string, native int)

stloc.1 // Сохранить результат в pNativeStrPtr,

// может быть равен pStackAllocPtr

ldc.i4 0x1 // IsSuccessful = 1 [положить 1 на стек]

stloc.0 // [сохранить в IsSuccessful]

nop

nop

nop

В следующем фрагменте заглушка получает из своего контекста указатель на неуправляемую функцию и вызывает ее. Инструкция вызова в действительности делает больше, чем кажется. Например, она изменяет режим работы сборщика мусора, перехватывает значение, возвращаемое неуправляемой функцией, и приостанавливает выполнение управляемого кода, пока сборщик мусора выполняет фазу, требующую этого:

ldloc.1 // Положить pStackAllocPtr на стек, для

// пользовательской функции, не для

// GetStubContext

call native int [mscorlib] System.StubHelpers.StubHelpers::GetStubContext()

ldc.i4 0x14 // Добавить 0x14 к указателю контекста

add // [фактическое сложение, результат на стеке

ldind.i // [разыменовать указатель, результат на стеке]

ldind.i // [разыменовать указатель на функцию, результат

// на стеке]

calli unmanaged stdcall int32(native int)

// Вызвать пользовательскую функцию

Следующий фрагмент фактически состоит из двух фрагментов, выполняющих обратное преобразование неуправляемых типов в управляемые для возвращаемого значения и выходных параметров, соответственно. В данном примере, неуправляемая функция возвращает значение типа int, не требующее преобразования (маршалинга), которое просто копируется в локальную переменную в исходном виде. Поскольку здесь нет выходных параметров, последний раздел остался пустым:

// UnmarshalReturn {

nop // возврат {

stloc.s 0x5 // Сохранить результат

// функции (int) в x, y и z

ldloc.s 0x5

stloc.s 0x4

ldloc.s 0x4

nop // } возврат

stloc.s 0x6

// } UnmarshalReturn

// Unmarshal {

nop // аргумент {

nop // } аргумент

leave IL\_007e // Выход из блока try

IL\_007e:

ldloc.s 0x6 // Поместить z на стек

ret // Вернуть z

// } Unmarshal

Наконец, в фрагменте завершения освобождается память, выделенная временно для нужд маршалинга. Эти операции выполняются в блоке finally инструкции try, поэтому они будут выполнены, даже если неуправляемая функция возбудит исключение. Имеется также возможность выполнять некоторые действия только в случае исключения. Во взаимодействиях с COM-объектами заглушки могут преобразовывать возвращаемое значение HRESULT, указывающее на ошибку, в исключение:

// ExceptionCleanup {

IL\_0081:

// } ExceptionCleanup

// Cleanup {

ldloc.0 // If (IsSuccessful && !pStackAllocPtr)

ldc.i4 0x0 // Вызвать ClearNative(pNativeStrPtr)

ble IL\_0098

ldloc.3

brtrue IL\_0098

ldloc.1

call void [mscorlib]

System.StubHelpers.CSTRMarshaler::ClearNative(native int)

IL\_0098:

endfinally

IL\_0099:

// } Cleanup

.try IL\_0010 to IL\_007e finally handler IL\_0081 to IL\_0099

Как видно, заглушка маршалера на языке IL имеет замысловатую организацию даже для функций с такой простой сигнатурой. Для сложных сигнатур генерируются еще более длинные и медленные заглушки.

Многие неуправляемые типы двоично совместимы с управляемым кодом. Эти типы, называемые двоично совместимыми (blittable), не требуют преобразования и передаются через границу между управляемым и неуправляемым кодом намного быстрее, чем двоично несовместимые (non-blittable) типы. В действительности заглушка маршалера может оптимизировать такую передачу еще больше, закрепляя управляемый объект и передавая неуправляемому коду указатель на него, исключая одну или две операции копирования (по одной для каждого направления передачи).

К двоично совместимым типам относятся:

System.Byte (byte)

System.SByte (sbyte)

System.Int16 (short)

System.UInt16 (ushort)

System.Int32(int)

System.UInt32 (uint)

System.Int64 (long)

System.UInt64 (ulong)

System.IntPtr

System.UIntPtr

System.Single (float)

System.Double (double)

Кроме того, одномерные массивы с элементами двоично совместимых типов (где все элементы принадлежат одному и тому же типу) также являются двоично совместимыми, так же как структуры или классы, состоящие только из полей двоично совместимых типов.

Тип System.Boolean (bool) не является двоично совместимым, потому что в неуправляемом коде он может иметь 1, 2 или 4-байтное представление. Тип System.Char (char) не является двоично совместимым, потому что может представлять символ ANSI или Юникода. Тип System.String (string) не является двоично совместимым, потому что его неуправляемое представление может состоять из символов ANSI или Юникода, и может быть строкой в стиле языка C или COM BSTR, а также потому, что управляемая строка должна быть неизменяемой.

Тип, содержащий поле со ссылкой на объект, не может быть двоично совместимым, даже если это ссылка на двоично совместимый тип или массив. При маршалинге двоично несовместимых типов требуется выделять память для сохранения преобразованных версий параметров, ее заполнения и последующего освобождения.

Наивысшей производительности можно добиться, реализовав маршалинг входных строковых параметров вручную (см. следующий фрагмент кода). Но при этом вызываемая неуправляемая функция должна принимать строку в кодировке UTF-16, в стиле языка C, и никогда не писать в память, занимаемую строкой, из-за чего такая оптимизация редко бывает применима. Чтобы выполнить маршалинг вручную, необходимо закрепить входную строку, изменить сигнатуру P/Invoke так, чтобы неуправляемая функция выглядела, как принимающая IntPtr вместо String, и передавать ей указатель на закрепленную строку:

class Win32Interop

{

[DllImport("NativeDLL.DLL", CallingConvention = CallingConvention.Cdecl)]

// IntPtr вместо string

public static extern void NativeFunc(IntPtr pStr);

}

// ...

// Управляемый код вызывает функцию P/Invoke внутри области видимости

// fixed, что обеспечивает закрепление строки

unsafe

{

string str = "MyString";

fixed (char\* pStr = str)

{

// pStr можно использовать в нескольких вызовах

Win32Interop.NativeFunc((IntPtr)pStr);

}

}

Преобразование неуправляемой строки UTF-16 в стиле языка C в управляемую строку также можно оптимизировать, применив конструктор System.String, принимающий параметр типа char\*. Конструктор System.String создает копию буфера, поэтому неуправляемую память, занимаемую строкой, можно освободить сразу после создания управляемой строки. Обратите внимание, что здесь не выполняется никакой проверки, чтобы убедиться, что строка содержит только допустимые символы Юникода.

Направление маршалинга, ссылочные типы и типы значений

Как упоминалось выше, параметры функции могут передаваться заглушкой маршалера в одном или в двух направлениях. Направление передачи параметра определяется рядом факторов:

- является ли параметр ссылочным типом или типом значения;

- передается ли параметр по ссылке или по значению;

- является ли тип параметра двоично совместимым или нет;

- применяются ли к параметрам атрибуты, изменяющие направление маршалинга (System.RuntimeInteropService.InAttribute и System.RuntimeInteropService.OutAttribute).

Для дальнейшего обсуждения будем обозначать направление маршалинга из управляемого кода в неуправляемый как «in»; а направление из неуправляемого кода в управляемый - как «out». Ниже приводится список правил определения направления маршалинга по умолчанию:

- параметры, передаваемые по значению, независимо от того, являются ли они ссылочными типами или типами значений, передаются только в направлении «in» (не требуется применять атрибут In вручную, StringBuilder является исключением из правила и всегда передается в направлении «in/out»);

- параметры, передаваемые по ссылке (с применением ключевого слова ref в C# или ByRef в VB.NET), независимо от того, являются ли они ссылочными типами или типами значений, передаются в направлении «in/out».

Добавление атрибута OutAttribute запрещает маршалинг в направлении «in», поэтому вызываемый неуправляемый код может не получить значение, переданное вызывающим кодом. Ключевое слово out в языке C# действует подобно ключевому слову ref, но добавляет атрибут OutAttribute.

Если типы параметров не являются двоично совместимыми в вызове P/Invoke и вам требуется организовать маршалинг только в направлении «out», ненужного маршалинга в направлении «in» можно избежать, использовав ключевое слово out вместо ref.

Из-за закрепления параметров двоично совместимых типов при маршалинге, как описывалось выше, для двоично совместимых ссылочных типов автоматически устанавливается направление «in/out», даже если правила выше утверждают иное. Однако не следует полагаться на эту особенность, когда требуется получить поведение «out» или «in/out» маршалинга, а вместо этого указывать направление явно, с помощью атрибутов, так как данная особенность перестанет действовать, если позднее вы добавите поле двоично несовместимого типа или если это вызов COM-объекта, пересекающий границы подразделений (apartments).

Разница между маршалингом типов значений и ссылочных типов заключается в особенностях их передачи через стек:

- типы значений, которые передаются по значению, копируются на стек, поэтому они всегда передаются в направлении «in», независимо от используемых атрибутов;

- типы значений, которые передаются по ссылке, и ссылочные типы, которые передаются по значению, передаются как указатель;

- ссылочные типы, которые передаются по ссылке, передаются как указатель на указатель.

Передача объемных типов значений (более десятка байт) по значению стоит дороже передачи их по ссылке. То же относится к объемным возвращаемым значениям, вместо которых может оказаться предпочтительнее использовать выходные параметры.

**Code Access Security**

Механизм .NET Code Access Security позволяет выполнять код, не вызывающий доверия, в изолированном окружении, называемом «песочницей» (sandbox), с ограниченным доступом к возможностям среды выполнения (например, P/Invoke) и BCL (например, доступ к файлам и реестру). Когда вызывается неуправляемый код, механизм Code Access Security требует, чтобы все сборки, чьи методы будут вызываться, имели право UnmanagedCode. Заглушка маршалера будет проверять это право для каждого вызова, что влечет за собой обход стека вызовов, чтобы убедиться, что весь код в цепочке вызовов обладает данным правом.

Если вы выполняете только код, пользующийся доверием, или у вас имеются иные средства, гарантирующие безопасность, вы можете значительно увеличить производительность, добавив атрибут SuppressUnmanagedCodeSecurityAttribute в объявление метода, класса (в этом случае данный атрибут применяется ко всем методам), интерфейса или делегата.

## Интеграция ngspice.dll в .NET C#

Исходный код API Ngspice содержится в файле sharedspice.c и в соответствующем файле заголовка sharedspice.h. Все объявления типов и функций содержатся в файле заголовка, полный листинг которого приведен в приложении А. Ниже приведенная таблица 2.1 содержит объявления структур и функций в файле заголовка и полученные их прототипы в управляемом коде.

Таблица 2.1 – Таблица маршалинга

|  |  |
| --- | --- |
| sharedspice.h | C# |
| struct ngcomplex {  double cx\_real;  double cx\_imag;  }; | [StructLayout(LayoutKind.Sequential)]  public struct ngcomplex  {  public double cx\_real;  public double cx\_imag;  } |
| typedef struct vector\_info {  char \*v\_name;  int v\_type;  short v\_flags;  double \*v\_realdata;  ngcomplex\_t \*v\_compdata;  int v\_length;  } vector\_info, \*pvector\_info; | [StructLayout(LayoutKind.Sequential)]  public struct vector\_info  {  /// char\*  [MarshalAs(UnmanagedType.LPStr)]  public string v\_name;  /// int  public int v\_type;  /// short  public short v\_flags;  /// double\*  public IntPtr v\_realdata;  /// ngcomplex\_t\*  public IntPtr v\_compdata;  /// int  public int v\_length;  } |
| typedef struct vecvalues {  char\* name;  double creal;  bool is\_scale;  bool is\_complex;  } vecvalues, \*pvecvalues; | [StructLayout(LayoutKind.Sequential)]  public struct vecvalues  {  /// char\*  [MarshalAs(UnmanagedType.LPStr)]  public string name;  /// double  public double creal;  /// double  public double cimag;  /// boolean  [MarshalAs(UnmanagedType.U1)]  public bool is\_scale;  /// boolean  [MarshalAs(UnmanagedType.U1)]  public bool is\_complex;  } |

|  |  |
| --- | --- |
| Продолжение таблицы 2.1 | |
| sharedspice.h | C# |
| typedef struct vecvaluesall {  int veccount;  int vecindex;  pvecvalues \*vecsa;  } vecvaluesall, \*pvecvaluesall; | [StructLayout(LayoutKind.Sequential)]  public struct vecvaluesall  {  /// int  public int veccount;  /// int  public int vecindex;  /// pvecvalues\*  public IntPtr vecsa;  } |
| typedef struct vecinfo  {  int number;  char \*vecname;  bool is\_real;  void \*pdvec;  void \*pdvecscale;  } vecinfo, \*pvecinfo; | [StructLayout(LayoutKind.Sequential)]  public struct vecinfo  {  /// int  public int number;  /// char\*  [MarshalAs(UnmanagedType.LPStr)]  public string vecname;  /// boolean  public bool is\_real;  /// void\*  public IntPtr pdvec;  /// void\*  public IntPtr pdvecscale;  } |
| typedef struct vecinfoall  {  char \*name;  char \*title;  char \*date;  char \*type;  int veccount;  pvecinfo \*vecs;  } vecinfoall, \*pvecinfoall; | [StructLayout(LayoutKind.Sequential)]  public struct vecinfoall  {  /// char\*  [MarshalAs(UnmanagedType.LPStr)]  public string name;  /// char\*  [MarshalAs(UnmanagedType.LPStr)]  public string title;  /// char\*  [MarshalAs(UnmanagedType.LPStr)]  public string date;  /// char\*  [MarshalAs(UnmanagedType.LPStr)]  public string type;  /// int  public int veccount;  /// pvecinfo\*  public IntPtr vecs;  } |
| typedef int (SendChar)(char\*, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int SendChar(IntPtr param0, int param1, IntPtr param2); |

|  |  |
| --- | --- |
| Продолжение таблицы 2.1 | |
| sharedspice.h | C# |
| typedef int (SendStat)(char\*, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int SendStat(IntPtr param0, int param1, IntPtr param2); |
| typedef int (ControlledExit)(int, bool, bool, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int ControlledExit(int param0, [MarshalAs(UnmanagedType.I1)] bool param1, [MarshalAs(UnmanagedType.I1)] bool param2, int param3, IntPtr param4); |
| typedef int (SendData)(pvecvaluesall, int, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int SendInitData(IntPtr param0, int param1, IntPtr param2); |
| typedef int (SendInitData)(pvecinfoall, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int BGThreadRunning([MarshalAs(UnmanagedType.I1)] bool param0, int param1, IntPtr param2); |
| typedef int (BGThreadRunning)(bool, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int BGThreadRunning([MarshalAs(UnmanagedType.I1)] bool param0, int param1, IntPtr param2); |
| typedef int (GetVSRCData)(double\*, double, char\*, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int GetVSRCData(IntPtr param0, double param1, IntPtr param2, int param3, IntPtr param4); |
| typedef int (GetISRCData)(double\*, double, char\*, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int GetISRCData(IntPtr param0, double param1, IntPtr param2, int param3, IntPtr param4); |
| typedef int (GetSyncData)(double, double\*, double, int, int, int, void\*); | [UnmanagedFunctionPointer(CallingConvention.Cdecl)]  public delegate int GetSyncData(double param0, IntPtr param1, double param2, int param3, int param4, int param5, IntPtr param6); |
| int ngSpice\_Init(  SendChar\* printfcn, SendStat\* statfcn, ControlledExit\* ngexit, SendData\* sdata, SendInitData\* sinitdata, BGThreadRunning\* bgtrun, void\* userData); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_Init",  SetLastError = true,  CallingConvention = CallingConvention.Cdecl)]  public static extern int ngSpice\_Init(IntPtr printfcn, IntPtr statfcn, IntPtr ngexit, IntPtr sdata, IntPtr sinitdata, IntPtr bgtrun, IntPtr userData); |
| int ngSpice\_Init\_Sync(GetVSRCData \*vsrcdat, GetISRCData \*isrcdat, GetSyncData \*syncdat, int \*ident, void \*userData); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_Init\_Sync",  CallingConvention = CallingConvention.Cdecl)]  public static extern int ngSpice\_Init\_Sync(IntPtr vsrcdat, IntPtr isrcdat, IntPtr syncdat, IntPtr ident, IntPtr userData); |

|  |  |
| --- | --- |
| Продолжение таблицы 2.1 | |
| sharedspice.h | C# |
| int ngSpice\_Command(char\* command); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_Command",  CallingConvention = CallingConvention.Cdecl)]  public static extern int ngSpice\_Command(  [MarshalAs(UnmanagedType.LPStr)] string command); |
| pvector\_info ngGet\_Vec\_Info(char\* vecname); | [DllImport("ngspice.dll",  EntryPoint = "ngGet\_Vec\_Info",  CallingConvention = CallingConvention.Cdecl)]  public static extern IntPtr ngGet\_Vec\_Info( [MarshalAs(UnmanagedType.LPStr)] string vecname); |
| int ngSpice\_Circ(char\*\* circarray); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_Circ",  CallingConvention = CallingConvention.Cdecl)]  public static extern int ngSpice\_Circ(IntPtr circarray); |
| char\* ngSpice\_CurPlot(void); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_CurPlot",  CallingConvention = CallingConvention.Cdecl)]  public static extern IntPtr ngSpice\_CurPlot(); |
| char\*\* ngSpice\_AllPlots(void); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_AllPlots",  CallingConvention = CallingConvention.Cdecl)]  public static extern IntPtr ngSpice\_AllPlots(); |
| char\*\* ngSpice\_AllVecs(char\* plotname); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_AllVecs",  CallingConvention = CallingConvention.Cdecl)]  public static extern IntPtr ngSpice\_AllVecs(string plotname); |
| bool ngSpice\_running(void); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_running",  CallingConvention = CallingConvention.Cdecl)]  [return: MarshalAs(UnmanagedType.I1)]  public static extern bool ngSpice\_running(); |
| bool ngSpice\_SetBkpt(double time); | [DllImport("ngspice.dll",  EntryPoint = "ngSpice\_SetBkpt",  CallingConvention = CallingConvention.Cdecl)]  [return: MarshalAs(UnmanagedType.I1)]  public static extern bool ngSpice\_SetBkpt(double time); |

**Выводы**

Для создания хороших обёрток над нативными библиотеками нужно обладать определёнными качествами и возможностями. Прежде всего, требуются хорошие способности в разработке архитектуры. Отображение нативной библиотеки один-к-одному редко является хорошим решением. Отбросив возможности, которые не нужны потребителям библиотеки, можно колоссально упростить задачу. Создание фасада для оставшихся возможностей библиотеки требует знания CLS, возможностей FCL и часто используемых в .NET подходов. Создание обёртки над нативной библиотекой также предполагает обёртывание нативных ресурсов (например, неуправляемой памяти, необходимой для создания оборачиваемых объектов).

Для интеграции API Ngspice был выбран второй вариант – механизм Platform Invoke как наименее затратный по времени. Возможностей P/Invoke хватает для обеспечения взаимодействия с неуправляемым кодом библиотеки.

В данном разделе был сделан маршалинг API структур и методов Ngspice. Данные объявления можно использовать в .NET C# проектах и вызывать методы из неуправляемого кода в проекте с управляемым кодом.

# Разработка архитектуры модуля

[uml-диаграммы]

# Заключение

# Сокращения, обозначения, термины и определения

# Список использованных источников

1. Норенков И. П. Основы автоматизированного проектирования: учеб. для вузов / И. П. Норенков. – 4-е изд., перераб. и доп. – М. : Изд-во МГТУ им Н. Э. Баумана, 2009. – 430 с.
2. Трифоненко И. М., Горячев Н. В., Кочегаров И. И., Юрков Н. К. Обзор систем сквозного проектирования печатных плат радиоэлектронных средств [Электронный ресурс] // КиберЛенинка. – 2012. – URL: <http://cyberleninka.ru/article/n/obzor-sistem-skvoznogo-proektirovaniya-pechatnyh-plat-radioelektronnyh-sredstv> (дата обращения 03.05.17)
3. Сальникова Н. А., Астафурова О. А. Методы моделирования в системах автоматизированного проектирования СВЧ-устройств [Электронный ресурс] // КиберЛенинка. – 2014. – URL: <http://cyberleninka.ru/article/n/metody-modelirovaniya-v-sistemah-avtomatizirovannogo-proektirovaniya-svch-ustroystv> (дата обращения 03.05.17)
4. Leon O. Chua, Pen-Min Lin. Computer-aided analysis of electronic circuits / Перевод с английского Е.В Виленкина, В.Н, Елисеева и др. – М.: «Энергия», 1980.
5. Атабеков Г. И. Теоретические основы электротехники. Линейные электрические цепи: учебное пособие / Г.И. Атабеков. – Спб.: Издательство «Лань», 2009. – 592 с.
6. Решение ТОЭ ОТЦ ТЛЭЦ Электротехника [Электронный ресурс]. – URL: <http://rgr-toe.ru> (дата обращения 23.04.17)
7. Getting Started Guide. Chapter 4. MWO: Using the Linear Simulator [Электронный ресурс]. – URL: [awrcorp.com/download/faq/english/docs/Getting\_Started/ch04.html](https://awrcorp.com/download/faq/english/docs/Getting_Started/ch04.html) (дата обращения 23.04.17)
8. Circuit Simulation [Электронный ресурс]. – URL: [literature.cdn.keysight.com/litweb/pdf/ads2003c/cktsim/index.html](http://literature.cdn.keysight.com/litweb/pdf/ads2003c/cktsim/index.html) (дата обращения 23.04.17)
9. Черкашин М.В. Модели и методы анализа проектных решений: учеб. пособие / М.В. Черкашин. – Томск: Томск. гос. ун-т систем упр. и радиоэлектроники, 2012. – 296 с.
10. SPICE (симулятор электронных схем) [Электронный ресурс]. – URL: <https://en.wikipedia.org/wiki/SPICE> (дата обращения 27.04.17)
11. Сайт ПАЯЛЬНИК [Электронный ресурс]. – URL: <http://cxem.net/> (дата обращения 27.04.17)
12. Ngspice – Википедия [Электронный ресурс]. – URL: <https://ru.wikipedia.org/wiki/Ngspice> (дата обращения 27.04.17)
13. Qucs – Википедия [Электронный ресурс]. – URL: <https://ru.wikipedia.org/wiki/Qucs> (дата обращения 04.05.17)
14. HSPICE [Электронный ресурс]. – URL: <http://dic.academic.ru/dic.nsf/ruwiki/1700378> (дата обращения 04.05.17)

# Приложение А

(справочное)

**Описание типов лицензий**

Лицензия **BSD** (англ. BSD license, Berkeley Software Distribution license – Программная лицензия университета Беркли) – это лицензионное соглашение, впервые применённое для распространения UNIX-подобных операционных систем BSD. Права на исходный дистрибутив BSD официально принадлежат «попечителям университета Калифорнии» (англ. Regents of the University of California) – управляющему органу университета Калифорнии.

По сравнению с другими распространёнными лицензиями на свободное программное обеспечение (например, GNU General Public License) лицензия BSD налагает меньше ограничений на пользователя. Лицензия BSD допускает проприетарное коммерческое использование ПО. Для ПО, выпущенного под этой лицензией, допускается встраивание в проприетарные коммерческие продукты. Работы, основанные на таком ПО, даже могут распространяться под проприетарными лицензиями (но всё же обязаны соответствовать требованиям лицензии). Можно применять к распространяемому продукту одновременно лицензию BSD и какую-то другую.

Бесплатное программное обеспечение (англ. **freeware**) – программное обеспечение, лицензионное соглашение которого не требует каких-либо выплат правообладателю. Бесплатное программное обеспечение обычно распространяется в бинарном виде, без исходных кодов и является проприетарным программным обеспечением.

GNU **General Public License** – лицензия на свободное программное обеспечение, созданная в рамках проекта GNU в 1988 г., по которой автор передаёт программное обеспечение в общественную собственность. Цель GNU GPL – предоставить пользователю права копировать, модифицировать и распространять (в том числе на коммерческой основе) программы, а также гарантировать, что и пользователи всех производных программ получат вышеперечисленные права. GNU GPL не позволяет включать программу в проприетарное ПО.

GPL предоставляет получателям компьютерных программ следующие права, или «свободы»:

- свободу запуска программы с любой целью;

- свободу изучения того, как программа работает, и её модификации (предварительным условием для этого является доступ к исходному коду);

- свободу распространения копий как исходного, так и исполняемого кода;

- свободу улучшения программы, и выпуска улучшений в публичный доступ (предварительным условием для этого является доступ к исходному коду).

В общем случае распространитель программы, полученной на условиях GPL, либо программы, основанной на таковой, обязан предоставить получателю возможность получить соответствующий исходный код.