Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский политехнический университет Петра Великого»

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

ВЫПУСКНАЯ РАБОТА БАКАЛАВРА

Разработка помехоустойчивого кодека с применением технологий дизассемблирования и реверс-инжиниринга

направление: 09.03.01 - Информатика и вычислительная техника

Выполнил:
Чеботарев Георгий Михайлович
Подпись_____
Руководитель: доцент, к.т.н.,
Богач Наталья Владимировна
Подпись_____

Санкт-Петербург

2016

Реферат

40 стр., 22 рис., 2 табл., 7 приложений

ОБРАТНАЯ РАЗРАБОТКА, ДИЗАССЕМБЛИРОВАНИЕ ПРОШИВКИ, РЕВЕРС-ИНЖИНИРИНГ, ПОМЕХОУСТОЙЧИВОЕ КОДИРОВАНИЕ, ЯЗЫК ПРОГРАММИРОВАНИЯ СИ/С++, РАЗРАБОТКА БИБЛИОТЕКИ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ.

В выпускной работе проводится разработка помехоустойчивого кодека на основе прошивки устаревшего устройства. К целям проведенной работы можно отнести:

- Раскрытие алгоритмов помехоустойчивого кодирования из прошивки.
- Внедрение промежуточного режима кодирования
- Разработка библиотеки модернизированного кодека на языке программирования Си.

Библиотека будет применяться при создании прошивки для новых устройств.

В работе проведён анализ существующих дизассемблеров. Приведены характеристики целевой аппаратной платформы. На простом примере рассмотрены основные принцип работы алгоритма Витерби.

Содержание

Список рисунков

Введение

Помехоустойчивое кодирование данных является повседневной задачей. Данные кодируется для любого типа каналов и видов передач. Без такого рода защиты от шума в канале передачи, отправка даже небольших объемов данных, на незначительное расстояние, представлялась бы невозможной задачей. Количество повторных передач возросло бы в сотни раз, что негативно отразилось бы на организации работы всех систем передачи данных.

Цель помехоустойчивого кодирования – это минимизация искажения данных, в следствии воздействия на них шумовых эффектов, в канале передачи. Для решения данной задачи применяются различные приемы передачи и их сочетание: кодирование данных, чередование закодированного сообщения, повторное кодирование и т.д.

При кодировании, данные представляются специальным образом. Ошибки, возникшие при передаче данных по каналу связи, при декодировании частично (или полностью) обнаруживаются и исправляются.

Одним из самых распространённых способов кодирования, для надежной передачи данных, является использование сверточного кодера и декодера Витерби. Такая организация работы кодека, гарантирует высокий процент исправления ошибок, однако обладает выраженным недостатком: избыточным кодированием. Повышение эффективности работы такого кодека является актуальной задачей.

Постановка задачи

Необходимо провести обратную разработку кодека, скрытого в прошивке устройства, базирующегося на основе микропроцессора TMS320C54. Необходимо дизассемблировать прошивку, определить границы кодека, проанализировать алгоритмы функций кодека и реализовать их на языке Си. Разработанный кодек должен поддерживать протокол работы кодека, скрытого в прошивке, для совместимости нового и старого устройства. Необходимо

разработать и внедрить промежуточный режим кодирования (декодирования). Конечная версия кодека должна быть оформлена в виде библиотеки. Взаимодействие с памятью должно осуществляться посредством передачи указателей на участки одномерного массива внутрь функций кодека. По завершению разработки, необходимо провести сравнительный анализ всех режимов кодирования.

1. Характеристика целевой аппаратной платформы

Прошивка была изъята из микропроцессора TMS320C54x. На ее основе был разработан более совершенный кодек, который может быть использован при создании прошивок различных устройств. Целевой аппаратной платформой, для которых будут создаваться прошивки, являются устройства на основе микропроцессора TMS320C54x. Этот выбор обусловлен следующими характеристиками:

Семейство микропроцессоров TMS320C54x ориентировано на быструю обработку цифровых сигналов. К их архитектурным особенностям можно отнести следующие:

- Модифицированная Гарвардская архитектура с раздельными 16-битными шинами памяти: одна под программу, три под данные
- Память и дополнительные периферийные устройства на чипе
- Наличие 7 видов адресации
- Параллельный умножитель 17×17 бит, соединенный с аккумулятором, позволяющий выполнять операцию умножения за один цикл
- Высокоскоростной АЛУ, обладающий высоким значением распараллеливания
- Специализированный блоки сравнения, выбора и хранения (CSSU) для ускорения выполнения операций сложения и сравнения в алгоритме Витерби
- Два 40-битных аккумулятора и восемь 16-битных регистров

- Инструкции повтора команд
- Инструкции с 32-битными операндами
- Инструкции, содержащие 2-3 операнда чтения
- Арифметические инструкции с параллельной загрузкой и параллельным выполнением
- Инструкции-условия

Совокупность данных характеристик позволяет выполнять кодирование данных максимально эффективно. Более подробное описание микропроцессора можно прочитать в источнике 1, а функциональная схема приведена на рисунке 22 (приложение 7).

2. Описание используемого в работе алгоритма декодирования

В ходе разработки кодека, было установлено, что для декодирования используется алгоритм Витерби. Для понимания работы кодека, и некоторых аспектах жизненного цикла, необходимо иметь некоторое представление о работе самого алгоритма декодирования.

Алгоритм Витерби — это алгоритм динамического программирования, выполняющий поиск наиболее подходящего списка состояний и получающий наиболее вероятную последовательность произошедших событий. Алгоритм получил своё название в честь создателя — американского инженера итальянского происхождения Эндрю Витерби [2].

Алгоритм декодирует последовательность, полученную в результате работы сверточного кодера. Рассмотрим работу кодера на несложном примере (см. Рис. 1).

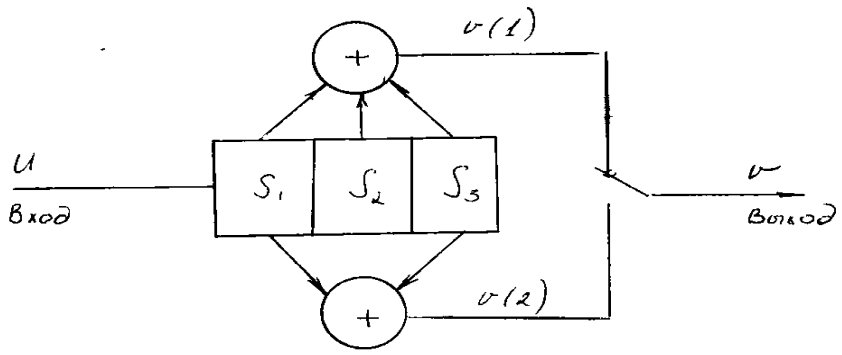

Рис. 1. Пример сверточного кодера

В зависимости от поступившего бита и линии задержки, создается закодированная пара бит, которая поступает на выход декодера. Для дальнейших пояснений кодер удобно будет представить в виде конечного автомата. (см. пример на Рис. 2).

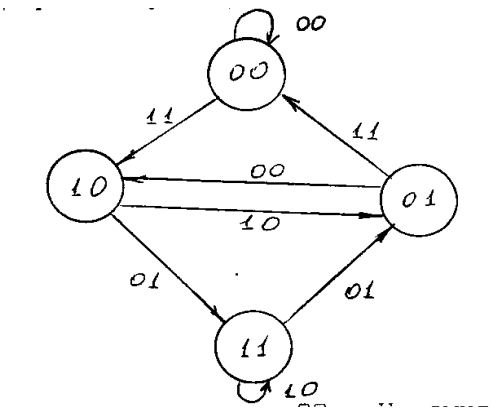

Рис. 2. Пример конечного автомата кодера

В вершинах графа указаны состояния кодера, дуги – это значения выходов с кодера. В зависимости от того, в каком состояний кодер находится и что было принято на вход - кодер переходит из одного состояния в другое, таким образом выполняя кодирования данных.

Изначально кодер находится в состоянии 00. Например, на вход поступит 1, тогда кодер перейдет в состояние 10, а на выходе кодера будет 11 (закодированный бит) и т.д.

Развертка диаграммы состояний во времени образует решетчатую диаграмму (см. Рис. 3).

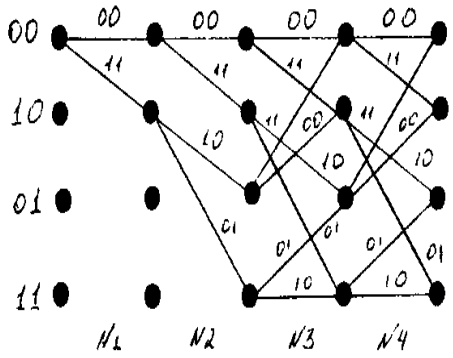

Рис. 3. Пример решетки Витерби

Каждое состояние решетки соответствует состоянию конечного автомата. Ширина решетки равна количеству состояний в конечном автомате. Движение по решетке осуществляется слева направо. Вход в решетку, как и в графе, расположен в состоянии 00. В зависимости от того, какая последовательность принимается на вход декодера, осуществляются переходы из одного состояния в другое. В случае же, если таких переходов из состояния нет, например, из текущего состояния, в графе предусмотрены переходы с весами 10 и 01, а на вход была принята последовательность 00, каждый из маршрутов, берущий начала из данного состояния, получает штраф, за один не соответствующий бит. При работе алгоритма генерируется N-ое количество путей. Для сокращения вычислительных ресурсов, отслеживается только K путей из всех возможных, $K \leq N$ (для данного примера $K = 4$), остальные маршруты отбрасываются. Таким образом в решетке остаются пути с наименьшим количеством штрафов.

По того, как вся решетка была построена, выбирается лучший путь (штрафы которого минимальны) и выполняется обратная проходка. При этом происходит восстановление и декодирование искомого сообщения.

3. Жизненный цикл разработки помехоустойчивого кодека

3.1. Дизассемблирование

Дизассемблирование является ключом к исходному коду любой программы, будь это компьютерный вирус или любой другой бинарный файл. Транслирование исходного файла на язык ассемблер позволяет увидеть и изучить алгоритмы работы программы. В связи с тем, что технология применяется не повсеместно – качественных сред дизассемблирования немного.

3.1.1. Выбор среды дизассемблирования

К наиболее известным и активно используемым дизассемблерам можно отнести: Sourcer, Hiew и IDA Pro. При разработке использоваться будет IDA Pro. Выбор обуславливается положительными и отрицательными сторонами всех трех дизассемблеров:

- *Sourcer* – дизассемблер, с простейшими функциями. Позволяет конвертировать исходный файл в ассемблерный код, однако, примерно в 30% случаев требуется вмешательство программиста для исправления ошибок дизассемблирования, что усложняет отладку большой программы. Невозможно добавлять комментарии или изменять название функций – это усложняет понимание кода и сказывается на скорости и разработки.
- *Hiew* – редактор двоичных файлов, ориентированный на работу с кодом. Имеет встроенный дизассемблер для x86, x86-64 и ARM, ассемблер для x86, x86-64 [3]. Стоит так же отметить, что данный редактор распространяется бесплатно. Однако с момента создания, в середине 90-х, в Hiew серьезных изменений не вносили, программа не удобна при работе: полное отсутствие интерактивности и командный режим. Количество поддерживаемых форматов входных файлов и процессоров жестко ограничено. Очевидно, что недостатки превышают достоинства.

Данный редактор подойдет для просмотра двоичного кода, но для дизассемблирования прошивки.

- *IDA Pro* (англ. Interactive DisAssembler) — интерактивный дизассемблер, отличается исключительной гибкостью, наличием встроенного командного языка. Поддерживает множество форматов исполняемых файлов для большого числа процессоров и операционных систем [4]. Среда рассчитана на подключение отдельных модулей, расширяющих функциональность среды. Последние версии (IDA SDK) дают возможность пользователям самостоятельно разрабатывать модули. Разрабатываемые модули могут быть отнесены к одному из трех классов:
    - процессорные модули – позволяют разбирать программы для новых процессоров;
    - загрузчики – необходимы для поддержки новых форматов входных файлов;
    - плагины – модули, расширяющие функциональность IDA (добавление новых команд, улучшение анализа посредством установления обработчиков событий [5])

На данный момент IDA является лидером среди дизассемблеров на мировом рынке. Дизассемблирование может быть проведено как в автоматическом режиме, так и в полуавтоматическом режиме, с участием пользователя. Продвинутый уровень интерактивности разрешает изменять названия функций и переменных, сохраняет позицию курсора при перезагрузке среды, записывает изменения в базу данных (с возможным экспортом для другой ПК с IDA), позволяет выполнять переходы по адресам двойным кликом мыши.

Кроме того, следует отметить, что существуют как платные, так и бесплатные версии среды. Благодаря этому, использовать ее могут все желающие. Автором дизассемблера является Ильфак Гильфанов,

разработчик с многолетним опытом и специалист по компьютерной безопасности [5].

Из существующих дизассемблеров, Interactive DisAssembler наиболее подходит для решения поставленной задачи. IDA не только обладает рядом функциональных преимуществ перед конкурентами, но и поддерживает дизассемблирование программ, написанных для искомого процессора TMS320C54.

3.1.2. Процесс дизассемблирования прошивки

Для дизассемблирования прошивки необходимо выполнить следующие действия:

1. Запустить среду IDA и перейти по вкладкам **File → New → Binary File.** Откроется окно выбора типа входного файл, как на Рис. 4. После подтверждения типа файла, откроется окно, в котором необходимо выбрать сам файл прошивки.

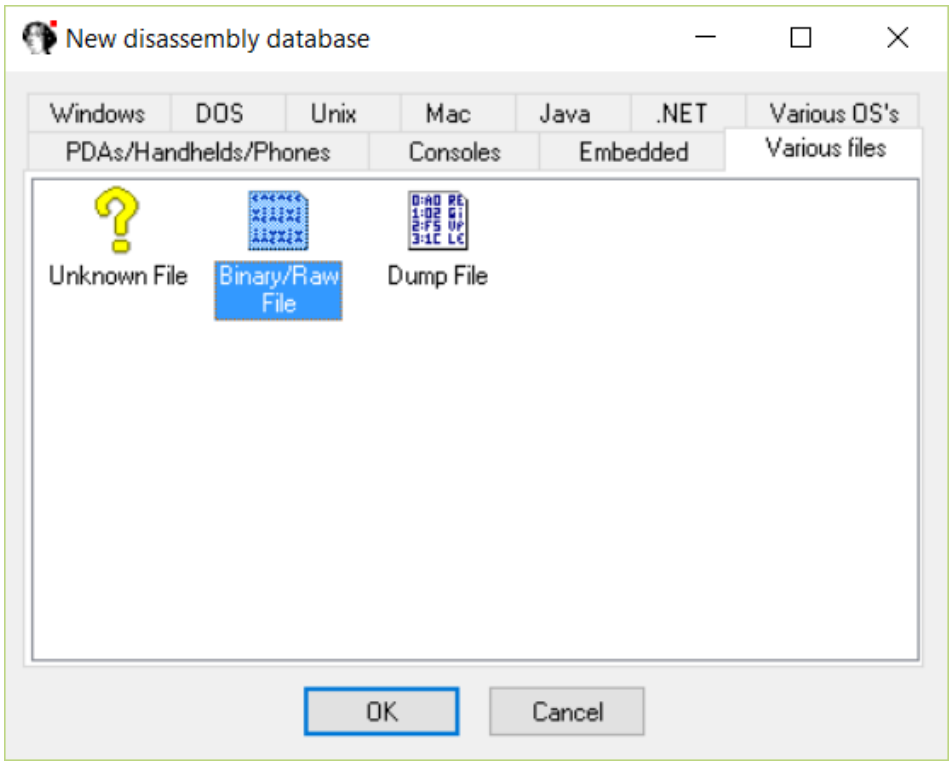

Рис. 4. Выбор типа входного файла

2. Затем откроется окно выбора типа процессора. Необходимо выбирать **TMS320C54** (см. Рис. 5).

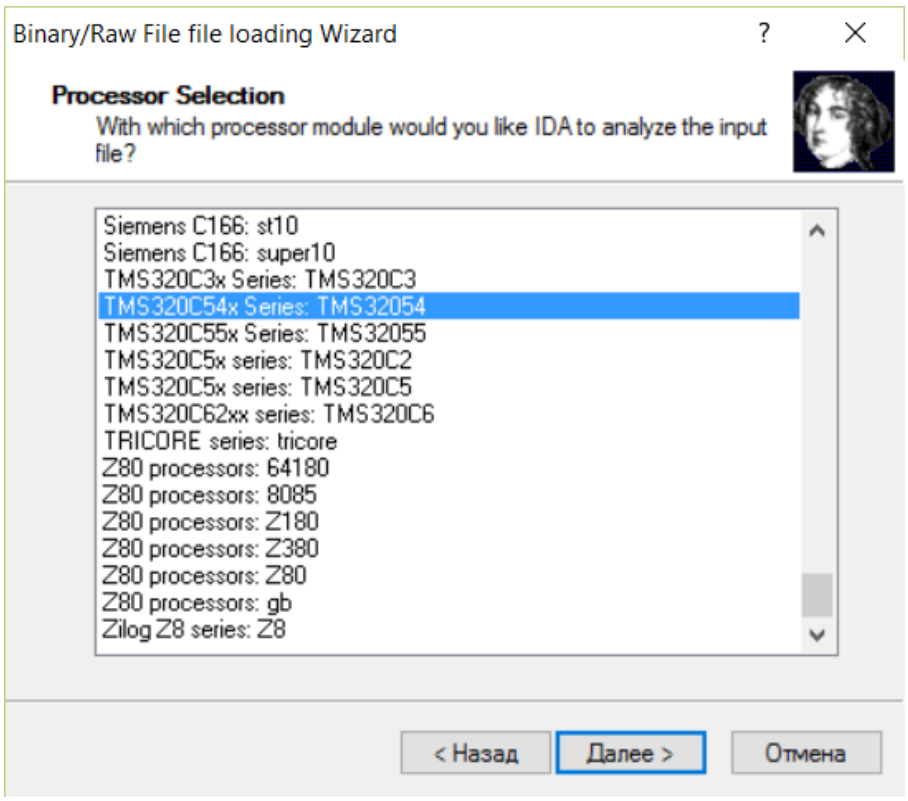

Рис. 5. Выбор нужного процессора

3. После выбора процессора **IDA** предложит указать участок памяти, куда будет загружена прошивка (см. Рис. 6). Так как задача - получить искомый код прошивки – никаких изменений вносить не нужно, необходимо оставить все по умолчанию.

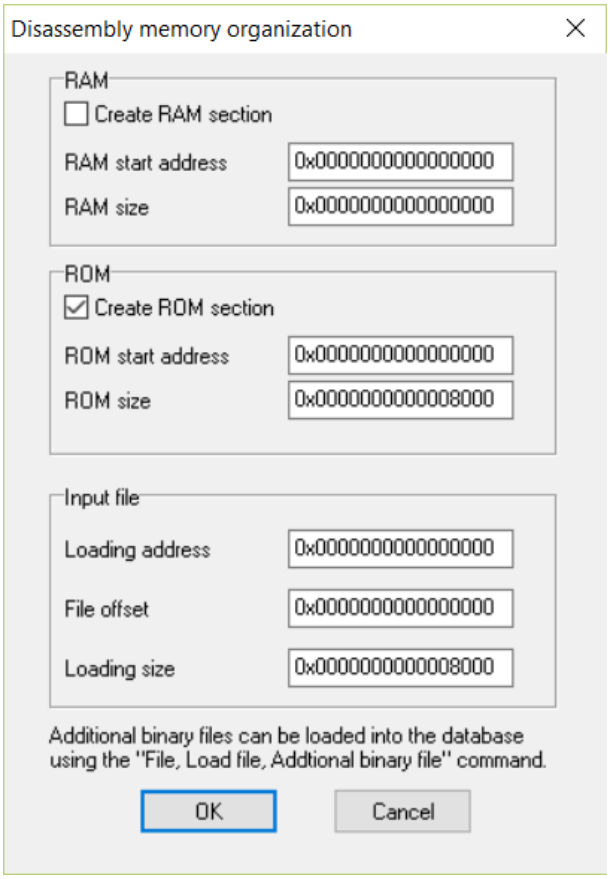

Рис. 6. Настройки адресов загрузки

4. Сразу после завершения настроек отроется окно IDA View-A (см. Рис. 7), в котором и будет отображаться дизассемблированный код.

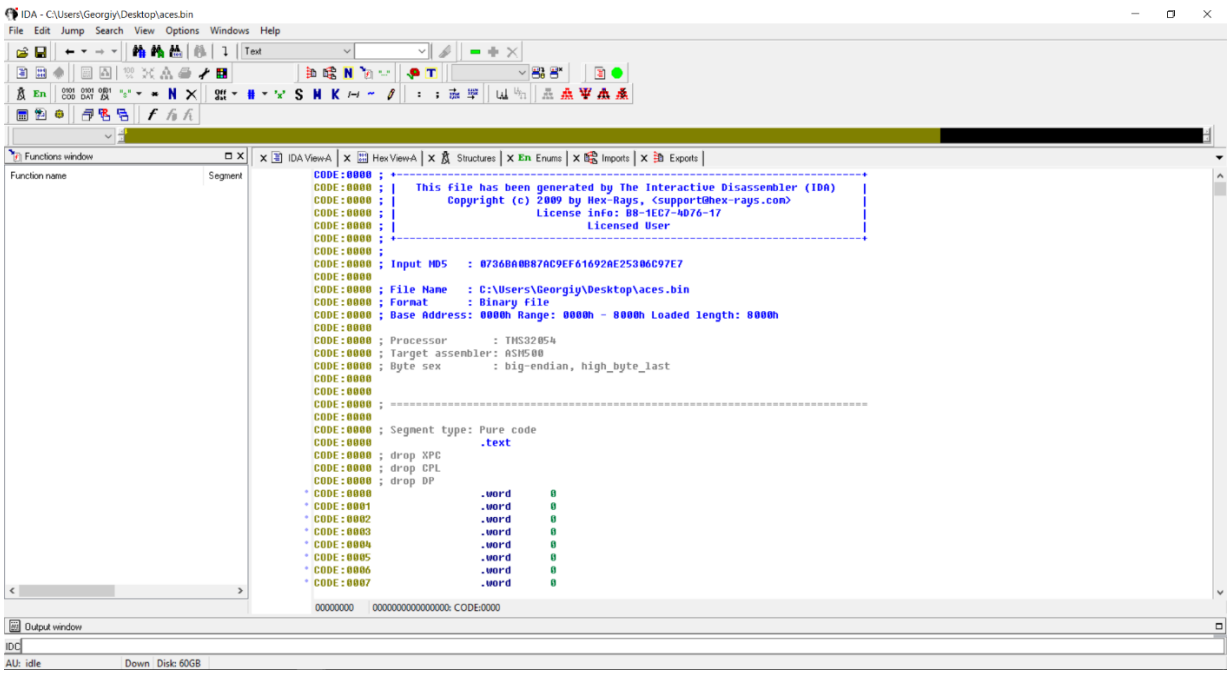

Рис. 7. Основное окно разработки IDA View

5. В открывшемся окне предоставлено содержимое бинарного файла. Необходимо установить курсор на нулевой адрес. По нажатию на кнопку "C" - небольшие участки кода начнут преобразовываться в ассемблерные команды, которые можно будет анализировать (см. Рис. 8). Перемещаясь по адресному пространству, необходимо выполнять преобразование бинарного кода в ассемблерные команды. IDA самостоятельно подписывает начало и конец функций, присваивая названия по адресу, например *sub_2010*. Среда так же позволяет изменять названия функций и добавлять комментарии. Если пользователь проанализировал и понял некоторую функцию, он может изменить ее название на более подходящее или добавить необходимый комментарий двумя кликами ПКМ. Проводить статический анализ кода, в котором часть функций подписана и прокомментирована – в разы удобнее и быстрее, нежели чистый ассемблер файл.

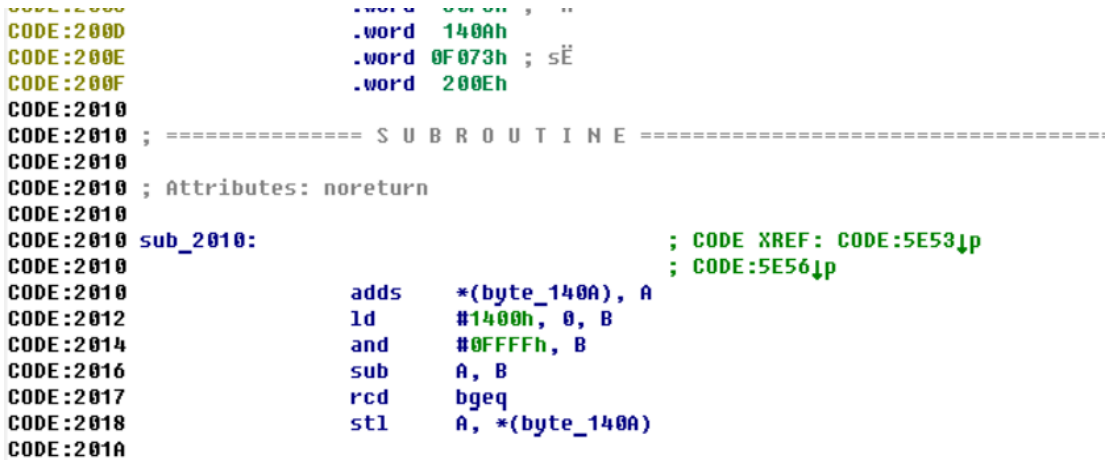

Рис. 8. Дизассемблированный участок кода

3.1.3. Выбор симулятора микропроцессора

Для того, чтобы найти кодек среди всей прошивки, одного статичного анализа недостаточно. Для удобного анализа кода потребуется симулятор микропроцессора. Симулятор позволит анализировать изменение регистров, аккумуляторов и памяти, что упростит первостепенную задачу – задачу

обнаружения кодека. В связи с тем, что микропроцессор TMS320C54 является разработкой компании Texas Instrument [6], в качестве симулятора будет использоваться Code Composer Studio [7] – симулятор, разработанный той же фирмой для тестирования своих собственных процессоров. Данная среда зарекомендовала себя на рынке достойным образом, поэтому выбор очевиден. Существуют как платные версии, так и бесплатные с ограниченным набором микропроцессоров (TMS320C54 попадает как в платную версию программы, так и в бесплатную).

3.1.4. Настройка среды симулятора

Среда состоит из двух модулей: ***CCStudio(CCS)*** и ***CCStudio Setup(CCSS)***. Перед использованием симулятора необходимо установить тип симулируемого процессора в CCSS (см. Рис. 9)

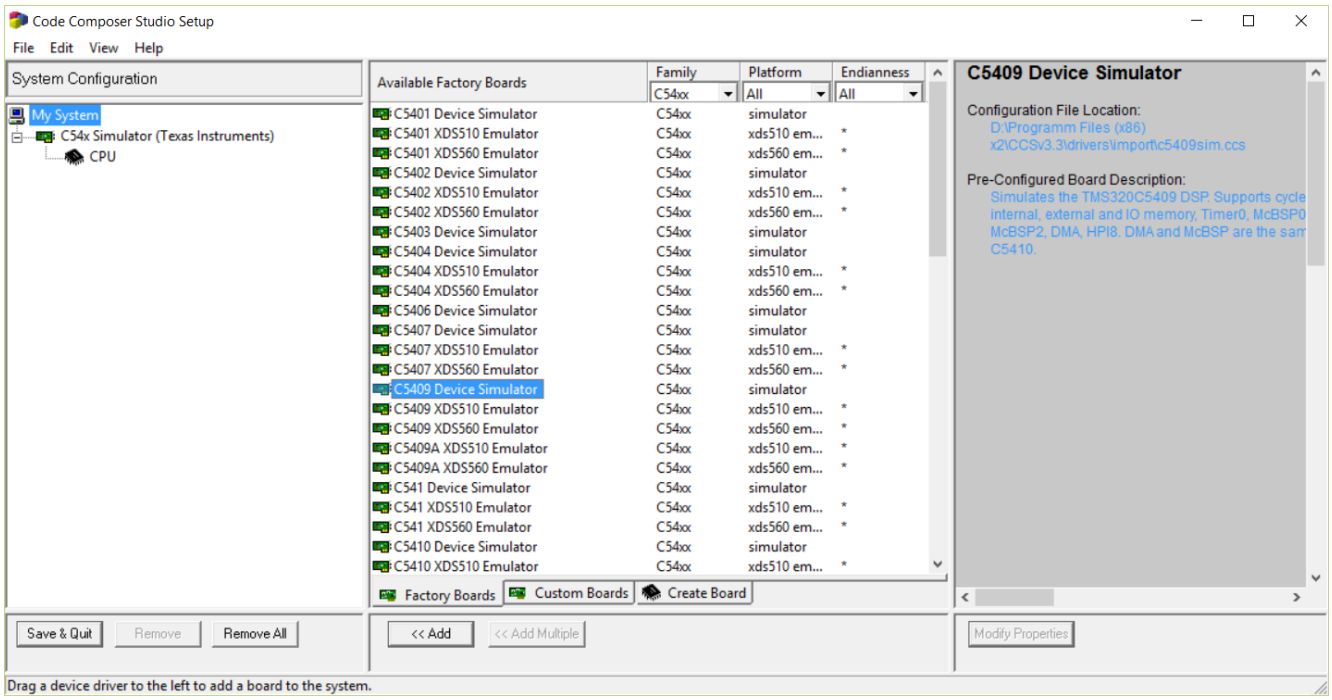

Рис. 9. Настройка CCS

Необходимо найти процессор ***TMS320C54*** и добавить его к симулируемым (**<<Add**), после чего перезапустить ***CCS*** (**Save&Quit**). После перезагрузки среды откроется основное окно CCS. Отобразить нужные окна, можно установив соответствующие модули во вкладке ***View***. Потребуется

следующие три окна: окно памяти, окно разборки и окно основных регистров (см. Рис. 10).

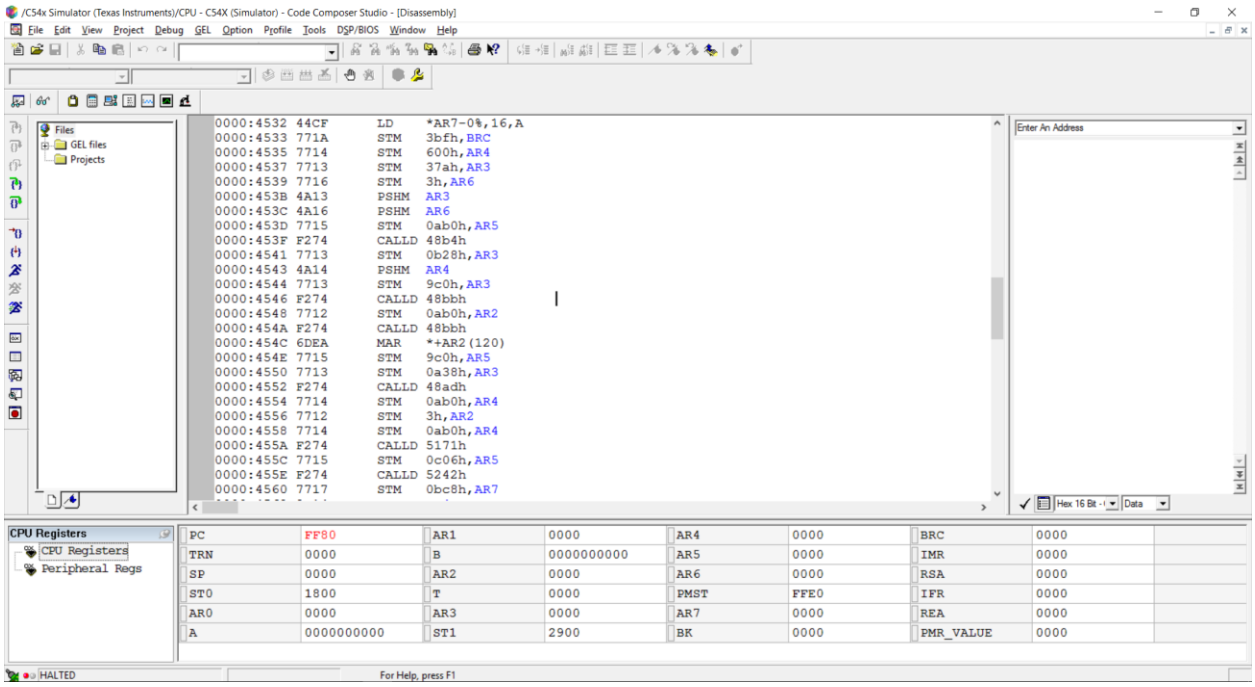

Рис. 10. Необходимые окна для разработки в CCS

После настройки среды необходимо загрузить саму прошивку и данные в симулятор. Дизассеблированная прошивка подается на вход CCS в виде dat-файла, содержащем необходимые заголовки, ассемблерные команды и данные в 16-ом коде. IDA способна сгенерировать файл содержащий все команды в 16-ом коде, однако ни заголовки, ни формат данных не подходит для CCS. Для корректной работы на вход CCS необходимо подать файл с 16-битным словами формата 1 (см. Рис. 11), с следующими заголовками:

- 1651 – корректная сигнатура для правильного распознавания формата данных и считывания CCS;
- 1 – формат подаваемых чисел. 1 – это hex формат. Команды можно подавать и в другом формате: 2 - int, 3-long, 4-float;
- 0 - начальный адрес (загрузка с нулевого адреса);
- 0 - текущая страница (нулевой адрес, следовательно, нулевая страница);
- FFFF - конечный адрес (крайний адрес в 16-битной системе) [8].

```
1651 1 0 0 ffff
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
0x1100
0x0000
```

Рис. 11. Формат dat-файла для CCS

IDA способна сохранить результат дизассемблирования в 16-битном формате (окно **hex view-A** → выделить код → **ПКМ** → **copy to file**) типа 2 (см. Рис. 12).

```
2185  7313 0062 F010 0003 8061 ED01 FA43 2191 E589 4592 4761 CB89 8793 F000 0002 8061
2195  8816 F000 0001 8814 7664 01FF 11E4 6524 8163 F77F 8910 7715 0011 7212 0060 7213
21A5  0062 721A 0061 F420 F272 21B4 7711 0040 6F85 0D59 1964 8914 6DB1 3093 28E4 6000
21B5  F00F 0001 8292 4910 6E8E 21A4 0163 8910 FC00 F000 0014 8060 F274 2318 F062 0008
21C5  FC00 F110 0018 FA4E 21D9 F020 6666 F300 0009 FA4F 21D9 F020 4CCC F300 000F 890E
21D5  F020 2666 F067 028F FC00 F6B8 F310 0001 891A F310 0008 890E F300 000F 44F8 000B
21E5  F48C F47F F57E F300 6320 8913 F030 0003 F562 890E 7714 0008 F004 FFFF F272 2201
21F5  1593 F764 F000 0001 962D FA20 2201 F180 8392 1193 F010 0004 F764 F7B8 FC00 F6B8
2205  F6B9 F310 0009 F761 F300 652F 8913 771A 0007 F272 2215 1093 3C83 F130 000F 8192
2215  F47C FE00 F7B8 F7B9 7311 0065 8064 8163 7312 0062 7313 0061 8060 F274 2318 F062
2225  0004 8066 3066 2061 F468 F00F 0001 8267 1064 F010 0001 881A 8064 7712 0068 F272
2235  2247 F420 F7B6 0067 F130 03FF 890E F576 0162 8913 2163 6F69 0D65 4563 4369 8368
2245  A598 B394 8391 F6B6 1065 8812 8813 6F66 0C5E 880E F420 4764 2892 F00F 0001 4409
2255  E732 ED00 4593 F520 4764 C798 FC00 7312 0066 7313 0068 7314 0069 E808 F074 2000
2265  806A F7B6 7212 0068 E808 F274 2183 7213 006A F420 8067 6F68 0D00 8914 6F69 0D00
```

Рис. 12. Формат сохранения результата от IDA

Конвертация формата 2 (IDA) в формат 1 (CCS) выполняется при помощи небольшой программы, написанной на языке программирования Си (приложение 1). Результатом работы программы станет dat-файл с необходимыми заголовками и нужным форматом данных, который будет подаваться на вход CCS.

3.2. Анализ полученного кода

Перед вторым этапом разработки были поставлены следующие задачи:

- Найти функции кодека среди прошивки;
- Подтвердить работоспособность кодека, полученного после дизассемблирования;
- Провести анализ структуры кодека и его составляющих.

3.2.1. Поиск функции кодека

Для упрощения задачи поиска функций кодека в прошивке, потребовалось еще несколько вариантов прошивки этого же процессора. Каждая из прошивок отличается внутренним состоянием области памяти. Удобно было взять:

- Состояние памяти, до и после кодирования;
- Состояние памяти, сразу после принятия данных и после декодирования.

Поиск некоторого логически связанного участка прошивки рассмотрим на примере кодера. Анализируя прошивку с данными до кодирования представилось возможным найти адреса, в которые записываются входные данные. Функции, которые обращаются к этим адресам были выделены в группу 1. Таких функций оказалось не много, так как прошивка занимает примерно четверть всей внутренне памяти микропроцессора, в следствии чего свободной памяти на кристалле в избытке, и функции редко обращаются к одним и тем же адресам без логической связи друг с другом. Затем была проанализирована прошивка, содержащая закодированные данные, которую получили, взяв слепок памяти после кодирования. Функции, которые «что-то» пишут в область памяти, где расположились закодированные биты, были выделены в группу 2. Сопоставив функции групп 1 и 2, обнаружились функции, которые одновременно обращаются и к первым и ко-вторым адресам – это и есть основные функции кодера, точнее некоторая их часть (они были выделены в новую группу 3). Следующим шагом стало точное определение границ кодера. Необходимо было определить, какие близлежащие функции являются

кодером, а какие нет. Выполняя и анализируя код функций группы 3 и прилежащих функции с помощью симулятора CCS, удалось с точностью до команды определить границы кодера. CCS не обладает интерактивностью, поэтому работу CCS удобно было комбинировать с IDA, оставляя пометки и комментарии в среде дизассемблера. Поиск декодера выполнялся подобным образом, на основании прошивок до и после декодирования.

3.2.2. Тестирование кодека

После того, как границы кодера и декодера были определены, потребовалось подтвердить работоспособность кода, полученного после дизассемблирования. Тестировать отдельные участки кода (например, кодер или декодер) намного проще, нежели всю прошивку целиком. Именно поэтому этап проверки кода идет только после обнаружения границ кодека.

В связи с тем, что IDA поддерживает TMS320C54, вероятность возникновения ошибки при дизассемблировании сводиться к нулю. Однако писать собственные модули, не проверив работоспособность ранее полученного кода не разумно - «*Здоровое недоверие — хорошая основа для совместной работы*» [9]. Для того, чтобы удостовериться в работоспособности кода, в симулятор (CCS) последовательно загружаются два dat-файла: первый - полученный код после дизассемблирования с помощью IDA, и второй - данные которые необходимо закодировать. Средствами симулятора выполняется кодирование/декодирование. Результат кодирования/декодирования должен совпадать с эталонной моделью (прошивки устройства в соответствующих состояниях) не только по содержанию (данные бит в бит), но и по адресам расположения данных. Т.к. задача стоит подтвердить корректность кода после дизассемблирования, нескольких тестов с успешным исходом хватит (при условии, что отсутствуют тесты с неуспешным исходом). Ошибки, возникающие в коде в процессе обратной разработки чаще всего, несут глобальный характер, как следствие уже первые тесты показывают некорректность кода. В таком случае необходимо проверить корректность

искомой прошивки и правильность настроек. Большинство ошибок допускаются именно на этих этапах, конкретного универсального решения таких проблем нет. Возможно стоит повторить весь процесс дизассемблирования с начала или же попробовать отладить код самому, что без понимания практически невозможно.

В случае, если проблем не возникло, или они были успешно решены, можно переходить к анализу кодека.

3.2.3. Анализ структуры кодека

После завершения дизассемблирования и тестирования работоспособности кода, был проведен полный анализ функций и алгоритмов кодирования и декодирования. В результате было выяснено, что кодек состоит из 12 функций и подразделяется на нескольких основных модулей: свёрточный кодер, модуль прямого чередования (interleaving), модуль обратного чередования (deinterleaving), декодер- Витерби.

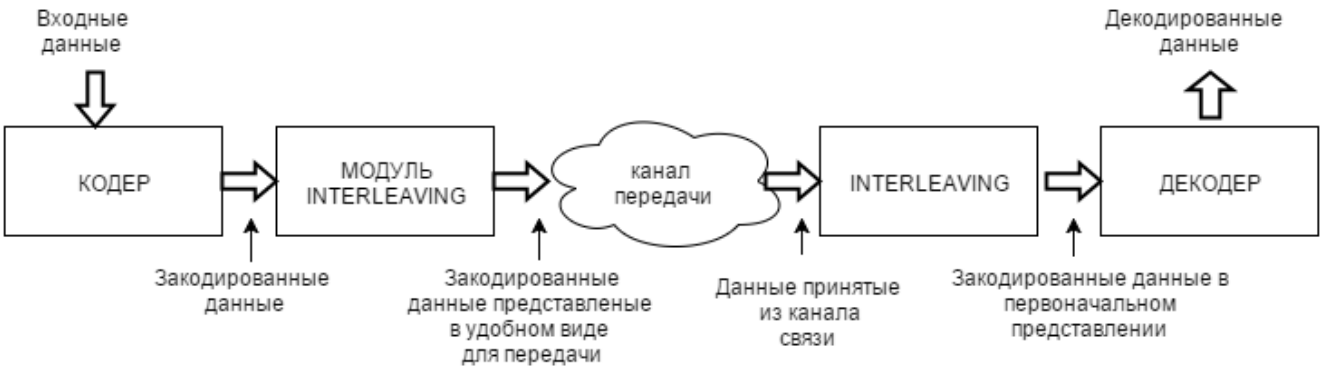

Рис. 13. Структурная схема работы кодека

Структура кодека приведена на Рис. 13. Ниже рассмотрим смысловую часть каждого из модулей.

3.2.3.1. Кодер

Кодер выполняет кодирование указанного участка памяти в одном из двух режимов: кодирование двумя битами или четырьмя. По завершению кодирования каждый из кодируемых бит будет представлен двумя или

четырьмя битами соответственно, то есть выполняется сверточное кодирование. В основе данного модуля лежит 4 полинома. Для режима кодирования двумя битами используется только два полинома, для кодирования четырьмя битами – все четыре. Полиномы зависят от предыдущих шести бит - это означает, что кодер с памятью.

3.2.3.2. Модули чередования

После того, как данные успешно закодированы, они подвергаются чередованию. Этот процесс необходим для снижения вероятности неудачного декодирования. Во время передачи данных, вследствие воздействия шумов происходят битовые ошибки, часть данных теряется или принимается неверно. Декодер способен восстановить часть потерянных данных за счет свойств кода, однако качество декодирования и распознавания ошибок, напрямую зависит от количества ошибочных бит, следующих подряд друг за другом (пачки ошибок). В канале же при возникновении зашумления – портится целая последовательность бит, длиной от нескольких десятков до нескольких сотен бит.

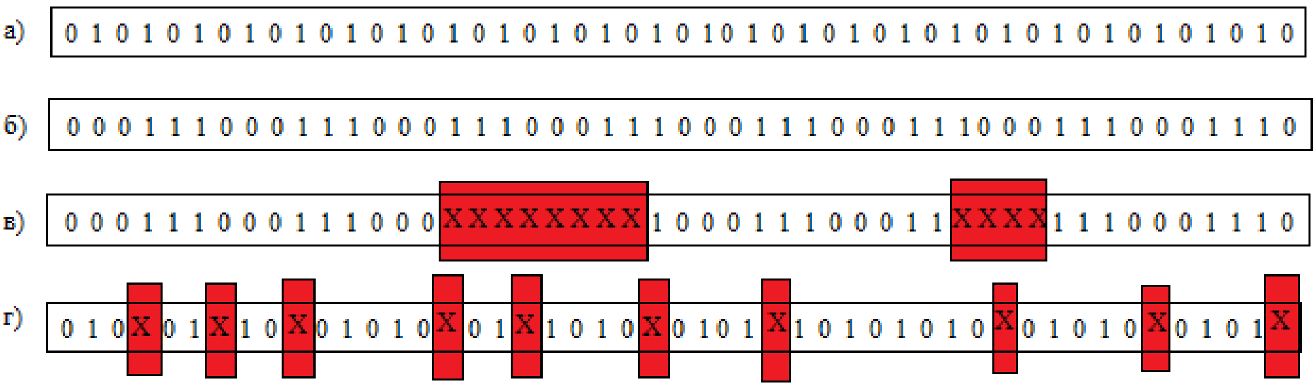

Рис. 14. а) Данные после кодирования; б) Данные после чередования; в) Данные после передачи по каналу связи; г) Данные после ликвидации чередования.

Для того, чтобы минимизировать длину последовательностей, ошибочных бит, применяются прямая и обратная операция чередования. Модуль чередования перемешивает биты в соответствии со стандартным

алгоритмом. Затем данные поступают в канал передачи, откуда считываются другим устройством. На принятом устройстве закодированные данные поступают в модуль обратного чередования, который восстанавливает исходную последовательность бит, которые в свою очередь уже могут быть декодированы. На Рис. 14 приведен простой пример работы данных модулей.

3.2.3.3. Декодер

На последнем этапе закодированные данные поступают в декодер, где согласно алгоритму Витерби производиться декодирование. Стоит отметить, что данная реализация декодера-Витерби является реализацией с мягкими весами, т.е. на вход декодера поступают биты с весами, которые присваиваются сторонними компонентами. В данном случае вес – это вероятностная мера, утверждающая с некоторой точностью, что принятый бит - это нуль или единица.

Тело алгоритма состоит из двух основных частей: прямой и обратной проходки. При прямой проходке выстраивается решетка Витерби - записываются метрики возможных путей декодирования. Решетка шириной 64 состояния, глубина декодирования составляет 6. Это означает, что на каждом шаге прямой проходки алгоритм отслеживает лучшие 64 пути (принимая решения на основе накопившихся ошибок). При обратной проходке, на основе полученных метрик, определяется путь с минимальной ошибкой. Выполняя проходку по решетке в обратном направлении, закодированное сообщение восстанавливается.

3.3. Разработка модулей кодека на языке Си

На основе логических связей функции были разбиты на 4 группы. Каждая из групп послужила образованию отдельного модуля: сверточного кодера, двух модулей чередования и модуля декодера. Перед каждым модулем поставлена отдельная задача, которую он должен выполнять максимально эффективно.

3.3.1. Кодер

С целью повышения эффективности, все возможные результаты работы полиномов рассчитаны заранее и сохранены в двумерном массиве 4x8, в виде 16-битных чисел (согласно спецификации процессора, под который ведется разработка). Каждый столбец соответствует конкретному полиному, а номер строки и бита – результату работы полинома. В зависимости от того какие биты были приняты на вход кодера ранее, определяется необходимая строка и бит. Для сверточного кодера был реализован следующий алгоритм (см. Рис. 15):

Рис. 15. Структура работы кодера

Тело кодера состоит из двух вложенных циклов. В теле внешнего цикла происходит определение номера строки и бита, в теле внутреннего - копирование необходимого бита для каждого из полиномов. Количество используемых полиномов зависит от режима кодирования.

3.3.2. Модули Чередования

На вход в качестве параметров модули принимают указатель на память и длину сообщения. Данные так же записываются в память на некотором смещении.

Алгоритм чередования состоит из двух частей (см. Рис. 16): диагонального и вертикального смешивания. Данные при этом представляются в виде условной матрицы Nx120, где каждая строка – это отдельный участок сообщения.

Рис. 16. Структура работы кодера

При диагональном чередовании выполняется считывание по диагоналям, в результате данные представляются в виде новой матрицы. Затем, при вертикальном смешивании выполняется транспонирование матрицы, полученной после диагонального смешивания. Результатом работы алгоритма становится перемешивание сообщения с двумя соседними. В случае же, если это было первое сообщение или последнее, в передаваемом потоке данных, то оно смешивается с нулевым пакетом (сообщение с нулями), который будет отброшен после передачи.

3.3.3. Декодер

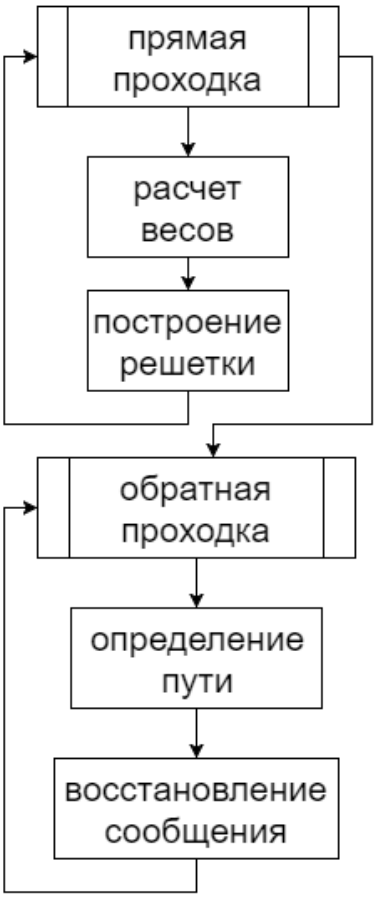

Рис. 17. Алгоритм декодирования

Модуль декодер состоит из 3 функций: 1 основной, которая включает в себя прямую и обратную проходку (см. Рис. 17), и 2-ух вспомогательных функции, которые рассчитывают все варианты ошибок на закодированном участке и соответствующие веса, необходимые для постройки решетки Витерби. Для решетки создается динамический массив TRN, размерность которого зависит от количества декодируемых бит. Решение о переходе состояний решетки принимается на основе рассчитанных весов закодированного участка бит. Метрики возможных путей декодирования при этом сохраняются в массиве TRN. При обратной проходке на основе метрик из массива TRN определяется верный маршрут (с наименьшей накопленной ошибкой) и данные восстанавливаются.

Кодек был оформлен в качестве библиотеки, написанной на языке программирования Си (Приложение 2). При этом каждый модуль представлен в виде отдельной функции или группы функций. Модуль-образующие функции принимают на вход: указатели на участки памяти (для чтения и записи), длину участка памяти, который необходимо подвергнуть обработке, и режим кодирования, необходимый только кодеру и декодеру. При разработке использовался язык программирования Си. Были подключены две сторонние библиотеки: <math.h> – для команды pow (возведение в степень) и <stdlib.h> - для выделения памяти под массив TRN в декодере.

3.4. Тестирование реализации на языке Си

По завершению разработки было проведено тестирование. Во-первых, необходимо было подтвердить факт работоспособности разработанного кодека, во-вторых подтвердить совместимость разработанного кодека и искомого устройства. Для того, чтобы выполнить поставленные задачи потребовалось написать отдельную программу на языке программирования C++ (приложение 3), которая запускает модули кодека в необходимом порядке, передавая на кодирование различные данные, сгенерированные псевдослучайным образом. Алгоритм разработанной программы показан на Рис. 18.

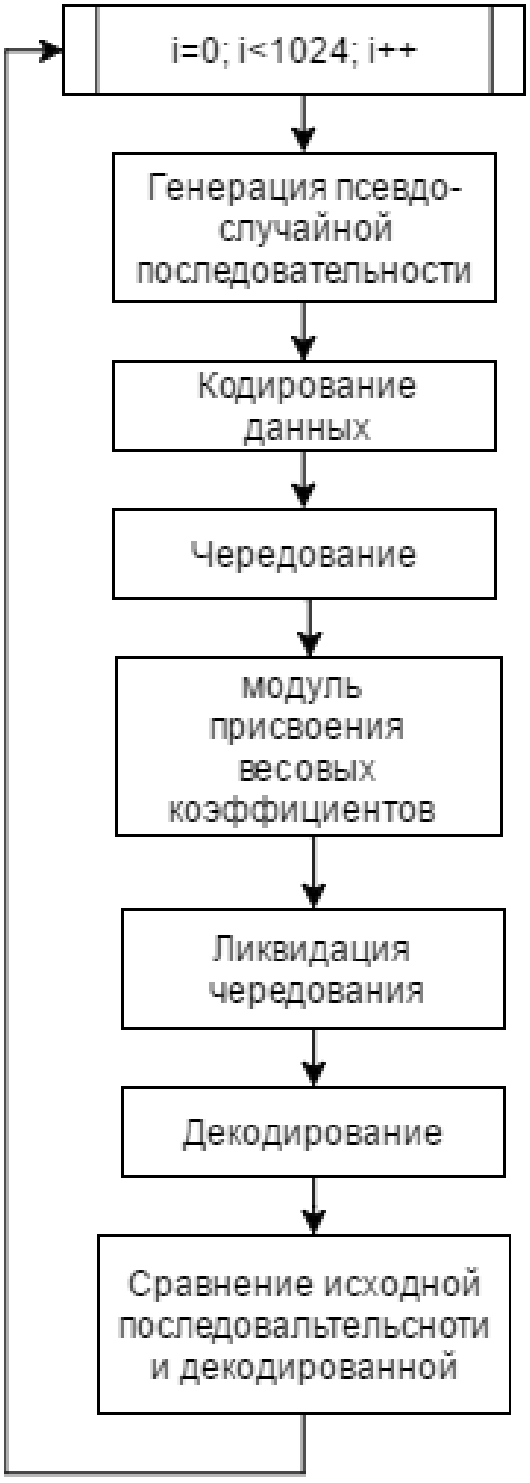

Рис. 18. Алгоритм тестирования кодека

Работа программы происходит следующим образом:

В цикле генерируется случайная последовательность нулей и единиц. Затем сгенерированная последовательность передается на вход кодера, где она подвергается кодированию и чередованию. После кодирования данные готовы

для отправки в канал. Модуль весовых коэффициентов, присваивающий однозначные веса передаваемым битам, написан на языке программирования C++ (приложение 4). После получения данных из «канала», происходит ликвидация чередования и декодирование данных. По завершению декодирования полученная и искомая последовательности сравниваются. В случае если они не совпали, номер теста и ключ последовательности выводятся на экран. Анализируя провалившиеся тесты с помощью отладчика и симулятора выполняется поиск и исправление допущенных ошибок в коде.

Результатом первого этапа тестирования стало обнаружение одной ошибки в реализации на языке Си. После ее исправления, тестирование было проведено повторено (с добавлением неудавшихся тестов) и ошибок не выявило.

На втором этапе тестирования необходимо было подтвердить возможность взаимодействия старого и нового устройства. В виду того, симулятор невозможно включить в цикл тестирования (т.к. CCS является GUI приложением), было проведено тестирование случайных 50 последовательностей. Данные кодировались кодером с прошивки, средствами симулятора CCS, и кодером написанном на языке Си, средствами компилятора C++, после чего результаты сравнивались. Различий обнаружено не было, из чего можно сделать вывод, взаимодействие между «старыми» и «новыми» устройствами будет возможно.

3.5. Модернизация кодека

Очевидный недостаток кодека, изъятого из прошивки – это отсутствие промежуточного режима кодирования кодирование трема битами. Наличие такого режима позволит сократить временные затраты на кодирование, передачу и декодирование в целом рядя случаев.

3.5.1. Кодер

Внедрение промежуточного режима должно минимально изменять структуру модулей, поэтому использовались первые три полинома из существующих. При этом, вся модернизация кодера заключается в изменении кол-ва возможных вложенных циклов (см. Рис. 19).

Рис. 19. Алгоритм кодера с промежуточным режимом.

3.5.2. Модули Чередования

Модули прямого и обратного чередования изменениям не подвергаются, так как они не зависят от режима кодирования.

3.5.3. Декодер

Для модернизации декодера необходимо было решить следующие две задачи: во-первых, необходимо было разработать вспомогательную функцию для нового режима, которая должна была бы, аналогично двум другим, рассчитывать веса возможных ошибок для построения решетки Витерби. А во-

вторых, рассчитать переходы по конечному автомату для используемых полиномов.

Для решения первой задачи потребовалось проанализировать алгоритмы двух существ функций. Рассмотрим алгоритм работы этих функций на примере функции режима 2. На вход поступает последовательность из двух бит, что является одним закодированным битом. Каждый принятый бит представляется 8 битами: один – под значение, семь – под весовой коэффициент, присвоенный после принятия бита из канала передачи. Функция разделяет значения и веса. Значения обоих бит конкатенируются и образуют *дугу перехода* (со значением от 0 до 3). На основе выделенных весов рассчитываются все варианты ошибок. В таблице 1 представлен пример расчета для режима 2.

Таблица 1. Пример расчета веса ошибок

1 бит	2 бит	Итоговый вес:
-	-	Вес 1 бита + Вес 2 бита
-	+	Вес 1 бита - Вес 2 бита
+	-	Вес 2 бита - Вес 1 бита
+	+	- Вес 1 бита - Вес 2 бита

Знак «+» означает, что бит принят верно, знак «-» - бит принят с ошибкой. После того, как все варианты ошибок были рассчитаны, в решетке выполняется полный перебор возможных путей. Полученная раннее *дуга перехода* сравнивается с возможными переходами из каждого текущего состояния решетки. Если один (или два) бита не совпадают, то к пути добавляется рассчитанное значение ошибки. Если же оба бита совпали – то от веса всего пути отнимается вес правильных бит. Такая организация работы цикла, построения решетки, позволяет сократить внутренние вычисления, так как вместо того, чтобы считать веса для каждого конкретного случая, вычисления проводятся вне цикла один раз.

Для решения второй задачи пришлось так же проанализировать последовательности переходов по конечному автомату для режима 2 и 4. Конечные автоматы для всех трех случае одинаковые, однако числа, которыми закодированы дуги – отличаются. Из анализа чисел стало понятно, что каждый бит числа – это результат работы одного из полиномов. Это позволило рассчитать веса дуг переходов для используемых полиномов.

Структура алгоритма при этом никак не изменилась.

По завершению разработки было проведено тестирование как уже существующих режимов, на случай если возникли непредвиденные ошибки при разработке, так и внедрённого режима. В результате было выявлено две ошибки внутри разработанного режима. После их исправления – тесты подтвердили работоспособность нового режима. Для тестирования использовалась программа из пункта 3.4.

4. Тестирование декодирующих способностей

В завершении разработка, было необходимо проверить и подтвердить восстанавливающие способности разработанного декодера и определить граничные значения восстанавливающих способностей.

4.1. Разработка программы тестирования

Для проведения тестирования, программа тестирования из пункта 3.4 (приложение 3) была модифицирована (приложение 6), а модуль присваивающий однозначные весовые коэффициенты (приложение 4) был заменен на модуль имитирующий шум канала (приложение 5).

4.1.1. Алгоритм программы

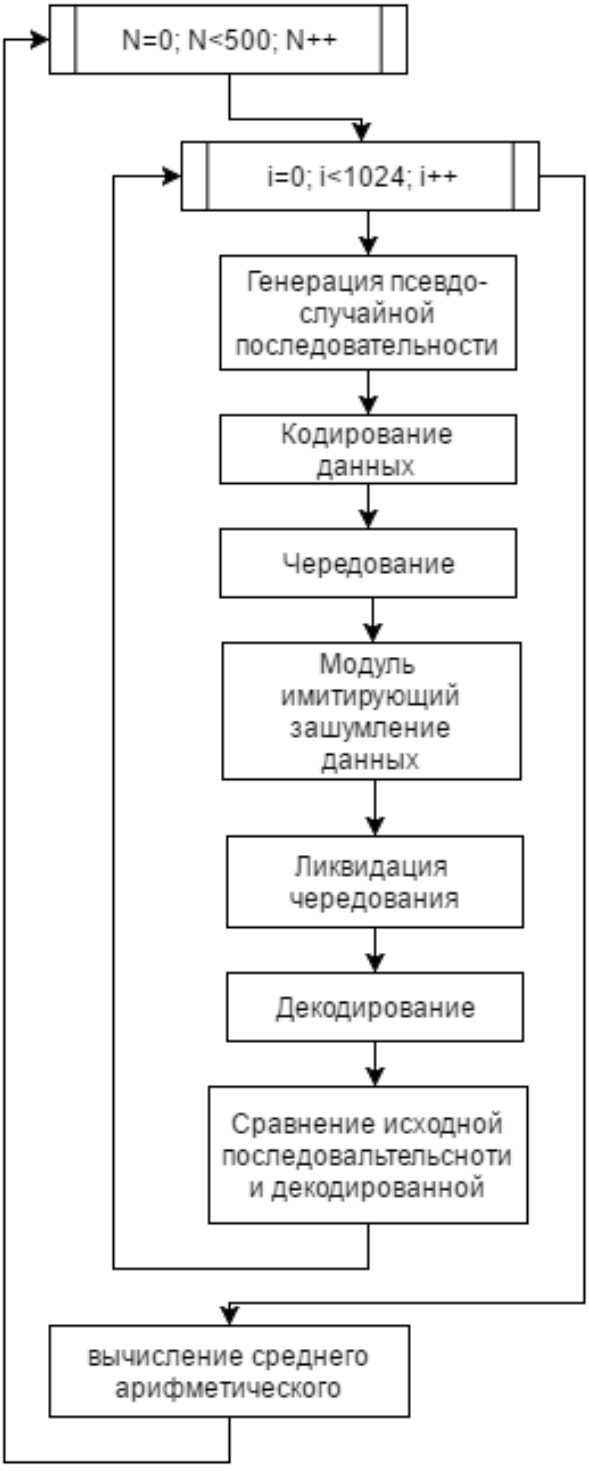

Рис. 20. Алгоритм тестирования декодирующих способностей

Алгоритм программы представлен на Рис. 20. Внутренний цикл генерирует случайную последовательность, после она кодируются. В закодированное сообщение вносится N ошибок (определяется значением из

внешнего цикла), затем декодируется и анализируется. На основе информации полученной, о количестве допущенных ошибок на 1024 опыта для N ошибок, высчитывается среднее арифметическое с округлением в большую сторону. Результат вычисления записывается в файл. После этого количество вносимых ошибок (N) увеличивается и опыт повторяется 500 раз (больше ошибок вносить смысла нет, т.к. ни один из трех режимов правильно декодировать сообщение не сможет).

На основе готового файла можно построить кривую, показывающую зависимость восстанавливающих способностей от количества допущенных ошибок.

4.1.2. Модуль, имитирующий шум в канале

Модуль имитирующий шум имеет три входных параметра, задаваемых перед запуском модуля: количество зашумлённых участков в передаваемой посылке, длина первого зашумленного участка (если участков n, каждый следующий будет на (100/n) % короче первого) и расстояние между зашумленными участками (в случае, если данный параметр не указан – зашумленные участки создаются на максимальном расстоянии друг от друга). Во избежание получения однотипных результатов, модуль содержит ряд случайно сгенерированных внутренних параметров (например, расстояние, от начала посылки, на котором генерируется первое зашумление). Биты, которые необходимо подвергнуть зашумлению – инвертируются и им присваивается псевдослучайный вес.

4.2. Постановка эксперимента.

Для тестирования восстанавливающих способностей декодера были проведены эксперименты со настройками модуля-шума, приведенными в таблице №2.

Таблица 2. Настройки модуля шума для трех групп опытов

№	Кол-во зашумленных участков	Кол-во испорченных бит	Расстояние между ошибками
1	1	1-492	-
2	2	1-360; 1-180;	0-120
3	3	1-240; 1-120; 1-60;	0-120

В результате проведенных опытов зависимость восстанавливающих способностей от количества зашумленных участков не выявилась. Результаты всех опытов оказались схожи, и зависят только от количества ошибочных бит. В связи с этим на Рис. 21 предоставлен результат только для первой группы опытов.

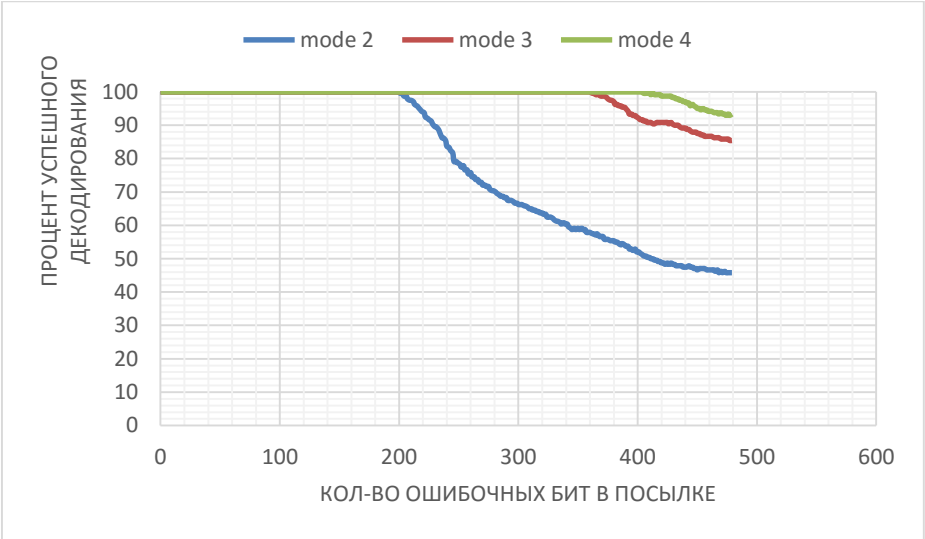

Рис. 21. Результат тестирования декодирующих способностей.

На Рис. 21 предоставлена визуальная характеристика декодирующих способностей различных режимов кодирования.

Эксперимент был поставлен следующим образом: генерируется порядка тысячи случайных последовательностей по 240 бит каждая. Последовательности кодируются, длина закодированного сообщения при этом составляет 1440; 1680; 1920. Такое увеличение обусловлено работой модуля

чередования (т.к. модуль чередования смешивает несколько сообщений сразу, или если это единственное сообщение – смешивает с нулевым пакетом, 960 нулевых бит). После чередования в каждую из закодированных посылок вносится N ошибок и сообщения декодируются. По завершению декодирования все последовательности сравниваются с исходными. В случае, если исходная и декодированная последовательность не совпали – подсчитывает кол-во несовпадений. Затем подсчитывается среднее арифметическое всех ошибок (с округлением в большую сторону). На основе среднеарифметического вычисляется % успешного декодирования.

4.3. Результаты

Режим 2 способен исправить до 201 ошибочных бит из 1440 бит передаваемого сообщение (240 * 2 + 960 бит нулевого пакета), что соответствует 13,9% исправления ошибок.

Режим 3, добавленный в ходе разработки, способен удачно декодировать сообщение при 360 ошибках из 1680 передаваемых бит (240*3+ нулевой пакет), что соответствует 21,4%.

Режим 4, удачно декодирует сообщение при немного большем, чем 400 ошибок из 1920 передаваемых бит, то есть при 20,8%.

Как видно, режим 3 не уступает в декодирующих способностях режиму 4, но выигрывает по скорости в 1,5 раза (нулевые пакеты не учитывают, в связи их бесконечно малым размером при полноценной передаче). Очевидно, что разработанный режим эффективен. Он позволит сократить затраты на кодирование, передачу и декодирование на 30-35%, для случаев где количество ошибочных бит колеблется от 200 до 360 бит на передаваемое сообщение. Таким образом, внедренный режим практически полностью вытесняет режим 4, за исключением тех случаев, когда количество ошибок на передаваемое сообщение изменяется от 360 до 400 бит.

Заключение

В ходе работы были выполнены следующие задачи:

- Было проведено дизассемблирование прошивки устройства, базирующегося на микропроцессоре TMS320C54.
- В полученном коде были определены функции, принадлежащие кодеку.
- Был проведен тщательный анализ искомого кодека.
- На основе полученных знаний, был разработан и внедрен промежуточный режим кодирования, показывавший свою эффективности уже при первых тестах.
- Кодек был реализован на языке программирования Си и оформлен в виде библиотеки.

Разработанная библиотека будет применятся для создания прошивок новых устройств, базирующихся как на основе TMS320C54, так и на основе любых других микропроцессоров.

Структура разработанного кодека позволяет, в случае необходимости, внедрить дополнительные режимы, отличающиеся полиномами кодирования или количеством используемых бит для кодирования.

Список использованных источников

1. Техническое описание микропроцессора TMS320C54: [Электронный документ]. (http://www.ti.com.cn/cn/lit/ug/spru172c/spru172c.pdf). Проверено 29.05.2016.
2. Статья «Алгоритм Витерби» на сайте «nlpub.ru»: [Электронный документ]. (https://nlpub.ru/Алгоритм_Витерби). Проверено 29.05.2016.
3. Статья «Hiew» на сайте «Wikipedia.org»: [Электронный документ]. (https://ru.wikipedia.org/wiki/Hiew). Проверено 29.05.2016.
4. Статья «IDA Pro - интерактивный дизассемблер» на сайте «www.idasoft.ru»: [Электронный документ]. (http://www.idasoft.ru/idapro/). Проверено 29.05.2016.
5. Интервью с Ильфаком Гильфановым, разработчиком IDA: [Электронный документ]. (http://fcenter.ru/online/softarticles/interview/6704). Проверено 29.05.2016.
6. Официальный сайт фирмы «Texas Instrument»: [Электронный документ]. (http://www.ti.com/). Проверено 29.05.2016.
7. Code Composer Studio (CCS) Integrated Development Environment (IDE): [Электронный документ]. (http://www.ti.com/tool/ccstudio). Проверено 29.05.2016.
8. Описание сигнатуры dat-файла: [Электронный документ]. (https://e2e.ti.com/support/development_tools/code_composer_studio/f/81/t/277303). Проверено 29.05.2016.
9. И.В.Сталин в беседе с Б. Куном; С. 190.

Приложение 1

Листинг 1. Файл convertor.h

```
#ifndef __CONVERTOR__
#define __CONVERTOR__

#include <stdio.h>

#endif
```

Листинг 2. Файл convertor.cpp

```
#include "convertor.h"

void convert(char * IDA_file, char * CCS_file)
{
	int bit = 0x0;
	FILE *IDA = fopen(IDA_file, "rb");
	FILE *CCS = fopen(CCS_file, "wb");

	fprintf(CCS, "1651 1 0 0 ffff\n");

	do{
		fscanf(IDA, "%x", &bit);
		fprintf(CCS, "0x%x\n", bit);
	} while (!feof(IDA));

	fclose(CCS);
	fclose(IDA);
}

int main(){
	printf("Converting starting...\n");
	convert("IDA.txt", "CCS.dat");
	printf("Proccess was finished!\n");
	getchar();
	return 0;
}
```

Приложение 2

Листинг 3. Файл codec.h

```
#ifndef __CODEC__
#define __CODEC__

      #include <math.h>
      #include <stdlib.h>

      int coder(int mode, unsigned int *memory_0xA00, unsigned
int *memory_0x600, unsigned int length);

      void interliving(unsigned int *memory_0x600, int length);
      void deinterliving(unsigned int *memory_0x600, int length);

      void Viterbi(unsigned int *inData, unsigned int *outDate,
int mode, int length);

#endif
```

Листинг 4. Файл codec.cpp

```
#include "codec.h"

int coder(int mode, unsigned int *memory_0xA00, unsigned int
*memory_0x600, unsigned int length)
      {
            int accumulatorA = 0;
            int accumulatorB = 0;
            int brc = mode;
            int ar0 = 0;
            int regT = 0;
            int count_bits = 0, i = 0, j = 0;

            int tableRepliesHalf[8][2] =   {{ 42330, 27030 },
                                           { 42330, 27030 },
                                           { 23205, 27030 },
                                           { 23205, 27030 },
                                           { 23205, 38505 },
                                           { 23205, 38505 },
                                           { 42330, 38505 },
                                           { 42330, 38505 } };

      int tableRepliesThree[8][3] = { { 42330, 27030, 42330, }
                                    { 42330, 27030, 23205 },
                                    { 23205, 27030, 42330 },
                                    { 23205, 27030, 23205 },
                                    { 23205, 38505, 23205 },
                                    { 23205, 38505, 42330 },
                                    { 42330, 38505, 23205 },
```

```
                                    { 42330, 38505, 42330 } };

int tableRepliesQuater[8][4] = {{ 42330, 42330, 26214, 27030 },
                                { 23205, 42330, 39321, 27030 },
                                { 42330, 23205, 39321, 27030 },
                                { 23205, 23205, 26214, 27030 },
                                { 23205, 23205, 39321, 38505 },
                                { 42330, 23205, 26214, 38505 },
                                { 23205, 42330, 26214, 38505 },
                                { 42330, 42330, 39321, 38505 } };

            switch (mode){
            case(2) :
                  for (i = 0; i < length + 6; i++){
                        accumulatorA = accumulatorA << 1;
                        accumulatorA = accumulatorA & 127;
                        accumulatorA |= memory_0xA00[i];
                        ar0 = (accumulatorA >> 4) & 7;
                        regT = accumulatorA & 15;
                        for (j = 0; j<brc; j++){
                              accumulatorB =
tableRepliesHalf[ar0][j] >> regT;
                              memory_0x600[count_bits] =
accumulatorB & 1;
                              count_bits++;
                        }
                  }
                  break;
            case(3) :
                  for (i = 0; i < length + 6; i++){
                        accumulatorA = accumulatorA << 1;
                        accumulatorA = accumulatorA & 127;
                        accumulatorA |= memory_0xA00[i];
                        ar0 = (accumulatorA >> 4) & 7;
                        regT = accumulatorA & 15;
                        for (j = 0; j<brc; j++){
                              accumulatorB =
tableRepliesThree[ar0][j] >> regT;
                              memory_0x600[count_bits] =
accumulatorB & 1;
                              count_bits++;
                        }
                  }
                  break;
            case(4) :
                  for (i = 0; i < length + 6; i++){
                        accumulatorA = accumulatorA << 1;
                        accumulatorA = accumulatorA & 127;
                        accumulatorA |= memory_0xA00[i];
                        ar0 = (accumulatorA >> 4) & 7;
                        regT = accumulatorA & 15;
                        for (j = 0; j<brc; j++){
```

```
                              accumulatorB =
tableRepliesQuater[ar0][j] >> regT;
                              memory_0x600[count_bits] =
accumulatorB & 1;
                              count_bits++;
                        }
                  }
                  break;
            }
            return count_bits;
      }

void interliving(unsigned int *memory_0x600, int length){
      int CountRow = length / 120 + 15;
      int i,
            row = 0;
      do{
            i = row;
            for (int k = 0; k < 120; k++){
                  if (i - row == 8){
                        i = row;
                  }
                  memory_0x600[120 * row + k] = memory_0x600[120
* i + k];
                  i++;
            }
            row++;
      } while (row < CountRow - 7);

      int index = 0;
      for (int i = 0; i < CountRow - 7; i++){
            for (int j = 0; j < 120; j++){
                  memory_0x600[120 * (CountRow - 1) + j] =
memory_0x600[120 * i + index];
                  index = index + 10;
                  if (index > 119)
                        index = index - 119;
            }
            index = 0;
            for (int j = 0; j < 120; j++){
                  memory_0x600[120 * i + j] = memory_0x600[120 *
(CountRow - 1) + j];
            }
      }
}
void deinterliving(unsigned int *memory_0x600, int length){
      int CountRow = length / 120 + 7;
      int index = 0;
      for (int i = 0; i < CountRow - 7; i++){
            for (int j = 0; j < 120; j++){

                  memory_0x600[120 * (CountRow - 1) + index] =
```

```
memory_0x600[120 * i + j];
                  index = index + 10;
                  if (index > 119)
                        index = index - 119;
            }
            index = 0;
            for (int j = 0; j < 120; j++){
                  memory_0x600[120 * i + j] = memory_0x600[120 *
(CountRow - 1) + j];
            }
      }

      int i,
            row = CountRow - 8;
      do{
            i = row;
            for (int k = 0; k < 120; k++){
                  if (i - row == 8){
                        i = row;
                  }
                  memory_0x600[120 * i + k] = memory_0x600[120 *
row + k];
                  i++;
            }
            row--;
      } while (row >= 0);
}

int callA(unsigned int *inData, int *k, int *ar3);
int callA_3(unsigned int *inData, int *k, int *ar3);
int callA_4(unsigned int *inData, int *k, int *ar3);

void Viterbi(unsigned int *inData, unsigned int *outDate, int
mode, int length){

      int byte6E = -1;

      int bias[32] = { 3, 1, 3, 1, 0, 2, 0, 2, 0, 2, 0, 2, 3, 1,
3, 1, 2, 0, 2, 0, 1, 3, 1, 3, 1, 3, 1, 3, 2, 0, 2, 0 };

      int bias_3[32] = { 7, 3, 6, 2, 0, 4, 1, 5, 0, 4, 1, 5, 7,
3, 6, 2, 5, 1, 4, 0, 2, 6, 3, 7, 2, 6, 3, 7, 5, 1, 4, 0 };

      int bias_4[32] = { 15, 9, 5, 3, 2, 4, 8, 14, 2, 4, 8, 14,
15, 9, 5, 3, 12, 10, 6, 0, 1, 7, 11, 13, 1, 7, 11, 13, 12, 10,
6, 0 };

      length /= mode;


      switch (mode){
      case(2) :
```

```
            byte6E = 63;
            break;
      case(3) :
            byte6E = 63;
            for (int i = 0; i < 32; i++){
                  bias[i] = bias_3[i];
            }
            break;
      case(4) :
            byte6E = 31;
            for (int i = 0; i < 32; i++){
                  bias[i] = bias_4[i];
            }
            break;
      }

      int byte_68 = 0;
      unsigned int *TRN = (unsigned int*)calloc(length * 4,
sizeof(int));

      int state[128];
      state[0] = 0;

      for (int j = 1; j<64; j++)
            state[j] = -1000;

      int B, T, ar2,
            byte_64,
            k = 0;

      int x1, x2, y1, x_cnt, y_cnt, z_cnt;
      int ar3[16];
      z_cnt = 0;

      for (int ar0 = 0; ar0 < length; ar0++){
            switch (mode){
            case(2) :
                  byte_64 = callA(inData, &k, ar3);
                  break;
            case(3) :
                  byte_64 = callA_3(inData, &k, ar3);
                  break;
            case(4) :
                  byte_64 = callA_4(inData, &k, ar3);
                  break;
            }

            if (ar0 % 2 == 0){

                  x1 = 64; x2 = 96; y1 = 0;
            }
            else{
```

```
                  x1 = 0; x2 = 32; y1 = 64;
            }

            y_cnt = 0;
            x_cnt = 0;

            for (int ar1 = 0; ar1<4; ar1++){
                  for (int brc = 0; brc<8; brc++){

                        B = byte_64 ^ bias[x_cnt];
                        T = ar3[B];


                        if (state[y1 + y_cnt] + T > state[y1 +
y_cnt + 1] - T) {
                              state[x1 + x_cnt] = (state[y1 +
y_cnt] + T);
                              TRN[z_cnt] = TRN[z_cnt] << 1;
                              TRN[z_cnt] = TRN[z_cnt] & 0xFFFF;
                        }
                        else {
                              state[x1 + x_cnt] = (state[y1 +
y_cnt + 1] - T);
                              TRN[z_cnt] = (TRN[z_cnt] << 1) + 1;
                              TRN[z_cnt] = TRN[z_cnt] & 0xFFFF;
                        }

                        if (state[y1 + y_cnt] - T > state[y1 +
y_cnt + 1] + T) {
                              state[x2 + x_cnt] = state[y1 +
y_cnt] - T;
                              TRN[z_cnt] = TRN[z_cnt] << 1;
                              TRN[z_cnt] = TRN[z_cnt] & 0xFFFF;
                        }
                        else {
                              state[x2 + x_cnt] = (state[y1 +
y_cnt + 1] + T);
                              TRN[z_cnt] = (TRN[z_cnt] << 1) + 1;
                              TRN[z_cnt] = TRN[z_cnt] & 0xFFFF;
                        }
                        y_cnt += 2;
                        x_cnt++;
                  }
                  z_cnt++;
            }
            y_cnt = 0;
            x_cnt = 0;
      }

      int A = 0;
      int   ar4 = 0x0,
            byte_66 = 0,
```

```
            ar7 = 983,
            mask = 0,
            TC = 0;

      int jump[64] = { 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22,
24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48,
            50, 52, 54, 56, 58, 60, 62, 1, 3, 5, 7, 9, 11, 13,
15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35,
            37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61,
63 };

      for (int brc = length - 1; brc>-1; brc--){
            B = ar4;
            if (B - 0x20 >= 0)
                  outDate[brc] = 1;
            else
                  outDate[brc] = 0;
            B = (A >> 3) & 3;
            byte_66 = B;
            B = B ^ 3;
            ar7 = ar7 - B;
            A = jump[ar4];
            T = A & 0xF;
            mask = pow(2.0, (0xF - T));
            TC = (TRN[ar7] & mask) >> (0xF - T);

            if (ar4 - 0x20 >= 0){
                  if (TC == 0)
                        A = A - 1;
            }
            else{
                  if (TC == 1)
                        A = A + 1;
            }
            ar7 = ar7 + ~byte_66 + 1;
            ar4 = A;

            ar7--;
      }
}

int callA(unsigned int *inData, int *k, int *ar3){
      int A, B;
      A = inData[*k];
      B = (A >> 6) & 2;

      int byte_60 = A & 127;

      A = inData[*k + 1];
      int byte_64 = B + (A >> 7);

      int byte_61 = A & 127;
```

```
	ar3[3] = byte_61 + byte_60;
	ar3[0] = ~ar3[3] + 1;
	ar3[2] = byte_60 - byte_61;
	ar3[1] = ~ar3[2] + 1;

	*k += 2;
	return byte_64;
}

int callA_3(unsigned int *inData, int *k, int *ar3){
	int A, B = 0;
	int byte_mark[3];

	A = inData[*k];
	B += (A >> 5) & 12;
	byte_mark[0] = A & 127;
	*k += 1;

	A = inData[*k];
	B += (A >> 6) & 14;
	byte_mark[1] = A & 127;
	*k += 1;

	A = inData[*k];
	B += (A >> 7) & 15;
	byte_mark[2] = A & 127;
	*k += 1;

	int byte_64 = B;

	ar3[0] = -byte_mark[0] - byte_mark[1] - byte_mark[2];
	ar3[1] = -byte_mark[0] - byte_mark[1] + byte_mark[2];
	ar3[2] = -byte_mark[0] + byte_mark[1] - byte_mark[2];
	ar3[3] = -byte_mark[0] + byte_mark[1] + byte_mark[2];
	ar3[4] = byte_mark[0] - byte_mark[1] - byte_mark[2];
	ar3[5] = byte_mark[0] - byte_mark[1] + byte_mark[2];
	ar3[6] = byte_mark[0] + byte_mark[1] - byte_mark[2];
	ar3[7] = byte_mark[0] + byte_mark[1] + byte_mark[2];

	return byte_64;
}

int callA_4(unsigned int *inData, int *k, int *ar3){
	int A, B = 0;

	int byte_mark[4];

	A = inData[*k];
	B += (A >> 4) & 8;
	byte_mark[0] = A & 127;
	*k += 1;
```

```
      A = inData[*k];
      B += (A >> 5) & 12;
      byte_mark[1] = A & 127;
      *k += 1;

      A = inData[*k];
      B += (A >> 6) & 14;
      byte_mark[2] = A & 127;
      *k += 1;

      A = inData[*k];
      B += (A >> 7) & 15;
      byte_mark[3] = A & 127;
      *k += 1;

      int byte_64 = B;

      ar3[15] = byte_mark[0] + byte_mark[1] + byte_mark[2] +
byte_mark[3];
      ar3[0] = ~ar3[15] + 1;

      ar3[14] = byte_mark[0] + byte_mark[1] + byte_mark[2] -
byte_mark[3];
      ar3[1] = ~ar3[14] + 1;

      ar3[3] = ar3[1] + byte_mark[2] + byte_mark[2];
      ar3[12] = ~ar3[3] + 1;

      ar3[13] = ar3[12] + byte_mark[3] + byte_mark[3];
      ar3[2] = ~ar3[13] + 1;

      ar3[11] = byte_mark[0] - byte_mark[1] + byte_mark[2] +
byte_mark[3];
      ar3[4] = ~ar3[11] + 1;

      ar3[10] = byte_mark[0] - byte_mark[1] + byte_mark[2] -
byte_mark[3];
      ar3[5] = ~ar3[10] + 1;

      ar3[7] = ar3[5] + byte_mark[2] + byte_mark[2];
      ar3[8] = ~ar3[7] + 1;

      ar3[9] = ar3[8] + byte_mark[3] + byte_mark[3];
      ar3[6] = ~ar3[9] + 1;

      return byte_64;
}
```

Приложение 3

Листинг 5. Файл work_test.cpp

```
#include "codec.h"
#include "wt_function.h"
#include <fstream>

int main(int argc, char *argv[])
{
	int in_bit = 240,
		out_bit,
		mode = atoi(argv[1]),
		count_bits_from_chanel;

	switch (mode){
	case(2) :
		out_bit = 492;
		count_bits_from_chanel = 1440;
		break;
	case(3) :
		out_bit = 738;
		count_bits_from_chanel = 1680;
		break;
	case(4) :
		out_bit = 984;
		count_bits_from_chanel = 1920;
		break;
	}

	unsigned int memory[0xFFFF] = { 0 };
	int keyword = 0, shift = 0;

	int i,
		error = 0,
		count_tests = 1024,
		average_error = 0,
		count_error = 0;

	int length_message;

		for (int o = 0; o < count_tests; o++){
			for (i = 0; i< 240; i++){
				memory[i] = (keyword >> shift) & 1;
				shift++;
				if (shift>10)
					shift = 0;
			}

			length_message = coder(mode, &memory[0x0],
&memory[0x1000], in_bit);
			interliving(&memory[0xCB8], length_message);
```

```
                  wt_function(&memory[0xCB8],
count_bits_from_chanel);

                  deinterliving(&memory[0xCB8], 1920);

                  Viterbi(&memory[0x1000], &memory[0x2000], mode,
out_bit);

                  for (i = 0; i< 240; i++){
                        if (memory[i] != memory[0x2000 + i])
                              error++;
                  }
                  if (error > 0){
                        cout << "Error! Number of mistakes = " <<
error << ". Bad keyword = " << keyword << endl;
                        count_error++;
                  }
                  else{
                        cout << "It all ok " << keyword << endl;
                  }
                  keyword++;
            }

      cout << "Count of error: " << count_error << " out of " <<
count_tests << " Corl!" << endl;
      getchar();
      return 0;
}
```

Приложение 4

Листинг 6. Файл wt_function.h

```
#ifndef __NOISE_F__
#define __NOISE_F__

#include <iostream>
using namespace std;

      void wt_function(unsigned int *memory_0x600, int length);

#endif
```

Листинг 7. Файл wt_function.cpp

```
#include "wt_function.h"

void wt_function(unsigned int *memory_0x600, int length){

      do{
            if (memory_0x600[length] == 1)
                  memory_0x600[length] = 0xBF;
            else
                  memory_0x600[length] = 0x3F;
            length--;
      } while (length >= 0);
}
```

Приложение 5

Листинг 8. Файл noise_function.h

```
#ifndef __NOISE_F__
#define __NOISE_F__

#include <iostream>
using namespace std;

      void noise_function(int noise, int k_noise, int nois_dist,
int mode, unsigned int *memory_0x600);

#endif
```

Листинг 9. Файл noise_function.cpp

```
#include "noise_function.h"

void noise_function(int noise, int k_noise, int nois_dist, int
mode, unsigned int *memory_0x600){

      int CountRow = 12;
      switch (mode)
      {
      case(2) :
            CountRow = 12;
            break;
      case(3) :
            CountRow = 14;
            break;
      case(4) :
            CountRow = 16;
            break;
      default:
            break;
      }

      int count_noise, dynamic_noise, dynamic_step,
            count_dist, dist;
      bool can_break = true;
      int i, j;


      dynamic_noise = k_noise;
      count_noise = dynamic_noise;
      if (noise > 0){
            dynamic_step = k_noise / noise;

            dist = CountRow * 120 / (noise);
            if (dist > nois_dist)
                  dist = nois_dist;
```

```
            count_dist = dist;

      }
      else
      {
            can_break = false;
            dynamic_step = 0;

            dist = CountRow * 120;
            count_dist = dist;
      }


      for (i = 0; i < CountRow; i++){
            for (j = 0; j < 120; j++){
                  if (memory_0x600[120 * i + j] == 1)
                        memory_0x600[120 * i + j] = 0xBF;
                  else
                        memory_0x600[120 * i + j] = 0x3F;
            }
      }

      int random_int = rand();
      int ri_120 = (random_int % 120);
      noise--;

      for (i = 0; i < CountRow; i++){
            for (j = ri_120; j < 120; j++){

                  count_dist--;
                  if (can_break){
                        if (memory_0x600[120 * i + j] == 0xBF){
                              memory_0x600[120 * i + j] = 0x15;

                        }
                        else{
                              memory_0x600[120 * i + j] = 0x95;

                        }
                        if (count_noise <= 0){
                              can_break = false;

                              dynamic_noise = dynamic_noise -
dynamic_step;
                              count_noise = dynamic_noise;

                        }
                        else{
                              count_noise--;
                        }
                  }
                  else
```

```
                    {
                        if (count_dist <= 0 && noise > 0){
                            count_dist = dist;
                            can_break = true;
                            noise--;
                        }
                    }
                }
                ri_120 = 0;
        }
}
```

Приложение 6

Листинг 10. Файл mode_test.h

```
#include "codec.h"
#include "noise_function.h"
#include <fstream>

int main(int argc, char *argv[])
{
	int in_bit = 240,
		out_bit,
		mode = atoi(argv[1]),
		count_bits_from_chanel;

	switch (mode){
	case(2) :
		out_bit = 492;
		count_bits_from_chanel = 1440;
		break;
	case(3) :
		out_bit = 738;
		count_bits_from_chanel = 1680;
		break;
	case(4) :
		out_bit = 984;
		count_bits_from_chanel = 1920;
		break;
	}

	unsigned int memory[0xFFFF] = { 0 };
	int keyword = 0, shift = 0;

	int i,
		error = 0,
		count_tests = 1024,
		average_error = 0,
		count_error = 492;

	int length_message;

	int nois = 2,
		noise_k,
		nois_dist;

	ofstream fout("result_of_the_test.txt");

	for (int noise_test = 0; noise_test < 492; noise_test++){
		for (int o = 0; o < count_tests; o++){
			for (i = 0; i< 240; i++){
				memory[i] = (keyword >> shift) & 1;
				shift++;
				if (shift>10)
```

```
                              shift = 0;
                  }

                  length_message = coder(mode, &memory[0x0],
&memory[0x1000], in_bit);
                  interliving(&memory[0xCB8], length_message);


                  noise_k = noise_test;

                  nois_dist = rand() % 80;

                  noise_function(nois, noise_k, nois_dist, mode,
&memory[0xCB8]);

                  deinterliving(&memory[0xCB8], 1920);

                  Viterbi(&memory[0x1000], &memory[0x2000], mode,
out_bit);

                  for (i = 0; i< 240; i++){
                        if (memory[i] != memory[0x2000 + i])
                              error++;
                  }
                  keyword++;
            }
            average_error = error / 1024;

            fout << average_error << "    " << noise_test <<
endl;
            error = 0;
            cout << noise_test << endl;

      }
      fout.close();
      return 0;
}
```

Приложение 7

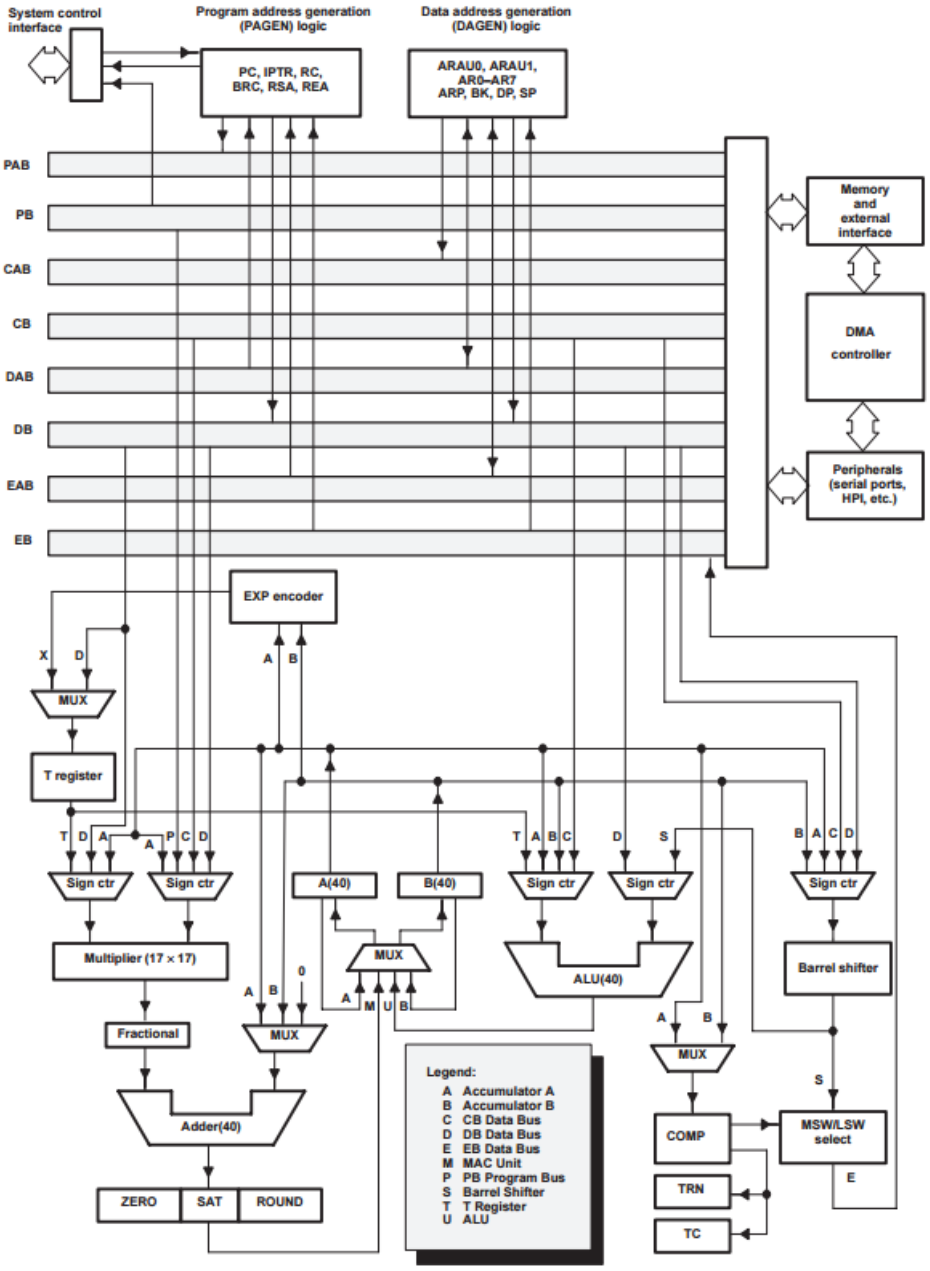

Рис. 22. Функциональная схема TMS320C54

Заключительный лист работы

Выпускная работа выполнена мною самостоятельно. Использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

Список использованных источников содержит _____ наименований.

Работа выполнена на_____ страницах.

Приложения к работе на _____ страницах.

Один экземпляр сдан в директорат.

Подпись_____/_____/

(фамилия, инициалы)

Дата «_____» _____ 20_____г.