Преподу: Основной кусок бака. Читать и ругать только то, что до pause. Сейчас повествования начинается с выбора среды дизассемблирования -> выбор симулятора и связка -> анализ полученного кода –> разработка модулей на языке Си –> тестирование реализации на языке Си

Задача:
Провести обратную разработку кодека, скрытого в прошивке устаревшего устройства на базе TMS32054. Необходимо дизассемблировать прошивку, определить границы кодека, проанализировать алгоритмы и реализовать их на языке Си. Необходимо разработать и внедрить промежуточный режим кодирования (декодирования). По завершению разработки необходимо предоставить результат сравнения всех режимов.

В связи с поставленной задачей был разработан и предложен следующий цикл разработки программного обеспечения:

1) Дизассемблирование прошивки.
2) Анализ и тестирование полученного кода.
3) Разработка модулей кодека на языке Си.
4) Тестирование реализации на языке Си.
5) Модернизация кодека.
6) Тестирование декодирующих способностей разработанного режима.
7) Анализ полученных результатов.

Этап первый. 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. Несмотря на то, что технология является узкопрофильной, ее актуальность технологии бесспорна. В связи с тем, что технология применяется не повсеместно, сред дизассемблирования немного. К наиболее известным и активно используемым средам можно отнести: Sourcer, Hiew и IDA Pro. Каждый из них доказал свое право на существования, однако использоваться будет IDA Pro. Этот выбор обуславливается положительными и отрицательными сторонами всех трех дизассемблеров:

1) *Sourcer* –простейший дизассемблер. Позволяет конвертировать исходный файл в ассемблерный код, однако, примерно в 30% случаях требуется вмешательство программиста для исправления ошибок дизассемблирования, что усложняет отладку большой программы. Так же отсутствуют базы данных и интерактивность интерфейса – это неимоверно сказывается на скорости и качестве обратной разработки.
2) *Hiew* - редактор двоичных файлов, ориентированный на работу с кодом. Имеет встроенный дизассемблер для x86, x86-64 и ARM, ассемблер для x86, x86-64 [1]. Стоит так же отметить, что данный редактор распространяется бесплатно. Однако с момента создания, начале 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.
3) *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 [2]. 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 [3])

На данный момент IDA является лидером среди дизассемблеров на рынке. Позволяет в автоматическом и полуавтоматическом режиме проводить дизассемблирование. Продвинутый уровень интерактивности позволяет 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 данной среды. Это позволяет ее использовать всем желающим. Создателем среды является Ильфак Гильфанов, разработчик с многолетним опытом и специалист по компьютерной безопасности [3]. Из приведенных дизассемблеров, Interactive DisAssembler наиболее подходит для решения поставленной задачи.

Среда не только обладает рядом функциональных преимуществ перед конкурентами, но и поддерживает дизассемблирование программ, написанных для искомого процессора TMS32054. Для дизассемблирования прошивки необходимо выполнить следующие действия:

а. Запустить среду IDA. **File -> New ->Binary File** (Рис. 1). После выбирать файл прошивки (aces.bin);

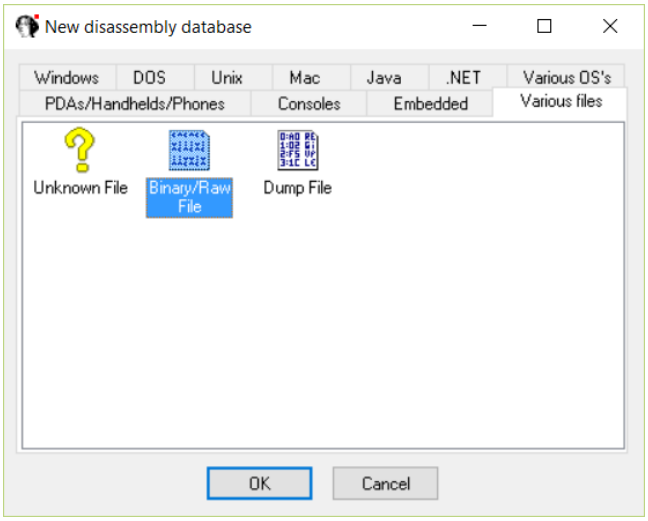

Рис. 1 – Выбор типа входного файла.

б. Затем откроется окно выбора типа процессора. Выбирать **TMS32054** (Рис. 2).

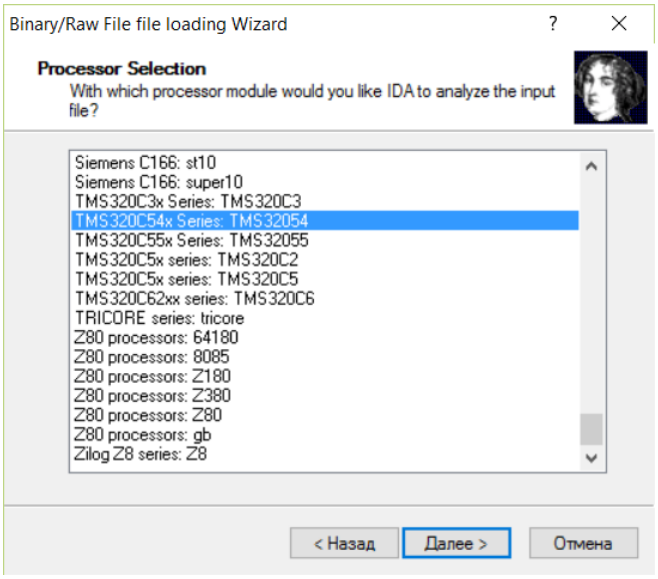

Рис. 2 – Выбираем нужный процессор.

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

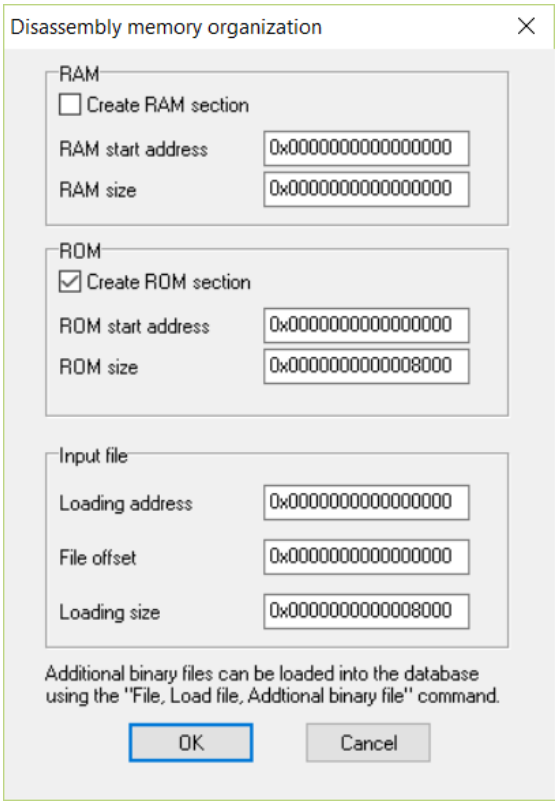

Рис. 3 – Выбираем нужный процессор.

г. Сразу после завершения настроек отроется окно IDA View-A (Рис. 4)

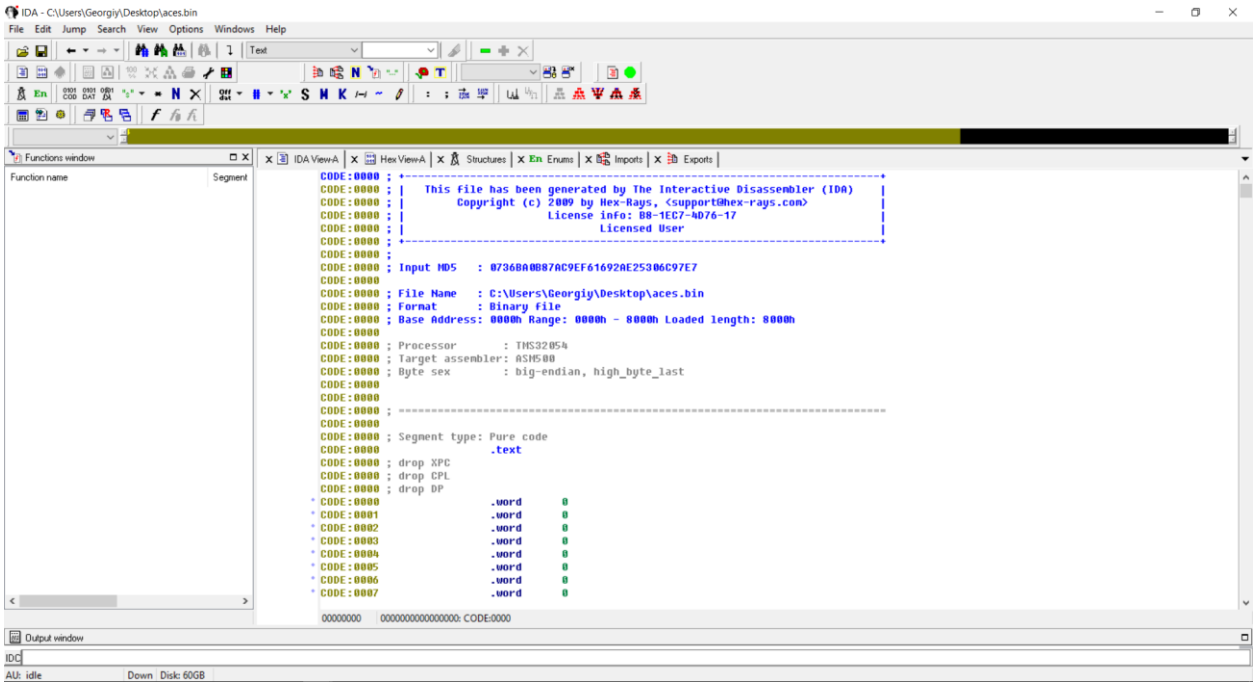

Рис. 4 – Основное окно разработки IDA View.

д. 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 (Рис. 5). 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.

```
CODE:200D                .word  140Ah
CODE:200E                .word  0F073h ; sË
CODE:200F                .word  200Eh
CODE:2010
CODE:2010 ; =============== S U B R O U T I N E =======================================
CODE:2010
CODE:2010 ; Attributes: noreturn
CODE:2010
CODE:2010 sub_2010:                                ; CODE XREF: CODE:5E53↓p
CODE:2010                                          ; CODE:5E56↓p
CODE:2010                adds    *(byte_140A), A
CODE:2012                ld      #1400h, 0, B
CODE:2014                and     #0FFFFh, B
CODE:2016                sub     A, B
CODE:2017                rcd     bgeq
CODE:2018                stl     A, *(byte_140A)
CODE:201A
```

Рис. 5 – Дизассемблированный участок.

Однако, для того, чтобы найти кодек среди всей прошивки, одного статичного анализа недостаточно. Для удобного анализа кода потребуется симулятор микропроцессора. Симулятор позволит анализировать изменения регистров, аккумуляторов и памяти, что упростит первостепенную задачу – задачу обнаружения кодека. В связи с тем, что микропроцессор TMS32054 является разработкой компании Texas Instrument [4], в качестве симулятора будет использоваться

Code Composer Studio [5] – 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. TMS32054 попадает как в платную версию программы, так и в бесплатную.

Среда состоит из двух модулей: ***CCStudio(CCS)*** и ***CCStudio Setup(CCSS)***. Перед использованием симулятора необходимо установить тип симулируемого процессора в CCSS (рис. 6)

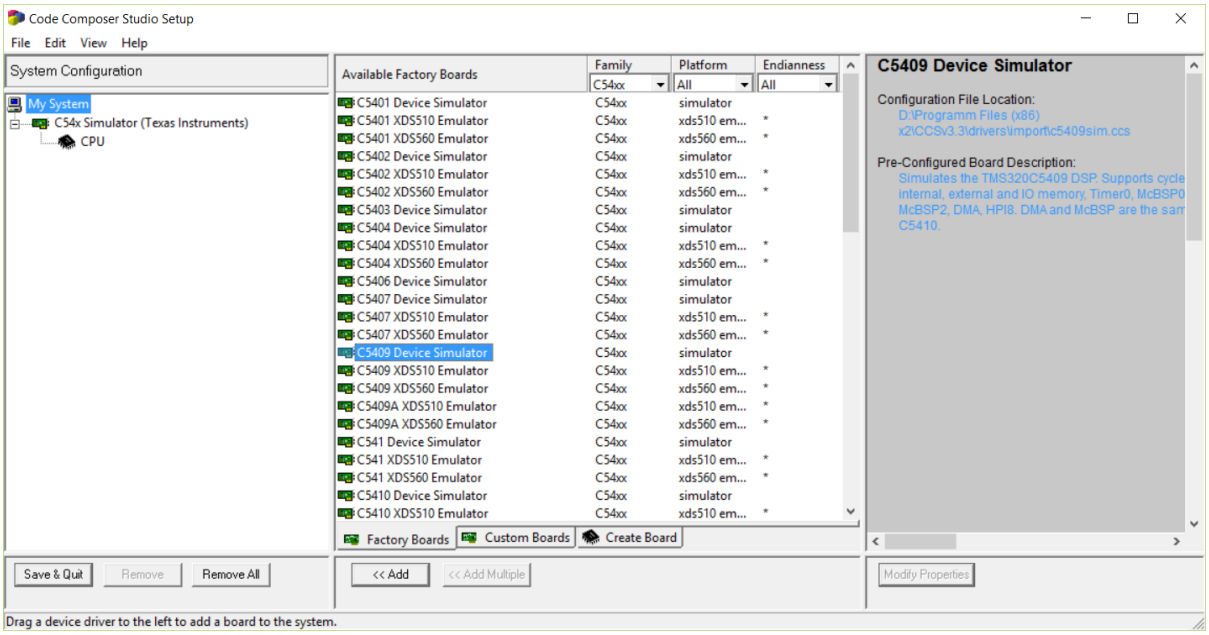

Рис. 6 – Настройка CCS.

Необходимо найти процессор TMS32054 и добавить его к симулируемым **(<<Add**), после чего перезапустить ***CCS*** (**Save&Quit**). Среда перезапуститься и 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 (рис. 7).

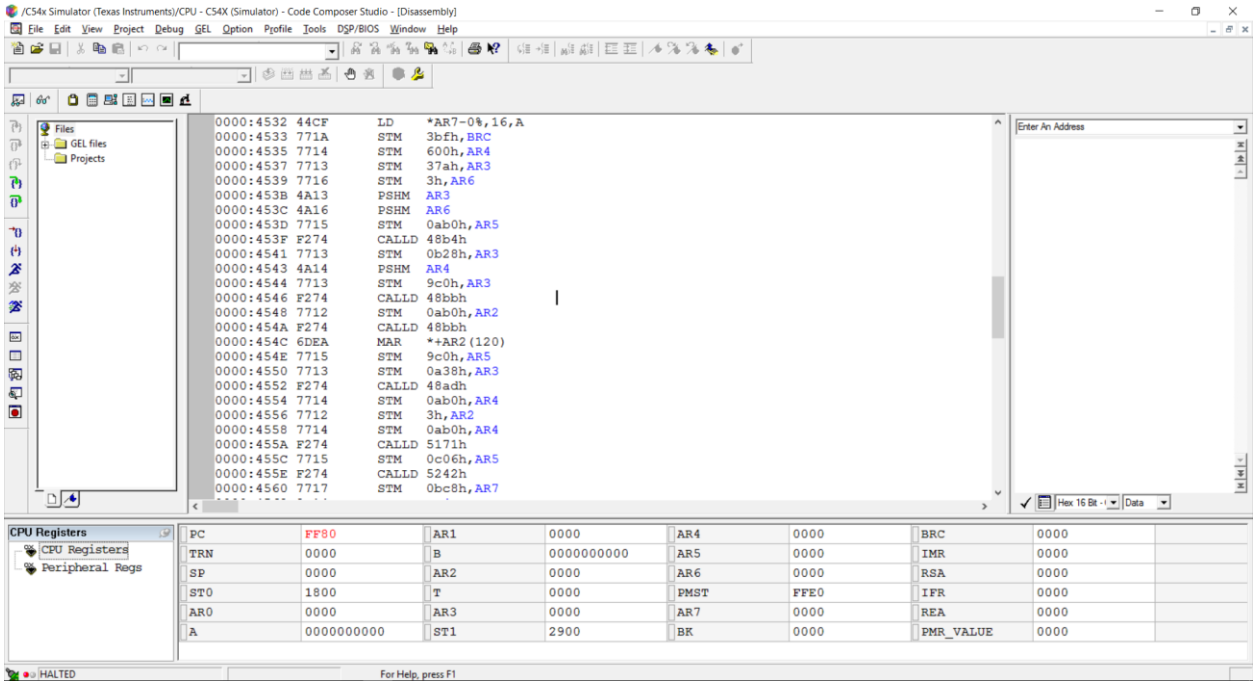

Рис. 7 – Окно Code Composer Studio.

После настройки среды необходимо загрузить саму прошивку и данные в симулятор. Дизассемблированная 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 (Рис. 8), 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) [6].

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

Рис. 8 – Формат dat-файла для CCS.

IDA способна сохранить результат дизассемблирования в 16-битном формате типа 2 (рис. 9).

```
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. 9 – Формат сохранения результата от IDA.

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

Этап второй. Анализ полученного кода.

Перед вторым этапом разработки был поставлен целый ряд непростых задач:

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

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:

1) Состояние памяти, до и после кодирования;
2) 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.

Подтвердить работоспособность 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) оказалось намного проще, нежели всю прошивку целиком. Именно поэтому этап проверки кода идет только после обнаружения границ кодека.

В связи с тем, что IDA поддерживает TMS32054, 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» [7]. 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 - файл с битами, 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 тест 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 следующему этапу.

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.

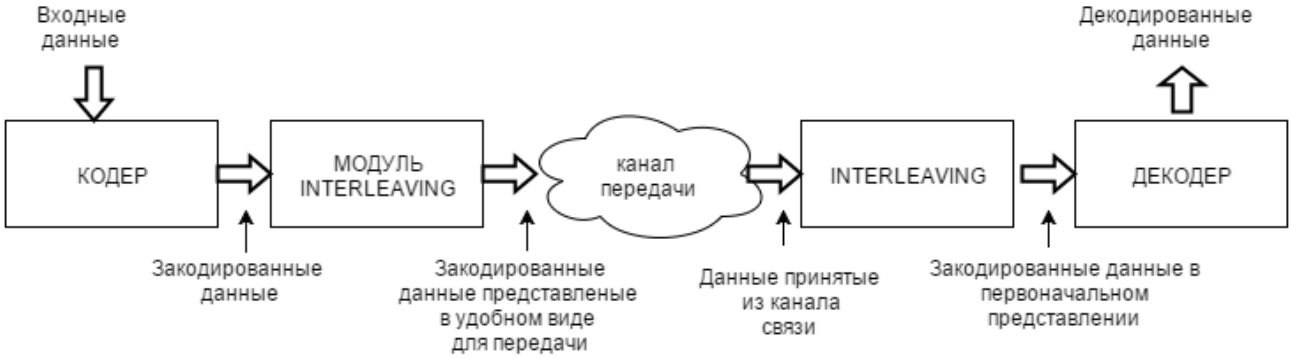

Рис.10. Структурная схема работы кодека.

Структура кодека приведена на рисунке 10. Ниже рассмотрим смысловую часть каждого из модулей.

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, то есть это конечный автомат. Количество возможных комбинаций ограничено 64 возможными, значит в конечном автомате 64 состояния.

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.

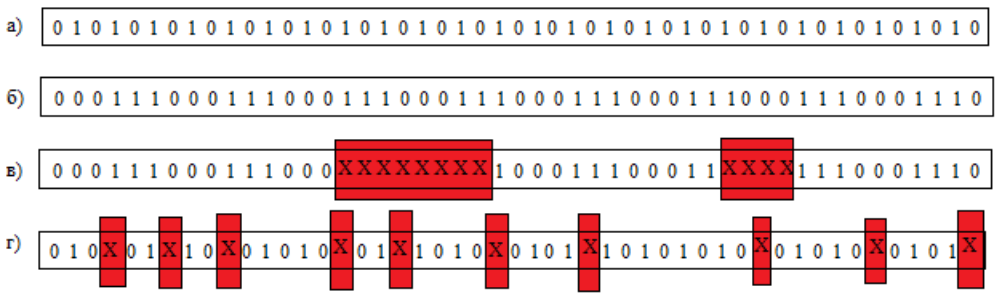

Рис. 11. а) Данные после кодирования; б) Данные после чередования;
в) Данные после передачи по каналу связи; г) Данные после ликвидации чередования.

Для того, чтобы минимизировать длину последовательностей, ошибочных бит, применяются прямая и обратная операция чередования. Модуль interleaving 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 рисунке 11 приведен простой пример работы данных модулей.

Алгоритмы чередования не представляют большого интереса, поэтому подробно не представлены. Можно только сказать, что основной задачей чередования является максимальный разброс соседних бит по посылке.

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. Декодер, содержит решетку на 64 состояния, глубина декодирования составляет 6. Это означает, что на каждом шаге алгоритм отслеживает лучшие 64 пути из возможных.

Тело алгоритма состоит из двух основных частей: прямой и обратной проходки. При прямой проходке выстраивается решетка Витерби - записываются метрики возможных путей декодирования. При обратной проходке, на основе полученных метрик, определяется путь с минимальной ошибкой. Выполняя проходку по решетке в обратном направлении, закодированное сообщение восстанавливается.

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.

Кодер

Для сверточного кодера был реализован следующий алгоритм (рис. 13):

Рис. 13. – Структура работы кодера.

С целью повышения эффективности, все возможные результаты работы полиномов рассчитаны заранее и сохранены в двумерном массиве 4x8, в виде 16-битных чисел (в соответствии со спецификацией TMS32054). 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. Тело кодер при этом состоит из двух вложенных циклов. В теле первого цикла происходит определение номера строки и бита, в теле

второго - копирование необходимого бита для каждого из полиномов. Количество используемых полиномов характеризуется режимом.

Модули Чередования

Алгоритмы используемые в модулях чередования идентичны алгоритмам дизассемблированной прошивки. На вход в качестве параметров модули принимают указатель на память и длину сообщения. Важно отметить, что алгоритмы чередования, в отличии от алгоритмов кодирования и декодирования, невозможно использовать в нескольких потоках для одной группы данных. Дело в том, что эти алгоритмы создают, своего рода, поток передаваемых бит, при этом каждое из чередуемых сообщений, частично смешивается с предыдущим. Это не позволяет организовать выполнение алгоритма в разных потоках, без дробления потока передаваемых данных на более мелкие. Однако при этом возникают потери во времени, так как первое сообщение смешивается с нулевым пакетом (сообщение, заполненное нулями), как следствие необходимо передать лишние 840 бит.

Декодер

Декодер состоит из 3 функций: 1 основной, которая включает в себя прямую и обратную проходку, и 2-ух вспомогательных, рассчитывающих все возможные варианты весов для решетки Витерби. Каждая из вспомогательной функции вызывается в соответствующем режиме. Алгоритмы работы декодера идентичны алгоритмам, скрытым в прошивке.

Для простоты использования, кодек был оформлен в качестве библиотеки (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 только язык Си. Были подключены две сторонние библиотеки: <math.h> – для команды pow (возведение в степень) и <stdlib.h> - для выделения памяти под массивы.

Тестирование реализации на языке Си.

По завершению разработки было проведено тестирование. Во-первых, необходимо было удостовериться в том, что разработанный кодек кодирует и декодирует информацию так же качественно, как и исходный, во-вторых, необходимо узнать границы декодирующих способностей двух исходных режимов кодирования. Для того, чтобы выполнить поставленные задачи потребовалось написать отдельную программу (приложение 3), которая бы запускала модули кодека в необходимом порядке, передавая на кодирование различные данные, сгенерированные псевдослучайным образом. Алгоритм разработанной программы показан на рисунке 14.

Рис. 14. 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. Так как были поставлены сразу две задачи (тестирование работоспособности и определение граничных возможностей режимов кодирования), модуль имитирующий шум в канале, при решении первой задачи запускаться не будет (о модуле шума ниже). 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 провалившихся тестов) и ошибок не выявило. Кроме того, необходимо было подтвердить возможность кодирования старым устройством и декодированием новым. В виду того, симулятор невозможно включить в цикл тестирования (т.к. CCS GUI приложением), было проведено выборочное тестирование 50 последовательностей. Данные кодировались кодером с прошивки, средствами CCS, и кодером-Си, средствами компилятора Си, после чего результаты сравнивались. Различий обнаружено не было, из чего можно сделать вывод, что закодированная последовательность на старом устройстве, легко декодируется на новом и наоборот.

На втором этапе тестирования необходимо проверить и подтвердить восстанавливающие способности реализованного декодера. Для этого в тело цикла тестирования включается модуль

имитирующий шум в канале (Приложение 4). Данный модуль имеет три входных параметра: 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, имитируя наихудший вариант. Это объясняется работой модуля чередования, т.к. он распределяет данные равномерно 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.

Для тестирования восстанавливающих способностей декодера были проведены эксперименты со следующими настройками модуля-шума:

№	Кол-во ошибочных последовательностей (ОП)	Кол-во испорченных бит	Расстояние между ОП
1	1	1-480	-
2	2	1-360; 1-180;	0-120
3	3	1-240; 1-120; 1-60;	0-120

После проведения экспериментов всех трех групп, выявилось, что количество зашумленных последовательностей никак не влияет на декодирующие способности, в отличии от количества зашумленных бит. В связи с этим на рисунке 15 предоставлена диаграмма с результатом только первой группы экспериментов.

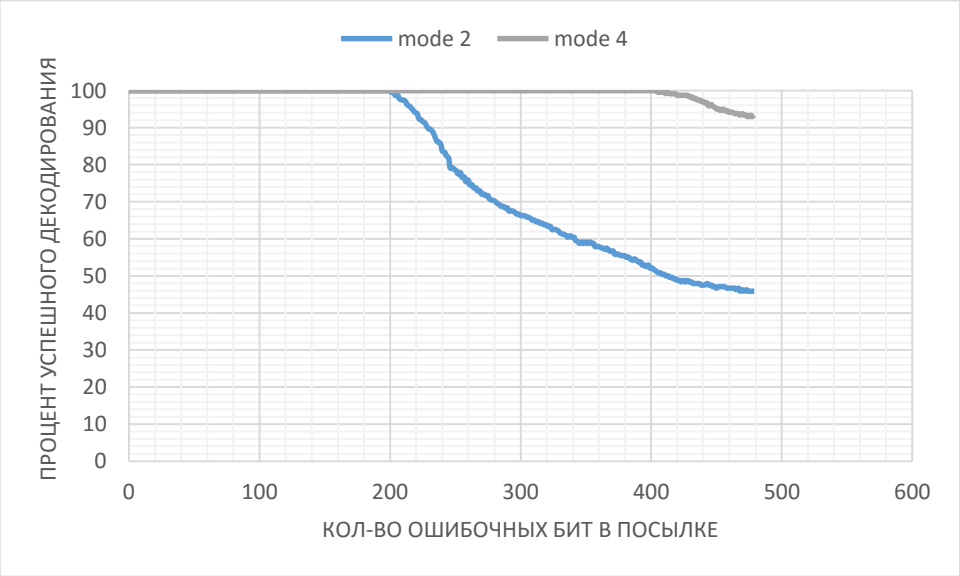

Рис. 15. Граничные значения декодирующих способностей режима 2 и 4.

На диаграмме (Рис. 15.) предоставлена визуальная характеристика декодирующих способностей различных режимов кодирования. Эксперимент был поставлен следующим образом:

Генерируется 1024 случайных последовательностей длиной по 240 бит каждая (длина 240 бит обуславливается средне статической длиной передаваемого сообщения). Последовательности кодируются и чередуются. Затем в каждую из закодированных посылок вносится N ошибок (ось абсцисс), после чего чередование ликвидируется и сообщения декодируются. По завершению

декодирования все последовательности сравниваются с исходными. В случае, если исходная и декодированная последовательность не совпали – подсчитывается кол-во несовпадений. Процент успешности декодирования (ось ординат) зависит от успешности декодирования каждой последовательности.

Из проведенных экспериментов следует:

// тут явный недочет по подсчету бит, необходимо пересчитать

- Режим 2 способен исправить до 201 ошибке из 1440 бит передаваемого сообщение (240 * 2 + 960 бит нулевого сообщения), что соответствует 14%.
- Режим 4, удачно декодирует сообщение при 411 ошибках из 1920 передаваемых бит (240*4+960), что соответствует же соответствует 41%. Длина допустимой ошибки вдвое больше, но и скорость передачи в два раза меньше.

Очевидно, что использованием обоих режимов для каналов связи, в которых в среднем ошибка составляет 200-300 бит, будет не рациональным. Использовать режим 2 не представляется возможным из-за превышения восстанавливающих способностей декодера, а использование режима 4 является не эффективным, т.к. вызывает значительное замедление работы системы из-за удвоения объема передаваемых данных. Возникает необходимость в промежуточном режиме.

Модернизация кодека

Промежуточный режим кодирования позволит ускорить передачу данных и сократить временные затраты на кодирование/декодирование. При этом структура основных модулей не должна быть изменена. Для минимизации изменений было решено использовать первые 3 полинома из существующих. В виду того, что реализация всего кодека и для режима 2 и для режима 4 значительно не отличается, добавить режим кодирования 3 битами труда не составит, однако реализация

PAUSE

Литература:

1) Краткое описание Hiew. URL: https://ru.wikipedia.org/wiki/Hiew
2) Статья о возможностях IDA. URL: http://www.idasoft.ru/idapro/
3) Интервью с Ильфаком Гильфановым. URL: http://fcenter.ru/online/softarticles/interview/6704
4) Сайт фирмы, разработавшей процессор. URL: http://www.ti.com/
5) Раздел сайта TI, посвящённый CCS. URL: http://www.ti.com/tool/ccstudio
6) Описание заголовка dat-файла. URL: https://e2e.ti.com/support/development_tools/code_composer_studio/f/81/t/277303
7) Сталин в беседе с Б. Куном; С. 190

Приложение 1 – программа для изменения формата текста

Приложение 2 – библиотека кодека

Приложение 3 – программа для тестирования

Приложение 4 – модуль имитирующий шум в канале