МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта
Кафедра автоматических систем

Система менеджмента качества обучения
СМКО МИРЭА 7.3/04. 3Дпр.5КУБ/О/220400.62-16

КУРСОВАЯ РАБОТА
по дисциплине «Сети и системы передачи информации»

ТЕМА: «___Манчестерский код_____»
(название темы)

Руководитель работы к.т.н.
старший преподаватель _____ ___Новоженин М. Б._____
(подпись) (Ф.И.О.)

Студент _____ ___Никишина А. А._____
(подпись) (Ф.И.О.)

Группа _ККСО-03-19_, шифр __19К0603__

МОСКВА 2022 г.

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА-Российский технологический университет»
РТУ МИРЭА

Институт кибернетики
Кафедра автоматических систем

УТВЕРЖДАЮ:
Заведующий кафедрой автоматических систем

_____
(Ф.И.О.)
«____» _____20__ г.

ЗАДАНИЕ
на выполнение курсовой работы по дисциплине
«Сети и системы передачи информации»

Студент: __Никишина Анна Александровна_____
(Ф.И.О.)
Группа _ККСО-03-19_, шифр __19К0603_____

Тема: «___Манчестерский код_____»
(название темы)

Перечень вопросов, подлежащих разработке:
- анализ принципов кодирования/декодирования;
- выбор и анализ алгоритмов работы кодера/декодера;
- разработка и описание аппаратной (программной) реализации кодера;
- разработка и описание аппаратной (программной) реализации декодера;
- разработка и описание примера кодирования/декодирования;
- оценка эффективности метода;
- анализ преимуществ и недостатков метода.

Срок представления к защите курсовой работы до «___»_____20__ г.

Задание на курсовую работу выдал _____ _____
(подпись) (Ф.И.О.)
«___»_____20___ г.
Задание на курсовую работу получил _____ _____
(подпись) (Ф.И.О.)
«___»_____20___ г.

Содержание

Введение.

Потребность кодировать информацию, т.е. преобразовывать ее для каких-либо целей, люди испытывали и в доцифровую эпоху. Даже наша письменность также является, способом кодировать сообщения для долговременного хранения, публикации или отправки адресату. То же применимо к цифрам, используемым для вычислений, записи дат. Музыка уже несколько веков записывается с помощью нот.

С развитием человечества в целом и данной сферы в частности появились специальные системы кодирования для передачи информации такие как азбука Морзе, азбука Брайля для незрячих, азбука жестов для глухонемых. По мере развития науки и техники в каждой отрасли человеческой деятельности появились свои сленги, профессиональные языки (ноты, формулы, языки программирования), которые помогают коллегам лучше понимать друг друга и унифицировать произведения их труда.

С цифровизацией кодирование информации также унифицировалось: все традиционные знаковые системы были оцифрованы и теперь кодирование для них производится посредством электрических импульсов внутри вычислительных устройств.

Сегодня кодирование – это своего рода способ перевода человеческого искусства на язык машин. К примеру, мы можем сфотографировать картину, скачать электронную книгу или послушать музыку в плеере.

В частности, Манчестерское кодирование применяется в технологии Ethernet, в стандартах сетевых протоколов Token Ring, в протоколах управления различными устройствами по инфракрасному каналу. Что делает его изучение актуальным на нынешний момент.

Целью моей курсовой работы является изучение Манчестерского кодирования и его программная реализация.
Для достижения данной цели следует выполнить следующие задачи:

1. Поиск источников и сбор информации.
2. Изучение алгоритма манчестерского кодирования.
3. Исследование его сферы применения.
4. Практическая реализация алгоритма.
5. Обобщение результатов проделанной работы.

Глава 1. Теоретическая часть.

В локальных сетях Манчестерский код характеризуется как весьма популярный. Данные передаются, когда между половинами такта возникают перепады потенциала.
При кодировании нуля происходит переход от высокого уровня к низкому, а при кодировании единицы, соответственно возникает обратная ситуация. Служебный перепад происходит в начале каждого такта в том и только в том случае, когда в текущем такте передаваемый бит имеет то же значение, что и в прошлом. Манчестерский код обеспечивает изменение уровня сигнала при представлении каждого бита. В то время как при передаче серий одноимённых битов происходит двойное изменение. Помимо этого, манчестерский код не имеет постоянной составляющей в спектре и обеспечивает наилучшие синхронизирующие свойства.

К особенностям манчестерского кода, в первую очередь, я бы отнесла следующие факты:

1. Из способа кодирования следует, что, вне зависимости от длинны исходной последовательности битов, в закодированной последовательности будет одинаковое количество нулей и единиц. Это делает возможным использование гальванически развязанных элементов для передачи.
2. Не менее важным свойством, на мой взгляд, является возможность самостоятельной синхронизации приемника. Проще говоря, приёмник не нуждается ни в информации о периоде следования битов в исходной последовательности, ни в дополнительной линии с тактовым сигналом.

Кодирование.
Кодирование манчестерского кода можно назвать относительно простым — оно производится применением логической операции «ИСКЛЮЧАЮЩЕЕ ИЛИ» (иначе: сложение по модулю два или XOR). Операция совершается над текущим кодируемым битом и битом тактового генератора.
При программной реализации с помощью программного таймера создаётся виртуальный тактовый генератор. При чём длительность выдержки таймера составляет половину периода.

Декодирование.
При декодировании существуют два варианта развития событий:

- На приёмном конце (т.е. декодировщику) известна длительность периода кодирования.
- И второй — декодировщику не известен период кодирования.

Случай 1.
В первом случае при декодировании вначале определяется середина периода – это происходит путём битовой синхронизации. Для этого декодировщик следит за спадами сигнала и фронтами. Признаком последовательно идущих 0 и 1 или 1 и 0 в потоке данных являются, разделенные одним периодом, два соседних

перепада с противоположными направлениями. Здесь есть небольшой аспект, заключающийся в том, что если первый из этих перепадов нарастающий, а второй — падающий, то мы можем сделать вывод о том, что в потоке данных это комбинация двух битов 1 и 0 или 0 и 1, случае когда мы наблюдаем убывание— нарастание. А также второй из этих перепадов указывает на середину периода кодирования. В двоичный поток декодированных данных записывается данная комбинация из двух битов. Затем декодировщик выжидает время примерно равное 3/4 периода и начинает следить за появлением перепада, при возникновении перепада, в том случае если это спад, это говорит о том, что очередной бит данных 0, и 1 — фронте. Одновременно по этому же перепаду осуществляется перезапуск таймера выдержки 3/4 периода и далее процесс повторяется.

Для приёма потоков битов данных недостаточно одной только лишь битовой синхронизации. Это всё из-за того, что во многих случаях в потоке однообразных битов нам не известно начало информационного блока. В качестве примера можно привести считывание данных с магнитного дискового накопителя. Из-за этого также применяется синхронизация по слову синхронизации.

При кодировании кодировщик вставляет бинарное слово синхронизации в начало блока данных. В некоторых случаях вставляется сигнатура синхронизации, при условии, что она известна декодировщику. Чтобы начать синхронизацию блока, декодировщик постоянно производит сравнение последовательности битов в сдвиговом регистре после каждого сдвига. Этот регистр имеет длину равную длине в которую, с заданным словом синхронизации, на каждом периоде сдвигается очередной бит. Признаком начала блока данных является совпадение сигнатуры или её инверсии со словом в регистре сдвига. После обнаружения этого совпадения декодировщик интерпретирует последующий двоичный поток данных как информационные. В разных системах длинна слова синхронизации может отличаться, но как правило она составляет не менее двух байтов. К примеру, в протоколе Ethernet эта длина принята равной 56 бит.

В потоке данных вероятность встречаемости слова синхронизации должна быть пренебрежимо мала, а в идеальном случае оно не должно встречаться вовсе. Поэтому чем больше длинна сигнатуры, тем лучше, так как понижается вероятность её существования в потоке данных. Но, с другой стороны, чрезмерно длинная сигнатура – это тоже плохо, ведь она не несёт полезной информации и снижает скорость передачи данных. Для разрешения этого противоречия в том случае, если сигнатура короткая, производится проверка потока данных перед кодированием на случайное наличие последовательности бит, которая совпадала бы с сигнатурой. В случае обнаружения такой коллизии блок перекодируется, чтобы исключить сигнатуру, или же при появлении синхронизации по началу информационного блока, до окончания приёма всего

блока данных, декодировщик не обращает внимания на попадающиеся сигнатуры.

Случай 2.
При изначально неизвестном периоде кодирования, в первую очередь декодировщик измеряет данный временной промежуток между соседними перепадами. В силу временны́х колебаний моментов перепадов в реальных сигналах из-за помех, джиттера, нерегулярных программных задержек в программном кодировщике и прочих причин, измерение лишь нескольких пар соседних перепадов становится недостаточным. В связи с этим, требуется накопление некоторой статистики по информационному потоку при измерении половины периода и периода.

В правильном коде, при условии не слишком большого джиттера, гистограмма измерений будет представлена двумя группами выборок. Заметим, что чем меньше джиттер, тем, соответственно, уже группы на гистограмме. Длительности полупериода отвечает первая группа, в то время как вторая отвечает целому периоду. Исходя из результатов статистических рассчётов по данной гистограмме, выявляется период и исходя из его значения настраивается интервальный таймер декодировщика или его внутренний тактовый генератор.

Вычисление оценки периода по результатам измерений периодов $T_{ср}$ и при необходимости половины периода $T_{ср1/2}$ производится по формуле среднего арифметического взвешенного.

$$T_{ср} = \frac{\sum_{i=1}^{K}(T_{min} + i\Delta T)n_i}{N}$$

где K — число интервалов по времени выборок интервалов гистограммы (карманов),
Δ T — ширина интервала,
N — общее число выборок
n_i — число выборок попавшее в i-й интервал.

K и Δ T выбираются так, чтобы T_{min} + K Δ T = T_{max} , например, по рисунку $T_{min} = 30$мкс и $T_{max} = 50$ мкс.

Схожим методом вычисляется усреднённое по статистической совокупности значение полупериода $T_{ср1/2}$.
После вычисления периода последующее декодирование не отличается от случая с заранее известным периодом, описанного выше.

Глава 2. Аналитическая часть.

Проанализировав теоретическую часть, попробуем разделить его особенности на основные преимущества и недостатки преобразования данных в Манчестерский Код:

1. В начале и в конце даных не возможно совпадение двух логических уровней - только 10 или 01.
2. Двоичная комбинация логических уровней из двух единиц говорит о последнем принятом нуле, а комбинация из двух нулей также однозначно говорит о единице. А значит после одной из таких комбинаций приёмник синхронизируется.
3. Количество логических 0 всегда равно количеству логических 1, а значит у такого сигнала будет отсутствовать постоянная составляющая, что крайне важно для электрических цепей и радиоволн.
4. Не может идти последовательно более двух одинаковых логических уровней, т.е. комбинация типа 111 или 000 невозможна.
5. Размер данных удваивается, что негативно сказывается на скорости передачи.

Глава 3. Практическая часть.

Для решения данной задачи применим технику Mock-объектов, в связи с этим для проведения тестов будем использовать связку gmock (Google Mocking Framework) + gtest (Google Test). В перечне требований к системе значится лишь единственный - совместимый C / C++ компилятор, например самый распространённый Visual Studio C++ для Windows или gcc g++ для Linux. Так (см. Приложение 1) выглядит простое консольное приложение, которое будет собираться вместе с тестами и запускать их.

Теперь перейдём непосредственно к тестам.
Преобразование данных в Манчестерский Код (encode), на мой взгляд, кажется более простой задачей, в сравнении с декодированием. Исходя из этого с неё и начнём. (см. Приложение 2).

Все тесты помещены в макрос TEST_F(). С помощью EXPECT_CALL(), в начале теста, необходимо установить ожидаемое поведение. В процессе преобразования 0 => 1010101010101010 и при условии, что старший бит идёт первым (MSB), ожидается вызов методов: сперва On_Man_Encode_One(), а затем On_Man_Encode_Zero() и так повторить восемь раз. После того, как ожидаемое поведение описано, необходимо вызвать проверяемый метод Man_Encode(). В случае если ожидаемое поведение не совпадает с реальным, во время проведения тестов об ошибке будет сообщено. Также по завершении каждого теста осуществляется проверка условия, что количество еденицы и нулей совпадает.

Как мы поняли из теоретической части преобразовать данные в Манчестерский Код не составляет труда (с помощью XORa), что и показано в Приложении 3.

Реализация задачи декодирования данных (см. Приложение 4) в оригинальную кодировку из Манчестерского Кода несколько сложнее. Прежде чем начть передачу данных, необходимо произвести синхронизацию с приёмником

сигнала. В реализации тестов нас не сильно заботит как именно приёмник синхронизируется и в какой последовательности будут вызваны (если вообще будут) On_Man_Decode_Add_0() и On_Man_Decode_Add_1()- для этой цели наши предварительные ожидания обозначим как testing::AtMost(1). По завершении синхронизации, можно точно описать соответствующие ожидания и спрогнозировать процесс декодирования при помощи EXPECT_CALL(). Также как и в предыдущем случае в конце каждого теста проверяется отсутствие постоянной составляющей (количество единиц и нулей должно совпадать).

Реализация процесса декодирования Манчестерского кода выглядит примерно так, как описано в Приложении 5.

Запуск:

- Если Вы используете связку Visual Studio + Windows, необходимо выполнить следующее:

```
@set GTEST_HOME=gtest-1.6.0
@set GMOCK_HOME=gmock-1.6.0

cl /EHsc /I%GTEST_HOME% /I%GTEST_HOME%/include -I%GMOCK_HOME% ^
-I%GMOCK_HOME%/include main.cpp src/manchester/Man_Encode.cpp ^
src/manchester/Man_Decode.cpp %GTEST_HOME%/src/gtest-all.cc ^
%GMOCK_HOME%/src/gmock-all.cc

main.exe
```

  В результате должен появиться вывод как в Приложении 6.

- Если Вы используете связку Linux + gcc g++, необходимо выполнить следующее:

```
GTEST_HOME=gtest-1.6.0
GMOCK_HOME=gmock-1.6.0

g++ -g -I$GTEST_HOME -I$GTEST_HOME/include -I$GMOCK_HOME \
-I$GMOCK_HOME/include -pthread main.cpp src/manchester/Man_Encode.cpp \
src/manchester/Man_Decode.cpp $GTEST_HOME/src/gtest-all.cc \
$GMOCK_HOME/src/gmock-all.cc

./a.out
```

  В результате должен появиться вывод как в Приложении 7.

Заключение.

В ходе выполнения Курсовой работы на тему «Код Манчестера» я изучила теоретическую часть данного алгоритма, ознакомилась с аппаратными реализациями, а также сделала программную реализацию данного алгоритма.

Основываясь на новых знаниях код Манчестера кажется мне достаточно актуальным. По найденным мною данным скорость приема/передачи не высока и составляет приблизительно 520,3 бит/сек. Что говорит о нецелесообразности данного алгоритма в случаях, когда требуемая скорость кратно выше.

На мой взгляд данный код хорошо подходит для простых прикладных задач или для случаев, когда объём кодируемых данных не слишком велик.

Список использованных источников.

1. Городов И. С., Бирюлин И. В., Лазарева О.В. Пути повышения конкурентоспособности в отрасли телекоммуникаций // Наукоемкие технологии в космических исследованиях земли 2012, т. 4, No2, с. 29-31.
2. Болданова Е. В. Тенденции в мировых телекоммуникациях // Baikal esearch journal 2017, т. 8, No1, с. 1-8.
3. Жакишева Т. М., Погожев А. О. «ОСНОВНЫЕ СПОСОБЫ КОДИРОВАНИЯ ИНФОРМАЦИИ ПРИ ПЛАНИРОВАНИИ ПРОИЗВОДСТВЕННОГО ПРОЦЕССА»
4. https://radiohlam.ru/manchester/
5. http://forum.easyelectronics.ru/viewtopic.php?f=16&t=36021
6. http://proiot.ru
7. https://docs.microsoft.com

Приложение.

Приложение 1.

/manchester/tests/gtest/main.cpp

```
#include "gtest/gtest.h"

int main(int argc, char* argv[]) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Приложение 2.

/manchester/tests/gtest/src/manchester/Man_Encode.cpp

```
#include <gtest/gtest.h>
#include <gmock/gmock.h>

extern "C"
{
    #include "../../../../manchester/src/tx/Man_Encode.c"
}

class I_Manchester_Encode {
public:
        virtual void Manchester_Encode_One() = 0;
        virtual void Manchester_Encode_Zero() = 0;
};

/* Mock implementation */

ACTION_P2(Action_Inc_1, T, V) {
    ON_CALL(*T, Manchester_Encode_One())
        .WillByDefault(Action_Inc_1(T, testing::ByRef(++V)));
}
```

```
ACTION_P2(Act_Inc_0, T, V) {
    ON_CALL(*T, Manchester_Encode_Zero())
        .WillByDefault(Act_Inc_0(T, testing::ByRef(++V)));
}

class Manchester_Encode_Mock : public I_Manchester_Encode {
    public:
        MOCK_METHOD0(Manchester_Encode_One,void());
        MOCK_METHOD0(Manchester_Encode_Zero,void());

        Manchester_Encode_Mock() {
            _full_1 = _full_0 = 0;
            ON_CALL(*this, Manchester_Encode_One())
                .WillByDefault(Action_Inc_1(this, testing::ByRef(_full_1)));
            ON_CALL(*this, Manchester_Encode_Zero())
                .WillByDefault(Act_Inc_0(this, testing::ByRef(_full_0)));
        }

        void Exp_full_one_and_zero_Eq(int full) {
            EXPECT_EQ(_full_1, full);
            EXPECT_EQ(_full_0, full);
        }

    private:
        int _full_1, _full_0;
};

/* Fixture class for each test */

class Manchester_Encode_Test_F : public testing::Test {
    public:
        static Manchester_Encode_Mock* get_Mock() {
            return _Manchester_Encode_Ptr;
        }

    protected:
        virtual void SetUp() {
            _Manchester_Encode_Ptr = &_Manchester_Encode;
        }

    private:
        Manchester_Encode_Mock _Manchester_Encode;
        static Manchester_Encode_Mock* _Manchester_Encode_Ptr;
};

Manchester_Encode_Mock* Manchester_Encode_Test_F::_Manchester_Encode_Ptr;

/* Man_Encode externs (events) */

void On_Man_Encode_One() {
    Manchester_Encode_Test_F::get_Mock()->Manchester_Encode_One();
}

void On_Man_Encode_Zero() {
    Manchester_Encode_Test_F::get_Mock()->Manchester_Encode_Zero();
}

/* 0 => 1010101010101010(manchester) */

TEST_F(Manchester_Encode_Test_F, Send_0) {
    testing::InSequence s;
    for (int i = 0; i < 8; i++) {
        EXPECT_CALL(*get_Mock(), Manchester_Encode_One()); // MSB
        EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero());
    }
```

```
    Man_Encode(0);

    get_Mock()->Exp_full_one_and_zero_Eq(8);
}

/* 255(dec) => 11111111(bin) => 0101010101010101(manchester) */

TEST_F(Manchester_Encode_Test_F, Send_255) {
    testing::InSequence s;
    for (int i = 0; i < 8; i++) {
        EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero()); // MSB
        EXPECT_CALL(*get_Mock(), Manchester_Encode_One());
    }

    Man_Encode(255);

    get_Mock()->Exp_full_one_and_zero_Eq(8);
}

/* 170(dec) => 10101010(bin) => 0110011001100110(manchester) */

TEST_F(Manchester_Encode_Test_F, Send_170) {
    testing::InSequence s;
    EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero()); // MSB
    for (int i = 0; i < 3; i++) {
        EXPECT_CALL(*get_Mock(), Manchester_Encode_One()).Times(2);
        EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero()).Times(2);
    }
    EXPECT_CALL(*get_Mock(), Manchester_Encode_One()).Times(2);
    EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero());

    Man_Encode(170);

    get_Mock()->Exp_full_one_and_zero_Eq(8);
}

/* 85(dec) => 01010101(bin) => 1001100110011001(manchester) */

TEST_F(Manchester_Encode_Test_F, Send_85) {
    testing::InSequence s;
    EXPECT_CALL(*get_Mock(), Manchester_Encode_One()); // MSB
    for (int i = 0; i < 3; i++) {
        EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero()).Times(2);
        EXPECT_CALL(*get_Mock(), Manchester_Encode_One()).Times(2);
    }
    EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero()).Times(2);
    EXPECT_CALL(*get_Mock(), Manchester_Encode_One());

    Man_Encode(85);

    get_Mock()->Exp_full_one_and_zero_Eq(8);
}

/* 84(dec) => 01010100(bin) => 1001100110011010(manchester) */

TEST_F(Manchester_Encode_Test_F, Send_84) {
    testing::InSequence s;
    EXPECT_CALL(*get_Mock(), Manchester_Encode_One()); // MSB
    for (int i = 0; i < 3; i++) {
        EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero()).Times(2);
        EXPECT_CALL(*get_Mock(), Manchester_Encode_One()).Times(2);
    }
    EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero());
    EXPECT_CALL(*get_Mock(), Manchester_Encode_One());
    EXPECT_CALL(*get_Mock(), Manchester_Encode_Zero());
```

```
    Man_Encode(84);

    get_Mock()->Exp_full_one_and_zero_Eq(8);
}
```

Приложение 3.

/manchester/src/tx/Man_Encode.c

```
#include "Man_Encode.h"

/*****************************************************************
 *      Function Name:  Man_Encode                               *
 *      Return Value:   no                                       *
 *      Parameters:     character to transmit                    *
 *      Description:    Convert char to Manchester Code (2 chars)*
 *                      MSB is first to convert                  *
 *****************************************************************/

void Man_Encode(register char character) {
    register unsigned char bitcount = 8;

    while (bitcount--) {
        if (character & 0x80) {
            On_Man_Encode_Zero();
            On_Man_Encode_One();

        } else {
            On_Man_Encode_One();
            On_Man_Encode_Zero();
        }
        character <<= 1;
    }
}
```

Приложение 4.

/manchester/tests/gtest/src/manchester/Man_Decode.cpp

```
#include <gtest/gtest.h>
#include <gmock/gmock.h>

extern "C"
{
    #include "../../../../manchester/src/rx/Man_Decode.c"
}

class I_Manchester_Decode {
    public:
        virtual void Manchester_Decode_Add_1() = 0;
        virtual void Manchester_Decode_Add_0() = 0;
};

/* Mock implementation */

class ManDecodeMock : public I_Manchester_Decode {
    public:
        MOCK_METHOD0(Manchester_Decode_Add_1,void());
        MOCK_METHOD0(Manchester_Decode_Add_0,void());
};
```

```
/* Fixture class for each test */

class Manchester_Decode_Test_F : public testing::Test {
    public:
        static ManDecodeMock* get_Mock() {
            return _ManDecodePtr;
        }

    protected:
        virtual void SetUp() {
            _full_1 = _full_0 = 0;
            _ManDecodePtr = &_ManDecode;
        }

        void Exp_full_one_and_zero_Eq(int full) {
            EXPECT_EQ(_full_1, full);
            EXPECT_EQ(_full_0, full);
        }

        void Perf_Stable_Zero(unsigned char stage) {
            _full_0 += stage;
            Manchester_Decode_Stable_Zero(stage);
        }

        void Perf_Stable_One(unsigned char stage) {
            _full_1 += stage;
            Manchester_Decode_Stable_One(stage);
        }

    private:
        ManDecodeMock _ManDecode;
        static ManDecodeMock* _ManDecodePtr;
        int _full_1, _full_0;
};

ManDecodeMock* Manchester_Decode_Test_F::_ManDecodePtr;

/* Man_Decode externs (events) */

void On_Man_Decode_Add_1() {
    Manchester_Decode_Test_F::get_Mock()->Manchester_Decode_Add_1();
}

void On_Man_Decode_Add_0() {
    Manchester_Decode_Test_F::get_Mock()->Manchester_Decode_Add_0();
}

/* Helpers for TEST_F */

#define EXPECT_MANCHESTER_SYNCH(x,y)                                  \
    EXPECT_CALL(*get_Mock(), Man_Dec_Add_##x())                        \
        .Times(testing::AtMost(1)); /* Clay? */                       \
    EXPECT_CALL(*get_Mock(), Man_Dec_Add_##y())                        \
        .Times(testing::AtMost(1)); /* Clay? */                       \
    EXPECT_CALL(*get_Mock(), Man_Dec_Add_##x()); /* Sync  */ \
    EXPECT_CALL(*get_Mock(), Man_Dec_Add_##y())  /* Sync  */ \

#define PERF_MANCHESTER_SYNCH(x,y)                                    \
    Perf_Stable_##x(1);  /* Clay balance (1 - 1 = 0) */   \
    Perf_Stable_##y(2);  /* Sync balance (2 - 2 = 0) */   \
    Perf_Stable_##x(2);  /* Sync */                       \
    Perf_Stable_##y(1)   /* Clay */                       \

/* 1010101010101010(manchester) => 0 */
```

```
TEST_F(Manchester_Decode_Test_F, Decode_0) {
    testing::InSequence s;
    EXPECT_MANCHESTER_SYNCH(1,0);
    EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_0()).Times(8);

    PERF_MANCHESTER_SYNCH(One, Zero);
    for (int i = 0; i < 8; i++) {
        Perf_Stable_One(1);
        Perf_Stable_Zero(1);
    }

    Exp_full_one_and_zero_Eq(11); // Sync(3) + Byte(8)
}

/* 0101010101010101(manchester) => 11111111(bin) => 255(dec) */

TEST_F(Manchester_Decode_Test_F, Decode_255) {
    testing::InSequence s;
    EXPECT_MANCHESTER_SYNCH(0,1);
    EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_1()).Times(8);

    PERF_MANCHESTER_SYNCH(Zero, One);
    for (int i = 0; i < 8; i++) {
        Perf_Stable_Zero(1);
        Perf_Stable_One(1);
    }

    Exp_full_one_and_zero_Eq(11); // Sync(3) + Byte(8)
}

/* 0110011001100110(manchester) => 10101010(bin) => 170(dec) */

TEST_F(Manchester_Decode_Test_F, Decode_170) {Manchester_Decode_Test_F
    testing::InSequence s;
    EXPECT_MANCHESTER_SYNCH(0,1);
    for (int i = 0; i < 4; i++) {
        EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_1());
        EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_0());
    }

    PERF_MANCHESTER_SYNCH(Zero, One);
    Perf_Stable_Zero(1);
    for (int i = 0; i < 3; i++) {
        Perf_Stable_One(2);
        Perf_Stable_Zero(2);
    }
    Perf_Stable_One(2);
    Perf_Stable_Zero(1);

    Exp_full_one_and_zero_Eq(11); // Sync(3) + Byte(8)
}

/* 1001100110011001(manchester) => 01010101(bin) => 85(dec) */

TEST_F(Manchester_Decode_Test_F, Decode_85) {
    testing::InSequence s;
    EXPECT_MANCHESTER_SYNCH(1,0);
    for (int i = 0; i < 4; i++) {
        EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_0());
        EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_1());
    }

    PERF_MANCHESTER_SYNCH(One, Zero);
    Perf_Stable_One(1);
    for (int i = 0; i < 3; i++) {
        Perf_Stable_Zero(2);
```

```
        Perf_Stable_One(2);
    }
    Perf_Stable_Zero(2);
    Perf_Stable_One(1);

    Exp_full_one_and_zero_Eq(11); // Sync(3) + Byte(8)
}

/* 1001100110011010(manchester) => 84(dec) => 01010100(bin) */

TEST_F(Manchester_Decode_Test_F, Decode_84) {
    testing::InSequence s;
    EXPECT_MANCHESTER_SYNCH(1,0);
    for (int i = 0; i < 3; i++) {
        EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_0());
        EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_1());
    }
    EXPECT_CALL(*get_Mock(), Manchester_Decode_Add_0()).Times(2);

    PERF_MANCHESTER_SYNCH(One, Zero);
    Perf_Stable_One(1);
    for (int i = 0; i < 3; i++) {
        Perf_Stable_Zero(2);
        Perf_Stable_One(2);
    }
    Perf_Stable_Zero(1);
    Perf_Stable_One(1);
    Perf_Stable_Zero(1);

    Exp_full_one_and_zero_Eq(11); // Sync(3) + Byte(8)
}
```

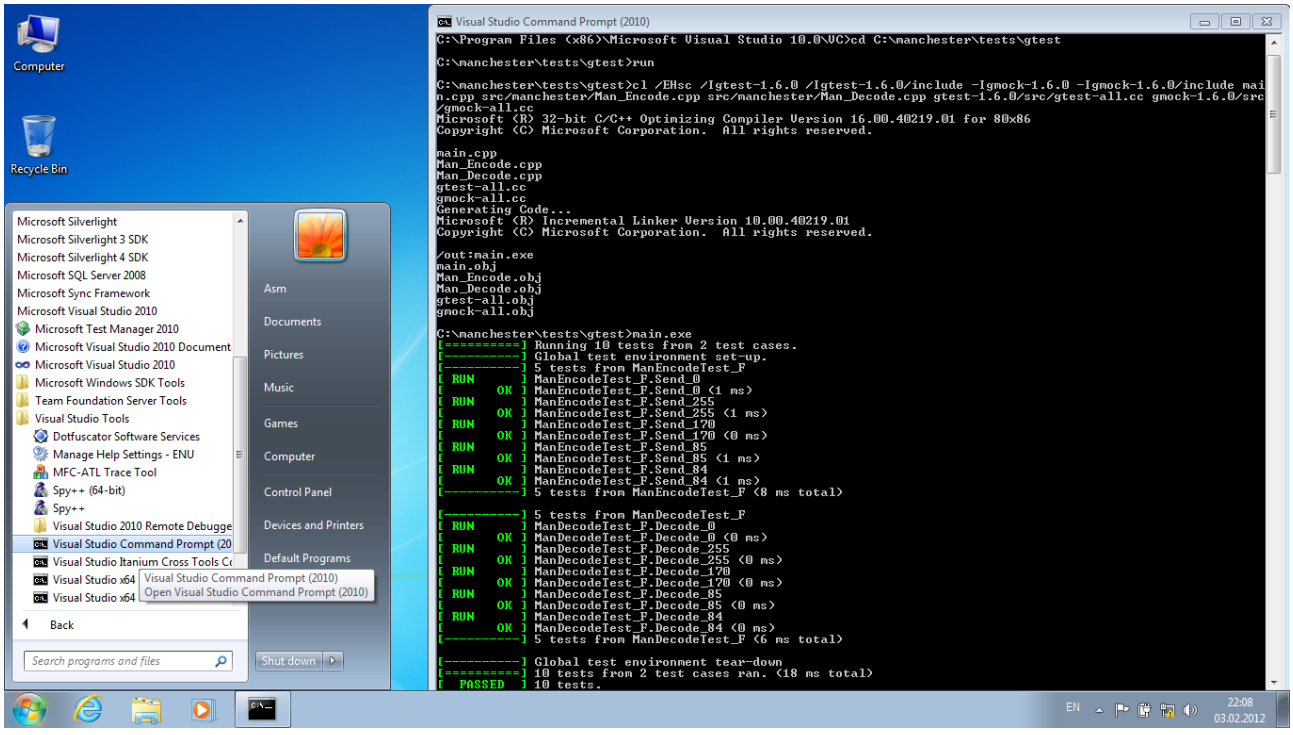
Приложение 5.

/manchester/src/rx/Man_Decode.c

```
#include "Man_Decode.h"

static bool check_LB;

/*****************************************************************
 *      Function Name:  Manchester_Decode_Stable_Zero             *
 *      Return Value:   no                                        *
 *      Parameters:     Stable digital input stage. Ideal 1 or 2  *
 *      Description:    Convert signal from Manchester Code.      *
 *                      Fire according On_Man_Decode_Add_1()      *
 *                      callback event.                           *
 *****************************************************************/

void Manchester_Decode_Stable_Zero(register unsigned char stage) {
    if ( stage ) {
        if ( !--stage ) {
            if ( check_LB ) {
                On_Man_Decode_Add_1();
                check_LB = 1;
            }
        } else if ( !--stage ) {
            On_Man_Decode_Add_1();
            check_LB = 1;
        }
    }
}
```

```
/*****************************************************************
 *       Function Name:  Manchester_Decode_Stable_One            *
 *       Return Value:   no                                      *
 *       Parameters:     Stable digital input stage. Ideal 1 or 2*
 *       Description:    Convert signal from Manchester Code.    *
 *                       Fire according On_Man_Decode_Add_0()    *
 *                       callback event.                         *
 *****************************************************************/

void Manchester_Decode_Stable_One(register unsigned char stage) {
    if ( stage ) {
        if ( !--stage ) {
            if ( !check_LB ) {
                On_Man_Decode_Add_0();
                check_LB = 0;
            }
        } else if ( !--stage ) {
            On_Man_Decode_Add_0();
            check_LB = 0;
        }
    }
}
```

Приложение 6.



Приложение 7.