Реверс-инжиниринг встраиваемых систем

Алексей Усанов

ДМК Пресс
издательство

Алексей Усанов

Реверс-инжиниринг встраиваемых систем

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2023

УДК 62-1/9
ББК 30.1
У74

У74 Усанов А. Е.
Реверс-инжиниринг встраиваемых систем. – М.: ДМК Пресс, 2023. – 296 с.: ил.

ISBN 978-5-93700-231-0

Перед вами руководство по погружению в мир встраиваемых систем – от их первоначального анализа и получения прошивки до нейтрализации механизмов защиты от реверс-инжиниринга и модификации. Приводится базовый набор оборудования и ПО, с помощью которого можно проводить исследования большинства систем. Опытному читателю книга пригодится в качестве справочника, а начинающим исследователям будет полезно изучить ее от начала до конца.

Издание адресовано инженерам и разработчикам встраиваемых систем; также оно пригодится студентам технических вузов.

ISBN 978-5-93700-231-0

Оглавление

Книга посвящена информационной безопасности встраиваемых систем. Считается, что история этого термина начинается в 1966 году, когда в США был разработан бортовой управляющий компьютер для лунной программы «Аполлон». В этом 16-битном компьютере впервые применялись интегральные схемы вместо отдельных транзисторов и электронных ламп, что снизило массу, уменьшило габариты и позволило назвать его первой встраиваемой системой, или встраиваемым устройством. С тех пор такой подход начал применяться в военной технике, авиации и автомобилях.

Следующий рывок связан с распространением интернета. Наличие высокоскоростного доступа к сети привело к взрывному росту встраиваемых устройств, так как теперь они объединялись в сложные системы на базе «интернета вещей» (Internet of things, IoT). Снижение стоимости производства привело к тому, что уже сложно обозначить границу между тем, что является, а что не является встраиваемыми устройствами – такие системы повсюду. Сразу после появления их начали пытаться нелегально копировать, но это не имело массового характера, и производители особо не беспокоились о защите. Однако доступность интернета изменила все – теперь хакеры могли взламывать их удаленно и получать огромное количество информации, которая хранится или обрабатывается на встраиваемых устройствах. Производители же стали использовать различные способы для усложнения анализа и взлома – зашифровывать прошивки устройств (firmware), внедрять электронные подписи и регулярно обновлять устройства через интернет, устраняя найденные уязвимости. Но количество успешных атак показывает, что достаточный уровень защиты не обеспечивается. Часто производители считают, что безопасность через неясность (Security through obscurity) отпугнет хакеров и никто не начнет исследовать устройство, так как это требует специфических навыков и оборудования. Книга покажет, насколько такое предположение ошибочно.

Литературы, рассказывающей о безопасности встраиваемых систем, не так много. «Реверс-инжиниринг встраиваемых систем» – одна из тех книг, которые позволяют вам понять, как происходят атаки на такие системы, как от них можно защититься, и даже поможет исследовать «подозрительное» устройство у вас дома. Профессионалы найдут в данной книге информацию, которая будут полезной в повседневной работе, – современные методы атак на микросхемы и способы защиты, ссылки на современные исследования. Если же вы только начинаете свой путь в исследовании встраиваемых систем, то сэкономите время и бюджет, применяя рекомендации из этой книги. После прочтения вы будете смотреть на свою систему видеонаблюдения или чайник с Bluetooth совсем по-другому.

Я благодарен автору за то, что он предоставил мне возможность ознакомиться с этой книгой в числе первых.

Максим Горячий (@h0t_max),
Device Security Engineer

Intro (Зачем исследовать встраиваемые системы?)

Что движет людьми, которые занимаются реверс-инжинирингом? На мой взгляд, желание разобраться, как устроен мир, и понять, как мыслят другие люди, создающие что-то. С детства я всегда разбирал все, что попадало под руку. Эта тяга осталась до сих пор, только игрушки стали другими. Мне повезло познакомиться и работать с людьми, которые также имеют нездоровую тягу к исследованию внутренностей электронных устройств. Накопленный опыт хочется систематизировать и изложить в максимально доступном виде.

Сразу сделаю уточнение: я не хотел писать академическую книгу сухим языком. Ведь общаемся мы совсем по-другому. Второе уточнение – это слово «реверс-инжиниринг» в названии книги. Лучше бы подошло слово «исследование», ведь в основном люди занимаются исследованием с помощью реверс-инжиниринга. Однако большинству людей понятнее слово реверс-инжиниринг, поэтому пусть в названии будет оно :)

Под словосочетанием встраиваемые системы (англ. embedded) мы будем понимать именно цифровые устройства. Некоторые из них могут иметь аналоговые блоки, выполняющие определенные функции, но темой данной книги будет исследование именно цифровых устройств. Почему именно цифровые устройства? Наверное, потому что, в отличие от реверс-инжиниринга ПО, есть возможность «пощупать» объект исследований, получить тактильные ощущения, совершить с ним различные манипуляции. Рано или поздно увидеть волшебный дым, на котором работает электроника (**https://en.wikipedia.org/wiki/Magic_smoke**). А еще потому, что в современном мире цифровые устройства стали максимально распространены. Они применяются в транспорте, умных домах, игрушках, IoT-системах, заводах и т. д. Какое электронное устройство в руки не возьми – везде встраиваемые системы. Практически в любой современной информационной системе корнем доверия является уровень «железа», т. е. аппаратного обеспечения, лежащего в основе цифровых устройств. Компрометация этого уровня или самих устройств приведет к компрометации всей информационной системы.

Книга была написана после сильного изменения взаимоотношений в мире в 2022 году. Многие технологии для разработчиков в нашей стране стали недоступны. Поэтому реверс-инжиниринг, в том числе встраиваемых систем, становится еще актуальнее. Ведь какие-то из устройств могут перестать работать и надо найти технические пути для их возвращения «в строй». Ну и конечно же, нельзя забывать про аудит безопасности устройств. Иногда проведение иссле-

дований для оценки защищенности – единственный способ удостовериться, что устройство всегда будет работать так, как заявлено. Не менее важен аудит защищенности российских устройств, особенно в контексте возрастающей угрозы атак на цепочку поставок. Все эти проблемы стали сверхактуальными для российского (да и мирового) рынка в последние годы.

Для кого эта книга? Я позиционирую ее как первичное руководство-справочник по погружению в мир цифровых устройств для исследователей, уже попробовавших реверс-инжиниринг ПО. Имеющих представление, что такое дизассемблер и декомпилятор, зачем нужны регистры процессора, из чего состоит бинарный исполняемый файл и что такое виртуальная память. Хотя бы на начальном уровне. В этой книге не будет глав про процесс реверс-инжиниринга ПО, интерфейсы дизассемблеров и т. д., на эту тему написано большое количество книг и статей. Другая, не менее важная целевая аудитория – это разработчики встраиваемых систем. Ведь мало грамотно спроектировать устройство, надо еще его качественно защитить от копирования или исследования. Кто лучше всех знает, как защищаться? Тот, кто умеет нападать. Разработчик, понимающий логику и подходы исследователя, сможет гораздо лучше защитить свое устройство. Также книга будет полезна студентам, обучающимся на кафедрах по специальностям «Информационная безопасность» и «Проектирование электронных устройств».

Я ставил себе задачу показать людям, никогда не имевшим дела с исследованием «железа», что это не сложно, но нужно получить знания во многих областях, которых практически не касаешься в большинстве случаев исследования или разработки ПО. Если у читателя был опыт программирования каких-то микроконтроллеров (хотя бы проекта Arduino), то многие вещи уже будут знакомы и понятны. Фактически данная книга – это методология, показывающая один из вариантов пути исследования электронных устройств, а также объясняющая, почему путь именно такой, какие на этом пути есть распространенные ошибки и как их избежать. Этот путь наверняка не единственный, но он показал свою эффективность более чем за 10 лет практических исследований устройств, так почему бы не начать свои шаги с него? Когда я начинал заниматься исследованием устройств, вся информация собиралась от старших коллег, разрозненных источников, из множества экспериментов, зачастую уводящих в сторону от результата, из «убитых» устройств и размышлений, можно ли было этого избежать. Все это позволило сформировать подход, описываемый в данной книге. На стажировках я рассказывал, почему именно такой порядок действий будет эффективным, но единого материала, который можно было бы прочитать и понять путь и методы проведения исследования цифровых устройств, не было.

При проведении любых исследований мы можем иметь разный уровень знаний об объекте исследования. В зависимости от количества имеющихся знаний о встраиваемой системе исследования могут проводиться с помощью методов белого, черного и серого ящика. Если у нас есть вся информация о системе, то это исследование методом «белого ящика». Если информации минимум, то исследование проводится методом «черного ящика» – анализируется реакция на воздействия, и постепенно получаем все больше информации об объекте исследования. Метод «серого ящика» является промежуточным, когда мы смог-

ли получить какую-то информацию о системе. Большинство описываемых в книге подходов применимы для любого метода исследований, но часть из них будет весьма полезна для исследований методами черного и серого ящика. Исследование методом «черного ящика» является наиболее сложным, особенно для начинающих исследователей цифровых устройств. Поэтому мы сконцентрируемся на том, чтобы максимально быстро получить прошивку устройства и перейти к исследованию методом «серого ящика», основываясь на ее анализе. На схеме представлен путь, которым я предлагаю пройти читателю при первых попытках исследования устройств.

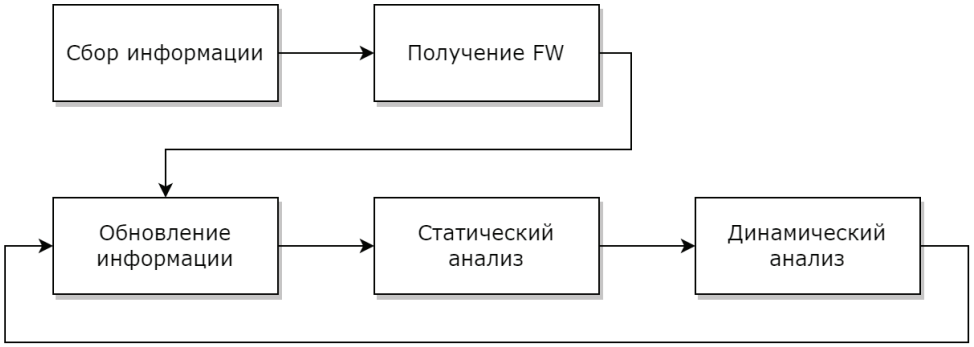


Рис. I.1. Вариант пути проведения исследований встраиваемых систем

Структура книги соответствует предложенной методологии и состоит из пяти основных глав-уровней и нескольких вспомогательных разделов.

В самом начале, в разделе «*Background*», мы договоримся о том, что такое цифровой электрический сигнал и какие существуют особенности его передачи.

Level 0. Первоначальный анализ. С него начинается исследование цифрового устройства. Мы пройдем по пути анализа основных компонентов устройства и восстановим структурную, возможно функциональную, и даже некоторые части принципиальной схемы. (Конечно же не полностью, но на достаточном для понимания базовой структуры устройства уровне. Подробно про разные типы схем можно прочитать в статье по адресу: **https://habr.com/ru/post/451158/**.) В этой главе я расскажу, из чего состоит большинство устройств, как делать этот анализ и почему эти знания помогут в дальнейшем исследовании.

Level 1. Добываем прошивку. Логика работы цифрового устройства заложена в программном обеспечении, оно же прошивка (firmware). Пока у нас ее нет, исследования проводить намного сложнее. В этой главе мы рассмотрим, как прошивку можно получить. От очевидных до весьма нестандартных путей.

Level 2. Начинаем статический анализ. К этой главе мы уже получили образ прошивки, но пока не знаем, что с ней делать. (Ок, «грузить» в дизассем-

блер, но что мы там увидим и что должны увидеть?) Узнаем, какие бывают типы прошивок, как прошивка устроена и как это связано с «железом» устройства.

Level 3. Настраиваем связь с внешним миром (динамический анализ). К этой главе мы уже загрузили прошивку в дизассемблер и даже что-то поискали. Можно и дальше смотреть в дизассемблер, но гораздо эффективнее (и приятнее) заставить устройство что-то нам рассказать о себе в процессе работы, в том числе за счет написания собственных патчей для прошивки.

Level 4. Механизмы защиты встраиваемых систем. Какое хорошее исследование заканчивается без запуска своего кода? Производители устройств очень не хотят, чтобы у нас это получилось. Рассмотрим, какие существуют способы защиты прошивок и устройств от модификации и что с ними можно сделать.

В разделе *«Well Done»* подводим итоги пройденного пути и смотрим, как дальше можно развиваться в исследовании встраиваемых систем.

В *приложении* приведен базовый набор оборудования и ПО, с помощью которого можно проводить исследования большинства встраиваемых систем.

Как читать книгу?

Я старался сделать главы максимально независимыми, чтобы опытный читатель мог использовать книгу как справочник и сам решить, какие блоки информации он знает и может пропустить, а каким стоит уделить внимание. В то же время для начинающих исследователей структура книги позволяет получить знания, объединенные единым подходом-методологией, отвечающей на вопрос «почему последовательность получения информации выстроена в таком порядке?».

Благодарности

Спасибо Юрию Васину, Максиму Горячему, Павлу Иванникову, Алексею Ковриных и Кириллу Малышеву за техническую редактуру книги. Ваши замечания и рекомендации сильно повлияли на итоговое содержание книги и сделали ее лучше.

Почта для связи с автором: **book.re.au@yandex.ru**.

Background: особенности цифровых электрических сигналов

Перед началом исследований цифровых устройств необходимо разобраться, как выглядит цифровой электрический сигнал и какие особенности существуют при его передаче. Я постарался максимально простым языком рассказать самые важные концепции, для более глубокого изучения темы цифровой схемотехники рекомендую начать с книги «Цифровая схемотехника и архитектура компьютера» Дэвида Харриса и Сары Харрис. Ну а мы начнем с определения и различий терминов шина, интерфейс и протокол обмена.

Под понятием «шина» (bus в англоязычной литературе) обычно подразумевают коммуникационную систему для передачи данных между несколькими (>= 2) функциональными блоками. В случае устройства в качестве функциональных блоков могут выступать как компоненты на плате устройства (внутренняя шина), так и само устройство, подключаемое к компьютеру, датчикам или другим устройствам (внешняя шина). В устройстве шины можно различить механический (разъемы и проводники на плате), электрический (физический) и логический (управляющий, или протокольный) уровни. Понятие шина может использоваться как для описания топологии соединения, так и для набора электрических сигналов.

В отличие от прямого соединения точка-точка, позволяющего соединить два функциональных блока, к шине обычно можно подключить несколько устройств по одному набору проводников. Каждая шина предполагает определенные протоколы взаимодействия между подсоединенными к ней функциональными блоками. Протокол – это система правил и соглашений о кодировании, синхронизации и логической организации передаваемой информации.

Интерфейс в общем случае также имеет несколько уровней: механический, электрический, протокольный и управления (программы или функционального блока, непосредственно выполняющего обмен). Несмотря на немного отличающиеся понятия, четко разграничить области применения интерфейса и шины достаточно сложно. Особенно сложно разобраться в переведенной на русский язык иностранной литературе, в которой часто слово interface переводится как шина. Поэтому в реальной жизни между понятиями «шина» и «интерфейс» в большинстве случаев не делается различия и они означают одно и то же: набор проводников, электрических сигналов и протокола, обеспечивающих информационный обмен. При этом также часто шиной называется только набор проводников… В общем, существует большая путаница понятий, мы будем стараться придерживаться следующих общепринятых определений:

- шиной мы будем называть набор электрических проводников и сигналов, передающихся по ним;
- протокол – набор правил, соглашений, сигналов, сообщений и процедур, определяющий взаимодействие между соединяемыми функциональными блоками;
- интерфейс – совокупность программных и аппаратных средств, необходимых для осуществления информационного обмена между функциональными блоками.

Кодирование цифровых сигналов

Для передачи одного бита информации (т. е. 0 и 1) по проводнику необходимо как-то закодировать эти значения. Например, с помощью разных уровней потенциалов относительно «земли», т. е. в общем случае 0 В. Для внешних интерфейсов, применяющихся для связи разных устройств, могут использоваться и отрицательные значения, однако они все равно считаются относительно земли. Для интерфейсов внутри одной платы в основном уровень логического нуля будет от 0 В до некоторого порогового значения $V_{пор}$, а уровень логической единицы – от $V_{пор}$ до значения напряжения питания компонента V_{CC}, как показано на рисунке.

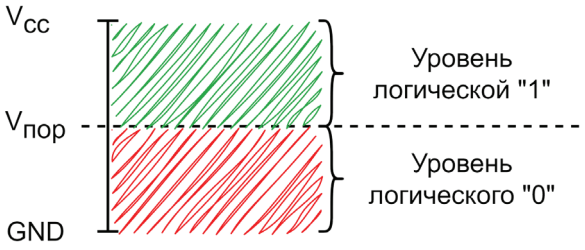


Рис. В.1. Кодирование уровней логического 0 и 1

Чтобы исключить ошибку определения уровня около порогового значения $V_{пор}$, определяют некоторую «буферную зону» напряжений, в которых значение не интерпретируется (зона гистерезиса). Тогда разрешенный диапазон напряжений уровней логического нуля (от GND до V_L) и единицы (от V_H до V_{CC}) будет меньше:

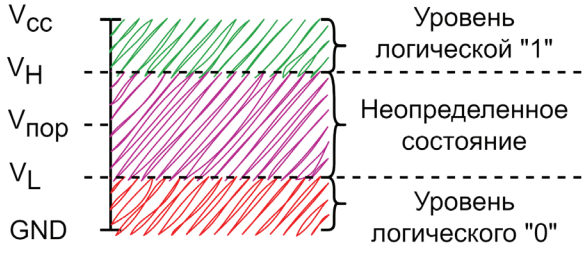


Рис. В.2. Кодирование уровней логического 0 и 1 с зоной неопределенного состояния

Для разных интерфейсов значения уровней логического нуля и единицы могут отличаться. Также уровни напряжений зависят от типов структур транзисторов, из которых состоят микросхемы. Наиболее распространенными на данный момент являются типы КМОП – комплементарная структура металл–оксид–полупроводник (CMOS, complementary metal-oxide-semiconductor) и ТТЛ – транзисторно-транзисторная логика (Transistor-Transistor Logic, TTL). Большинство современных цифровых компонентов построены на базе КМОП-логики. Диапазоны напряжений для ТТЛ- и КМОП-логик представлены на рисунках.

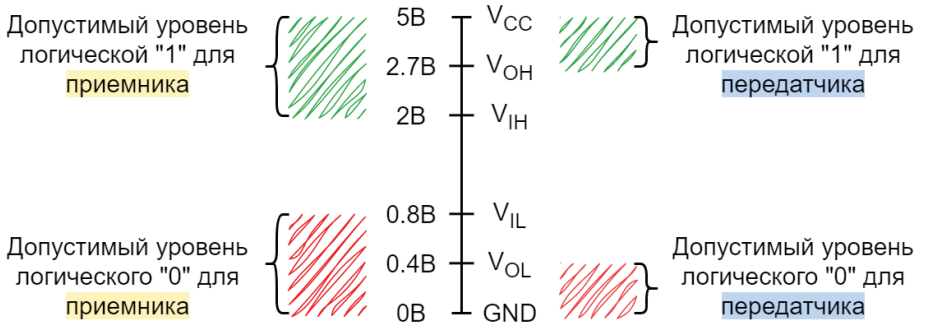


Рис. В.3. Диапазоны уровней напряжений для ТТЛ

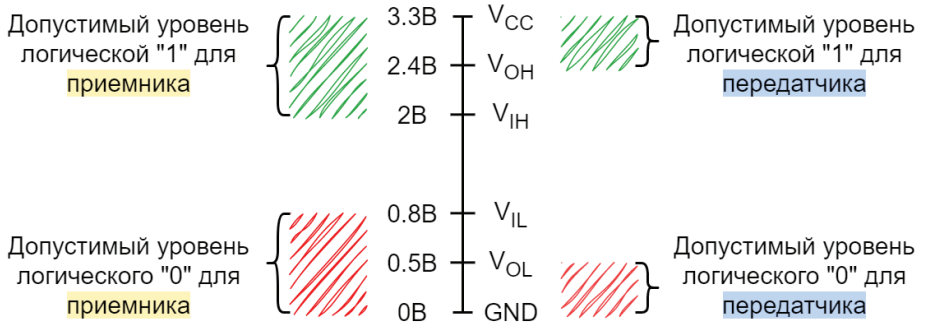


Рис. В.4. Диапазоны уровней напряжений для КМОП

Так как в любых проводниках могут быть потери на сопротивление и рассеивание сигнала, уровни минимального выходного и минимального входного сигналов отличаются. Например, для КПОМ-логики минимальный уровень выходного сигнала логической единицы (V_{OH}) равен 2.4 В, а минимальный уровень входного сигнала логической единицы (V_{IH}) – 2 В. То есть сигнал может «потерять» до 0,4 В при передаче, но будет корректно обрабатываться приемником.

Как отличить электрические сигналы, соответствующие логическому 0 и 1, мы разобрались. Но как отличить один передаваемый бит (со значением 0 или 1) от другого? На выручку приходят временное разделение и синхронизация сигналов.

Синхронизация сигналов

Представим ситуацию, когда нам надо передать один байт (8 бит) информации по одному проводнику. Самый очевидный вариант – использовать для передачи каждого бита информации какой-то временной промежуток, например 1 мс. Получается, что для передачи 8 бит будет достаточно 8 равных временных отрезков по 1 мс, т. е. 8 мс. Принимающая сторона должна проверять уровень напряжения (соответствующий логическому 0 или 1) с аналогичной периодичностью. На рисунке показана временная диаграмма подобной передачи для значения 0x9C (10011100b).

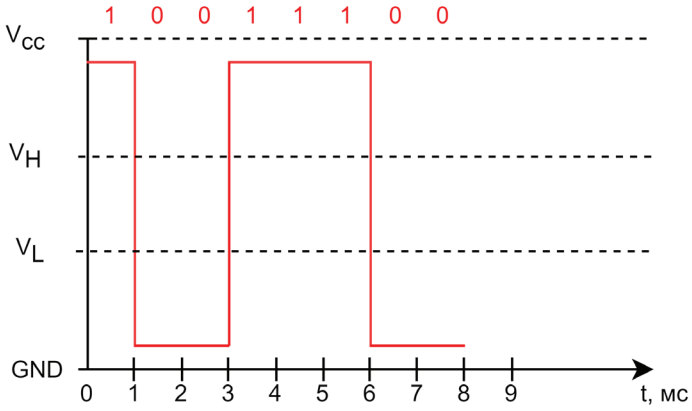


Рис. В.5. Пример временной диаграммы передачи значения 0x9C

Но электрический сигнал не может мгновенно изменить свое состояние с 0 на 1 и наоборот. Всегда существует время, в течение которого происходит переходный процесс, т. е. изменение реального значения напряжения в проводнике.

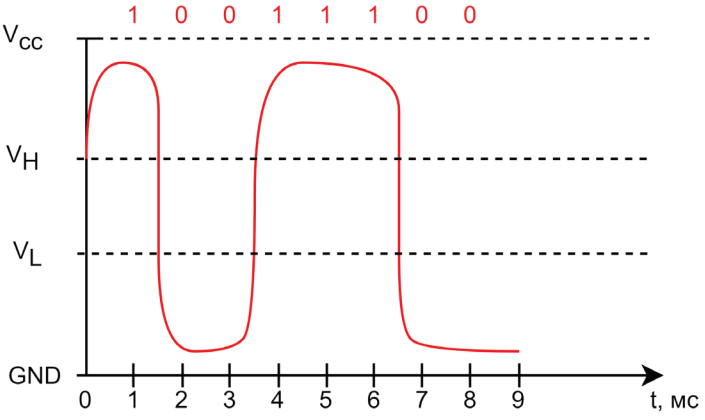


Рис. В.6. Пример переходных процессов изменения сигнала при передаче значения 0x9C

Следовательно, проверять значение уровня напряжения приемнику лучше всего в середине временного отрезка передачи бита. В нашем случае время передачи составляет 1 мс, значит, смотреть значение лучше всего со смещением 0,5 мс от начала передачи бита. Так как любая система не идеальна и всегда могут встречаться какие-то задержки, допустимый временной интервал расширяют до некоторого значения, например ±0,2мс. Получается, что, начиная с 0,3 мс от начала передачи бита и заканчивая 0,7 мс, приемник ожидает корректное значение напряжения, попадающего в границы напряжения для логического 0 и 1 выбранной логики (мы для примера возьмем КМОП). Внешний вид сигнала, передающего первые 4 бита информации (не будем перегружать схему отрисовкой всех 8 бит) байта 0x9C со скоростью 1 Кбит/с (т. е. 1 бит за 1 мс), показан на рис. В.7.

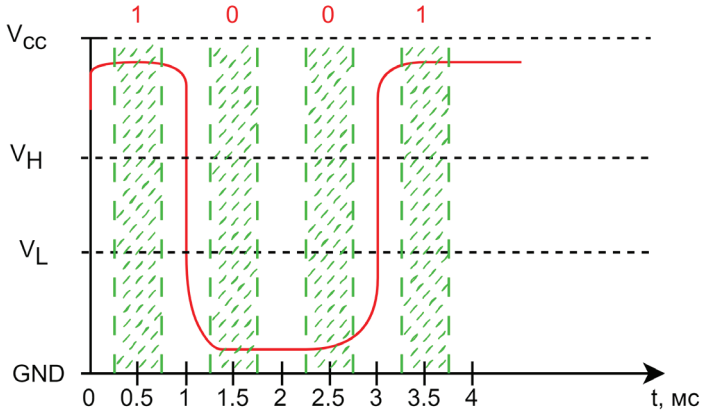


Рис. В.7. Зоны корректных значений сигнала

Этот сигнал соответствует исходной цифровой интерпретации передачи байта 0x9C. Поздравляю, мы с вами придумали простейший интерфейс с временной синхронизацией. Основным минусом подобного типа передачи является то, что приемник и передатчик обязательно должны быть заранее сконфигурированы на одну частоту, которую они должны довольно точно выдерживать. Если приемник или передатчик не могут гарантировать допустимые временные интервалы корректного сигнала, передача будет идти с ошибками. На рис. В.8 показана ситуация, когда время установки сигнала «плавает» и приемник не может корректно интерпретировать подобный сигнал.

Именно из-за отсутствия единой точки отсчета для синхронизации такие интерфейсы не могут похвастаться частотами передачи больше нескольких мегагерц. Для решения этой проблемы существуют различные механизмы, например посылка специальных данных с известным значением при старте передачи и настройка параметров приемника с учетом особенностей сигнала принятых от передатчика данных. Этот подход требует довольно сложной аппаратной поддержки и поэтому используется только в высокоскоростных интерфейсах. Мы рассмотрим другое решение – добавление специального синхронизирующего сигнала отдельным проводником. В таком случае электрический сигнал будет иметь примерный вид, как показано на рис. В.9.

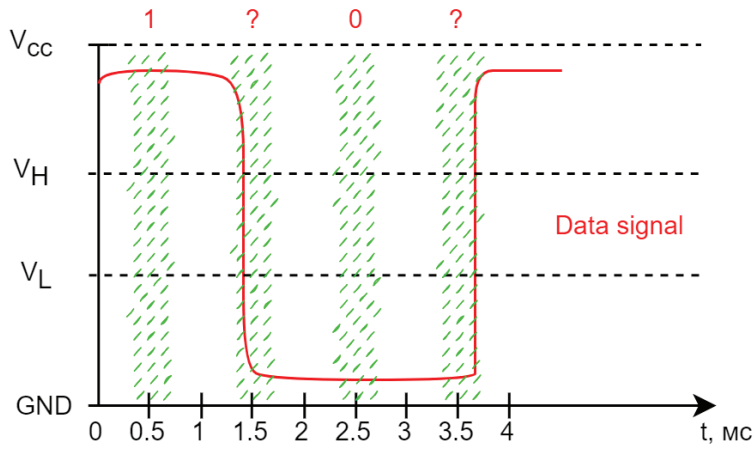


Рис. В.8. Зоны с неопределенными значениями сигнала

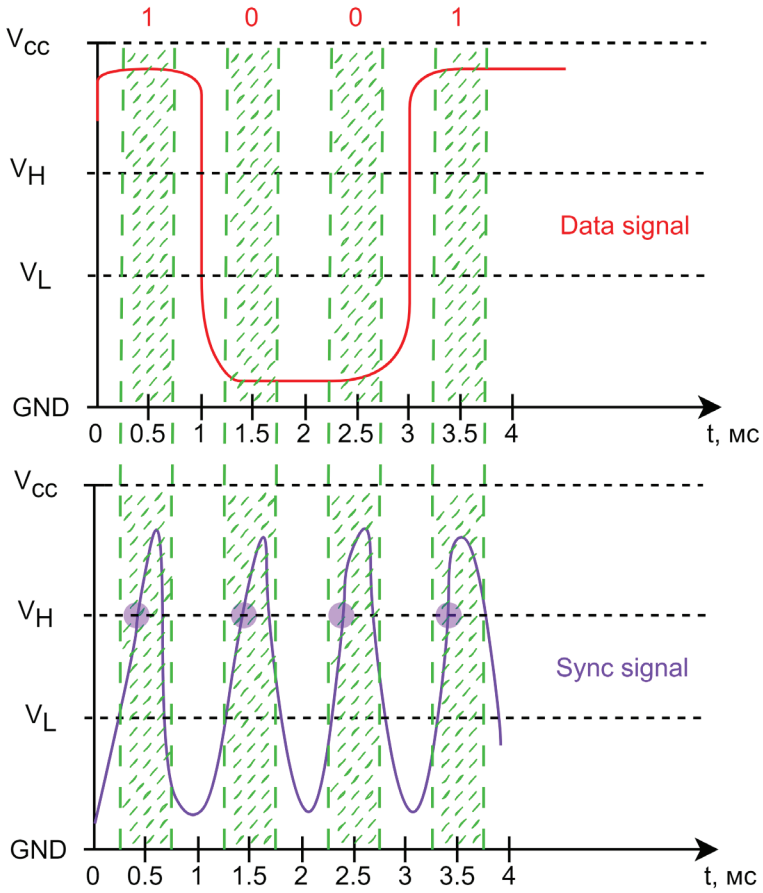


Рис. В.9. Добавление синхронизирующего сигнала

Цифровой сигнал нашего интерфейса примет следующий вид:

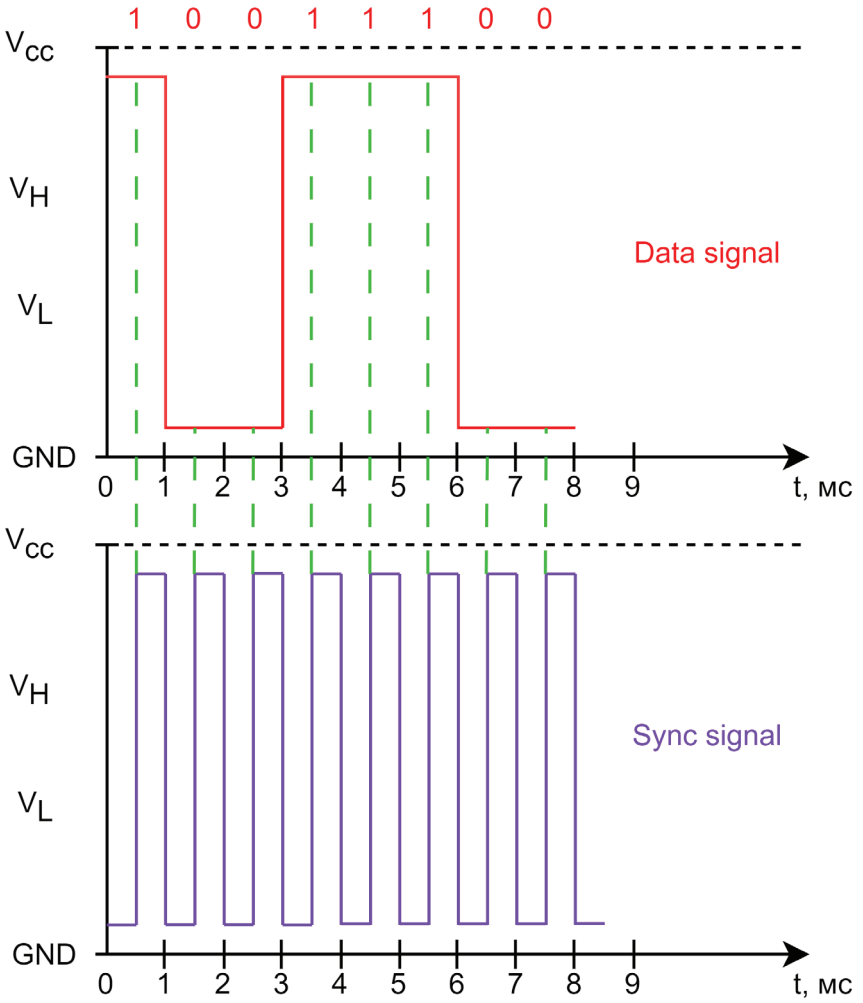


Рис. B.10. Добавление синхронизирующего сигнала

Как видите, теперь для приемника признаком наличия корректного значения на линии данных является изменение сигнала линии синхронизации с уровня логического 0 на 1. Такой сигнал называют синхронизацией по фронту (т. е. по возрастанию сигнала), обратная ситуация носит название по спаду (т. е. по убыванию сигнала с уровня 1 до 0). Существует механизм увеличения пропускной способности передачи, когда сигнал на линии данных меняется и по фронту, и по спаду, т. е. за 1 период изменения сигнала синхронизации передается 2 бита информации.

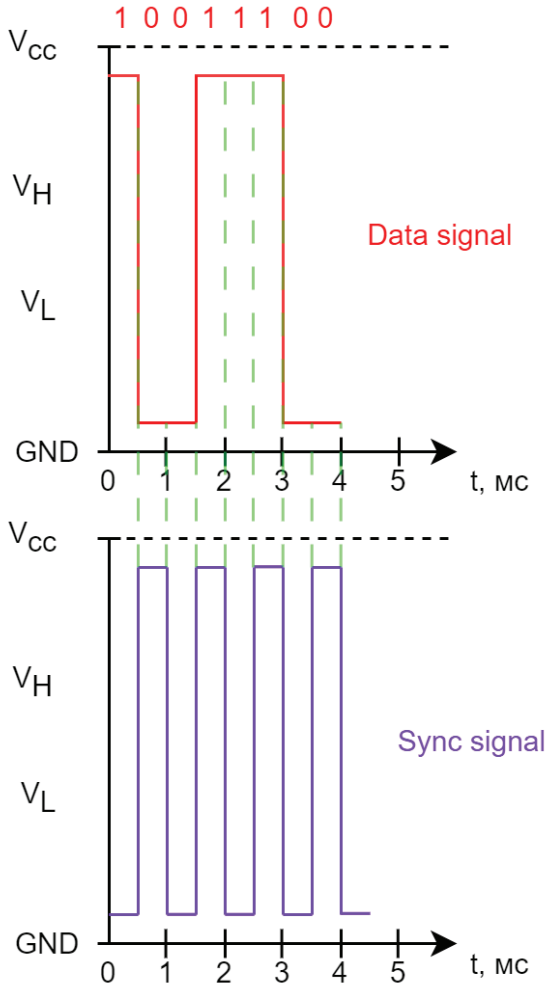


Рис. В.11. Режим передачи сигнала DDR

Фактически, без изменения частоты сигнала синхронизации, происходит удвоение пропускной способности интерфейса, называемое на английском языке хорошо вам известным сокращением DDR (Double Data Rate). Именно этот механизм используется в оперативной памяти современных компьютеров. Обычный вариант передачи данных по фронту или по спаду носит название SDR (Single Data Rate).

Синхронизация с помощью дополнительного проводника широко используется во многих интерфейсах цифровых устройств, их мы подробно рассмотрим чуть позже. А пока давайте разберем, чем последовательные интерфейсы отличаются от параллельных.

Параллельные и последовательные интерфейсы

Снова представим, что нам надо передавать определенный объем информации, чем быстрее, тем лучше. За один период тактового сигнала по двум линиям передачи информации (одна – для передачи данных и одна – для передачи сигнала синхронизации) в режиме SDR передается только один бит информации. Но ведь можно использовать несколько линий для передачи данных? Тогда пропускная способность увеличится, и на той же частоте с двумя линиями данных мы сможем передать уже 2 бита информации, с четырьмя линиями данных (и одним сигналом синхронизации) – 4 бита и т. д.! Вот мы с вами и изобрели параллельный интерфейс передачи данных. На рисунке показана диаграмма – пример передачи значения 0x9C75A16B по интерфейсу с четырьмя линиями данных и одной линией синхронизации:

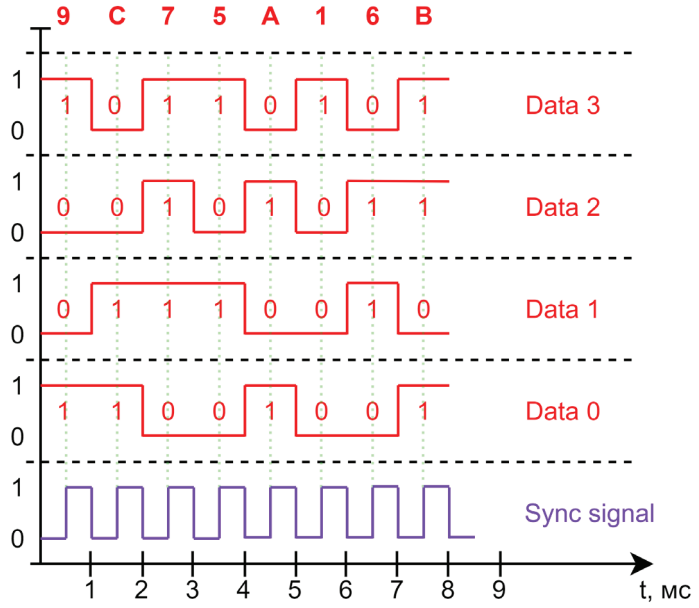


Рис. В.12. Параллельная передача сигнала

Масштабируя количество линий передачи данных, мы легко увеличиваем пропускную способность интерфейса, не меняя частоту передачи. Однако такая логика работает только до определенного порога частот, после которого требования к плате устройства существенно вырастают, т. к. влияние электрических помех все сложнее компенсировать. Это влечет за собой увеличение стоимости разработки устройства. Поэтому параллельные шины на высокоскоростных интерфейсах оправданы только в том случае, если нужна высокая пропускная способность. Например, от оперативной памяти до процессора, где используется несколько десятков параллельных проводников, сигнал по которым идет одновременно.

Поэтому чаще всего используют современные последовательные шины с набором нескольких дифференциальных пар, работающих на гигагерцовых частотах (например, как сделано в знакомом вам USB 3.2). Что такое дифференциальные пары и за счет чего достигаются настолько высокие частоты работы шин на их основе? Одна дифференциальная пара – это набор двух проводников, сигнал в одном из которых инвертирован по отношению к сигналу в другом проводнике. Пример показан на рисунке ниже (верхний график – хорошо знакомый нам обычный сигнал, нижний – дифференциальный сигнал).

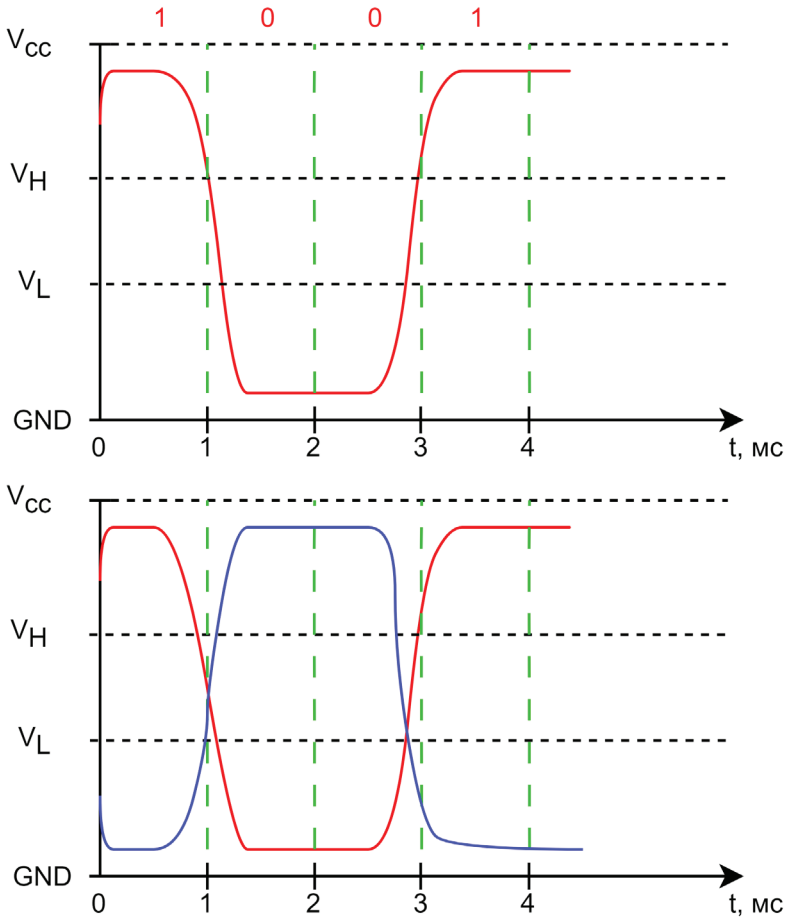


Рис. В.13. Дифференциальный сигнал

При таком способе передачи сигнала существенно снижается вероятность появления ошибок при передаче (т. к. уровень разности потенциалов всегда максимальный и вычитается помеха, влияющая одновременно на оба проводника), следовательно, можно значительно увеличивать частоту передачи. Данные по дифференциальной паре идут последовательно, но в современных высокоскоростных интерфейсах дифференциальных пар может быть несколько и работают они параллельно, например в интерфейсе PCI-Express.

Механизмы детектирования, снижения и исправления ошибок передачи

Фактически многие уже описанные решения были придуманы для минимизации ошибок, неизбежно возникающих при увеличении скорости передачи информации. Использование дифференциальных пар или режима DDR – отличные примеры подобных инженерных решений. Дополнительно во многих интерфейсах используются механизмы контроля целостности и исправления ошибок в передаваемых данных, основанные на помехоустойчивом кодировании:

- коды обнаружения ошибок (контрольные суммы) передаваемых данных. На практике используются циклические коды, известные как CRC8/16/32 и код контроля четности (над всеми битами передаваемых данных проводится операция исключающего ИЛИ (XOR), результат передается вместе с данными и проверяется приемником);
- коды коррекции ошибок (Error correction code, ECC). Позволяют не только обнаружить, но и исправить определенные битовые ошибки при передаче.

Подробнее про помехоустойчивое кодирование можно узнать, прочитав книгу «Коды, исправляющие ошибки» (Питерсон У., Уэлдон Э.). В зависимости от используемого интерфейса в цифровых устройствах могут применять различные механизмы и реализации (аппаратные и программные) обработки ошибок и даже их комбинации. Рассмотрев основы передачи цифровых сигналов, приступаем непосредственно к исследованию цифровых устройств.

Level 0

Первичный анализ

Исследование электронных устройств отличается от исследования высокоуровневого ПО как минимум тем, что нужно принимать во внимание аппаратные особенности устройства, зачастую недокументированные. Именно поэтому нужно получить как можно больше информации обо всем объекте исследований. Устройство далеко не всегда бывает автономно. Оно может подключаться к компьютеру и настраиваться через ПО управления. А может иметь модули связи и подключаться к облаку производителя. Чем больше информации вы соберете вначале, тем проще будет находить закономерности в ходе исследований. Мысль очевидная, но почему-то в пылу «пощупать» что-то руками многие часто забывают об этом шаге. А зря.

Сбор информации

Что мы должны попытаться найти в первую очередь? Правильно, документы. Чем больше – тем лучше. И под документами мы понимаем не всегда самые очевидные вещи, это:

- руководства (пользователя, администратора и т. п.);
- техническая документация (схемы для сервисных центров, информация на форумах ремонтников и т. п.);
- патенты и сертификационные отчеты, например FCC;
- результаты исследований (кто сказал, что мы первые?);
- любая информация о технологиях, применяемых у вендора (используемые ОС, стеки, фреймворки и т. д.). Полезно посмотреть список скиллов у сотрудников вендора на LinkedIn или их персональные блоги.

Вам пригодится умение правильно «гуглить». И пользоваться переводчиком с китайского (китайские форумы – кладезь полезной информации, которой больше нет нигде). Не забывайте про GitHub. Как корпоративный, так и личные репозитории работников вендора. История знает немало случаев, когда там хранились критические данные, например ключи шифрования.

Фактически вы проводите расследование и ищете любую информацию, которая может быть полезна. Полезно или нет, поймете потом, а сейчас просто сохраняйте все, что имеет отношение к исследованию. И еще один тезис: старайтесь мыслить широко. Первичная задача любого производителя устройства – получить прибыль. А как ее получить? Унифицировать все максималь-

но и использовать имеющиеся наработки. Поэтому даже если вы не можете найти информацию по конкретно вашему объекту исследований, посмотрите соседние модели и даже серии устройств. Например, Canon использует DryOS во множестве своих продуктов: от фотоаппаратов до принтеров, а сопроцессор безопасности внутри iPhone (на процессоре Apple A10) и MacBook (в виде чипа Apple T2) используется один и тот же. Ведь если работает, зачем что-то менять?

Ну и конечно же, не забывайте про любой софт. ПО управления, администрирования, *обновления*, *восстановления*. Все это пригодится. Помните, что версии под разные ОС могут значительно отличаться функциональными возможностями, а где-то даже могут быть исходники и отладочные символы.

Инженерный анализ устройства

Наконец-то начинается что-то, имеющее отношение к исследованию встраиваемых систем. Устройство лежит перед нами на столе, и не терпится побыстрее его потрогать. На этом этапе важно помнить о двух вещах.

Статическое электричество. Наш большой враг, ведь «убить» статикой железку очень легко. Поэтому ОБЯЗАТЕЛЬНО соблюдаем технику безопасности:

- «разряжаемся», правильно выбираем одежду и используем антистатические браслеты;
- лишний раз не трогаем выводы антенн и интерфейсные разъемы.

Неизвестное состояние устройства. Даже новое устройство из упаковки производителя может быть бракованным. Поэтому нельзя однозначно сказать, что устройство пришло в негодность после выполнения нашего инженерного анализа или оно уже было неработоспособно. Напрашивается очевидная мысль: перед инженерным анализом взять и проверить работоспособность устройства. Мысль абсолютно правильная, но есть нюанс. Устройство может иметь разные состояния до первого включения и после. И иногда состояние после первого включения содержит в себе меньше информации, чем до. Поясню: устройство может переписывать конфигурацию при каждом включении, перетирать логи (а логи заводских тестов бывают интересными) или иным образом менять свое внутреннее состояние. Возникает дилемма: не включишь – не узнаешь, а включишь – можешь потерять часть данных, полезных для анализа. Выход есть, и он очевидный. Нужно два и более устройств. Одно проверяем, другое препарируем. Но не всегда есть возможность добыть больше одного устройства, или его стоимость превышает стоимость квартиры. Как тут поступить – выбор каждый делает сам. Но из практики лучше все же сначала проверить работоспособность.

Отлично! Про статику помним, железок лежит штуки три, с чего же начинать инженерный анализ? И что мы должны получить на выходе? На выходе мы должны получить перечень основных компонентов устройства, схемы структурную (можно сразу функциональную) и даже принципиальную (в каком-то приближении), понимание организации основных информационных потоков между компонентами устройства. Это программа минимум.

Начнем мы с инвентаризации компонентов на печатной плате. Печатная плата (ПП, по-английски Printed Circuit Board, PCB) в общем случае – это кусок текстолита с проводящими электрический ток дорожками из меди, на котором смонтированы электронные компоненты. Устройства могут быть простые и сложные. В простых плата, скорее всего, одна, компонентов мало, монтаж компонентов неплотный.

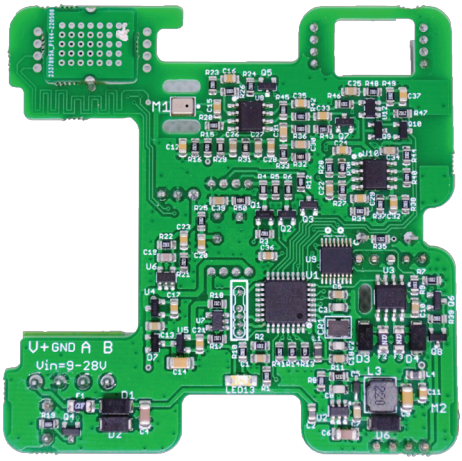


Рис. 0.1. Внешний вид платы простого устройства

В сложных устройствах могут быть различные ухищрения для миниатюризации устройства и эффективной компоновки внутри его корпуса. Размеры компонентов едва различимы на глаз, расстояния между ними практически нет. Например, современные телефоны используют «бутерброд» из двух печатных плат, спаянных между собой. Для примера приведем фото платы iPhone X (видите «точки» по периметру плат? Это линии передачи данных).

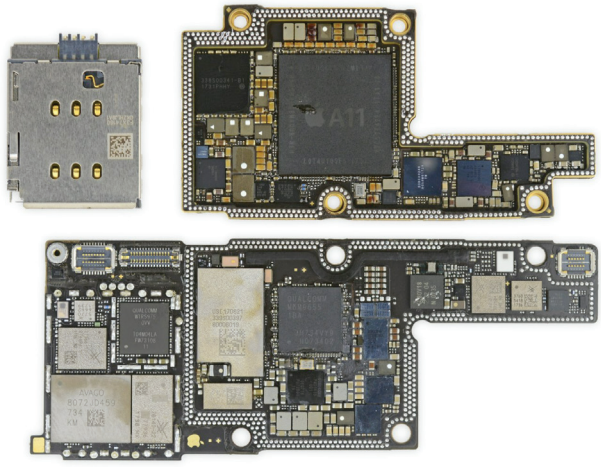


Рис. 0.2. Плата iPhone X как пример сложного устройства[1]

[1] https://ifixit.com/Teardown/iPhone+X+Teardown/98975.

Исследование подобных устройств на порядок сложнее, хотя нет ничего невозможного для человека с опытом и нужным оборудованием. Чтобы добраться до печатной платы, устройство надо сначала разобрать. И тут есть несколько типовых ошибок, которые могут привести к фатальному результату.

Вскрываем корпус и разбираем устройство

Первым делом мы должны обесточить устройство. То есть убрать питающее напряжение («питание»). Если питание внешнее – отключаем и даем пару минут полежать (чтобы конденсаторы разрядились). А если питание автономное (аккумуляторное) и внутри корпуса, то переходим к разбору, но помним, что первым делом после вскрытия надо будет отсоединить аккумулятор. Обратите внимание, что не стоит включать устройство с беспроводными модулями связи без антенн (или стоит использовать аттенюаторы). Это может привести к выходу радиочасти устройства из строя.

В большинстве случаев корпус устройства состоит из нескольких частей, скрепленных между собой как минимум защелками. Существуют специальные инструменты для вскрытия корпусов, продаются в наборах и легко находятся в интернете.

Рис. 0.3. Набор инструментов для вскрытия корпусов электронных устройств[1]

Корпус только на защелках – скорее редкость. Почти всегда к ним добавлены винты, и их надо открутить в первую очередь. Винты бывают разные, но маловероятно, что вы не сможете найти подходящую отвертку в большом наборе. А вот расположение винтов иногда бывает неочевидным. Например, под наклеенными резиновыми ножками на днище устройства. Или под этикеткой. Главное – помнить две вещи: в интернете есть инструкции по разбору многих устройств (гуглим по словам teardown и disassemble), возможно, вы даже нашли что-то на этапе сбора информации. И второе: «сила есть – ума не надо», это не про нас. Сломать можно все, что угодно, мы, собственно, за этим тут и собрались. Но не в смысле «пополам». Поэтому если устройство не хочет разби-

[1] https://www.ifixit.com/Store/Tools/Pro-Tech-Toolkit/IF145-307.

раться, усилие надо прикладывать постепенно. Может, где-то все-таки остался винт, скрепляющий половинки корпуса.

Кстати, еще один подвох: разная длина винтов. При сборке обратно можно легко и непринужденно убить устройство, вкрутив другой винт, имеющий длину даже на 1 мм больше. Оригинальный винт не доходит до печатной платы, а более длинный замыкает на ней что-нибудь. Видел последствия подобных ошибок, максимально обидно. Поэтому фотографируйте процесс разбора, подписывайте винты. Можно раскладывать по коробочкам, можно клеить их на малярный скотч. Каждый винт должен вернуться на свое место.

Случаи склеенных корпусов тоже бывают. Тут уже приходится действовать деструктивно (например, с помощью гравера с микрофрезой), использовать растворители клея или прогревать корпус паяльным феном. Хорошо, что таким приходится заниматься нечасто.

В некоторых встраиваемых системах применяются специальные механизмы защиты от вскрытия (tamper protection), стирающие прошивку, ключи шифрования или просто устанавливающие флаг о вскрытии устройства. Например, в ряде дорогих ноутбуков бизнес-класса есть специальный контакт-тампер, срабатывающий на вскрытие задней крышки ноутбука. Подробнее мы рассмотрим механизмы защиты от вскрытия в главе 4.

На этом этапе считаю, что корпус мы вскрыли без потерь, перед нами открылся вид на печатную плату устройства. На всякий случай напоминаю про статическое электричество и необходимость отсоединения аккумулятора. Держите подальше металлический инструмент: отвертки, пинцеты, скальпели. Замкнуть что-то на печатной плате устройства, уронив на нее металлический инструмент, очень легко. У вас точно получится когда-нибудь, не сомневайтесь.

Если печатных плат несколько, то между собой они могут быть соединены разъемами, например именно таким образом соединяются платы промышленного стандарта PCI/104 (он даже взят за основу форм-фактора для сверхмалых спутников CubeSat размером 10×10×10 см, **https://www.mdpi.com/2076-3417/9/15/3110**). Внешний вид современного компьютера (включающего в себя Intel Core i7 8665UE и 64GB RAM) в форм-факторе PCIe/104 показан на фото:

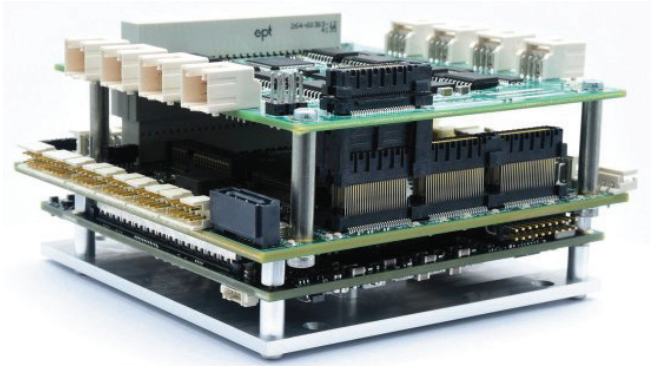


Рис. 0.4. Компьютер в форм-факторе PCIe/104[1]

[1] https://diamondsystems.com/products/gemini.

В большинстве устройств платы соединяются между собой шлейфами и проводами, подключенными к разъемам (коннекторам), пример фотоаппарата Canon – на фото ниже.

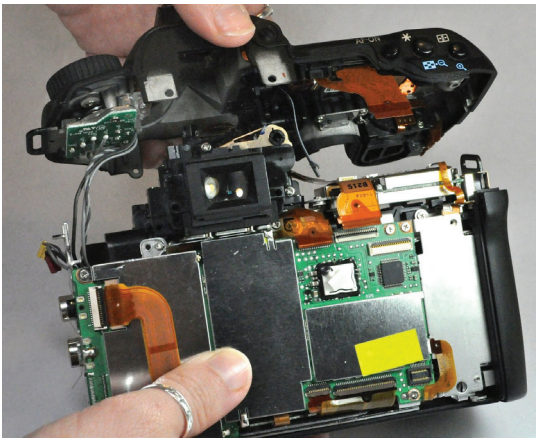


Рис. 0.5. Гибкие шлейфы в устройстве фотоаппарата Cannon[1]

В ряде устройств встречаются гибко-жесткие (rigid-flex) печатные платы. В этом случае шлейф (гибкая часть) является частью печатной платы и не может быть отсоединен. Существуют и чисто гибкие (flex) печатные платы, но в большинстве случаев они используются в качестве шлейфа для соединения жестких плат. Как правило, гибко-жесткие или гибкие платы применяются при плотной компоновке нескольких плат внутри корпуса небольшого устройства, т. к. позволяют располагать платы максимально близко друг к другу. Пример гибко-жесткой печатной платы показан на рисунке.

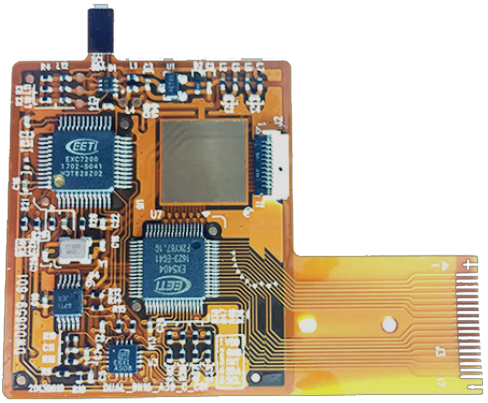


Рис. 0.6. Гибко-жесткая плата[2]

Если разъемы есть, нам нужно их разъединить. Тут все просто, существует всего несколько правил. Первое: найти фиксаторы (например, черная пласти-

[1] https://ifixit.com/Guide/Canon+EOS+40D+Shutter+Button+Replacement/50726.

[2] https://www.fanypcb.com/2-layer-rigid-flex-pcb.html.

ковая деталь на картинке ниже) и разблокировать их. Они могут выглядеть по-разному, но основной смысл – фиксировать соединение. Конструктив бывает разный, информация по разъемам легко находится. Отличным материалом для обучения будут видеоинструкции по разборке устройств на YouTube или обзоры на **https://ifixit.com/**.

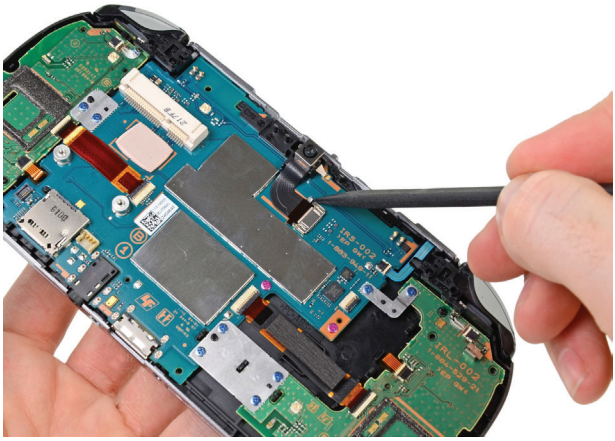


Рис. 0.7. Разъемы на плате устройства[1]

Второе: надо быть аккуратными. Пленочные шлейфы (так называются потому, что проводящие дорожки нанесены на гибкую пленку) легко рвутся или портятся при «переламывании». И если не повезет, можно надломить его и потом долго искать причину, почему устройство не работает, ведь внешне все будет выглядеть целым. Особенно обидно сломать шлейф гибко-жесткой печатной платы, ведь его нельзя заменить.

Не забываем фотографировать все шаги по разборке, времени займет немного, но потом сильно пригодится при обратной сборке устройства.

Из чего состоит плата устройства?

Зачем вообще делать инженерный анализ платы устройства? Часто иначе не достать прошивку, а без прошивки нам дальше делать нечего. Но даже если прошивку уже удалось получить, без информации об аппаратной составляющей устройства исследование может идти медленнее или вообще не получится. Поэтому начинаем анализ платы с изучения общих принципов компоновки.

Плата устройства состоит из непосредственно печатной платы с проводниками и напаянных на нее электронных компонентов. Притом часто говорят «печатная плата», подразумевая плату устройства в сборе с компонентами (еще точнее было бы ввести определения ячейки или электронного модуля, но мы будем оперировать распространенными названиями, пусть и в ущерб точным определениям).

Печатная плата представляет собой «пирог» из проводящих слоев металла (меди) с вытравленными дорожками проводников (4) и слоями диэлектри-

[1] https://ifixit.com/Teardown/PlayStation+Vita+Teardown/7872.

ков (5). Сверху и снизу плату покрывают защитной маской (она еще определяет цвет печатной платы) и наносят надписи с помощью метода шелкографии. На данном этапе нам важно знать следующее: проводящих электричество слоев может быть от одного (тогда это однослойная печатная плата) до нескольких десятков (в очень сложных устройствах, например в материнской плате компьютера). Наиболее распространенным количеством слоев проводников в большинстве печатных плат устройств являются 4, 6 и 8 (так и говорим: «четырёхслойная печатная плата»). Конфигурация «пирога» печатной платы называется стек. Внутренние слои с проводниками позволяют добавить третье измерение в печатную плату и существенно упростить разводку устройства, уменьшить его габариты.

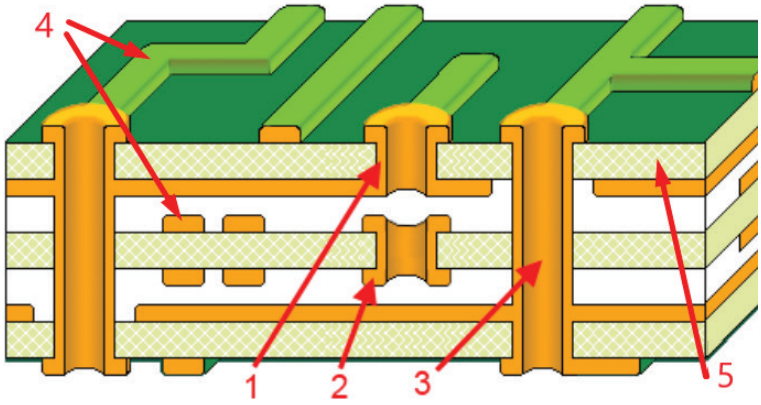


Рис. 0.8. Структура печатной платы[1]

Как правило, часть внутренних слоев используется для передачи питания: слои для «земли» (ground, GND) и для питающих напряжений (V_{CC}). Очевидно, что слои проводников должны как-то соединяться между собой, иначе весь смысл теряется. Делается это с помощью переходных отверстий (VIA) (1, 2, 3). Притом на сложных печатных платах могут быть глухие (1) или скрытые (2) переходные отверстия только между внутренними слоями (но такие ПП намного дороже в производстве). Подобный подход позволяет использовать пространство на внешних слоях печатной платы для размещения компонентов.

Переходные отверстия могут совмещаться вместе с контактной площадкой (VIA in PAD). В таком случае визуально определить наличие переходного отверстия сложно (потому что VIA прячется под дополнительным слоем меди). Кажется, что ножка микросхемы никуда не подключена. Подобное соединение можно обнаружить только прозвонкой или рентгеном (конечно, если нет добытой, например с форума ремонтников, схемы).

Процесс проектирования печатной платы называется трассировкой и делается в специальном ПО (Altium Designer, Cadence Allegro, KiCad и пр.). Часто на печатной плате можно встретить дорожки странного вида, как будто кто-то сделал причудливый узор, на первый взгляд не имеющий практического смысла. Пример распространенных вариантов показан на рисунке ниже.

[1] https://www.altium.com/documentation/altium-nexus/blind-buried-micro-vias.

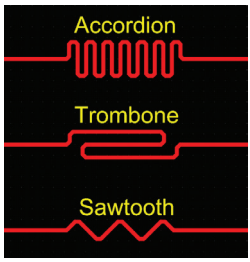


Рис. 0.9. Шаблоны для выравнивая длин проводников на печатной плате устройства[1]

Это выравнивание длин проводников, сигналы по которым должны идти одновременно. Несмотря на то что электрические сигналы распространяются практически со скоростью света (скорость распространения зависит от типа проводника, но в большинстве случаев примерно равна ⅔ от скорости света), при высоких частотах интерфейсов передачи данных дорожки, отличающиеся даже на несколько миллиметров, уже могут быть причиной некорректной работы устройства, т. к. сигналы по ним будут приходить в разное время. Наиболее часто выравнивание встречается для сигналов, передающихся по дифференциальным парам и высокоскоростным параллельным интерфейсам, например DDR. Поэтому когда мы видим на печатной плате эти странные узоры, то сразу понимаем, что это линии какого-то высокоскоростного интерфейса. На фото представлен фрагмент платы Microsoft XBOX, содержащий выровненные по длине проводники.

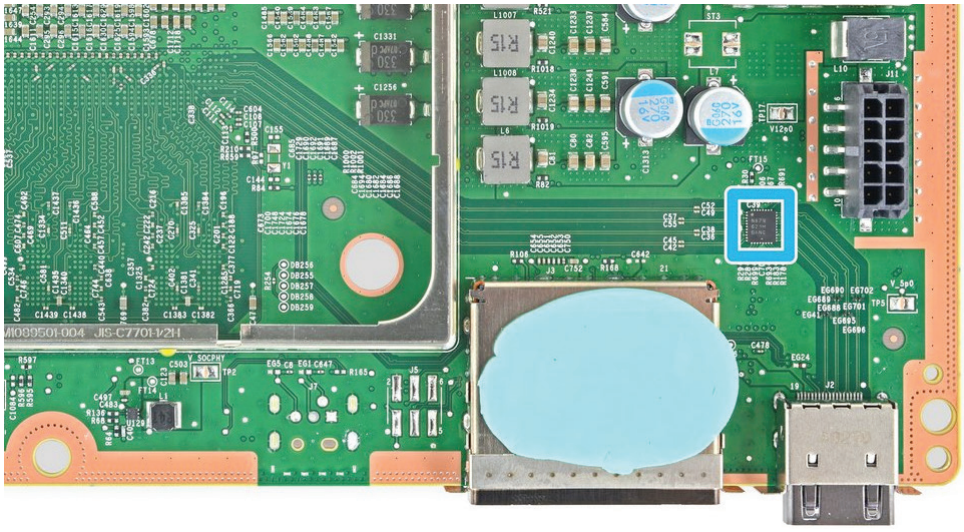


Рис. 0.10. Выровненные проводники на плате устройства[2]

Теперь перейдем к электронным компонентам. Их огромное количество, как и типов корпусов, в которых они существуют. Почти все компоненты мо-

[1] https://www.altium.com/ru/documentation/altium-designer/length-tuning-pcb.
[2] https://ifixit.com/Teardown/Xbox+Series+X+Teardown/138451.

гут быть «упакованы» в разные типы корпусов. Сейчас нас интересует следующее деление: компоненты *выводного* и *поверхностного* монтажа. Отличие – в выводах («ножках») этих компонентов. Наверное, вы видели платы старых устройств, там выводы проходили сквозь печатную плату и запаивались к проводящим слоям с обратной стороны.

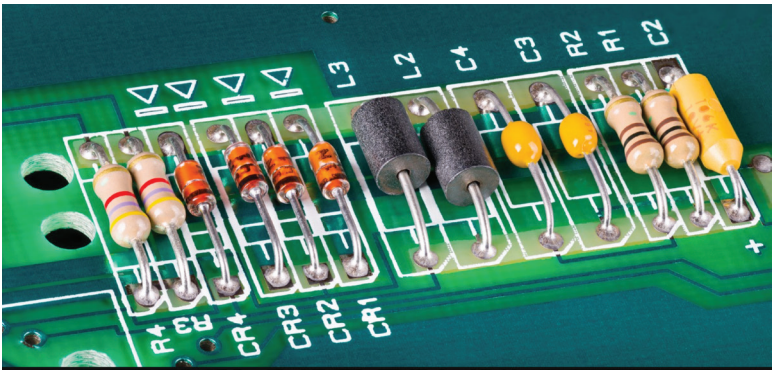


Рис. 0.11. Выводной монтаж компонентов[1]

Сейчас таких компонентов почти не осталось (но, например, их можно встретить в источниках питания, где компоненты могут рассеивать большую мощность). Если компоненты запаиваются с той же стороны, где и находятся, то это *поверхностный* монтаж (такой тип компонентов называется SMD – surface mount devices). Есть разные типы SMD-корпусов: выводные с ножками SOIC, TSOP или безвыводные BGA (Ball Grid Array), WLCSP с шариками контактов под компонентами. Отличие – в плотности и удобстве монтажа. Компонент с BGA-корпусом паяльником уже не запаяешь (да и не отпаяешь). Контакты на печатной плате, к которым подключаются выводы компонентов, называются футпринт (англ. footprint).

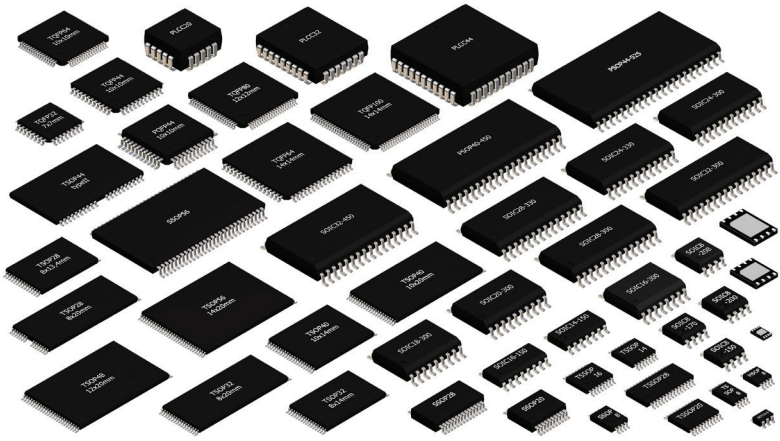


Рис. 0.12. Распространенные корпуса поверхностного монтажа[2]

[1] https://resources.altium.com/p/role-decoupling-inductor-and-resistor-pdn.
[2] https://tened.ru/blog-circuits.

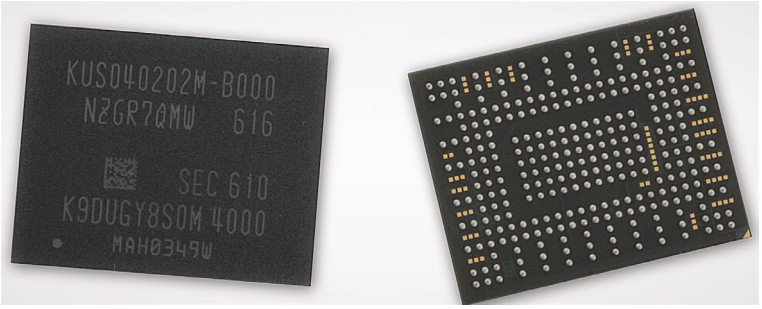


Рис. 0.13. Микросхема с корпусом BGA[1]

Описание процесса производства цифрового устройства

Мы рассмотрели, как выглядит типовое устройство, если вскрыть его корпус. Далее разберем, из каких типовых электронных компонентов состоит устройство, их назначение и какими интерфейсами они могут быть связаны между собой. Но перед этим познакомимся с тем, как выглядит процесс производства устройства на фабриках (не забывая, что разные части устройства могут производиться на разных фабриках).

Типовое цифровое устройство состоит из одной или нескольких электронных плат, установленных в корпус. В первую очередь инженеры разрабатывают электрическую принципиальную схему устройства. После выполняется трассировка печатной платы, результатом которой является полное описание конструкции печатной платы. Далее формируется список компонентов (Bill of Materials, BOM), включающий в себя все компоненты, используемые в плате устройства. Для унификации любое ПО автоматизации проектирования электронных устройств позволяет выгрузить файлы стандарта Gerber, хранящие информацию о проводящем рисунке каждого слоя, отверстия и т. д. Именно эти файлы вместе со списком BOM отправляются на производство. Пример отображения файла стандарта Gerber в ПО просмотра показан на скриншоте ниже.

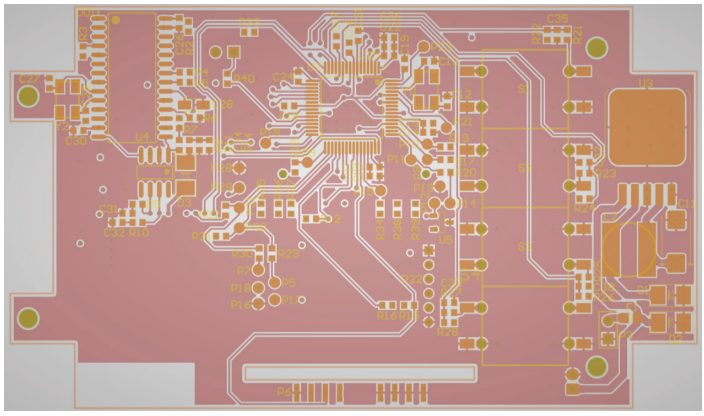


Рис. 0.14. Просмотр файла формата Gerber

[1] https://news.samsung.com/us/512-gigabyte-bga-nvme-ssd-pm971-nvm/.

На выходе первого этапа производства получается печатная плата устройства, полностью покрытая специальной защитной маской, за исключением контактных площадок.

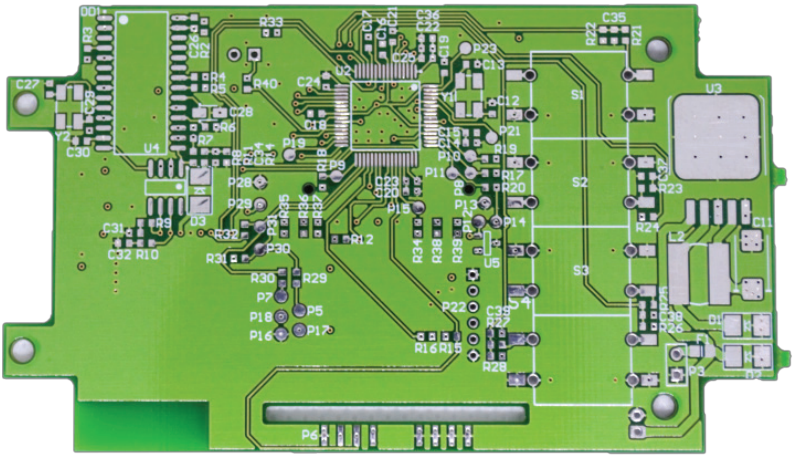


Рис. 0.15. Печатная плата устройства, произведенная на основе файла с форматом Gerber

Далее на контактные площадки наносится паяльная паста (смесь микроскопических шариков припоя с паяльным флюсом) и устанавливаются электронные компоненты (с помощью специальных линий автоматического монтажа) из списка BOM. Печатные платы с установленными компонентами отправляются в специальную печь, где припой, расплавляясь, припаивает выводы компонентов к контактным дорожкам печатной платы. В случае если на печатной плате должны быть установлены элементы, которые не могут подвергаться нагреву в печи, они могут быть запаяны вручную или селективным способом после этапа «запекания» в печи. Далее платы отправляются на процедуры отмывки от остатков флюса и сушки.

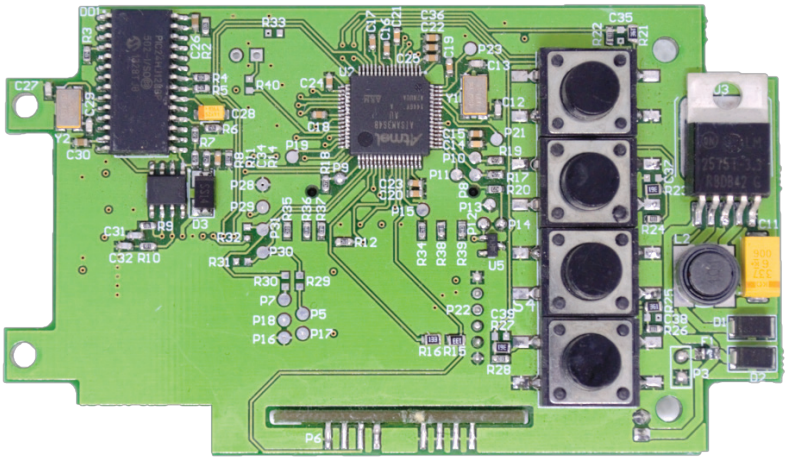


Рис. 0.16. Собранная плата устройства с компонентами

На каждом этапе производства платы устройства существует множество вариантов контроля – оптический, электрический или рентген. Поэтому процент брака при серийном производстве устройства крайне мал. Собранные платы устройства отправляются заказчику или тестируются (и даже программируются) на фабрике. После чего происходят установка плат в корпус и финальное тестирование устройства.

Ознакомившись с общим описанием типового процесса производства серийного цифрового устройства, можно переходить к детальному рассмотрению используемых электронных компонентов. Все компоненты делятся на активные и пассивные. Активные компоненты (микроконтроллер, память, стабилизаторы и т. п.) могут усиливать или преобразовывать электрические сигналы и требуют внешнего источника питания для своей работы. Пассивные элементы (резисторы, конденсаторы, разъемы и т. п.) не нуждаются во внешнем питании. Нас больше всего интересуют активные компоненты. Сейчас наша задача – идентифицировать маркировку основных компонентов и понять их назначение, вбивая маркировку в поисковый запрос и изучая документацию на микросхемы, называемую даташит (datasheet).

Маркировка компонентов на плате устройства

Довольно часто на печатной плате можно встретить надписи (состоящие из одной или нескольких букв и порядкового номера), по которым можно восстановить тип электронного компонента. Ранее мы рассматривали, что процесс нанесения краски на печатную плату устройства называется шелкографией. Рассмотрим основные обозначения на примере отладочной платы фирмы STM Nucleo (стоит учитывать, что иногда производители отходят от распространенных обозначений и маркируют компоненты по-своему):

- U – микросхема (реже встречается обозначение DD для цифровых микросхем и DA для аналоговых);
- J, CN – разъем, JP – перемычка;
- C – конденсатор;
- R – резистор;
- L – индуктивность;
- Q – транзистор;
- VD, D – диод;
- LD – светодиод;
- F – предохранитель;
- X – кварцевый резонатор.

Более подробный список представлен в Википедии (**https://en.wikipedia.org/wiki/Reference_designator**), а полный перечень обозначений элементов, принятых у нас в стране, можно посмотреть в ГОСТ (**http://www.robot.bmstu.ru/files/GOST/gost_2.710-81.pdf**).

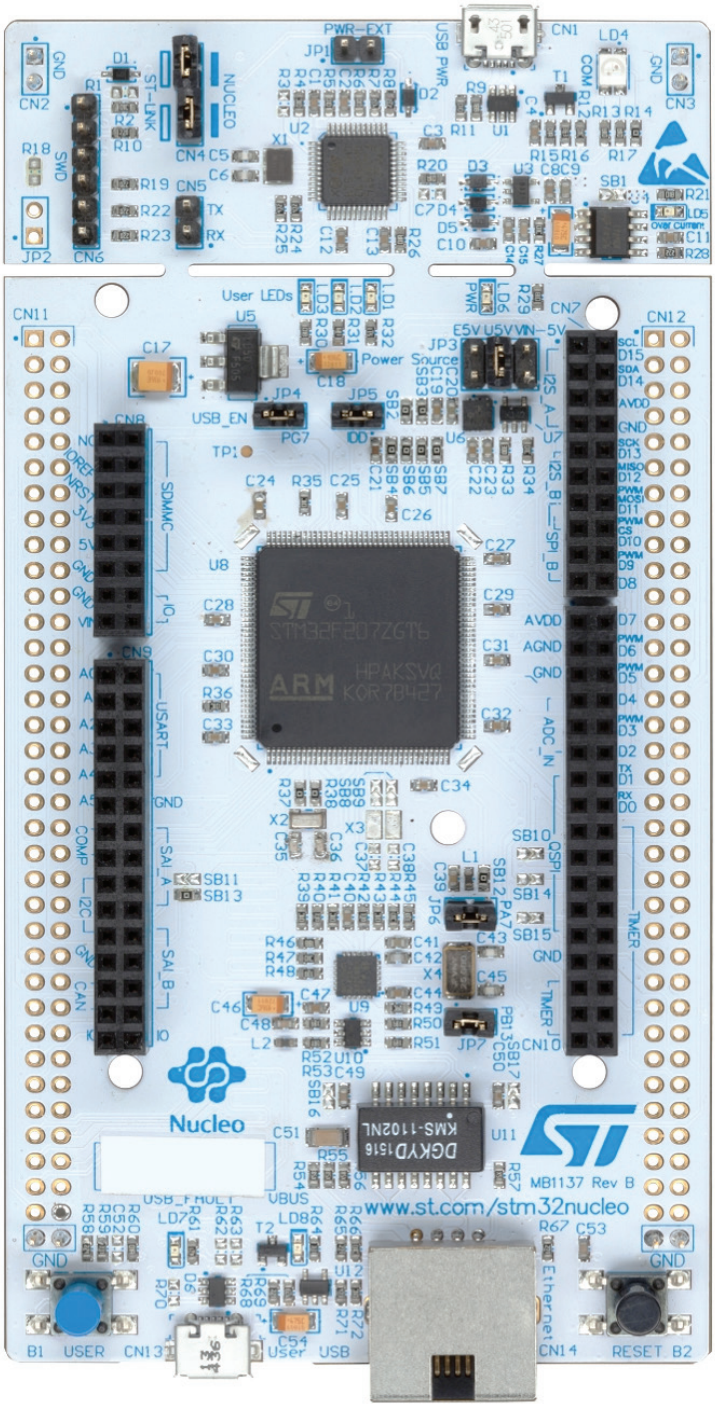


Рис. 0.17. Обозначения компонентов на плате[1]

[1] https://www.codeinsideout.com/blog/stm32/prepare/.

Что же касается маркировки на самом компоненте, то иногда она может быть нанесена очень плохо или умышленно стерта. В таких случаях может помочь немного спирта, нанесенного на корпус микросхемы. Спирт «проявляет» неровности, и под определенными углами обзора маркировка становится гораздо более читаемая. Также может помочь обыкновенный канцелярский корректор. Капните краской на верхнюю часть микросхемы и разотрите ее. Если маркировка имеет углубления, то она проявится. Отдельно отмечу компоненты с малым размером корпусов, на которых маркировка может быть не более трех символов. Если их назначение очень важно для исследования, можно попробовать поискать их маркировку вместе с фразой «ic marking».

Помимо определения маркировки и назначения основных электронных компонентов, нам нужно установить их связь, т. е. соединение между собой. В простых случаях достаточно визуального осмотра, т. к. дорожки на печатной плате устройства хорошо видны. Если плата сложная, то уже не обойтись без «прозвонки» мультиметром. Перед нами сейчас не стоит задача идентифицировать все связи между компонентами, скорее, мы должны ответить на вопрос «Какие есть основные микросхемы и как они связаны между собой?». Зная эту информацию, мы сможем составить функциональную схему устройства. Пример выделения основных компонентов на плате показан на фото.

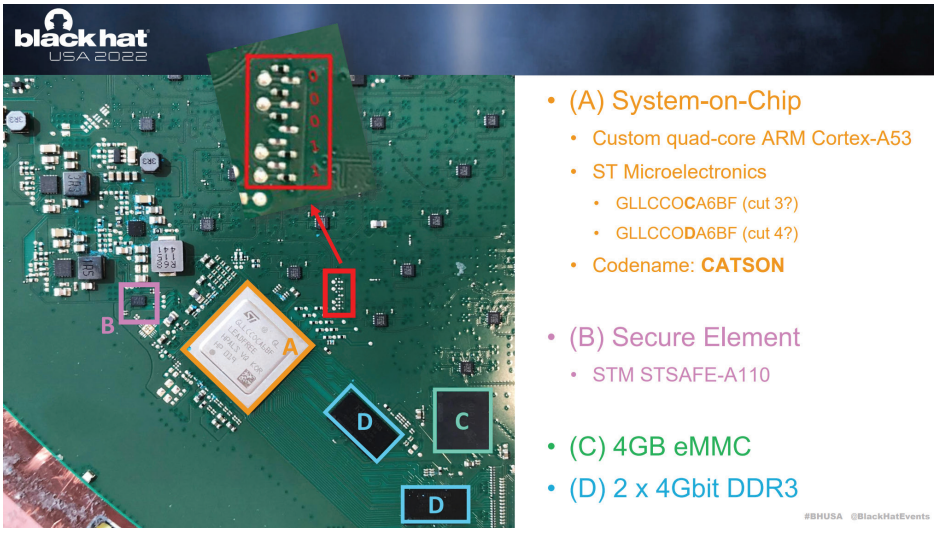


Рис. 0.18. Выделение ключевых компонентов устройства[1]

«Прозвонка» платы устройства

Основной инструмент, позволяющий восстановить схему подключения компонентов на плате устройства, – это мультиметр. Наверняка вы знаете, что им можно измерять множество величин – напряжение, ток, сопротивление, емкость и т. д. Но также в нем есть режим проверки целостности электрической цепи (иногда объединенный с режимом проверки диода), как правило, обозначаемый символом нескольких звуковых волн (показан на рис. 0.19).

[1] https://i.blackhat.com/USA-22/Wednesday/US-22-Wouters-Glitched-On-Earth.pdf.

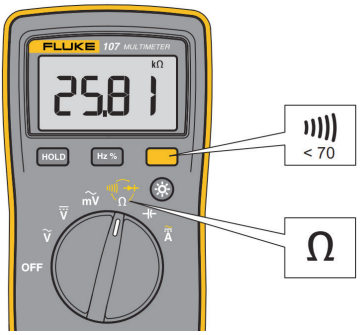


Рис. 0.19. Выбор режима проверки целостности электрической цепи[1]

В режиме проверки целостности, при замкнутой цепи и сопротивлении между щупами мультиметра меньше 70 Ом, звучит звуковой сигнал, отсюда и название режима – «прозвонка». Когда звука нет – цепь разомкнута или в ней присутствуют компоненты, увеличивающие сопротивление или не пропускающие ток. Например, обычный диод пропускает ток только в одном направлении.

> ***Не проверяйте целостность цепи при подключенном питании устройства! Это почти гарантированно приведет к выходу из строя отдельных блоков внутри микросхем.***

Допустим, нам надо найти, куда подключается определенная ножка интересующей нас микросхемы. Активируем на мультиметре режим проверки целостности цепи и один щуп мультиметра располагаем на интересующем нас выводе микросхемы. Вторым щупом мультиметра проверяем (можно аккуратно проводить по всем ножкам микросхемы) потенциальные места, куда должен приходить интересующий нас электрический сигнал. Если раздается длительный звуковой сигнал, значит, между контактами, к которым подключены щупы мультиметра, есть связь. Пример показан на фото.

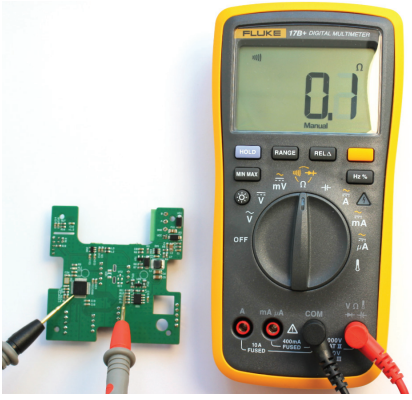


Рис. 0.20. Процесс восстановления схемы электрической цепи с помощью мультиметра

[1] www.vseinstrumenti.ru/instruction/fluke-107-695427.pdf.

Данный способ восстановления схемы электрической цепи имеет ряд особенностей, про которые надо помнить при работе.

1. Диод пропускает ток только в одном направлении – от анода к катоду. Но иногда диоды выходят из строя и начинают пропускать ток в обратном направлении.
2. Прозвонка цепи имеет полярность – ток течет от плюса к минусу (точнее, от большего потенциала к меньшему). Поменяв щупы местами в их точках подключения к плате, вы можете получить другие результаты, в том числе и из-за диодов.
3. Выводы микросхем могут быть окислены или покрыты лаком / несмытым флюсом. Это может затруднить нахождение нужного вывода из-за плохого контакта.

Зная возможные варианты маркировки основных компонентов на плате устройства и вооружившись мультиметром с режимом проверки целостности электрической цепи, пора переходить к идентификации ключевых компонентов и связей между ними.

Микроконтроллер

У любого цифрового устройства должен быть кто-то главный, отвечающий за основную логику работы. Этим главным будет микроконтроллер. Внутри микроконтроллера есть исполнительное ядро. Оно может быть построено на базе различных архитектур (например, ARM), и подробнее мы об этом поговорим в главе 2, когда перейдем к анализу прошивки. Сейчас нам надо запомнить, что микроконтроллер состоит из процессорного ядра, ОЗУ (RAM), скорее всего, ПЗУ (ROM и EEPROM) и набора периферийных интерфейсных контроллеров, позволяющих подключать к ядру внешние устройства посредством стандартных интерфейсов (I2C, SPI и т. д.). В этом заключается принципиальное отличие микроконтроллера от микропроцессора, у которого память и периферийные контроллеры не расположены на одном кристалле с вычислительным ядром.

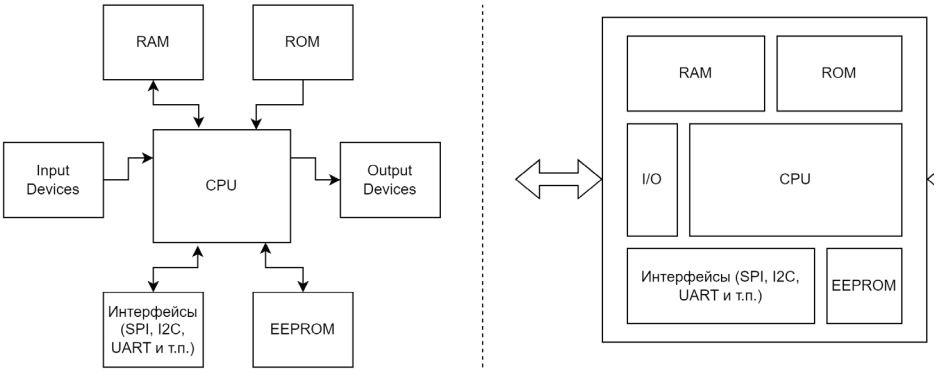


Рис. 0.21. Структурные схемы микропроцессора (слева) и микроконтроллера (справа)

Конечно, на плате может быть несколько микроконтроллеров, занимающихся обработкой разной информации. Но главный должен быть кто-то один.

> *В особо критичных условиях применения используется подход, когда главного нет, а есть, например, три идентичные схемы с отбраковкой одного «выбивающегося» результата работы, если он есть. Так называемый мажоритарный принцип, принцип большинства. Такую архитектуру можно увидеть только в контроллерах АСУТП, авионике и других крайне специфических устройствах, вряд ли вы начнете свое погружение в мир исследований встраиваемых систем с подобных устройств.*

Как и любой другой активный компонент, микроконтроллер на плате обозначается как U (или DD). Отличить его от иных элементов можно по двум косвенным признакам: первый – это большое количество выводов / большой физический размер (конечно, у этого правила есть исключения, но в 90 % случаев самый «ногастый» элемент – это микроконтроллер). Второй признак – «центральное» расположение микроконтроллера на плате. Так как микроконтроллер должен подключаться к другим компонентам, то наиболее логично поместить его в центр платы, а остальные элементы расположить вокруг.

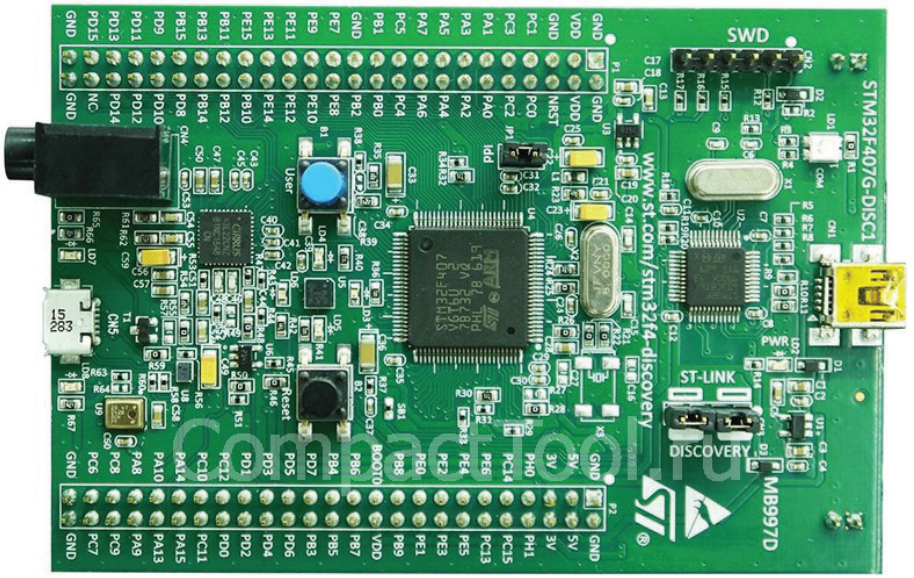


Рис. 0.22. Расположение микроконтроллера в центре платы устройства[1]

Одними из наиболее распространенных микроконтроллеров в простых устройствах являются микроконтроллеры фирмы STM на базе ядра ARM. Они бывают разных серий, но все обозначения начинаются с букв ST. Увидели такой – перед вами хороший кандидат на роль «главного». В последнее время появилось много клонов и аналогов микроконтроллеров STM32, их маркировка очень похожа и в большинстве случаев начинается с букв GD (GigaDevice), CS (CKS), CH (WCH) и т. д.

[1] https://compacttool.ru/otladochnaya-plata-stm32f4discovery.

Так как больше всего нас интересует прошивка, надо понять, где она может храниться. Как правило, большинство микроконтроллеров имеют встроенный первичный загрузчик (bootloader), расположенный в Read Only Memory (ROM), неизменяемой памяти внутри микроконтроллера. В случае если bootloader хранится в ROM, он носит название BootROM. Если не вдаваться в технические особенности (потому как на самом деле все может быть сложнее, но сейчас нам это не важно), основной смысл загрузчика – быть первым кодом при старте микроконтроллера и загрузить прошивку (или другой загрузчик). Если микроконтроллер имеет *специальное назначение* (т. е. изначально спроектирован для работы в определенном классе устройств и «заточен» под них), то, скорее всего, в ROM будет храниться часть прошивки. Ее функции гораздо шире, чем у простого bootloader'а, и будут «зашиты» на этапе производства микроконтроллера и не могут быть изменены впоследствии, т. к. используемый для этих целей тип памяти не предполагает возможности изменения. Например, BootROM может проверять целостность загружаемых далее загрузчиков и выполнять первичную инициализацию устройства. В случае микроконтроллеров *общего назначения*, т. е. не спроектированных изначально для выполнения конкретных функций, в BootROM расположен только простой загрузчик или ROM-памяти (и загрузчика) может не быть вообще. Вместо него, как правило, внутри микроконтроллера есть перепрограммируемая flash-память, содержимое которой можно менять и код из которой может быть исполнен сразу после включения микроконтроллера. Причины такого разделения сугубо экономические. Перепрограммируемая память стоит значительно дороже, а когда микроконтроллеры выпускаются сотнями миллионов экземпляров, экономится каждый цент.

Дамп памяти (memory dump) BootROM представляет для нас интерес, и мы будем пытаться считывать его в главе 1. Функции устройств и, следовательно, их прошивок бывают разные. Поэтому кому-то может не хватить встроенной памяти внутри микроконтроллера для реализации всех функций. Что делать в этом случае? Правильно, использовать внешнюю память, расположенную в отдельной микросхеме.

Память

Память бывает энергозависимая (т. е. для хранения содержимого ей нужно питание) и энергонезависимая (питание нужно только для изменения состояния). Энергозависимая память в подавляющем большинстве случаев представлена в виде оперативной памяти (ОЗУ, RAM). Наверняка вы слышали про DDR-память в своем компьютере. Вот это она. Питание отключили – содержимое пропало. Из плюсов – быстрый доступ к содержимому памяти. В ней хранятся данные в процессе работы микроконтроллера (и, возможно, какая-то часть кода прошивки). Аналогично энергонезависимой памяти (flash или ROM), определенный размер RAM часто есть внутри кристалла микроконтроллера, но не всегда его достаточно, поэтому для расширения можно использовать микросхемы внешней оперативной памяти. Если перевести на язык «взрослых» систем, т. е. x86-64, стоящих в большинстве компьютеров, то DDR – внешняя память RAM, а кеш процессора – внутренний RAM.

Независимо от типа памяти, «упакована» она в отдельную микросхему и должна как-то подключаться к микроконтроллеру. Обозначается также буквой U (DD) и, как правило, располагается недалеко от микроконтроллера. А вот интерфейсы подключения могут быть параллельные и последовательные. Рассмотрим основные типы памяти в разрезе физических интерфейсов, используемых для подключения памяти к микроконтроллеру.

I2C

Если вы увидели восьмиконтактную микросхему с маркировкой *24*, значит, существует большая вероятность, что вы нашли микросхему ПЗУ с интерфейсом I2C.

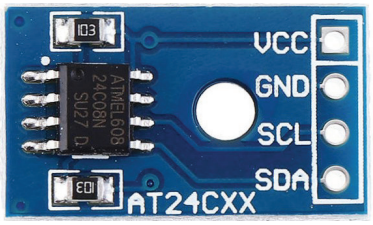


Рис. 0.23. Микросхема ПЗУ с интерфейсом I2C[1]

Поддержка I2C есть практически в любом микроконтроллере, ее довольно просто реализовать программно. Из минусов памяти с таким интерфейсом можно отметить малый объем. Обычно он составляет десятки-сотни килобит, хотя есть экземпляры на пару мегабит. Учитывая скромные скоростные характеристики интерфейса, делать больший объем просто нет смысла. Корпуса, как правило, типа SOIC или WLCSP, но есть и редкие, четырехконтактные. Подробнее сам интерфейс I2C мы рассмотрим в разделе «Интерфейсы связи компонентов» немного позднее.

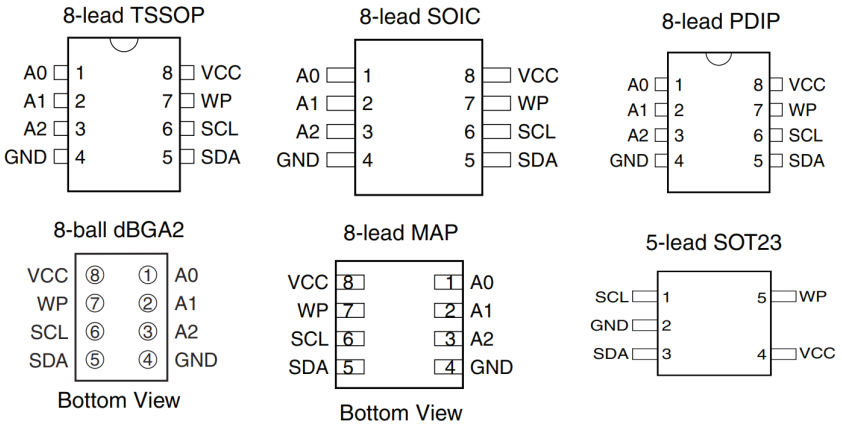


Рис. 0.24. Распространенные типы корпусов и назначение выводов микросхем ПЗУ с интерфейсом I2C[2]

[1] https://aliexpress.ru/item/1005002613837668.html.
https://static.chipdip.ru/lib/115/DOC013115377.pdf.

SPI

ПЗУ с этим интерфейсом очень распространены. Если в I2C зацепкой для нас была маркировка микросхемы **24**, то в SPI одни из самых распространенных видов микросхем будут иметь маркировку *25*, *45* и те же 8 ножек-контактов.

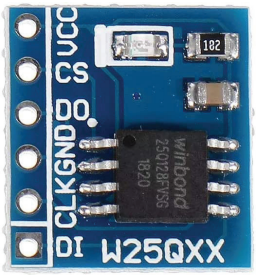


Рис. 0.25. Микросхема ПЗУ с интерфейсом SPI[1]

Частота работы интерфейса уже может составлять десятки и сотни мегагерц, что накладывает определенные ограничения на трассировку и требует выравнивания дорожек на печатной плате. Объем памяти может достигать нескольких сотен мегабит.

Интересная особенность интерфейса заключается в «костыле», который придумали для увеличения скорости работы уже существующего интерфейса – увеличения количества передающих линий. Так появились режимы работы Dual SPI (две линии передачи данных) и Quad SPI (четыре линии). Подробнее интерфейс SPI мы рассмотрим в разделе «Интерфейсы связи компонентов».

Самые распространенные корпуса микросхем памяти с SPI-интерфейсом – это восьмиконтактные SOIC, но встречаются и другие:

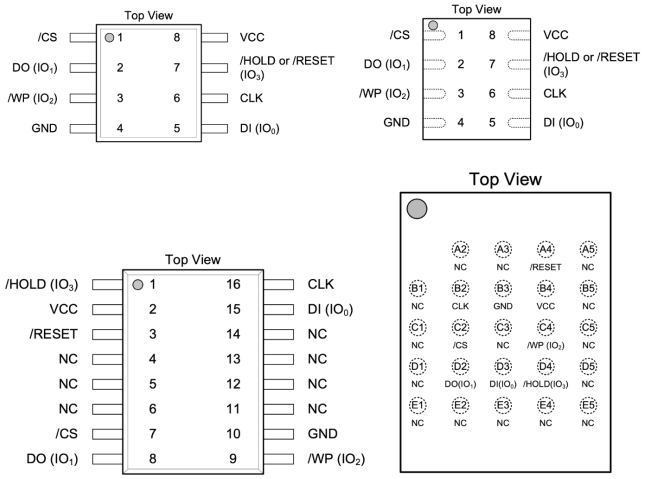


Рис. 0.26. Распространенные типы корпусов и назначение выводов микросхем ПЗУ с интерфейсом SPI[2]

[1] https://www.amazon.co.uk/W25Q32-W25Q128-Capacity-Storage-Interface/dp/B082G4F9SC.

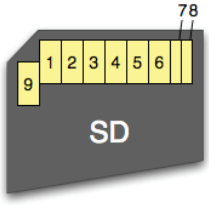
[2] https://www.compel.ru/item-pdf/7ef65a44791dfea423f712ef1bd7931b/pn/winbond~w25q32fvssig-.pdf.

SD и eMMC

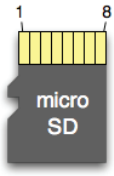
Эти интерфейсы используются в хорошо знакомых всем нам картах памяти SD и микро-SD (давным-давно их предшественники назывались MMC – Multi Media Card). Основное отличие карт памяти SD от eMMC (embedded MMC) – это их форм-фактор.

Рис. 0.27. Внешний вид карт памяти SD, микро-SD и микросхемы eMMC[1]

Карты памяти имеют интерфейсный разъем и устанавливаются в слот (специальный коннектор) на плате устройства, как правило, их всегда можно извлечь без вскрытия корпуса устройства. Память с интерфейсом eMMC выполнена в виде микросхемы в корпусе BGA и предназначена для запаивания на печатную плату устройства. При этом количество распространенных корпусов микросхем eMMC уже достигло десятка. Если внимательно посмотреть на распиновку микросхем eMMC, то можно заметить, что из примерно 150 выводов используется не больше 20, притом половина из них служит для питания.



Pin	SD	SPI
1	CD/DAT3	CS
2	CMD	DI
3	VSS1	VSS1
4	VDD	VDD
5	CLK	SCLK
6	VSS2	VSS2
7	DAT0	DO
8	DAT1	X
9	DAT2	X



Pin	SD	SPI
1	DAT2	X
2	CD/DAT3	CS
3	CMD	DI
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	DO
8	DAT1	X

Рис. 0.28. Назначение выводов карт памяти SD и микро-SD[2]

[1] https://4pda.to/forum/index.php?showtopic=1043267&st=0.
[2] https://4pda.to/forum/index.php?showtopic=1043267&st=0.

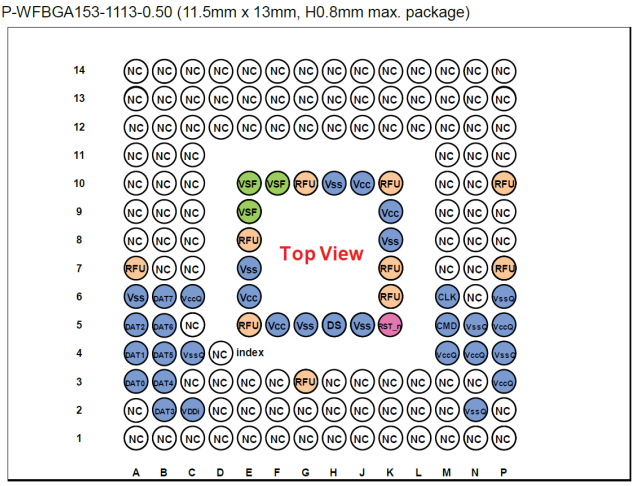


Рис. 0.29. Назначение выводов микросхемы eMMC в корпусе BGA-153[1]

Интересная особенность памяти с интерфейсом SD/eMMC заключается в том, что при инициализации данные передаются только через одну линию DAT0, но после могут быть переключены на четырех- (DAT0-3) или восьми- (DAT0-7, только для eMMC) битные режимы. На этом особенности не заканчиваются. Помните, мы говорили, что SPI – самый широко применяемый интерфейс? Так вот, память SD/eMMC умеет работать через него (очень редко встречаются дешевые экземпляры, не поддерживающие режим SPI)! Да, скорость значительно упадет, но и поддержки полноценного интерфейса SD со стороны микроконтроллера не нужно. Поэтому во многих устройствах, где используются «простые» микроконтроллеры, память SD работает именно через SPI.

Parallel NOR Flash

Как правило, память с интерфейсом данного типа имеет маркировку x29xxx, x39xxx и т. п., встречается в микроконтроллерах, обладающих большим количеством выводов, т. к. для подключения используются шины адреса и данных, состоящие из пары десятков параллельных проводников (отсюда название типа памяти).

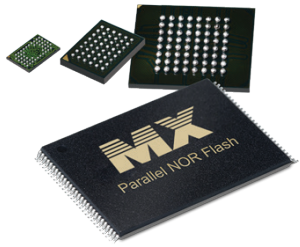


Рис. 0.30. Распространенные типы корпусов ПЗУ типа Parallel NOR Flash[2]

[1] https://www.tindie.com/products/voltlog/emmc-wfbga153-to-microsd-card-adapter-set-of-2/.
[2] https://www.mxic.com.tw/en-us/products/NOR-Flash/Parallel-NOR-Flash/Pages/default.aspx.

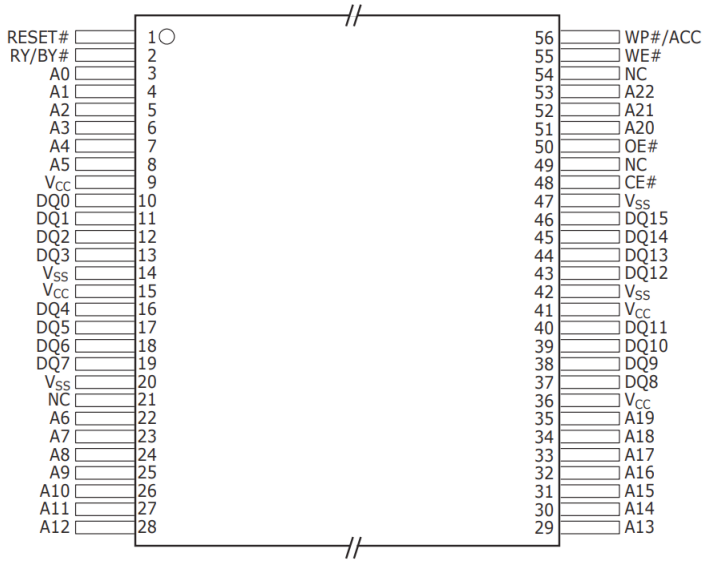


Рис. 0.31. Назначение выводов микросхемы ПЗУ NOR flash в корпусе TSOP-56[1]

Объем памяти может составлять от нескольких мегабит у устаревших моделей до нескольких гигабит у современных. Память данного типа имеет простой интерфейс, который может быть реализован программно в прошивке микроконтроллера. Распространенные корпуса – TSOP для памяти маленьких объемов и BGA для больших.

NAND Flash

Один из самых распространенных типов энергонезависимой памяти. Память именно этого типа находится внутри SSD, USB и SD-флешек (и eMMC).

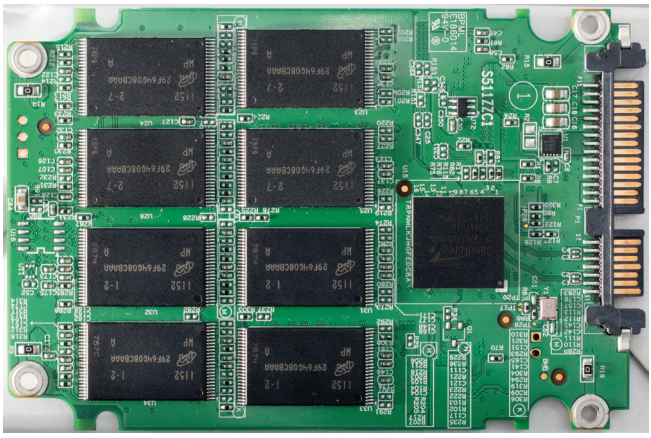


Рис. 0.32. ПЗУ NAND flash на плате SSD диска[2]

[1] https://www.infineon.com/dgdl/Infineon-S29PL-J_128-_128-_64-_32-MBIT_(8_8_4_2M_X_16_BIT)_3_V_FLASH_WITH_ENHANCED_VERSATILEIO-DataSheet-v07_00-EN.pdf.

[2] http://vlo.name:3000/hw/ssd/pcb/sf/adata-s511-120(2281b1-16xCBAAA)-pcb0.JPG.

Интерфейс памяти параллельный, с шириной линий данных 8–16 бит и общим количеством задействованных линий около 20–30 шт., в зависимости от используемой ширины линий данных.

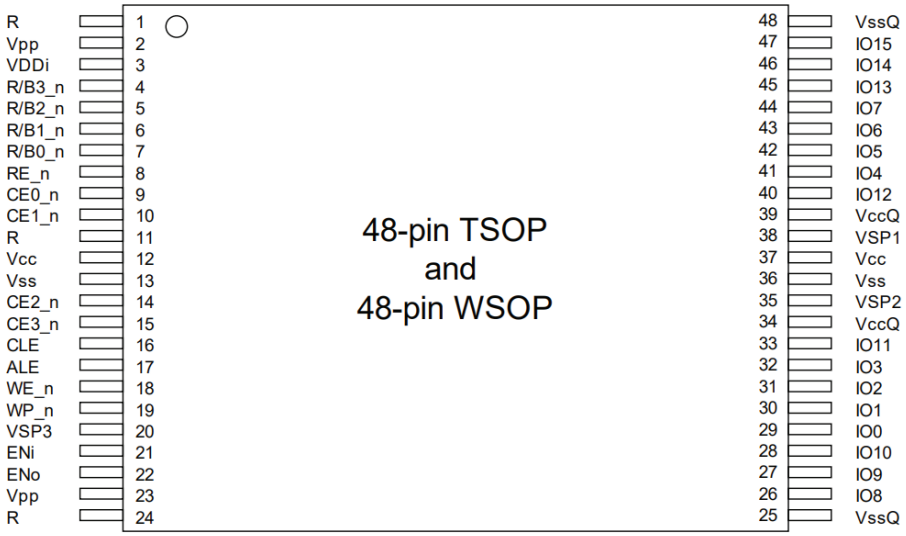


Рис. 0.33. Назначение выводов микросхемы ПЗУ NAND flash в корпусе TSOP-48[1]

Наиболее распространены выводной корпус TSOP48 или различные BGA-корпуса. Самыми известными производителями памяти являются Micron (Intel), Toshiba, Hynix и Samsung.

Для работы с данным типом памяти внутри микроконтроллера должен быть специальный NAND-контроллер, т. к. интерфейс и система команд довольно сложны как с точки зрения реализации логики работы, так и с точки зрения схемотехники (8–16 бит данных/адреса, сигналы выбора банков памяти, синхронизации, выбор типа передачи адрес/данные и т. д.). Поэтому такая память поддерживается только «продвинутыми» микроконтроллерами, которые предполагается использовать для обработки и хранения больших объемов информации.

Память с интерфейсом PCI-e (NVMe) / UFS

Когда скоростей передачи данных перечисленных ранее интерфейсов стало не хватать, на свет появились микросхемы ПЗУ с интерфейсом PCI-e (Non-Volatile Memory Express) и UFS. Такая память используется только в высокопроизводительных устройствах, обрабатывающих и хранящих большие объемы данных. В большинстве случаев именно такой интерфейс памяти используется в вашем мобильном телефоне или планшете. Интерфейс подключения таких микросхем представляет собой несколько дифференциальных пар. Во встраиваемых системах встречается не так часто, т. к. требует поддержки со стороны микроконтроллера. Корпуса – только BGA.

[1] https://media-www.micron.com/-/media/client/onfi/specs/onfi_5_0_gold.pdf?la=en&rev=b9d79143b14143a7a8253c1ae20b247c.

SRAM

Static Random Access Memory. Первый представитель оперативной памяти в нашем списке. Отличительная особенность большинства микросхем – нестандартное расположение выводов корпуса TSOP (хотя есть исключения), выводы расположены на длинных сторонах корпуса. Интерфейс памяти параллельный, отсюда и такое количество выводов. Огромный плюс SRAM-памяти – это простой интерфейс для общения (установка адреса на одних линиях связи и доступ к данным на других), в отличие от SDRAM. А также быстрый доступ к любому адресу за одинаковое время. Из минусов – малый объем и высокая цена.

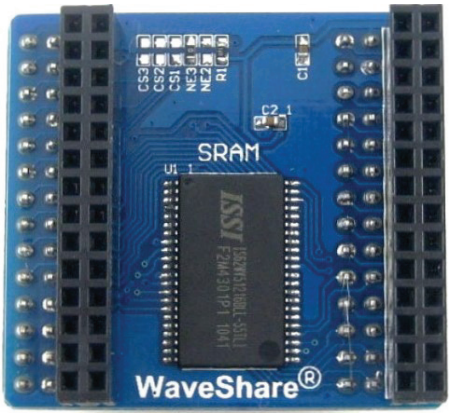


Рис. 0.34. Микросхема ОЗУ SRAM с параллельным интерфейсом[1]

Распространенные корпуса – TSOP для памяти маленьких объемов и TQFN/BGA для больших.

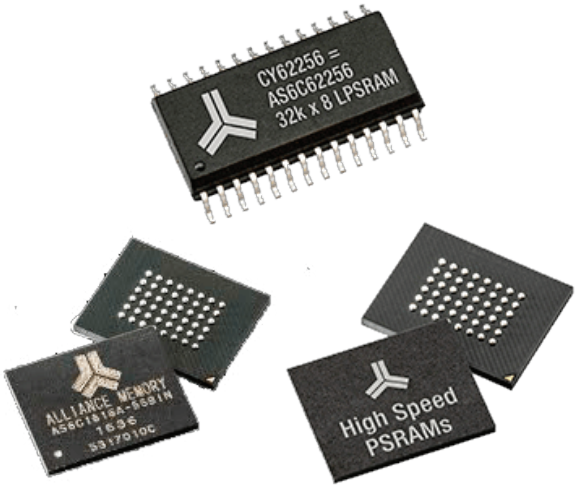


Рис. 0.35. Распространенные типы корпусов для микросхем ОЗУ SRAM[2]

[1] https://www.waveshare.com/is62wv51216bll-sram-board.htm.

[2] https://www.neumueller.com/en/produktgruppe/sram-memory-ics.

SDRAM

Synchronous Dynamic Random Access Memory. Тип памяти с произвольным доступом, которая должна постоянно обновляться, чтобы хранить информацию (т. к. в основе ячеек памяти лежат конденсаторы, быстро теряющие заряд). Несколько десятков лет назад существовали микросхемы асинхронной DRAM-памяти, развитием которой как раз является SDRAM. Микросхемы памяти типа SDRAM стоят в хорошо всем известных планках ОЗУ персональных компьютеров. Во встраиваемых системах может встречаться как обычная SDRAM-память, так и DDR SDRAM. Существует несколько стандартов DDR SDRAM: DDR, DDR2, DDR3, DDR4 и даже DDR5. Уже ведется разработка стандарта DDR6. Для нас важно знать, что отличия между ними – в частоте работы интерфейса, следовательно, вырастают требования к печатной плате и уровню специалистов, занимающихся ее трассировкой. Интерфейс параллельный. SDRAM-память требует наличия специального блока внутри микроконтроллера, что сразу ограничивает ее применение в дешевых устройствах. Микросхемы SDRAM-памяти старых стандартов бывают в корпусе TSOP, а новые – только в корпусе BGA. Во встраиваемых системах в основном применяются микросхемы поколений SDRAM DDR-DDR3.

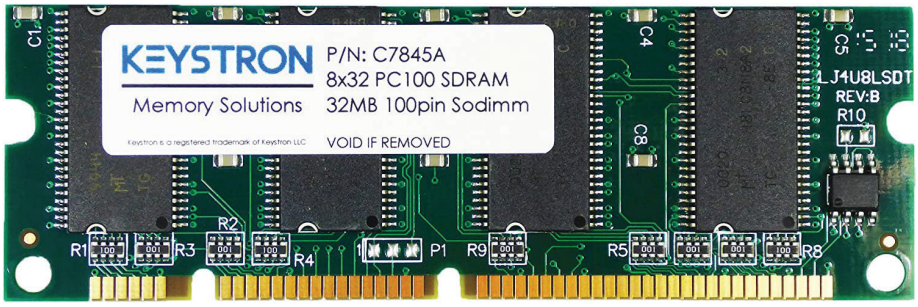


Рис. 0.36. Микросхемы ОЗУ SDRAM в корпусе TSOP на плате модуля памяти[1]

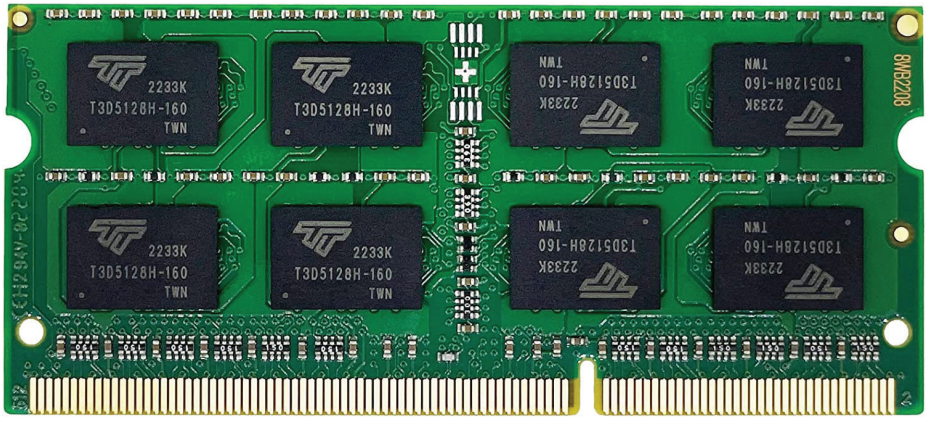


Рис. 0.37. Микросхемы ОЗУ SDRAM в корпусе BGA на плате модуля памяти[2]

[1] https://www.amazon.com/Keystron-Compatible-C7845A-C4143A-Laserjet/dp/B00FAZ0W2Q.

[2] https://www.amazon.com/Timetec-PC3L-12800-Unbuffered-Notebook-Computer/dp/B0145WDNI4.

В устройствах, где инженерам требуется расположить на печатной плате электронные компоненты максимально плотно, встречаются микросхемы, внутри которых могут быть упакованы несколько кристаллов памяти. Например, в мобильных телефонах часто используется BGA-микросхема SDRAM DDR4/DDR5 памяти, совмещенная с NAND- или eMMC-памятью (такой чип носит название eMCP). В большинстве случаев кристаллы микросхем сильно меньше корпусов, в которые они упакованы. Поэтому в одной микросхеме могут сосуществовать два разных типа памяти, ведь количества выводов BGA-микросхем с запасом хватит на всех.

FPGA

Field-Programmable Gate Array (программируемая пользователем вентильная матрица) – микросхема, внутреннее устройство которой («схема») может быть сконфигурировано после изготовления. FPGA являются более мощной и современной версией микросхем CPLD (Complex Programmable Logic Device, программируемая логическая интегральная схема (ПЛИС)). В большинстве современных упоминаний под ПЛИС подразумевается FPGA.

Зачем нужны микросхемы FPGA (CPLD)? Если говорить упрощенно, они позволяют программировать логику работы своей аппаратной схемы. То есть, с одной стороны, чип как бы имеет физическую реализацию схемы и позволяет быстро обрабатывать любую информацию (аппаратная реализация схемы всегда быстрее программной). С другой стороны – разработчик может менять физическую реализацию схемы с помощью переконфигурирования (перепрограммирования). За счет этого достигается высокая скорость исправления ошибок (не нужно делать новую версию микросхемы для исправления ошибки в логике ее работы). Также такой подход оправдан при мелкосерийном производстве, ведь разработка новой микросхемы с определенной логикой работы стоит сотни тысяч и даже миллионы долларов, а микросхемы FPGA стоят десятки и сотни долларов.

Микросхемы FPGA, как правило, используются в устройствах, где нужна высокоскоростная обработка сигналов или сложная коммутация множества сигналов, особенно в динамически меняющихся условиях. Например, они стоят в сетевых коммутаторах и межсетевых экранах корпоративного класса, в промышленных контроллерах и в некоторых мультимедиаустройствах. Отдельно выделю микросхемы типа CPLD – они могут стоять в более широком спектре устройств, так как стоят единицы долларов, но позволяют решить вопрос с коммутацией небольшого количества сигналов или ускорить работу простой аппаратной схемы. Еще одно преимущество микросхем CPLD – низкое время готовности к работе, фактически сразу после подачи питания. В отличие от них, микросхемам FPGA требуется значительное (несколько десятков или даже сотен миллисекунд) время на загрузку и настройку своей конфигурации.

Некоторые цифровые устройства могут даже не иметь управляющего микроконтроллера, вся управляющая логика работы устройства может быть построена на базе микросхемы FPGA. При этом существуют два подхода. Первый – это реализация логики устройства на основе конечного автомата внутри FPGA. Такой подход будет оправдан только для малого количества состояний, т. е. довольно простых с точки зрения логики работы устройств. Второй подход

основан на использовании программной реализации микропроцессора внутри FPGA. Современные FPGA настолько мощные, что позволяют реализовать внутри полноценный микропроцессор, работающий с частотой несколько сотен мегагерц.

Найти микросхемы FPGA (CPLD) при первичном анализе устройства легко, т. к. они, как правило, выполнены в корпусах с большим количеством выводов, не важно, планарных или BGA (в этом случае стоит ориентироваться на значительную площадь микросхемы). Большое количество выводов нужно, т. к. изначально микросхемы предназначены для коммутации большого количества линий данных и обработки передающейся по ним информации.

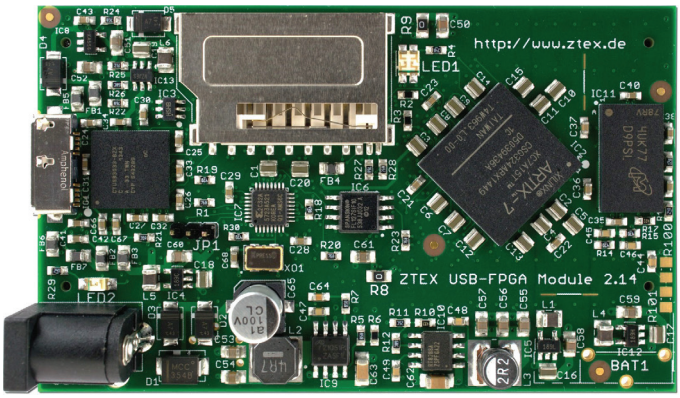


Рис. 0.38. Устройство на базе микрохемы FPGA[1]

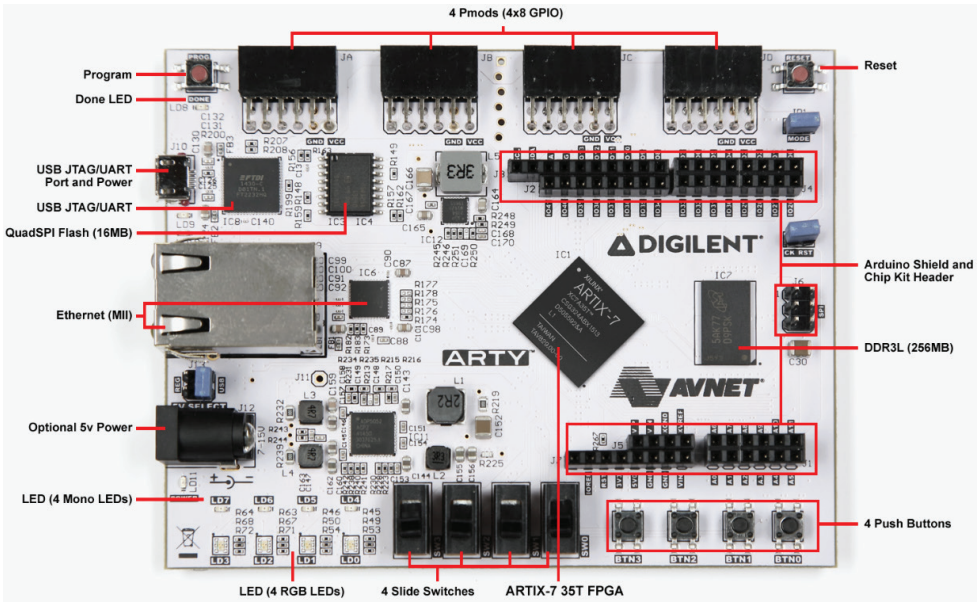


Рис. 0.39. Отладочная плата на базе микросемы FPGA[2]

[1] https://www.ztex.de/usb-fpga-2/usb-fpga-2.14.e.html.

[2] https://www.xilinx.com/products/boards-and-kits/arty.html.

Абсолютное большинство микросхем FPGA производится двумя компаниями – Xilinx (куплена AMD) и Altera (куплена Intel). Существует еще несколько производителей подобных микросхем, однако их суммарная доля рынка относительно невелика.

Рядом с микросхемой FPGA в большинстве случаев будет находиться микросхема ПЗУ с интерфейсом SPI. В ней хранится конфигурация, загружаемая при старте микросхемы FPGA и настраивающая логику работы схемы.

Существуют микросхемы FPGA с внутренним ПЗУ для хранения конфигурации. Микросхемы CPLD всегда содержат внутреннее интегрированное ПЗУ для хранения конфигурации.

Модули связи + антенны

Более правильным названием модулей связи являются приемопередающие модули. Найти их при первичном анализе устройства сильно проще, чем специализированные компоненты, т. к. когда вы анализируете устройство, вы в любом случае что-то знаете о его характеристиках и функциональных возможностях. То есть вы вряд ли будете анализировать устройство и с удивлением обнаружите, что оно умеет работать по интерфейсу Bluetooth, Wi-Fi, LoRa или какому-то другому беспроводному интерфейсу. Из описания устройства вы уже знаете, что где-то должен быть расположен модуль связи.

Модуль связи, как правило, представляет из себя отдельную печатную плату, на которой находится специализированный микроконтроллер, обрабатывающий беспроводные протоколы. На плате модуля связи также находится набор элементов, необходимых для формирования приемопередающих радиотрактов (резисторы, конденсаторы, катушки индуктивности, усилители и т. д.). Модули связи чувствительны к помехам (и при этом сами их создают), поэтому часто имеют защитный металлический экран. Но не под всеми экранами на устройстве обязательно расположены приемопередающие модули.

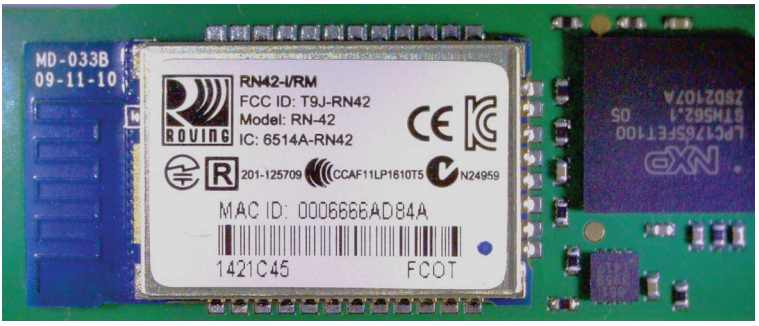


Рис. 0.40. Приемопередающий модуль Bluetooth на плате устройства

Любой модуль связи должен иметь антенну. По внешнему виду антенны легко определить местоположение модуля на ПП устройства.

Можно выделить три основных типа антенн:

1) внешние антенны, подключаемые к печатной плате устройства с помощью специальных высокочастотных разъемов (обычно типов UFL

или SMA). В таком случае вы сразу увидите разъем на ПП устройства и сможете определить расположение модуля связи:

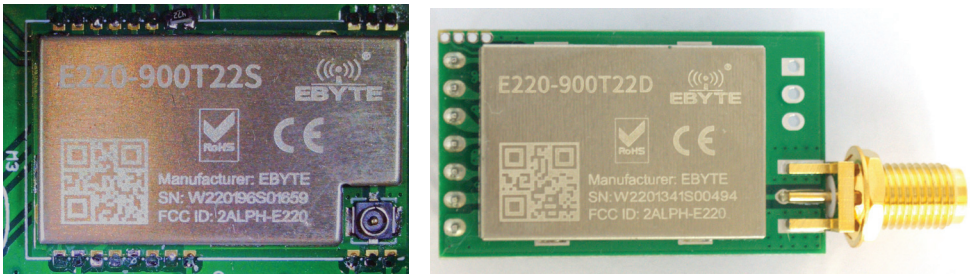


Рис. 0.41. Модули связи LoRa с разъемом UFL (слева) и SMA (справа)

2) SMD-антенны. Напаиваются на печатную плату устройства как компонент, имеют характерный внешний вид:

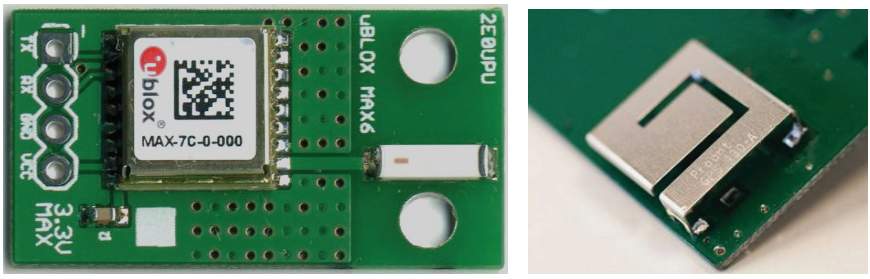


Рис. 0.42. Различные типы SMD-антенн[1]

3) PCB-антенна. Специальным образом сформированная дорожка на ПП устройства, выполняющая функцию антенны. Самый дешевый вариант, часто встречающийся в массовых устройствах (PCB-антенна может быть скрыта под слоем маски, как на первом фото в этом разделе).

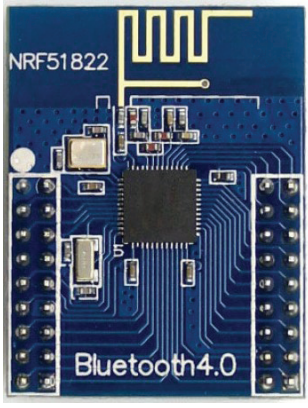


Рис. 0.43. PCB-антенна[2]

[1] https://ava.upuaut.net/?p=502 и https://www.saelig.com/product/ANTGPS004.htm.

[2] https://www.waveshare.com/product/iot-communication/short-range-wireless/bluetooth/core51822.htm.

Модуль связи не просто так выполнен в виде готовой печатной платы, запаиваемой на основную ПП устройства. Дело в том, что тонкая настройка радиотрактов за счет подбора характеристик его элементов – сложная задача, требующая специализированного оборудования. Следовательно, разработка радиотракта и его конфигурирование – сложная операция в жизненном цикле разработки устройства, оправданная только в специализированных и дорогих устройствах. Большинству производителей цифровых устройств проще купить уже готовый, проверенный и настроенный модуль связи с гарантированными характеристиками и установить его на ПП своего устройства.

В «простых» устройствах модуль связи подключается к основному микроконтроллеру (на котором исполняется прошивка) с помощью низкоскоростных интерфейсов: UART или SPI. Если устройство может обрабатывать данные с большой скоростью, то и интерфейс между центральным микроконтроллером и модулем связи, скорее всего, должен быть более быстрый: SDIO, USB или даже PCI-Express. Но высокоскоростные интерфейсы требуют соответствующей мощности от центрального микроконтроллера для обработки большого потока данных. Значит, вместо микроконтроллера уже может использоваться полноценный процессор, а в качестве ОС – Linux или что-то аналогичное. Все связано.

Ранее мы рассмотрели, что внутри модуля связи есть свой микроконтроллер, на котором исполняется прошивка, обрабатывающая пакеты беспроводной передачи данных. А можно ли на этом же микроконтроллере исполнять «пользовательский» код и обойтись без центрального микроконтроллера? Да, можно. Наиболее распространенный представитель такого универсального микроконтроллера – ESP32. Содержит в себе полноценный модуль связи Wi-Fi и Bluetooth, позволяет исполнять стороннюю прошивку, использующую функции связи. Очень удобно и существенно упрощает проектирование и производство устройства. Аналогичная ситуация с микроконтроллером NRF52, только поддерживаются другие стандарты беспроводной связи – Bluetooth LE и проприетарный 2,4 ГГц. Еще один пример – Wi-Fi-роутеры, в которых также стоит универсальный чип, отвечающий как за Wi-Fi, так и за прошивку с основной логикой устройства.

Модули связи не имеют какого-то универсального футпринта, поэтому ориентироваться при первичном инженерном анализе устройства надо на признаки, перечисленные в начале раздела (наличие антенн, металлических экранов и конструктив в виде отдельной платы).

Менее интересные компоненты

Помимо ключевых компонентов (микроконтроллера, памяти, модулей связи и т. п.), на печатной плате устройства находится большое количество «вспомогательных» компонентов, без которых устройство просто не будет работать. На данном этапе исследования их поиск и нахождение, скорее всего, не принесут дополнительной информации, но базовое представление, какие компоненты существуют и какие функции они выполняют, иметь нужно. Далее – краткая справка для ориентира и понимания, что искать в интернете, если нужно больше сведений.

«Питание»

Любое электронное устройство не может работать без питания. И чем сложнее устройство, тем большее количество номиналов напряжения ему нужно. Помните блок питания компьютера? Из сети он получает 220 В переменного напряжения, а выдает 3,3 В, 5 В и 12 В постоянного напряжения для питания. Чем сложнее микроконтроллер (микропроцессор), тем больше номиналов питания надо подавать. И даже последовательность подачи питания важна (и описана в документации). Наиболее распространенными вариантами являются 3,3 В, 1,8 В, 1,5 В и 1,2 В. При этом устройство имеет единственный вход питания, например 5 В от USB. Как же получить другие напряжения? Для этого служат специальные элементы, называемые преобразователи постоянного напряжения (DC-DC converter, LDO). Они бывают нескольких видов (наиболее распространены линейные и импульсные), но нам на данном этапе это не важно. Главное – понять принцип. Устройство может принимать на вход питание с определенными характеристиками, например 5–9 В и 400 мА. Преобразователи напряжений из разрешенного диапазона входного напряжения формируют фиксированные значения напряжений, необходимые для работы устройства. Внешний вид преобразователей напряжений весьма разнообразен, наиболее распространенные корпуса (серий small outline transistor (SOT) показаны на рисунке ниже.

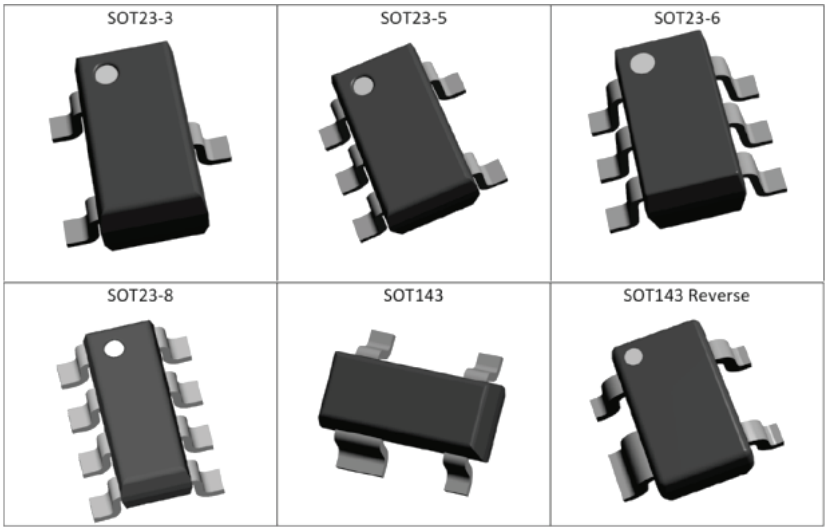


Рис. 0.44. Распространенные типы корпусов преобразователей напряжений[1]

Главное при анализе устройства – помнить, что питание надо отключать, т. к. случайно замкнуть линию питания (или любой ВЫХОДНОЙ сигнал микроконтроллера) на землю очень легко, что, скорее всего, приведет к выходу устройства из строя.

Если устройство подключается к 220 В без внешнего блока питания, т. е. блок питания скрыт в корпусе устройства или, что еще хуже, расположен на основ-

[1] https://a-contract.ru/publikacii/standartnye-semi-komponentov-dlja-pechatnykh-plat-chast-1/.

ной плате устройства, исследователю необходимо быть особенно осторожным. Лучше всего после анализа схемы питания найти место (например, на выходе схемы питания) для подключения более низкого постоянного напряжения. Таким образом получится отключить схему питания от 220 В без потери функциональности устройства, и процесс проведения исследований станет более безопасным.

Пассивные компоненты и не вошедшие в предыдущие разделы

Любое цифровое устройство на своей печатной плате имеет огромное количество пассивных компонентов, выполняющих различные функции. К таким компонентам относятся резисторы, конденсаторы, индуктивности, транзисторы и т. п., не вошедшие в предыдущее описание.

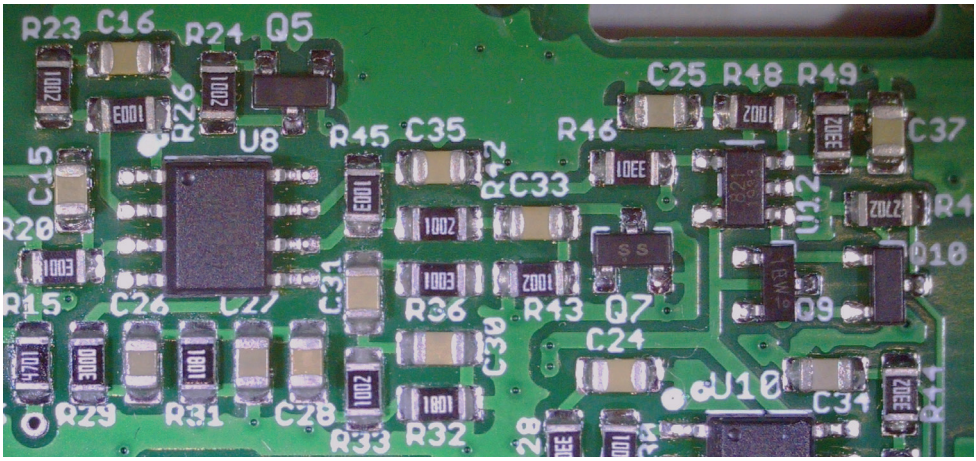


Рис. 0.45. Резисторы и конденсаторы на плате устройства

Резисторы могут выполнять множество функций, например ограничивать ток, формировать делитель напряжения, «подтягивать» линию к питанию в интерфейсе связи или блоке конфигурации микроконтроллера. Как правило, для первичного анализа устройства эти компоненты не так важны, как описанные ранее. Но иногда они могут подсказать или ускорить процесс анализа, если исследователь понимает возможные варианты их использования. В документации на микроконтроллер приводится минимально необходимая «обвязка» из элементов, необходимая для запуска микроконтроллера. Это необходимые фильтры питания, подтяжки нужных линий (например, сигнала сброса ядра микроконтроллера, reset) и блок генерации сигнала синхронизации для тактирования микроконтроллера.

На рисунке 62 приведена минимальная схема для микроконтроллера STM32F303. Для запуска и работы микроконтроллера достаточно подключить питание, конденсаторы для фильтрации питания, кварцевый резонатор для генерации тактирующего сигнала и несколько конфигурационных пинов (BOOT, NRST).

Зачастую по подобным схемам можно определить список компонентов, которые можно отбросить при анализе. Оставшиеся же могут дать полезную

информацию. Например, один-два резистора могут задавать режим конфигурации загрузки микроконтроллера, замыкая определенные ножки на питание (сигнал логической «1») и землю (сигнал логического «0»). Другие резисторы могут конфигурировать отключение отладочных интерфейсов JTAG и т. д.

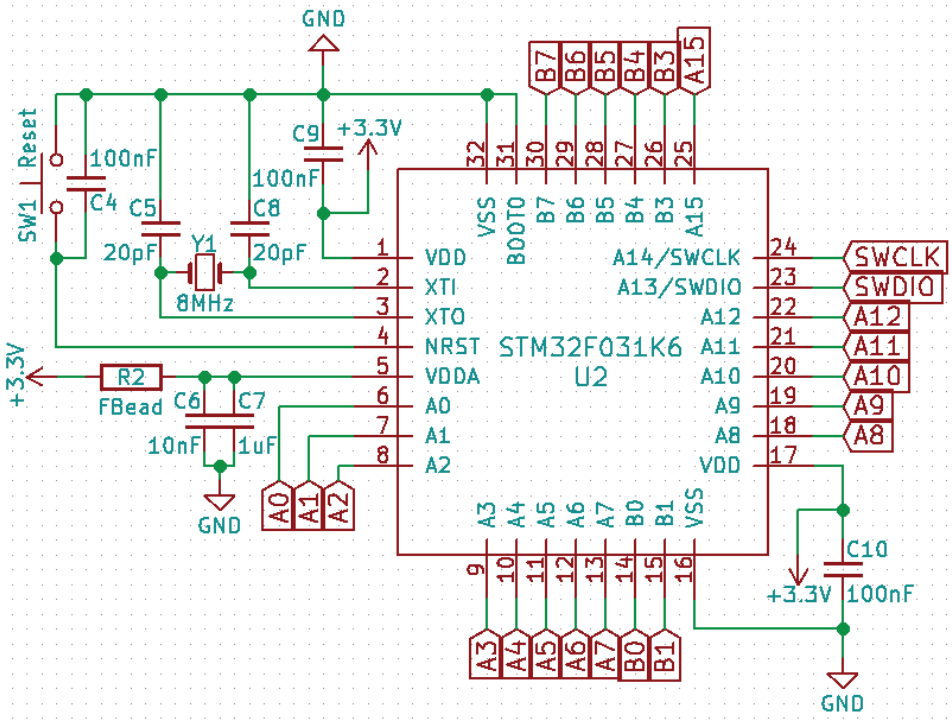


Рис. 0.46. Пример минимальной схемы, необходимой для работы микроконтроллера STM32F303

Во многих микроконтроллерах есть контакты конфигурирования (strapping pins). Выше мы увидели, что с помощью подобных контактов можно, например, сконфигурировать источник загрузки микроконтроллера. Они могут иметь внутренние резисторы, подтягивающие контакты к VCC или GND, поэтому даже если они оставлены неподключенными на плате устройства, это не значит, что они не задают какую-то конкретную конфигурацию. Найти описание strapping pins можно в документации на микроконтроллер.

SoM, SoC, SiP и другие типы компоновки

В мире встраиваемых систем широко распространены концепции System on a Module (SoM, система на модуле) и System on a Chip (SoC, система на кристалле). SoM представляет собой отдельную плату, способную выполнять какую-то функцию. Типичным представителем SoM является плата универсального вычислителя, на которой находится микроконтроллер (микропроцессор или FPGA), микросхемы оперативной и постоянной памяти, все необходимые для работы микроконтроллера компоненты.

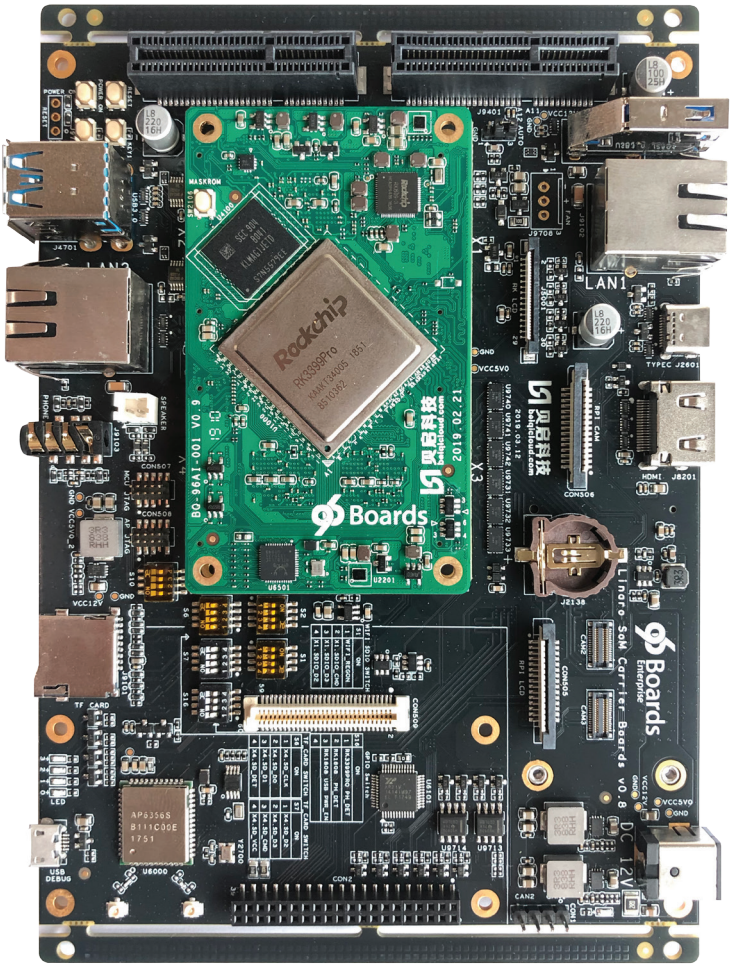


Рис. 0.47. Модуль SoM, установленный в периферийную плату с набором различных интерфейсов[1]

На плате SoM находятся разъемы со всеми доступными выводами микроконтроллера, к которым может быть подключено множество периферии. Плата SoM устанавливается в соответствующий разъем на периферийной плате, на которой разведены преобразователи интерфейсов, кодеки и другие компоненты и разъемы, необходимые для работы конкретного устройства. Использование SoM позволяет разработчику устройства сконцентрироваться на производстве периферийной платы для конкретного устройства, не погружаясь в сложный процесс разработки платы для современного процессора. Такой подход существенно упрощает разработку устройства и экономит время, однако при больших партиях серийно выпускаемых устройств обходится дороже.

Концепция SoC похожа на SoM с одним существенным отличием: все компоненты, необходимые для выполнения какой-либо функции, располагаются на

[1] https://www.96boards.org/product/96boards-som-carrier-board/.

одном кристалле (или в одном корпусе микросхемы, тогда они называются SiP (System in a Package) или MCP (Multi Chip Package), MCM (Multi-chip Module).

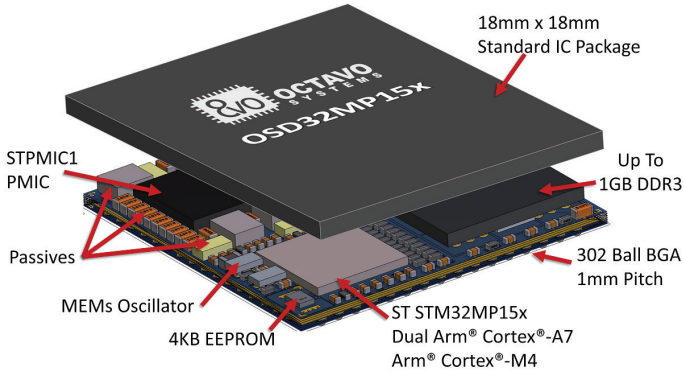


Рис. 0.48. Схема компоновки System in a Package[1]

Примером «стека» из нескольких микросхем, наложенных друг на друга, является компоновка PoP (Package on Package). При такой компоновке в большинстве случаев сверху располагаются кристаллы памяти, а снизу – микропроцессор.

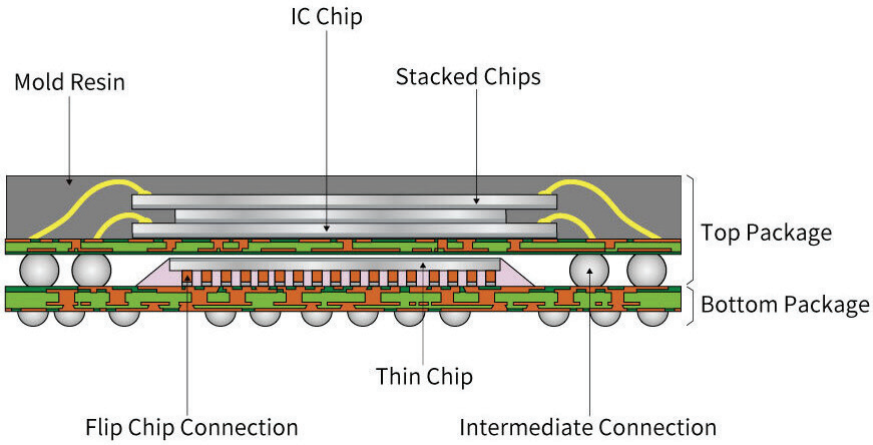


Рис. 0.49. Схема компоновки Package in a Package[2]

Типичными представителями SoC являются модули связи или высокоинтегрированные вычислительные системы современных мультимедийных устройств (смартфонов, планшетов и т. д.). Например, SoC Apple M1 и M2, использующиеся в современных компьютерах и планшетах Apple. Эти SoC объединяют в себе основной микропроцессор, память, видеоускоритель и большой набор интерфейсных контроллеров. На рисунке показаны внутреннее устройство SoC Apple M1 Pro и материнская плата Apple MacBook Pro на его основе.

[1] https://octavosystems.com/octavo_products/osd32mp15x/.
[2] https://www.shinko.co.jp/english/product/package/assembly/mcep.php.

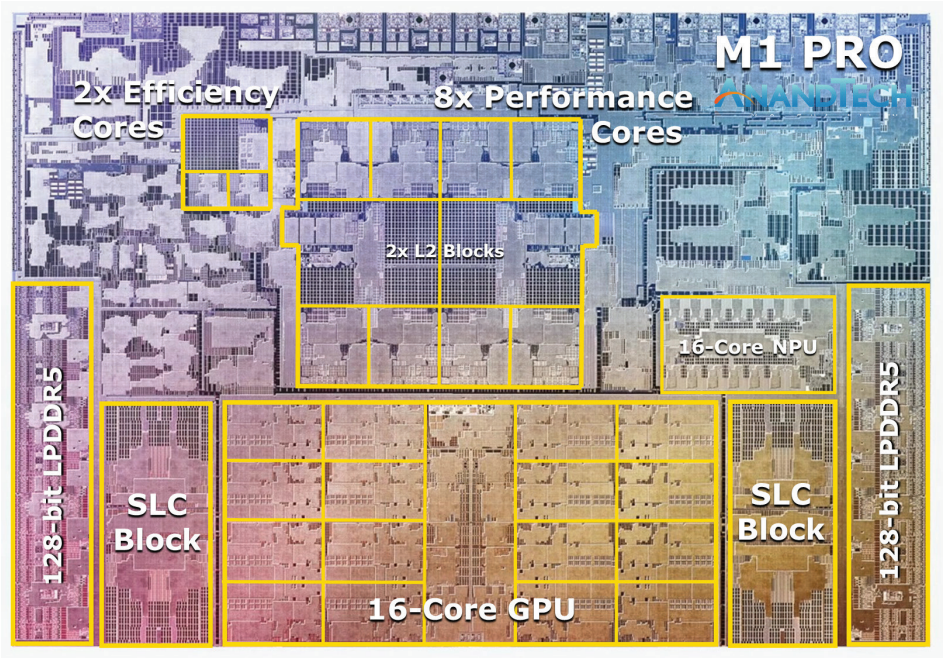


Рис. 0.50. Структура кристалла SoC Apple M1 Pro[1]

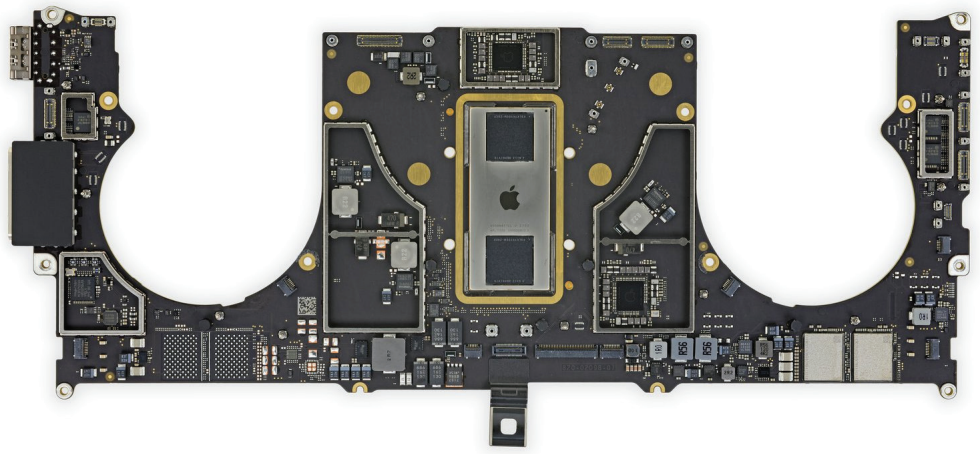


Рис. 0.51. Материнская плата Apple MacBook с SoC Apple M1 Pro[2]

Интерфейсы связи компонентов

В предыдущих разделах мы рассмотрели ключевые компоненты, которые стоит искать в первую очередь при первичном анализе устройства. Все компоненты на ПП устройства должны быть соединены в единую схему. Нас в первую

[1] https://www.anandtech.com/show/17019/apple-announced-m1-pro-m1-max-giant-new-socs-with-allout-performance.
[2] https://ifixit.com/News/54122/macbook-pro-2021-teardown.

очередь интересуют ключевые компоненты, т. е. выполняющие существенные с точки зрения функционирования устройства задачи. С высокой долей вероятности они будут соединяться с помощью набора электрических сигналов, по которым идет информационный обмен с помощью одного из распространенных стандартов. Пора рассмотреть наиболее часто встречающиеся в цифровых устройствах интерфейсы. На данном этапе исследования мы не делаем экспериментов с устройством и не пытаемся перехватить («заснифать», от sniff – нюхать) данные, передающиеся по интерфейсам (трафик, англ. traffic – движение, поток), с помощью логического анализатора или эмулятора. Это нас ждет в главе 1.

Преобразователи уровней сигнала

Иногда в устройстве могут присутствовать два компонента, имеющих разное напряжение питания (например, 3.3 В и 1.8 В) и, соответственно, разные уровни сигналов логических 0 и 1 на своих выводах. Просто так соединить эти устройства не получится, в лучшем случае информационный обмен будет некорректным, а в худшем устройство с более низким уровнем логических сигналов просто выйдет из строя. Решением является использование специальных микросхем (преобразователей, также называемых трансляторами, voltage level shifter), преобразующих входящий сигнал к требуемому уровню. Эти устройства никак не изменяют содержимое сигнала, они лишь меняют уровень логического 0 и 1.

I2C

I2C – универсальный последовательный интерфейс типа «шина», с помощью которого к микроконтроллеру можно подключить множество различных компонентов. В нем используются всего две линии (не считая питания): SDA (для передачи данных) и SCL (для тактирования).

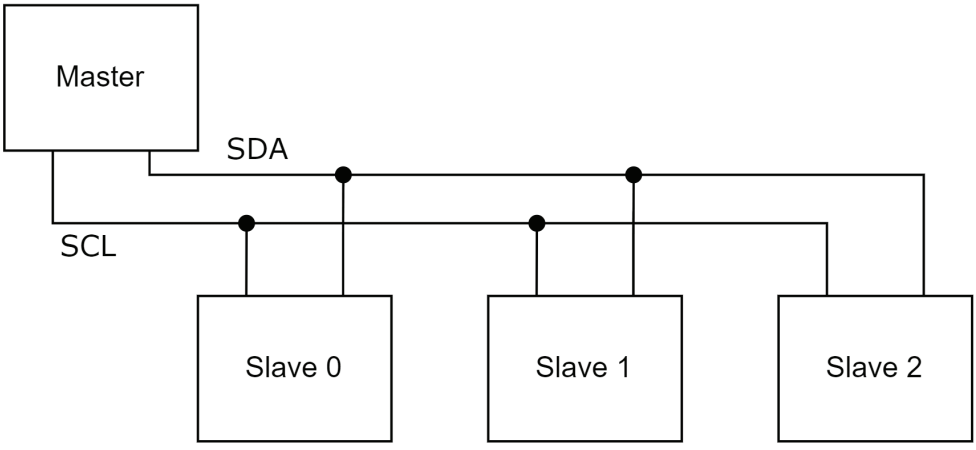


Рис. 0.52. Линии интерфейса I2C

Именно его простота – залог широкого использования. Частота работы интерфейса невелика, не более 1 МГц (наиболее распространенными вариантами являются 100 КГц, 400 КГц и 800 КГц). Поэтому нет проблем с «выравниванием» длин проводников на печатной плате и сложностей в ее трассировке. Впрочем, как и выдающихся скоростных характеристик. Но многим устройствам они и не нужны, а чем проще интерфейс, тем лучше. Существуют программные реализации интерфейса, но они используются редко, т. к. аппаратная поддержка I2C есть почти в каждом современном микроконтроллере. Устройства на шине подключаются параллельно (у каждого устройства есть семибитный адрес (однако есть варианты с восьми- или десятибитными адресами), как правило, настраиваемый подключением конфигурационных пинов к питанию и земле), а сама шина синхронная с главным устройством (master, именно оно инициирует обмен данными по шине) и ведомыми (slave). Пример выбора устройства с адресом 0x51 и передачи данных по шине I2C показан на рисунке ниже.

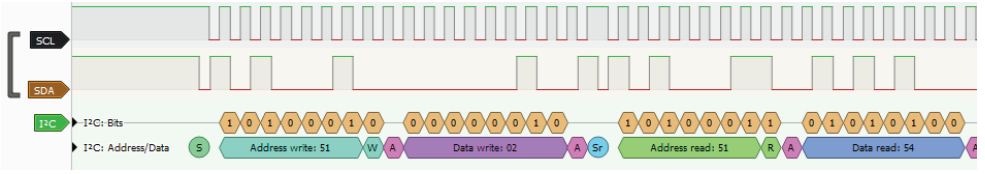


Рис. 0.53. Диаграмма передачи данных по интерфейсу I2C

Чаще всего по шине I2C подключаются:

- микросхемы ПЗУ (уже читали про это в соответствующем разделе);
- различные датчики (акселерометры, гироскопы, температурные датчики);
- расширители портов ввода-вывода (например, для реализации матричной клавиатуры);
- устройства с низкой скоростью обмена информацией (GPS-модули или низкоскоростные беспроводные модули, TPM-модули).

SPI

Serial Peripheral Interface. Еще один универсальный синхронный последовательный интерфейс. Используется не реже I2C, а его скоростные характеристики могут быть значительно лучше. В нем используется уже 4 линии (две для однонаправленной передачи данных, одна для синхронизации, одна для выбора микросхемы, не считая питания):

1) MOSI (*Master Output Slave Input*). Нужен для передачи данных от ведущего устройства ведомому;
2) MISO (*Master Input Slave Output*). Служит для передачи данных от ведомого устройства ведущему;
3) SCLK / SCK (*Serial Clock*). Служит для передачи тактового сигнала для ведомых устройств;
4) CS или SS – выбор микросхемы, выбор ведомого (*Chip Select, Slave Select*).

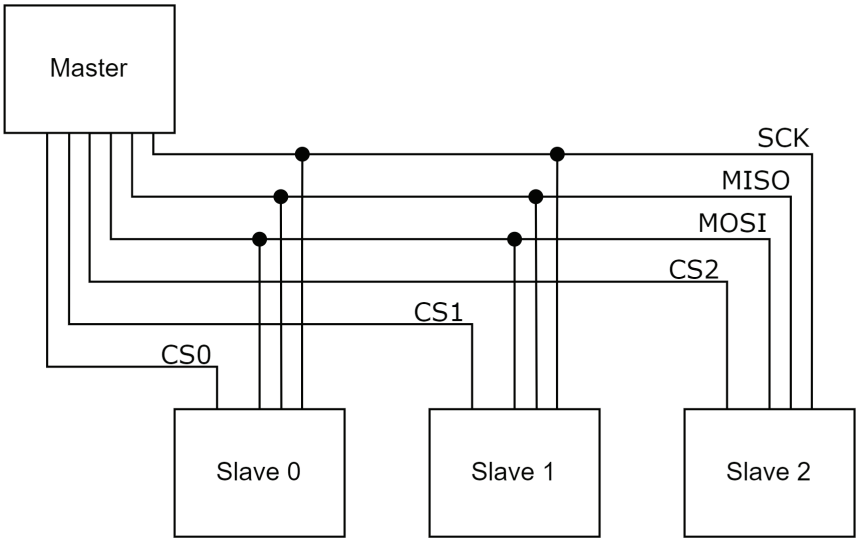


Рис. 0.54. Линии интерфейса SPI

Все информационные сигналы разных устройств (кроме CS) подключаются на шине параллельно. Выбор активного в данный момент устройства осуществляется сигналом CS.

Как таковых «стандартных» частот SPI-интерфейса нет, значения для конкретных реализаций могут составлять от сотен килогерц до десятков (и даже сотен) мегагерц. Существуют программные реализации интерфейса, но ввиду высоких скоростных характеристик они используются редко, тем более что аппаратный SPI-контроллер есть почти в каждом современном микроконтроллере. Пример передачи данных по шине SPI показан на рисунке ниже.

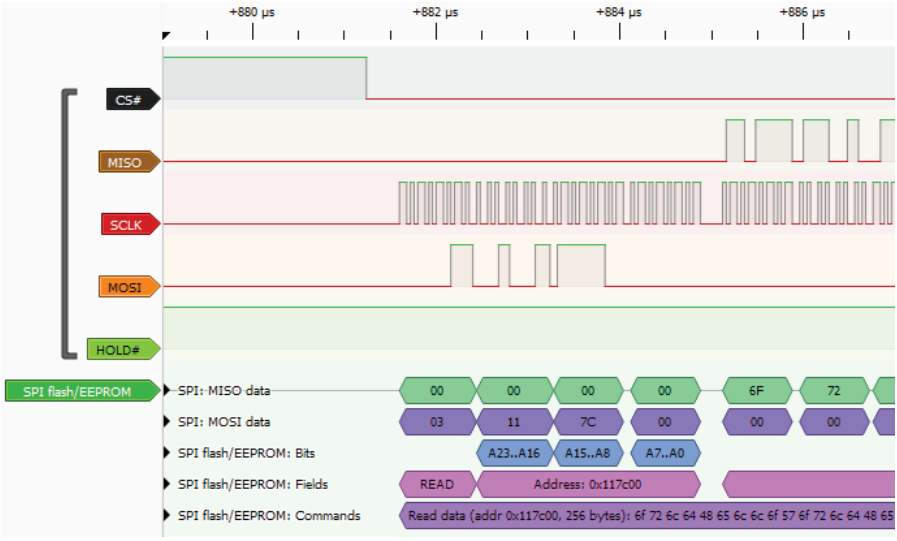


Рис. 0.55. Диаграмма передачи данных по интерфейсу SPI

SPI имеет четыре режима синхронизации. Режим определяется комбинацией битов CPHA и CPOL:

- CPOL = 0 – исходное состояние сигнала синхронизации – низкий уровень;
- CPOL = 1 – исходное состояние сигнала синхронизации – высокий уровень;
- CPHA = 0 – выборка данных производится по переднему фронту (переключению) сигнала синхронизации. То есть по переключению из основного в противоположное ему;
- CPHA = 1 – выборка данных производится по заднему фронту (переключению) сигнала синхронизации. То есть по переключению обратно к основному из противоположного.

Для обозначения режимов работы интерфейса SPI принято следующее соглашение:

- режим 0 (CPOL = 0, CPHA = 0);
- режим 1 (CPOL = 0, CPHA = 1);
- режим 2 (CPOL = 1, CPHA = 0);
- режим 3 (CPOL = 1, CPHA = 1).

Внешний вид разницы в сигналах при использовании режимов показан на диаграмме далее.

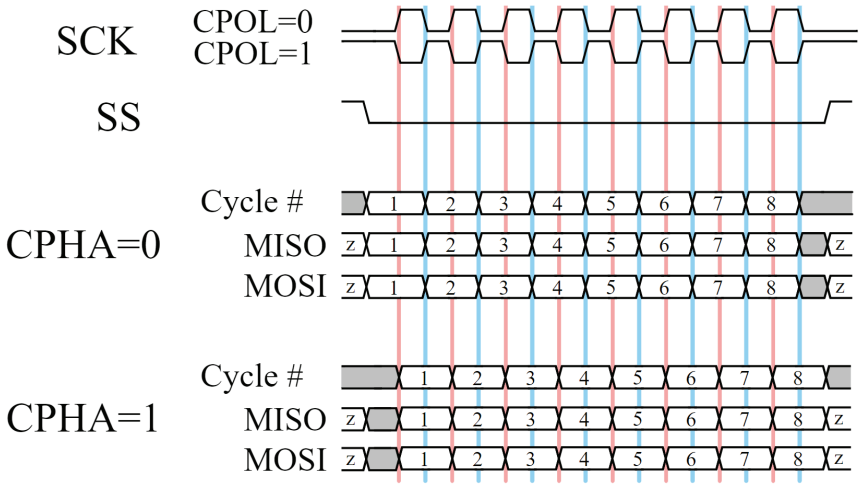


Рис. 0.55. Режимы синхронизации интерфейса SPI[1]

Для увеличения скорости работы уже существующего интерфейса (для быстрого считывания микросхем памяти) был придуман механизм увеличения количества передающих линий. Так появились режимы работы Dual SPI (две двунаправленные линии передачи данных) и Quad SPI (четыре двунаправлен-

[1] https://en.wikipedia.org/wiki/Serial_Peripheral_Interface.

ные линии), в которых вместо однонаправленных линий данных линии двунаправленные, т. е. могут передавать информацию в обе стороны. Именно такой реализацией SPI подключена микросхема ПЗУ BIOS к чипсету в вашем компьютере.

Список компонентов, подключаемых по шине SPI, сильно пересекается с перечисленными в разделе про I2C (часто они даже могут работать по обеим шинам, и выбор остается за производителем устройства):

1) память (ПЗУ) (отдельно стоит отметить возможность работы SD-карт и eMMC микросхем в режиме низкоскоростной передачи данных по SPI). В микроконтроллерах начального уровня нет отдельного аппаратного блока для работы по полноценному интерфейсу SD, а необходимость подключать SD-карты к устройству на их основе может быть;
2) различные датчики (акселерометры, гироскопы, температурные датчики);
3) вспомогательные микроконтроллеры (например, служащие для опроса датчиков и формирования единого пакета «состояния» для центрального микроконтроллера);
4) модули связи, в том числе среднескоростные (со скоростью в десятки мегабит).

Наверное, SPI наравне с UART – это самые распространенные интерфейсы связи компонентов в цифровых устройствах.

UART и USART

Universal (Synchronous) Asynchronous Receiver-Transmitter. Широко распространенный интерфейс во встраиваемых устройствах. Существует набор стандартных скоростей, измеряемых в бодах (грубо можно считать битах в секунду) передачи данных (9600 бод, 38 400 бод, 57 600 бод и 115 200 бод) и соответствующих им частот. Но фактически скорость может быть любая, как правило, не превышающая 1–2 Мбод. На электрическом уровне в самом распространенном варианте представлен в виде двух сигналов: TX (transmit) и RX (receive). Опять же, без линий питания. Обратите внимание, что сигнал TX передатчика соединяется с сигналом RX приемника, и наоборот. Это самая распространенная ошибка при анализе или работе с UART.

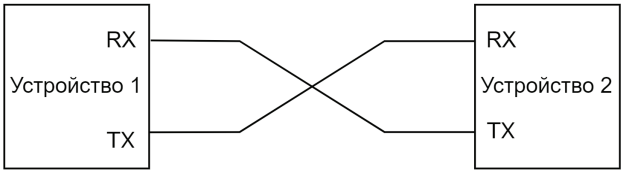


Рис. 0.56. Линии интерфейса UART

Устройства подключаются по принципу точка-точка, в отличие от рассмотренных ранее шинных интерфейсов I2C и SPI. Также обратите внимание на отсутствие сигнала синхронизации, т. е. оба устройства должны заранее знать, на какой скорости (и с какими параметрами) будет происходить информационный обмен. В очень редких случаях в устройствах могут присутствовать

алгоритмы автоопределения параметров передачи. Программная реализация возможна, но в микроконтроллерах почти всегда есть аппаратная реализация как минимум одного экземпляра UART. А в большинстве случаев – двух, трех и более. Пример передачи данных по интерфейсу UART показан на рисунке.

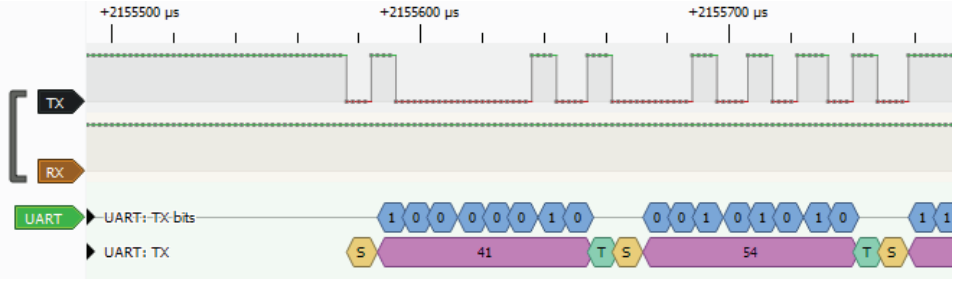


Рис. 0.57. Диаграмма передачи данных по интерфейсу UART

Стоит упомянуть также USART – Universal Synchronous / Asynchronous Receiver / Transmitter. Этот интерфейс похож на UART, но обладает гораздо большими возможностями. Он может работать в синхронном режиме, в котором передатчик генерирует сигнал синхронизации, извлекаемой приемником из потока данных без информации о скорости передачи. Возможен и другой вариант, когда для сигнала синхронизации выделяется отдельная линия связи. Использование внешнего сигнала синхронизации позволяет USART работать на скоростях до 10 Мбит/с.

Также не стоит путать интерфейсы UART/USART, работающие на уровне используемой в пределах одного устройства логики, и их возможные физические транспорты RS-232, RS-422, RS-485 и т. д. Например, интерфейс RS-232 предназначен для подключения периферийных устройств к ПК и имеет допустимые напряжения от –15 В до +15 В относительно земли. RS-485 используется для подключения удаленных устройств (до 1200 м) с помощью дифференциальной пары проводников. Поэтому, например, нельзя подключать адаптер USB-UART к порту RS232 устройства, несмотря на то что на уровне микроконтроллера он будет представлен в виде интерфейса UART.

Слово Universal в названии интерфейса U(S)ART абсолютно оправдано, во встраиваемых системах он применяется очень часто:

1) для передачи отладочной, и не только, информации на подключаемый компьютер (часто на этом интерфейсе «торчит» консоль устройств с Linux «на борту») или другие устройства;
2) для программирования прошивки микроконтроллера;
3) для связи нескольких микроконтроллеров в одном устройстве;
4) для общения с модулями связи (GSM, GPS, Bluetooth, CAN и т. д.).

SD(IO)/eMMC

Secure Digital. Не самый распространенный интерфейс, однако важный для понимания в контексте исследования устройств (т. к. на SD и eMMC (embedded Multi Media Card) часто хранится прошивка устройства). Одноименный стан-

дарт описывает хорошо знакомые всем нам SD- и MicroSD-карты памяти. Они работают по интерфейсу SD (существует и альтернативный низкоскоростной режим работы по SPI для подключения к микроконтроллерам начального уровня без поддержки интерфейса SD). Поддержка интерфейса SD есть в микроконтроллерах «продвинутого» уровня, т. к. сам протокол сложнее рассмотренных ранее SPI, I2C и UART. SDIO – это расширение для работы различных модулей по протоколу SD (изначально разработанному для подключения только карт памяти). eMMC – это (очень «грубо», но на данном этапе точные детали сильно усложнят понимание) SD-карта в форм-факторе BGA-чипа с расширенным набором линий данных (восемь вместо четырех). Интерфейс позволяет подключать устройства параллельно, однако я не встречал таких реализаций более чем за 15 лет работы. Всегда используется вариант точка-точка. Рассмотрим, как SD/EMMC выглядит на уровне линий связи.

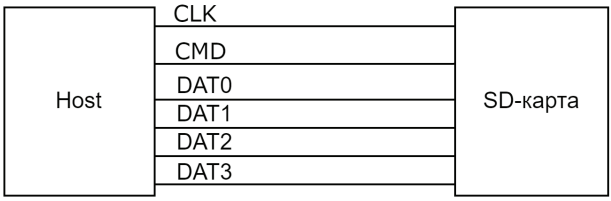


Рис. 0.58. Линии интерфейса SD

Обязательными для работы интерфейса являются три сигнала:

1) CLK – Clock, сигнал синхронизации (задается микроконтроллером);
2) CMD – Command. Служит для передачи команд протокола SD;
3) DAT0 – двунаправленный сигнал передачи данных. В начале своей работы SD-карта работает именно в однобитном режиме. А после прохождения нескольких команд инициализации может переключиться в четырехбитный режим (тогда используются уже 4 линии DAT0–DAT3). Аналогично работают EMMC-микросхемы, только у них линий данных восемь, соответственно, помимо одно- и четырехбитного, есть еще и восьмибитный режим.

При включении и работе в однобитном режиме частоты интерфейса обычно 100–400 КГц. Пример сигналов в данном режиме показан на рисунке далее.

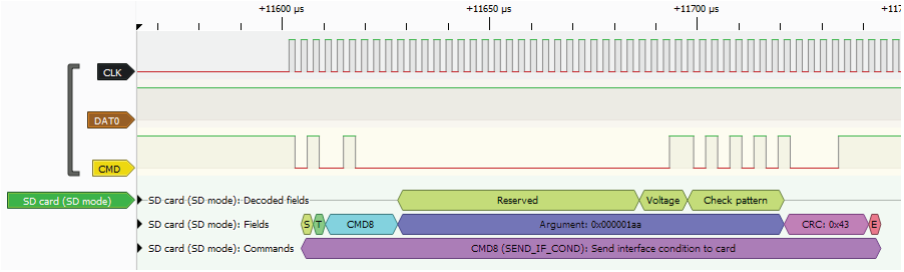


Рис. 0.59. Диаграмма передачи данных по интерфейсу SD в режиме однобитной шины

После прохождения процедуры инициализации SD-карты частоты могут составлять десятки и даже сотни мегагерц. Поэтому на ПП устройства дорожки интерфейса SD, как правило, выровнены по длине, т. к. на больших частотах при параллельной передаче данных все сигналы должны приходить в микроконтроллер одновременно для выполнения условий синхронизации.

USB

Да, USB широко используется во встраиваемых системах. Чаще всего в своих низко- и среднескоростных версиях USB 1.x и USB 2.0. Он может использоваться для подключения USB-флешек или модулей связи (например, Wi-Fi) к устройству для информационного обмена или обновления прошивки (устройство работает в роли USB host). Или для подключения самого устройства к компьютеру (устройство в режиме USB device). Не всегда на устройстве есть стандартные разъемы USB, встречаются и нестандартные, чтобы усложнить задачу подключения устройства, или если есть повышенные требования по вибростойкости (стандартные разъемы USB не обеспечивают надежного подключения в условиях, например, движущегося автомобиля). Максимальная скорость шины USB – 12 Мбит/с (Full Speed) для USB 1.1 и 480 Мбит/с (High Speed) для USB 2.0. Самый современный стандарт поколения USB 3.x (USB 3.2) со скоростью от 5 Гбит/с (SuperSpeed) до 20 Гбит/с (SuperSpeed+) встречается редко, обычно в одноплатных компьютерах, где требуется очень высокоскоростной интерфейс передачи данных. Аналогичным образом выглядит ситуация с USB4 – встретить его во встраиваемых системах, скорее всего, не придется.

USB Low Speed (1.5 Мбит/с) может быть реализован программно, т. к. скорость 1.5 Мбит/с, и соответствующая ей частота не является проблемой даже для микроконтроллеров базового уровня. USB 1.x и USB 2.0 требуют двух линий связи (опять же, помимо питания): D+ (также обозначаемой DP) и D– (DM).

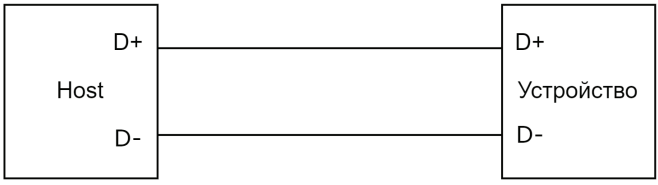


Рис. 0.60. Линии интерфейса USB 2.0

Это первый интерфейс в нашем обзоре, где сигнал передается по дифференциальной паре. Внешний вид сигнала показан на рисунке.

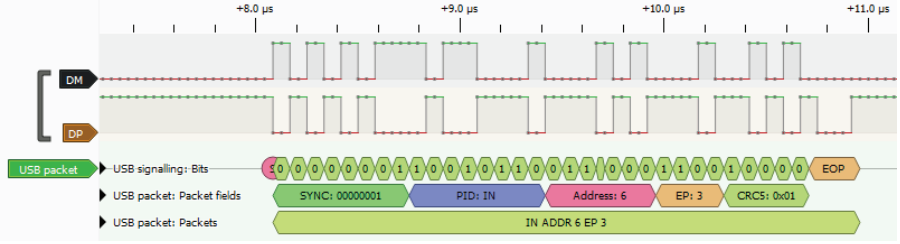


Рис. 0.61. Диаграмма передачи данных по интерфейсу USB 2.0

Интерфейс USB поддерживает несколько способов передачи пакетов (конкретная реализация которых называется каналами), отличающихся потенциальным назначением применения (нужна ли скорость и объем передачи данных, гарантированное время доставки или пропускная способность интерфейса):

- *control* – канал управления и обмена короткими сообщениями вида вопрос–ответ. Любое устройство должно иметь один control-канал (с адресом 0) для инициализации устройства на шине USB хоста;
- *bulk* – канал передачи блоков информации, гарантирующий доставку данных, в том числе приостановку передачи данных при неготовности устройства (переполнение или опустошение буфера). Не гарантирует скорость и задержки доставки данных. Используется в устройствах хранения информации, принтерах и т. п.;
- *isochronous* – канал непрерывной передачи. Позволяет доставлять пакеты без гарантии доставки и без ответов/подтверждений, но с гарантированной скоростью доставки в *N* пакетов на один период шины. Используется для передачи аудио- и видеоинформации;
- *interrupt* – позволяет доставлять короткие пакеты с гарантией времени доставки, но без гарантии самой доставки, используется в устройствах ввода (клавиатуры, мыши и т. п.).

Время шины USB делится на периоды, в начале периода контроллер передает всей шине пакет «начало периода». Далее в течение периода передаются пакеты interrupt, потом isochronous в необходимом количестве, в оставшееся время в периоде передаются control-пакеты и в последнюю очередь – bulk. Более подробно про USB можно узнать из подробнейшей документации (**https://www.usb.org/**) или отличной статьи «USB in a NutShell» (**https://www.beyondlogic.org/usbnutshell/usb1.shtml**), кратко рассказывающей о самых важных особенностях интерфейса.

CAN

Controller Area Network – синхронный последовательный интерфейс, широко использующийся в промышленных устройствах, автомобилях и устройствах умного дома. Не используется для связи компонентов в пределах платы одного устройства, но широко применяется для построения сетей связи (CAN-шин) в указанных областях применения. Например, современный автомобиль имеет десятки электронных блоков управления (ЭБУ), соединенных преимущественно с помощью CAN-шин. Начиная примерно с 2015 года в автоиндустрии идет активная замена CAN-шин их более высокоскоростными приемниками (FlexRay, BroadR-Reach (automotive ethernet)). Но даже в самых современных автомобилях более половины ЭБУ все еще соединяются CAN-шинами.

В большинстве реализаций CAN-шина представлена в виде дифференциальной пары проводников CANH и CANL (также обозначаемых буквами H и L). В микроконтроллерах реализована разная степень поддержки CAN (где-то нет вообще, в некоторых есть аппаратная поддержка всех уровней, кроме фи-

зического). Физическая поддержка интерфейса CAN реализуется с помощью специальных микросхем-трансиверов (драйверов) CAN-шины. Все устройства на CAN-шине соединяются параллельно, скорость передачи данных составляет от нескольких десятков Кбит/с до 1 Мбит/с.

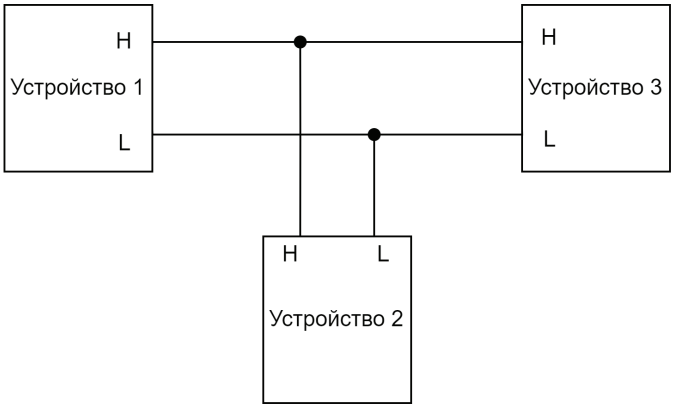


Рис. 0.62. Линии интерфейса CAN

Все общение по шине CAN построено по широковещательному принципу. Каждое подключенное устройство проверяет, что шина свободна (т. е. имеет состояние BUS IDLE) и начинает широковещательную передачу. В начале каждой посылки данных передается 11-битный (или 29-битный в стандарте CAN 2.0B) идентификатор message ID. Для разрешения коллизий на шине существует механизм неразрушающего арбитража, призванный гарантировать обнаружение ошибок и отключение некорректно ведущего себя на шине устройства. Когда происходит конфликтная ситуация при обращении к шине, CAN определяет победителя на основе побитного арбитража содержимого поля ID. Побеждает устройство с наивысшим приоритетом, то есть имеющее идентификатор с наименьшим числовым значением. Только победившее устройство продолжает передачу данных, остальные попытаются передать свои сообщения позже. На рисунке показан декодированный логический сигнал CAN (т. е. преобразованный из двух сигналов дифференциальной пары CANL и CANH):

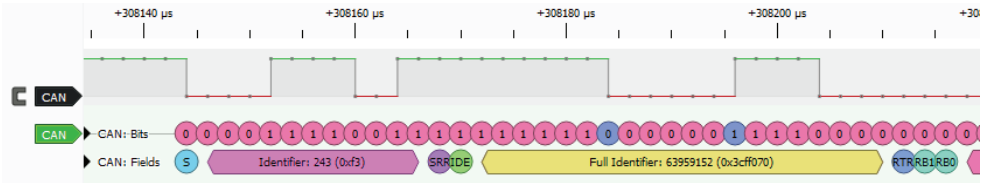


Рис. 0.63. Диаграмма передачи данных по интерфейсу CAN

PCI

Peripheral Component Interconnect – стандарт, появившийся в 1992 году и описывающий одноименный интерфейс связи устройств внутри ПК. Стандарт был объявлен открытым, поэтому любой желающий мог разрабатывать PCI-устройства без выплаты лицензионных отчислений.

PCI имеет 32 или 64 линии адреса/данных и несколько управляющих сигналов, представляет собой параллельную децентрализованную шину с возможностью пакетной коммутации данных. Сигналы PCI показаны на рисунке.

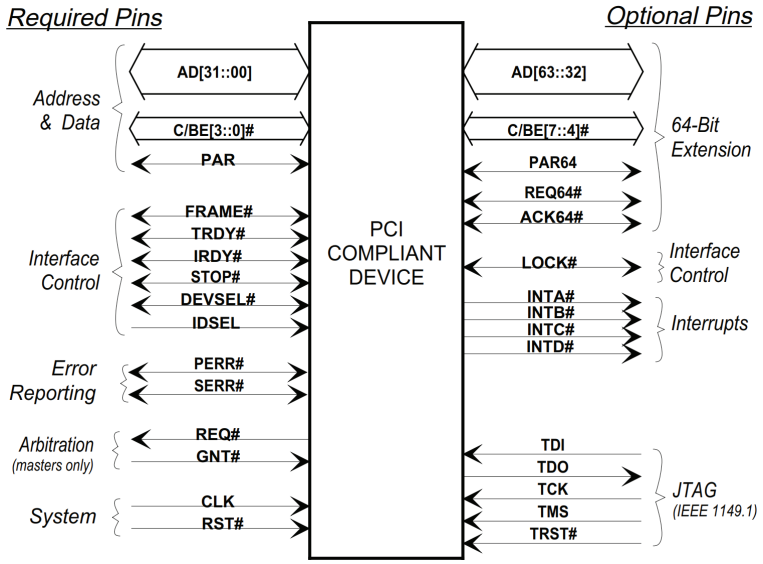


Рис. 0.64. Сигналы PCI[1]

Первая версия интерфейса работала на частоте 33 МГц и в теории обеспечивала скорость до 133 Мб/с. В версии 2.1+ поддерживалась работа с частотой 66 МГц и максимальная скорость передачи в 533 Мбайт/с (для 64-битного варианта). Структурная схема подключения PCI-устройств в виде топологии «шина» показана на рисунке.

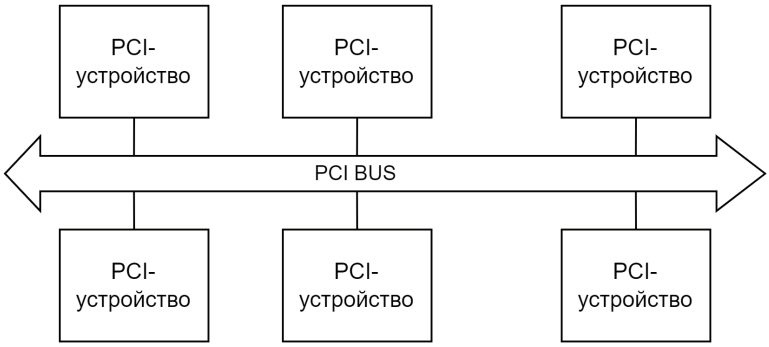


Рис. 0.65. Шина PCI

Интерфейс PCI чаще всего встречается во встраиваемых системах, построенных на базе процессоров с архитектурами x86, MIPS и PowerPC (подробнее про архитектуры мы поговорим в главе 2). С помощью PCI к процессору могут подключаться различные периферийные контроллеры.

[1] https://pcisig.com/specifications.

Существует несколько промышленных стандартов встраиваемых компьютеров, актуальных на момент написания книги, использующих подключение PCI, но механически не совместимых со стандартными разъемами.

1. CompactPCI – стандарт для разъемов и карт расширения, применяемый в промышленных, телекоммуникационных и встраиваемых компьютерах. Электрически шина отличается от PCI стандарта 2.2 тем, что позволяет подключить большее число устройств. Но в целом совместима и обычно использует тот же набор микросхем физического уровня. Форм-фактор и разъем позволяют использовать «горячее подключение» плат – то есть устанавливать и извлекать плату из шасси, не прерывая работоспособности компьютера. На фото показана типовая плата форм-фактора CompactPCI (обратите внимание на интерфейсный разъем на дальней части фотографии):

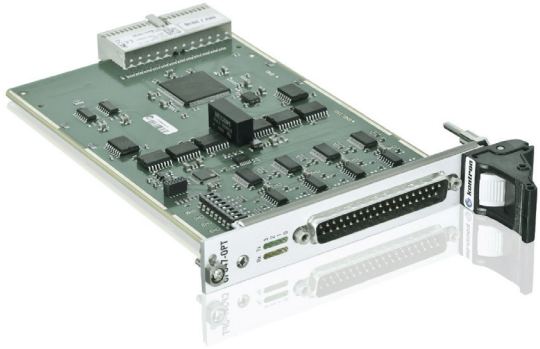


Рис. 0.66. Плата форм-фактора CompactPCI[1]

На следующем фото показано шасси CompactPCI с установленными процессорным модулем, модулями расширения и блоком питания:

Рис. 0.67. Шасси для установки плат форм-фатора CompactPCI[2]

[1] https://kontron.com.ru/catalog/apparatnye-komponenty/compactpci/3u-moduli-vvoda-vyvoda/cp347/.

[2] https://datarespons.solutions/product-solution/embedded-boards/compact-pci/enclosure/mh50c-mtcs-train-control-system-controller-cpci-serial-plus-io/.

2. Стандарты PC/104+ (PC/104Plus), PCI/104 и PCI/104-Express.

   Отличительной особенностью механического конструктива PC/104 является расположение контактов не на ребре платы, а перпендикулярно ей, что позволяет устанавливать платы друг на друга. Такая конструкция позволяет собрать до 3–6 плат в один большой «сэндвич» и разместить его в компактном корпусе.

Рис. 0.68. Плата форм-фактора PC/104+[1]

Несмотря на то что интерфейс PCI сложно встретить в потребительских устройствах, он до сих пор широко распространен в промышленных контроллерах и средствах автоматизации.

PCI Express

PCI Express (англ. Peripheral Component Interconnect Express), или PCIe, или PCI-e, – хорошо известный вам интерфейс связи компонентов внутри ПК, пришедший на смену PCI. Аналогично с заменой интерфейсом PCIe устаревшего интерфейса PCI, в промышленных стандратах CompactPCI и PCI/104 вместо (и вместе с) PCI начал использоваться PCIe. Он широко применяется во встраиваемых системах, обрабатывающих большие объемы данных (сетевые устройства, промышленные контроллеры, мобильные телефоны, устройства хранения информации и т. д.). С помощью него подключаются модули связи, накопители данных, камеры и другие интерфейсные устройства, которым необходимы высокие скорости передачи данных.

PCIe имеет ряд существенных отличий от PCI:

- вместо параллельного подключения используется последовательная пакетная передача данных по дифференциальным парам проводников. PCIe пересылает всю управляющую информацию через те же линии, что используются для передачи данных. Потоки приема и передачи имеют независимые каналы и одинаковые максимальные скорости;

[1] https://www.prosoft.ru/products/vstraivaemye-i-magistralno-modulnye-sistemy/single-board-computers/pc-104/moduli-obrabotki-graficheskoy-informatsii-pc-104/modul-graficheskogo-soprotsessora-vypolnennyy-v-standarte-pc-104-plus/.

- топология точка-точка вместо шины. Каждое устройство с интерфейсом PCIe подключается к коммутатору (свитчу) PCI-e. Структурная схема подключения PCI устройств показана на схеме:

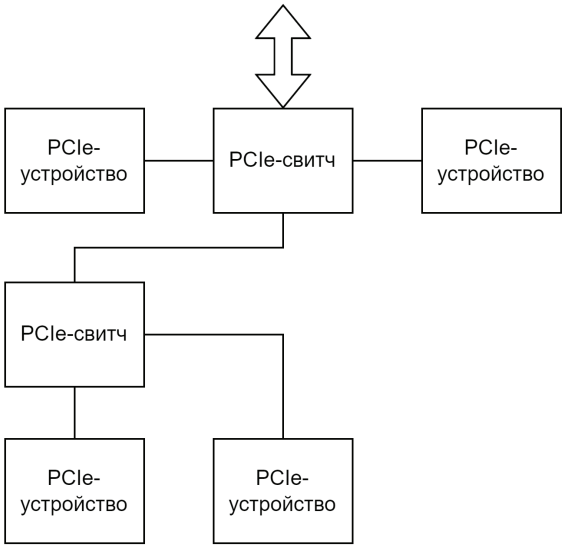


Рис. 0.69. Схема подключения PCIe-устройств

- PCI Express поддерживает горячую замену устройств, гарантированную полосу пропускания (QoS) и контроль целостности передаваемых данных.

Соединение между двумя устройствами PCI Express называется link и состоит из одного (называемого 1x) или нескольких (двух, четырех, восьми, шестнадцати) двунаправленных последовательных дифференциальных сигналов, называемых lane (линия). Каждое устройство должно поддерживать режим работы в соединении PCIe 1x. В минимальной конфигурации для установления соединения PCIe необходимо четыре сигнальные линии (по две для дифференциальных пар RX и TX).

В зависимости от версии интерфейса скорости PCIe могут существенно отличаться. Например, в самой первой спецификации PCI Express v 1.0 для соединения x1 теоретическая пропускная способность составляла 256 Мбайт/с, а x16 – 4 Гбайта/с соответственно. В спецификации PCI Express v7 приведены скорости 32 Гбайта/с для x1 и 512 Гбайт/с для x16.

Вероятность встретить интерфейс PCIe в современной высокопроизводительной встраиваемой системе весьма велика. В случае если в микроконтроллере нет аппаратной поддержки PCIe, она может быть реализована с использованием микросхемы FPGA или специальной микросхемы-моста (например, Broadcom PEX 8112, являющейся преобразователем PCIe в PCI и наоборот). На схеме показано использование микросхемы FPGA, подключающейся к микроконтроллеру по параллельному интерфейсу и реализующей преобразование транзакций для подключения PCIe-устройства:

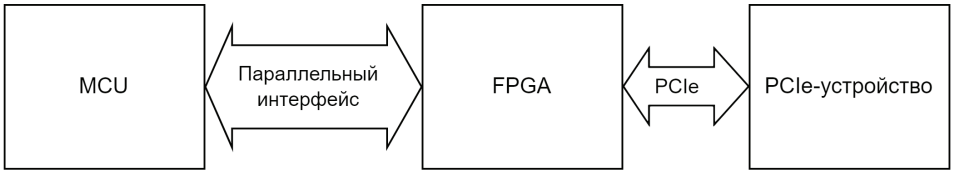


Рис. 0.70. Использование преобразователей для подключения PCIe-устройств

Отладочные и диагностические интерфейсы

Разъемы отладочных интерфейсов – наверное, это самая желанная сущность, которую хочет найти на печатной плате устройства исследователь любого уровня. И не спроста – отладочные интерфейсы открывают безграничные возможности по исследованию устройства, если, конечно, они не отключены.

Для серийно производящегося устройства в «релизной» конфигурации отладочные интерфейсы не нужны. Но т. к. мало кто из производителей специально делает параллельно два вида устройства – отладочную для разработки и релизную для серийного производства, их можно обнаружить практически на любом устройстве. Да и отлаживать релизную версию тоже как-то надо. А вот немного сэкономить на больших партиях устройств можно, не устанавливая отладочный разъем, а только оставив «пятачки» контактов на печатной плате устройства. Если разработчик хочет простыми способами защититься от начинающих исследователей железа, он может разорвать линии технологического интерфейса от микроконтроллера до контактных площадок, не устанавливая перемычки (или резисторы с номиналом 0 Ом). Внешне будет казаться, что разъем есть, а сигналы проходить не будут. Сможете найти места для установки перемычек на следующей картинке?

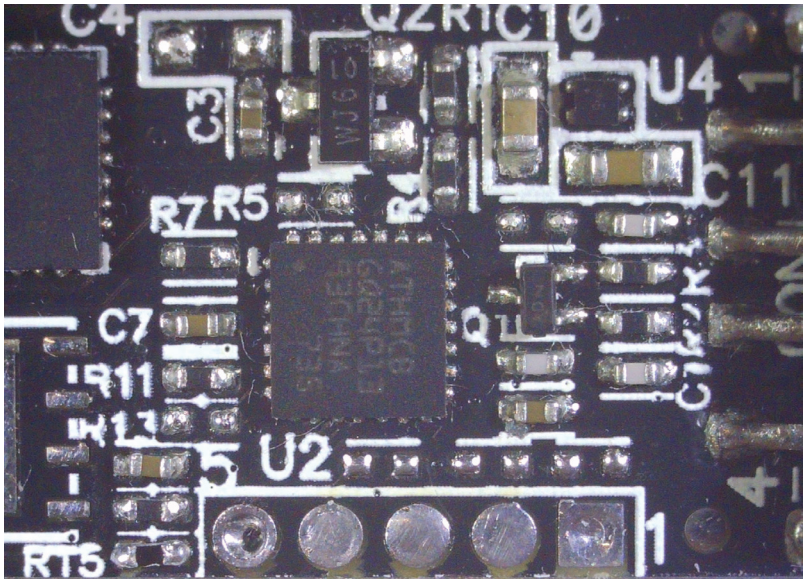


Рис. 0.71. Отладочный разъем с разорванными линиями связи

Чтобы восстановить электрические линии до разъема, нужно поставить три перемычки (или резистора номиналом 0 Ом) следующим образом:

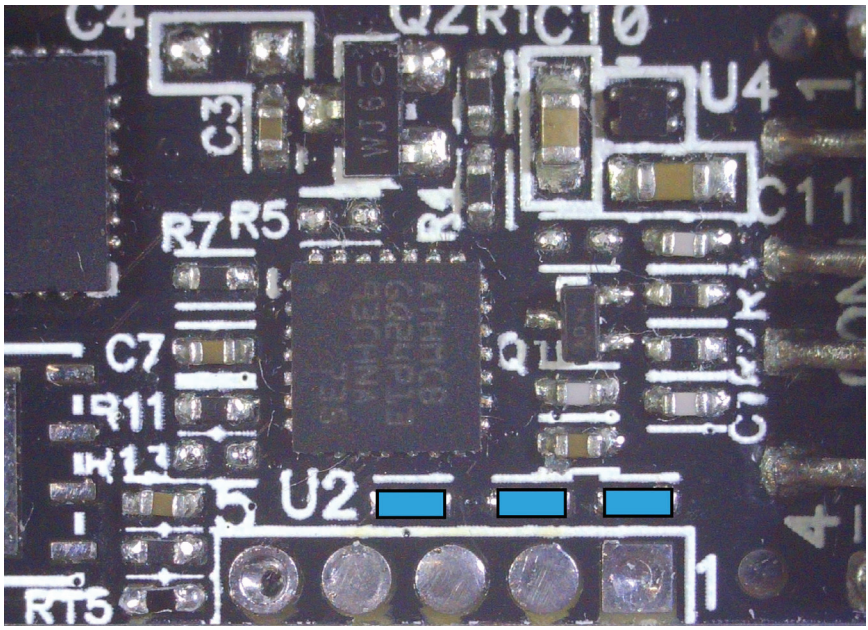


Рис. 0.72. Отладочный разъем с восстановленными линиями связи

Производитель может вообще не устанавливать разъем при разработке, а использовать специальную технологическую оснастку, которая подключается к контактным площадкам на плате устройства с помощью подпружиненных контактов – пого-пинов (pogo-pin). Пример показан на фото.

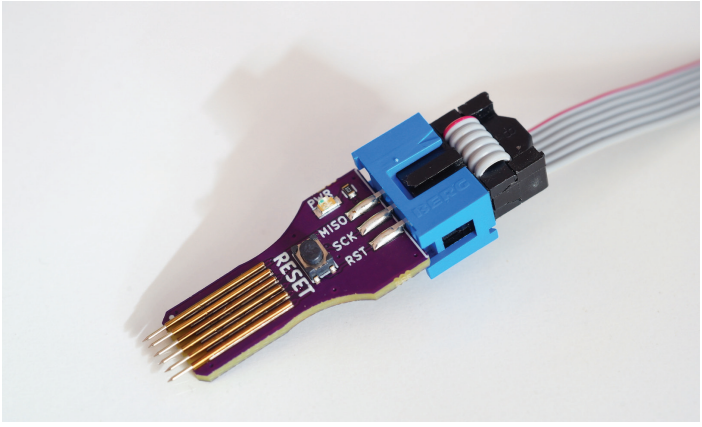


Рис. 0.73. Технологическая оснастка для подключения к плате без пайки[1]

Пора рассмотреть, какие технологические интерфейсы чаще других встречаются (и по факту являются стандартом) в цифровых устройствах и какие воз-

[1] https://www.waldgries.it/pogo-pin-programming-connector-k.html.

можности они могут дать исследователю безопасности. Производитель может использовать и нестандартные отладочные интерфейсы, особенно если архитектура микроконтроллера не сильно распространена.

JTAG

Joint Test Action Group – название группы специалистов по разработке стандарта Standard Test Access Port and Boundary-Scan Architecture. В итоге все начали называть стандарт названием группы. Люди придумали классную идею для тестирования устройств при производстве – возможность тестировать различные компоненты на устройстве через один разъем, при этом внутри компонента присутствует специальный аппаратный блок JTAG, позволяющий получить исчерпывающую информацию о состоянии компонента и каждого его вывода. Несмотря на всю мощь стандарта, как правило, при анализе простых устройств мы имеем дело только с одним микроконтроллером и подключаемся по интерфейсу JTAG только к нему. JTAG позволяет полноценно отлаживать прошивку микроконтроллера (читать и писать память, содержимое регистров микроконтроллера, ставить точки остановки и т. д.), а также работать с подключенной к микроконтроллеру flash-памятью.

Для работы JTAG нужно минимум четыре сигнала: TDI (test data input), TDO (test data output), TCK (test clock), TMS (test mode select). Иногда добавляется опциональный пятый сигнал TRST (test reset), и если он есть, его необходимо подключать. Также обязательно нужна линия GND. Стандартная схема подключения сигналов интерфейса JTAG показана на рисунке ниже.

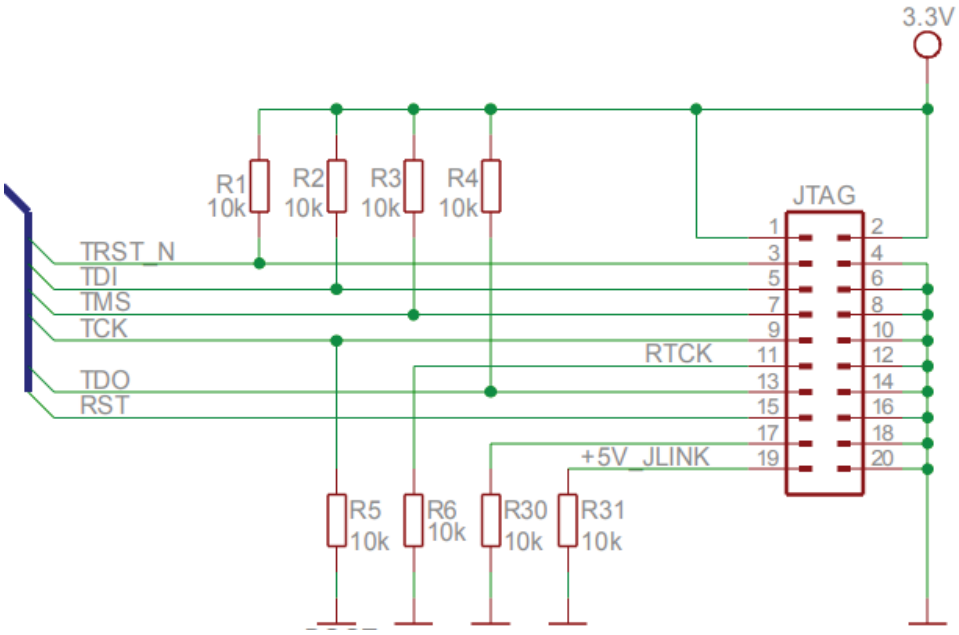


Рис. 0.74. Линии интерфейса JTAG, идущие от стандартного отладочного разъема[1]

[1] https://www.olimex.com/Products/ARM/NXP/LPC-H3131/resources/LPC-H3131-schematic.pdf.

Примеры сигналов интерфейса JTAG показаны на диаграмме:

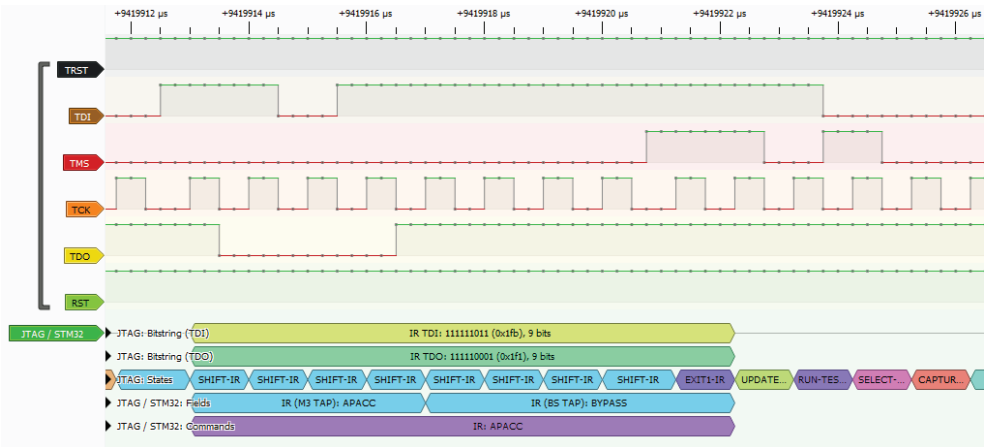
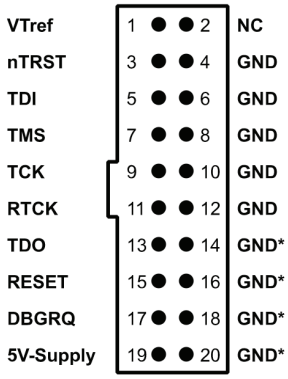


Рис. 0.75. Диаграмма передачи данных по интерфейсу JTAG

Стандартным разъемом для JTAG выступает 20-контактный разъем, почти половина контактов в котором является землей. Также применяются «урезанные» версии на 10 и меньше контактов.



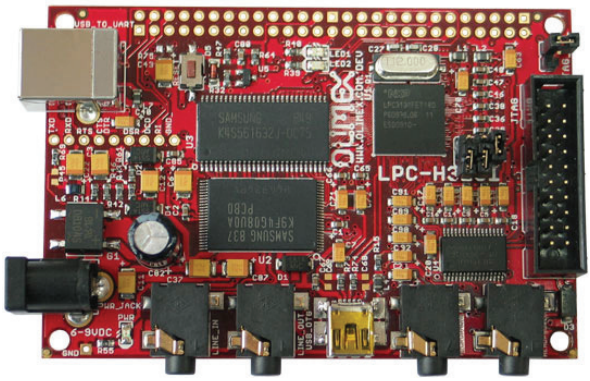


Рис. 0.76. Расположение контактов JTAG в стандартном 20-контактном разъеме JTAG и отладочная плата с 20-контактным разъемом для подключения отладчика[1]

Если мы встретим на плате группу из 4–6 контактных площадок – это хороший кандидат на JTAG-интерфейс. Часто эти контакты могут быть даже подписаны как JTAG или Debug (dbg). Однако иногда контакты JTAG могут быть рассредоточены по всей плате, пример – на рис. 0.77.

Для подключения к JTAG используется специальное устройство – аппаратный отладчик (дебаггер, debugger). Есть специализированные, работающие только с определенными процессорными ядрами, а есть универсальные, которые применяются наиболее часто, например JLink или что-то на базе чипа FTDI (допустим, FT2232). Последний не является полноценным отладчиком

[1] https://www.olimex.com/Products/ARM/NXP/LPC-H3131/.

сам по себе, но позволяет с помощью режима bitbang выставлять любое состояние на своих выводах, формируя в итоге требуемые для работы JTAG-сигналы.

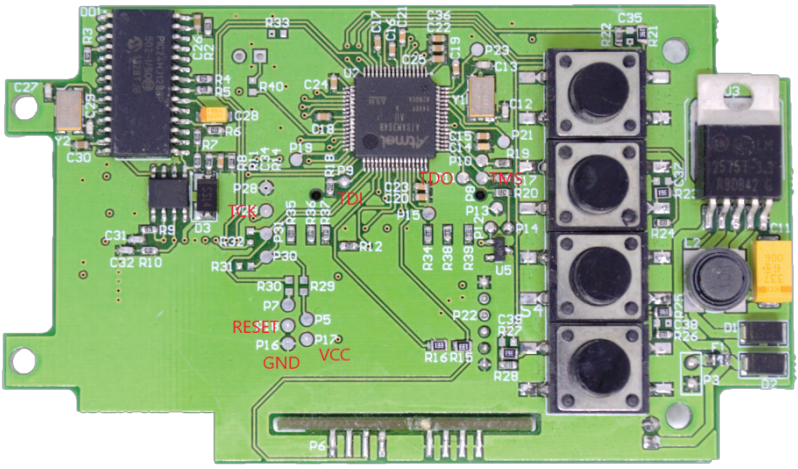


Рис. 0.77. Пример расположения контактов JTAG на плате устройства

Вернемся к фото контактных площадок (рис. 95). На этой фотографии контакты JTAG разбросаны по печатной плате в хаотичном порядке. Конечно, найти такой набор контактов сложнее, чем заботливо сгруппированный и подписанный производителем, но тоже можно. Для этого существуют специальные устройства – переборщики сигналов JTAG. Можно собрать самому (например, на базе проекта **https://github.com/szymonh/JTAGscan** или аналогичных), а можно купить. Проверенный и хорошо зарекомендовавший себя представитель – JTAGulator.

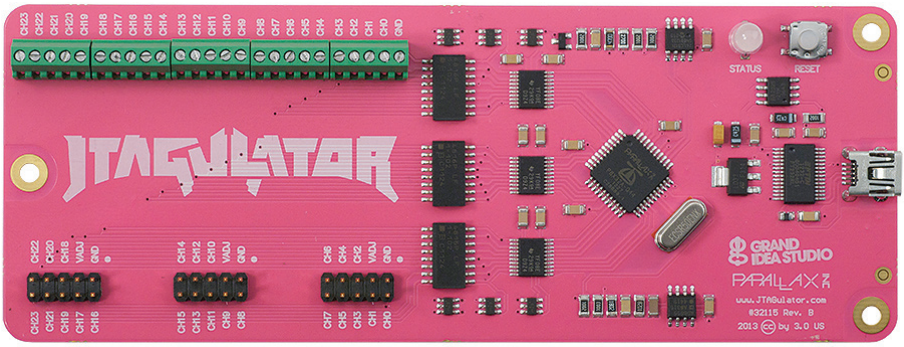


Рис. 0.78. Устройство поиска контактов JTAG – JTAGulator[1]

К нему подключается до 24 потенциальных контактов, среди которых мы надеемся найти JTAG, а он пытается определить, какой контакт соответствует каждому сигналу JTAG. И делает это весьма успешно! Конечно, если JTAG не отключен. Теперь об этой ложке дегтя в бочке меда отладочных интерфейсов. Так

[1] http://www.grandideastudio.com/jtagulator/.

как JTAG в большинстве случаев нужен только на этапе разработки устройства (иногда еще на производстве – для программирования), целесообразно его отключить, чтобы не дать лишней информации исследователям (или конкурентам, которые захотят получить максимум информации о продукте). Встречаются устройства, которые поддерживают обновление прошивки только через JTAG, что сопровождается выездом сервисного инженера производителя (т. е. платится много денег за поддержку). Но таких устройств все меньше и меньше, т. к. даже промышленные устройства теперь обновляются «программно» или через более привычные разъемы (например, с помощью подключенной USB-флешки). Вернемся к отключению JTAG. Микроконтроллер может отключить блок JTAG конфигурацией специальных программируемых битов. А управляет этим процессом программист, разрабатывающий прошивку устройства. Именно он должен не забыть в релизной конфигурации настроить микроконтроллер на отключение JTAG. При этом для некоторых ядер можно настраивать разные режимы отключения JTAG: полное (JTAGulator не сможет найти интерфейс) или варианты частичного. Например, запретив доступ к flash-памяти. Детальную информацию о режимах работы JTAG в конкретном случае можно найти в документации на чип микроконтроллера (если она есть) или хотя бы на процессорное ядро (например, ARM Cortex-M0).

Последнее, что нужно сказать в обзорном разделе про JTAG, – это частоты, на которых он работает. Как правило, это сотни килогерц, иногда – несколько мегагерц. Поэтому в большинстве случаев нет проблем с подключением достаточно длинных проводников к контактам JTAG, т. к. длинные проводники не создадут существенных помех на низких частотах. Однако если есть нестабильность работы интерфейса, всегда можно попробовать снизить частоту интерфейса, т. к. скорость интерфейса задается настройкой подключаемого отладчика.

SWD

Serial Wire Debug – отладочный интерфейс всего с двумя сигналами: SWDIO (SW data input-output) и SWCLK (SW clock). Существует опциональный сигнал трассировки SWO (SW trace output). Для корректной работы также часто нужен сигнал RESET. На следующем фото хорошо видны две дорожки: SWDIO и SWCLK, – подключенные к отладочным контактным площадкам.

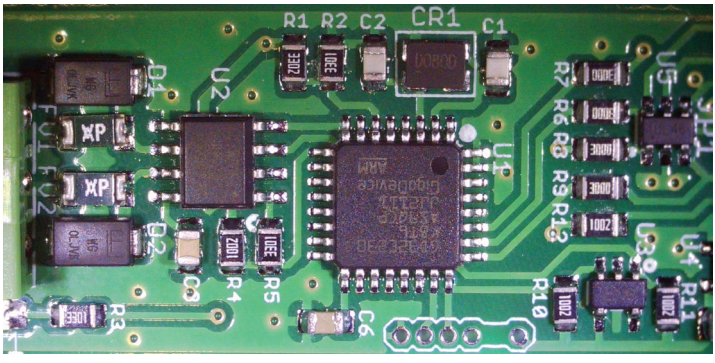


Рис. 0.79. Пример расположения контактов SWD на плате устройства

SWD – это реализация отладочного интерфейса для ядер с архитектурой ARM (в главе 1 мы рассмотрим оба интерфейса подробнее). Диаграмма с примером сигналов интерфейса SWD показана на рисунке:

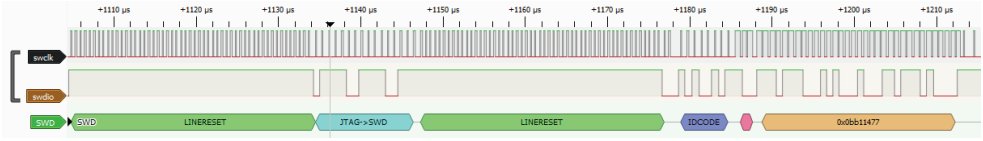


Рис. 0.80. Диаграмма передачи данных по интерфейсу SWD

Функциональные возможности аналогичны JTAG (полноценный доступ к регистрам микроконтроллера, памяти, управление потоком выполнения программы), за исключением периферийного сканирования. Частоты работы интерфейса, так же как и у JTAG, – единицы мегагерц. И да, JTAGulator (см. раздел JTAG) умеет искать и его. Для подключения отладчика к интерфейсу SWD может использоваться как стандартный 20-контактный разъем, так и сокращенные версии с меньшим количеством контактов.

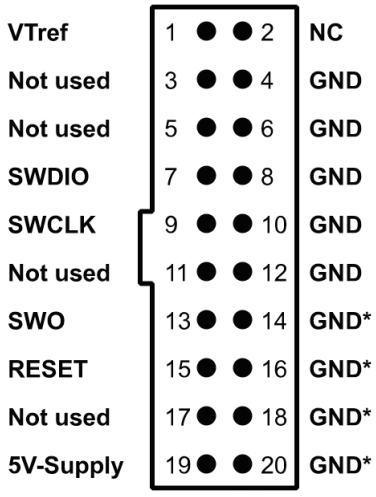


Рис. 0.81. Расположение контактов SWD в стандартном 20-контактном разъеме JTAG

UART

Почему UART – это отладочный интерфейс? Такой вопрос наверняка возникает при виде названия раздела. И этот вопрос абсолютно справедлив, ведь UART – не совсем полноценный отладочный интерфейс, т. к. через него нельзя как минимум управлять регистрами процессора. Однако в отладочный UART часто выводится лог загрузки устройства или даже отладочные сообщения в процессе работы устройства. Из плюсов – в большинстве случаев в UART пишутся человекочитаемые данные, т. е. мы можем их легко интерпретировать, просто посмотрев в консоль. Для подключения к UART достаточно двух контактов (TX и GND), если нам доступно только чтение данных. Если хотим что-

то писать на устройство, то нужен контакт RX. На плате устройства UART, как правило, выполнен в виде трех контактов – TX, RX и GND.

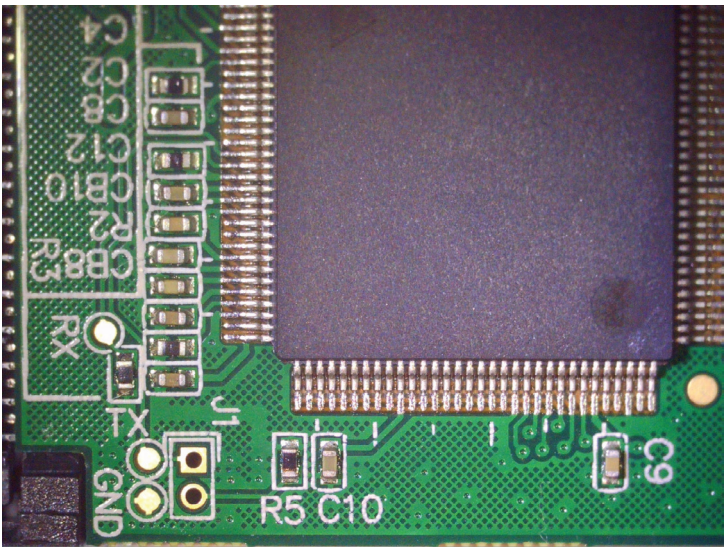


Рис. 0.82. Контактные площадки отладочного интерфейса UART на плате устройства

Диаграмма с примером сигналов интерфейса UART (только на линии TX) показана на рисунке:

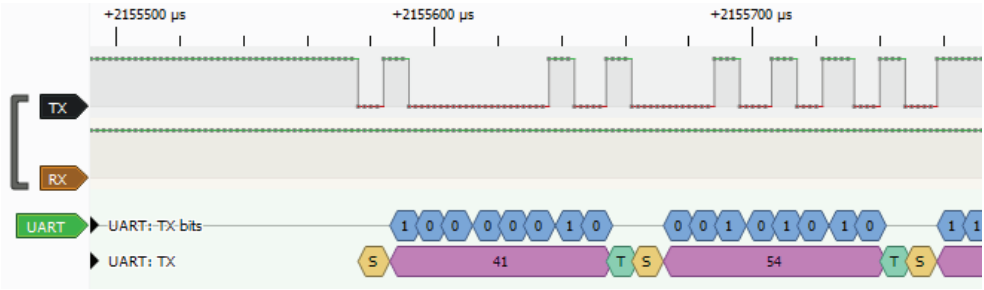


Рис. 0.83. Диаграмма передачи данных по интерфейсу UART

Если RX все же есть, то потенциально мы можем не только считывать данные из интерфейса, но и отправлять что-то, например команды. В отличие от JTAG и SWD, на отладочный UART нет никакой спецификации или стандарта, каждый производитель сам решает, какую информацию он выводит в UART и какой набор функциональных возможностей для отладки он реализует в устройстве с помощью Command Line Interface (CLI). Например, он может сделать ограниченный набор команд для получения статуса устройства или расширенный для считывания памяти устройства. А может подключить к отладочному UART консоль Linux, если устройство работает на этой ОС. В отличие от JTAG и SWD, вся логика работы отладочного UART (за исключением самого интерфейса UART) реализуется программно в прошивке устройства.

Для работы с UART можно использовать конвертеры USB в UART (большинство из них построено на базе микросхем FT232 и CP2102), пример подобного устройства показан на фото:

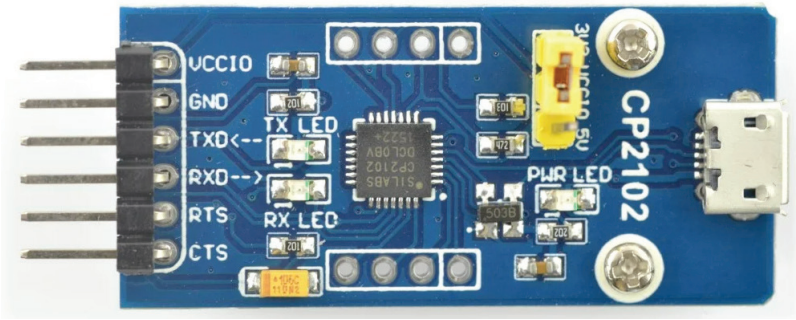


Рис. 0.84. Конвертер USB-UART[1]

Главное – не забыть, что сигналы TX приемника и RX передатчика замыкаются друг на друга, и наоборот. Ведь подключение сигналов TX-TX и RX-RX является самой распространенной ошибкой при работе с UART. Еще одной ошибкой может стать использование адаптера USB-UART для подключения к устройству с интерфейсом RS232 (который может быть помечен как RS232 UART). Стандарт RS232 допускает уровни сигналов, соответствующих логическим 0 и 1 от –15 В до +15 В, что гарантированно выведет из строя переходник USB-UART, скорее всего рассчитанный на уровни 0–3.3 В или 0–5 В.

Конечно, как и любой отладочный интерфейс, UART должен быть отключен на релизной версии устройства. Но в большинстве случаев он включен даже там, где отключен JTAG. И это отличная новость для любителей посмотреть, что же у устройства внутри.

Хардкор – рентген

Какое отношение рентген имеет к инженерному анализу устройства? На самом деле – самое прямое. Если каким-то образом у вас есть возможность сделать рентгеновский снимок печатной платы устройства, это может сильно помочь при анализе сложных случаев. Приведу несколько примеров.

1. Можно посмотреть, где проходят сигнальные линии на внутренних слоях печатной платы. Например, найти скрытые на внутреннем слое контакты JTAG-разъема. Или посмотреть трассировку интересующих интерфейсов без снятия BGA-чипа (конечно, если известна распиновка микросхемы). Ведь процедура выпаивания и запаивания BGA-чипа всегда может привести к выходу устройства из строя. А если оно у вас одно – почему бы не попробовать более безопасный способ?
2. С помощью рентгена можно определить наличие механизмов, срабатывающих при вскрытии корпуса устройства (тамперов) и стирающих важные для исследователя данные (ключи шифрования и т. п.).

[1] https://botland.store/uart-rs232-rs485-converters/5339-converter-usb-uart-cp2102-microusb-port--5903351248754.html.

3. Рентген позволит определить расположение кристалла микроконтроллера внутри корпуса чипа. Что очень полезно при неинвазивных атаках, например при атаке с помощью электромагнитных импульсов (об этом мы поговорим в разделе «Side Channel-атаки» в главе 1).
4. Рентген позволит проконтролировать корректность запаивания BGA-чипов, если пришлось их снимать. Опция рентген-контроля даже есть на фабриках, производящих сборку устройств.
5. Рентгеновское изображение исследуемого устройства круто смотрится в отчете об исследовании, статье или презентации :)

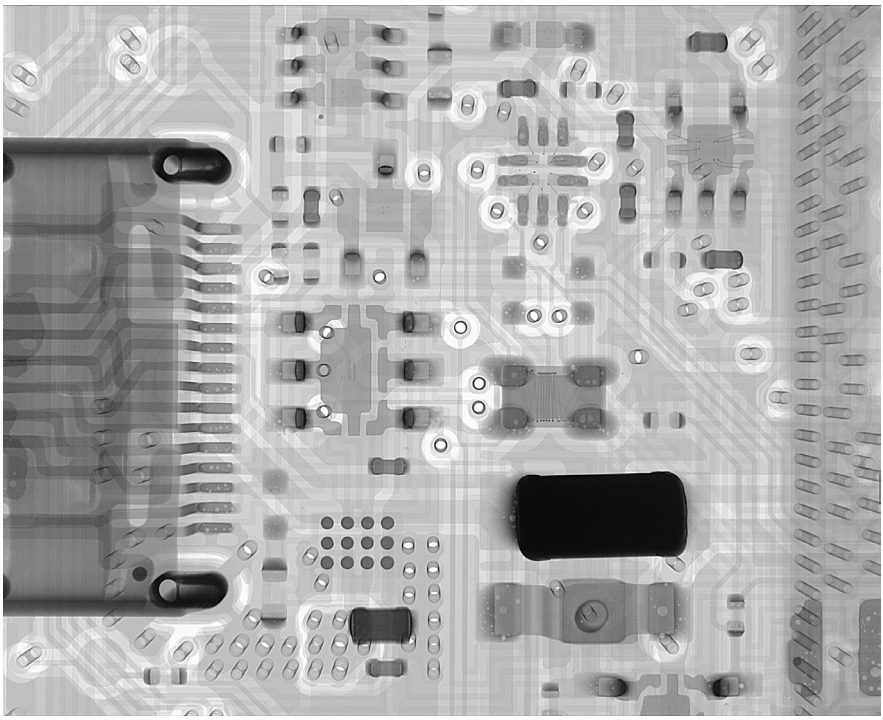


Рис. 0.85. Рентгеновский снимок печатной платы устройства. Слева – контакты разъема USB Type-C

Где можно сделать рентген устройства? Существуют специальные лаборатории, предоставляющие подобные услуги. Можно найти знакомых в институтах, оснащенных нужным оборудованием. Тут каждый исходит из своих возможностей, желаний и потребностей.

Level Up!

В разделе «Level Up!» мы будем подводить итоги главы, в данном случае главы «Level 0. Первичный анализ». В ходе первичного анализа мы должны были сделать несколько вещей, подготавливающих нас к следующей главе «Level 1. Добываем прошивку».

Первое, что мы сделали, – это *собрали информацию*. Документацию, описания и любые другие материалы, относящиеся к исследуемому устройству. Возможно, нашли утилиты обновления прошивки или сервисные утилиты для диагностики устройства. А может быть, скачали репозиторий github производителя устройства. Собранное мы уже посмотрели и представляем, что из этого нам может пригодиться и для чего.

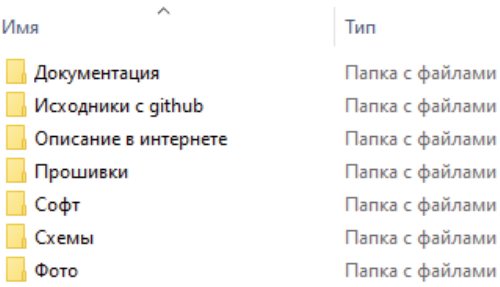


Рис. 0.86. Пример структурирования информации при проведении исследований

Второе – *составили список основных компонентов устройства* и даже отметили, где и что находится. Идентифицировали микроконтроллер, модули связи, память и даже отладочные интерфейсы устройства. Все, что вызывает у нас интерес и может ответить на вопрос о том, какие информационные потоки циркулируют в устройстве.

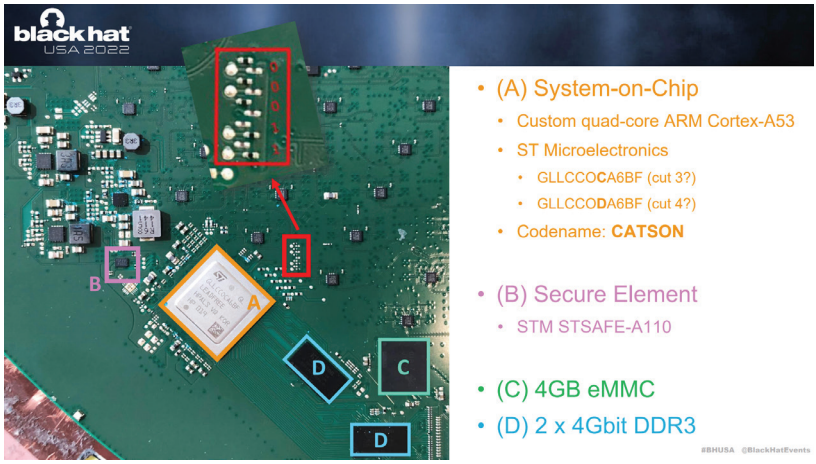


Рис. 0.87. Пример идентификации ключевых компонентов на плате устройства[1]

Третье – *составили структурную схему устройства и отметили основные информационные потоки, шины и протоколы* (на этом этапе как смогли, учитывая, что мы не смотрели устройство в динамике). Например, если стоит модуль приема сигналов GPS и по даташиту мы знаем, что он имеет интерфейс UART, – можем смело отмечать. Пример подобной схемы показан на рисунке:

[1] https://i.blackhat.com/USA-22/Wednesday/US-22-Wouters-Glitched-On-Earth.pdf.

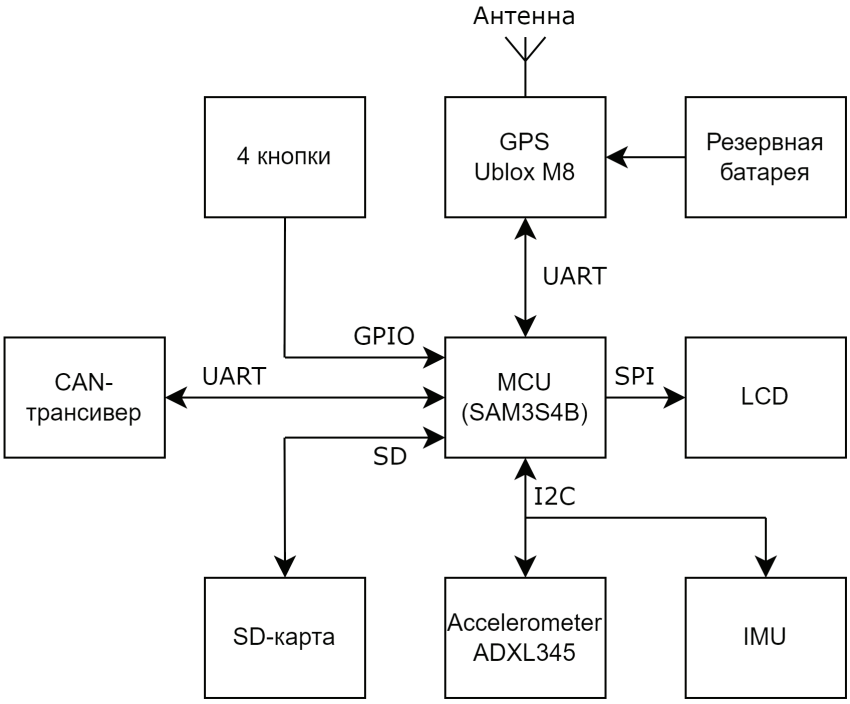


Рис. 0.88. Пример составленной структурной схемы устройства

Четвертое – *составили список необходимого оборудования* для дальнейшего исследования. Основным источником знаний для этого служат структурная и функциональная схемы. Так можно подобрать источник питания, конвертеры для работы с интерфейсами (UART, JTAG), а также устройства для анализа сигналов на шинах данных (логический анализатор или осциллограф).

Теперь у нас есть знание о функциональных возможностях устройства, о том, из каких ключевых компонентов оно состоит и как они связаны между собой. Самое время попытаться получить прошивку устройства и понять логику его работы.

Level 1

Добываем прошивку

В ходе первичного анализа мы уже собрали часть информации о функциональных возможностях устройства, но так как мы стараемся провести исследование методом «серого ящика» и получить как можно больше информации о внутреннем устройстве объекта исследований, самый лучший путь – попытаться получить прошивку устройства. Ответ на вопрос «Как именно устройство работает?» в большинстве случаев может быть получен только после дизассемблирования прошивки с восстановлением основных алгоритмов ее работы. В этой главе мы рассмотрим, как прошивку можно получить, от очевидных до нестандартных и даже недоступных многим исследователям встраиваемых систем путей. Знание подобных подходов позволяет комплексно посмотреть на проблему получения прошивки и придумать свой вариант решения этой задачи.

В большинстве микроконтроллеров есть внутренний начальный загрузчик (BootROM), формируемый на этапе производства чипа. Так как в большинстве случаев он является неизменным (в некоторых чипах существует механизм применения «патчей», хранящихся в однократно записываемой памяти внутри микроконтроллера, называемой fuse memory), то и имеет в своем названии аббревиатуру ROM (Read Only Memory). Его основная задача – найти и загрузить основную прошивку устройства (или загрузчик следующего уровня). В зависимости от микроконтроллера прошивка может храниться во внутренней flash-памяти микроконтроллера или загружаться с подключенной внешней микросхемы памяти. Как достать содержимое из внутренней памяти микроконтроллера, мы рассмотрим в этой главе, но чуть позже. А пока разберемся с самым простым случаем – прошивкой на внешней микросхеме ПЗУ.

Считывание из ПЗУ

В главе, посвященной первичному анализу устройства, мы рассмотрели основные типы чипов ПЗУ, которые могут встретиться при анализе цифровых устройств. Для считывания микросхемы ее (скорее всего) придется демонтировать, т. е. выпаять. Существуют специальные клипсы, позволяющие подключиться к выводам определенных корпусов микросхем (как правило, TSOP или SOIC) и считать содержимое чипа памяти без выпаивания. Внешний вид подобной клипсы для подключения показан на фото.

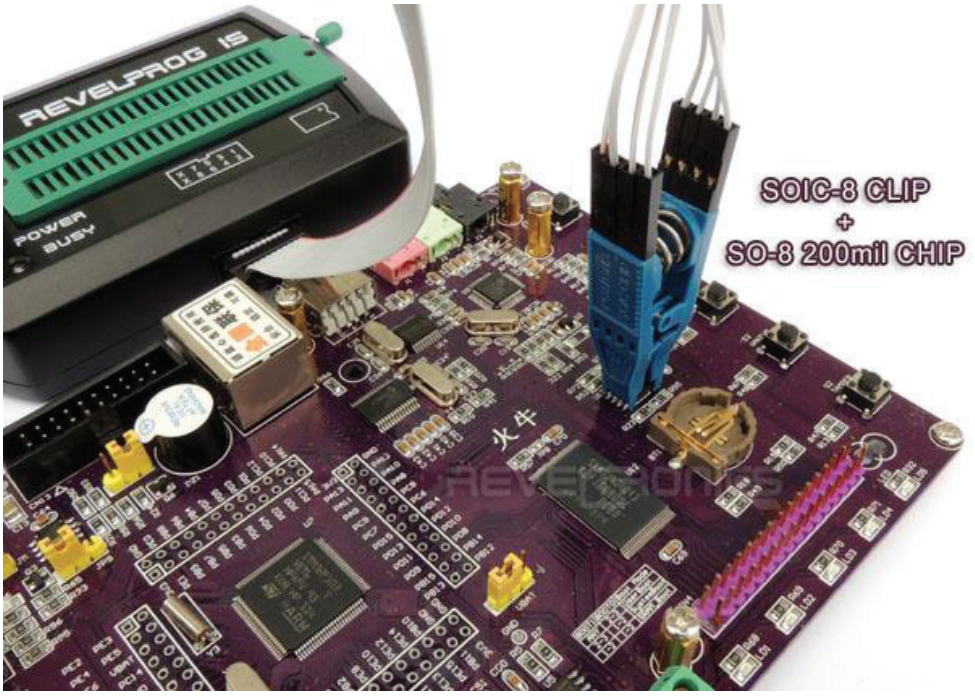


Рис. 1.1. Клипса для считывания микросхемы памяти без выпаивания[1]

Но если есть возможность выпаять микросхему и считать ее без подключения клипсы – лучше делать так. К тому же далеко не всегда электрическая схема платы позволяет считать содержимое микросхемы памяти без выпаивания, т. к., скорее всего, схема не рассчитана на такое подключение. Например, сигнал питания от программатора может идти на всю плату, и его мощности не хватит для корректного считывания. Описание процесса выпаивания в этой книге приводиться не будет, на YouTube существует много хороших обучающих видео. Всегда можно найти знакомого, «дружащего» с паяльником, или обратиться в сервис по ремонту техники, у них есть все необходимые знания и оборудование.

Допустим, микросхему ПЗУ мы выпаяли, теперь надо ее считать. Для этого существуют специальные устройства – программаторы микросхем памяти. Они могут не только считывать, но и записывать содержимое чипов памяти с помощью специального ПО. Программаторов существует огромное множество, ценой от нескольких сотен рублей на aliexpress до сотен тысяч рублей на специализированных сайтах. Их основное отличие – в количестве поддерживаемых типов микросхем памяти, поддержке редких микросхем, скорости и стабильности работы и готовности производителя добавить поддержку микросхемы по запросу пользователя. Для проведения исследований большинства устройств, особенно для начинающего исследователя, будет достаточно хорошо зарекомендовавших себя китайских устройств, например программаторов производства XGecu с расширенным набором адаптеров.

[1] https://www.reveltronics.com/en/shop/20/7/chip-programmers/clip-soic-8-pomona-5250-detail.

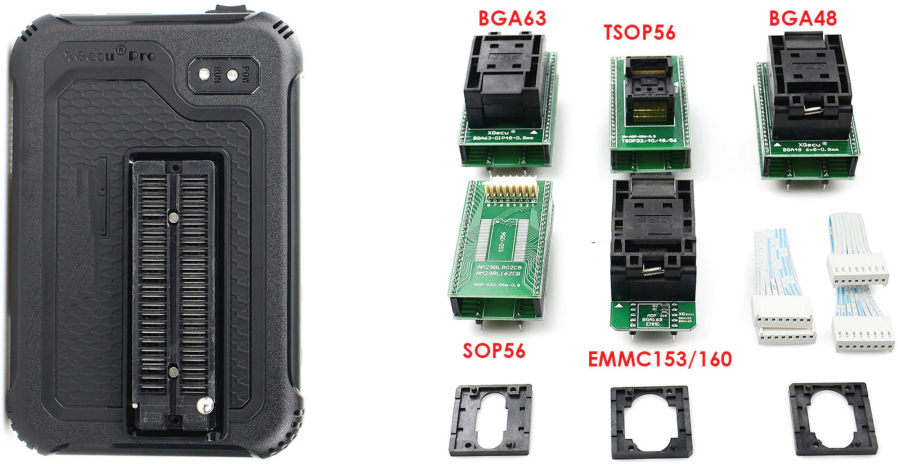


Рис. 1.2. Внешний вид программатора XGecu T56
с набором адаптеров для считывания различных типов микросхем[1]

Для управления программаторами используется специальное ПО. Например, для программаторов XGecu применяется программа XGpro, позволяющая задавать нужные опции при работе с разными типами микросхем памяти.

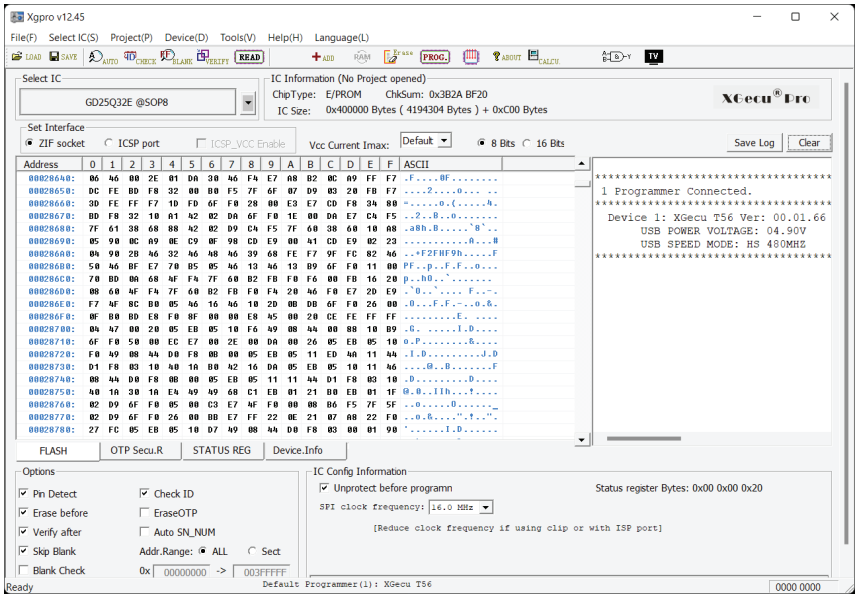


Рис. 1.3. Внешний вид главного окна ПО Xgrpo со считанным образом ПЗУ

Для разных типов микросхем могут применяться адаптеры (также называемые «кроватками»), позволяющие подключать различные типы микросхем к унифицированному разъему программатора. В кроватках каждому контакту микросхемы памяти соответствует специальный подпружиненный контакт.

[1] http://xgecu.com/.

Если кроватка предназначена для подключения к микросхемам с корпусами типа BGA, скорее всего, микросхему придется предварительно отреболить (от англ. reballing), т. е. восстановить шарики на контактах.

Если в устройстве стоит память типа EMMC (а из главы 0 мы знаем, что EMMC практически идентичны SD), то считать ее можно в обычном картридере для чтения карт памяти с помощью специального переходника EMMC-SD. Они бывают двух видов – дорогие, с кроватками, и дешевые, в которых микросхему памяти нужно запаивать.

Рис. 1.4. Пример переходника EMMC-SD с кроваткой[1]

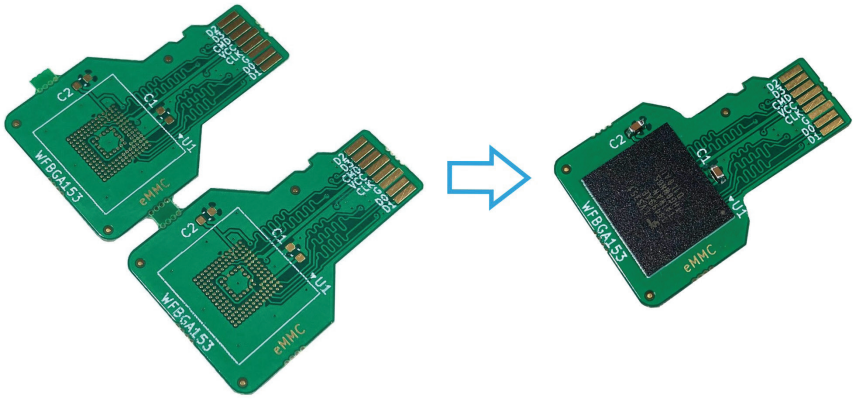


Рис. 1.5. Пример адаптера EMMC-SD с запаиваемой микросхемой[2]

Отдельно хочется отметить микросхемы NAND-памяти. Они являются наиболее сложными для считывания информации, и, по возможности, не стоит выбирать устройство с такой памятью для знакомства с миром реверс-инжиниринга встраиваемых систем. На это есть ряд причин.

Во-первых, микросхемы NAND-памяти не содержат внутри себя контроллера, исправляющего ошибки чтения информации (в отличие от, например, микросхем EMMC, внутри которых стоят такие же микросхемы NAND-памяти, но

[1] https://siredatech.en.made-in-china.com/productimage/oXPnxLfCsvUA-2f1j00iQHRVvecYsuD/China-Flash-Memory-Reader-Emmc-Adapter-to-SD-Card-Emmc-Test-Socket-Compatible-for-BGA153-and-BGA169-Perform-Forensic-Data-Recovery.html.

[2] https://www.tindie.com/products/voltlog/emmc-wfbga153-to-microsd-card-adapter-set-of-2/.

контроллер исправляет все ошибки перед отправкой данных в интерфейс). Почему именно память типа NAND склонна к появлению ошибок? Минимальной физической единицей flash-памяти является ячейка. Ячейки хранят определенный электрический заряд, уровень которого соответствует определенному двоичному значению, в простейшем случае это логический ноль (нет заряда) и логическая единица (есть заряд). Такая ячейка называется одноуровневой, или SLC (single-level cell), она обладает хорошими скоростными характеристиками, высокой надежностью и помехоустойчивостью, но имеет один значительный недостаток – высокую стоимость. Поэтому был придуман механизм, позволяющий повысить плотность хранения информации, не увеличивая количество ячеек, – использование нескольких уровней заряда в одной ячейке.

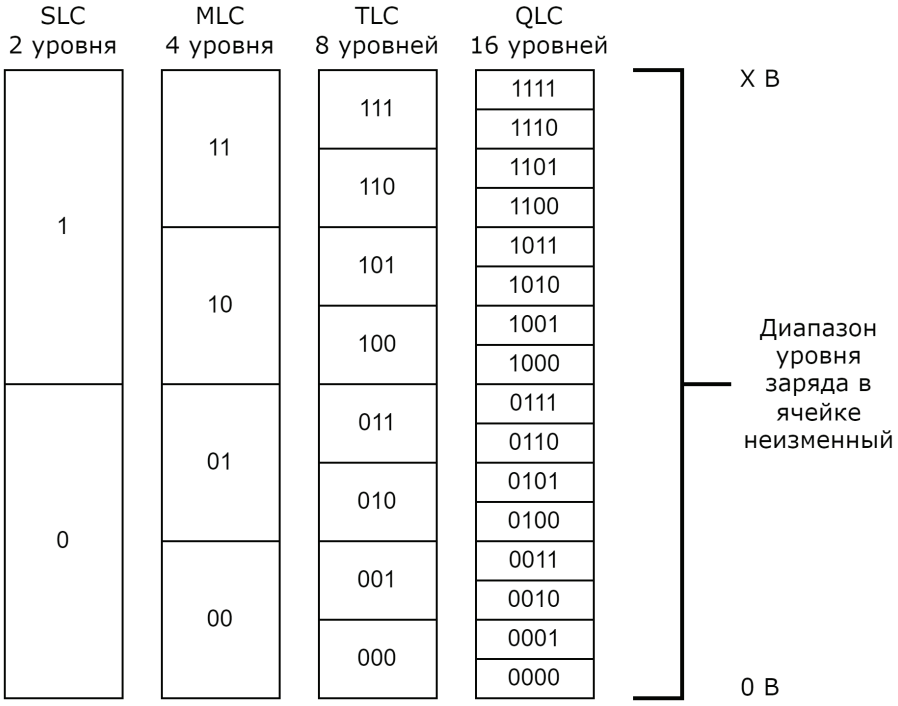


Рис. 1.6. Уровни заряда в ячейках NAND различных типов

Четыре уровня заряда позволяют хранить в одной ячейке уже два бита информации, такие ячейки называются многоуровневыми, или MLC (multi-level cell). Уступая SLC в скорости и надежности, они позволили существенно снизить стоимость NAND-памяти. Дальнейшее развитие технологий привело к созданию трех- и четырехуровневых ячеек: TLC (triple-level cell) и QLC (quad-level cell). При этом требуемое количество вариантов заряда растет пропорционально степени двойки, для TLC это 8, а для QLC – 16. В настоящее время разрабатываются пятиуровневые ячейки – PLC, которым потребуется уже 32 варианта заряда. При этом параллельно уменьшаются физические размеры транзисторов, из которых состоит ячейка NAND-памяти, т. к. постоянно идет переход на новые нормы техпроцесса производства кристаллов.

С одной стороны, такие подходы позволяют существенно увеличить плотность хранения информации, именно поэтому мы видим современные SSD-диски объемом в десятки терабайт. С другой стороны, такая память существенно медленнее и на порядки менее надежна, по сравнению с SLC. Чем больше уровней заряда, тем меньше разница между ними, а следовательно, ниже уровень помехоустойчивости, так как даже небольшой уровень помехи будет способен изменить состояние ячейки.

Для ячеек типа QLC абсолютно нормальным показателем является до 100 битовых ошибок на 1 Кбайт данных для памяти, только вышедшей с завода. Для их исправления применяются специальные математические механизмы исправления ошибок на основе корректирующих кодов (ECC). Только эти механизмы работают непосредственно в микроконтроллере, а мы же считываем память, выпаяв ее с платы устройства. Поэтому при считывании мы увидим все ошибки. Во многих современных микросхемах NAND существует специальный режим, использующийся для хранения особо важных данных, например прошивки. В этом режиме некоторые участки микросхемы (блоки) могут работать в режиме SLC, что существенно увеличивает помехоустойчивость хранимых данных. К сожалению, команды активации подобного режима, как и многих других, не стандартизованы (vendor specific), а документацию на конкретное семейство микросхем NAND бывает достать не так просто. Считав подобные блоки микросхемы памяти без активации данного режима, мы получим «мусор» вместо данных. Несмотря на то что существует Open NAND Flash Interface Specification (доступная на сайте **https://www.onfi.org/**), которую поддерживают основные вендоры, производящие микросхемы NAND-памяти, проприетарные команды активации нестандартных режимов работы найти бывает нелегко.

Вторым фактором, затрудняющим интерпретацию содержимого NAND-памяти, является то, что в большинстве случаев информация в ней хранится в модифицированном виде (как минимум в инвертированном или с наложенной битовой псевдослучайной «маской»). Таким способом разработчики пытаются «усреднить» заряд в ячейках и предотвратить его «утекание». Также, помимо данных, в NAND-памяти обычно хранятся специальные блоки с контрольными данными, нужными для исправления ошибок. То есть, даже считав содержимое микросхемы корректно (без битовых ошибок), нам придется отделять данные (data area), соответствующие прошивке, от служебных данных контроллера (spare area).

В-третьих, система адресации в таких микросхемах довольна сложна. Адрес состоит из нескольких частей, описывающих определенную иерархию структур. Разобраться в них не так просто, потому что вид иерархии адресов может зависеть от конкретной реализации кристалла микросхемы. Если у вас возникло желание глубоко разобраться в устройстве NAND-памяти и особенностях ее применения в устройствах хранения данных, советую прочитать книгу «Inside NAND Flash Memories» (**https://link.springer.com/book/10.1007/978-90-481-9431-5**).

Надеюсь, я убедил вас, что все эти факторы существенно усложняют получение корректного дампа микросхемы NAND-памяти.

Не рекомендую начинать свой путь исследователя встраиваемых систем со столь сложных вещей. Лучше выбрать устройство с SPI/I2C/EMMC-памятью и поскорее приступить к анализу прошивки (оптимальным выбором для начала будут устройства с 25-й серией SPI-памяти).

Сниффинг интерфейсов

Не всегда есть возможность выпаять микросхему памяти или отсутствует целесообразность, т. к. под рукой находится логический анализатор, он же сниффер (sniffer, от английского sniff – нюхать). Внешний вид анализатора показан на фото:

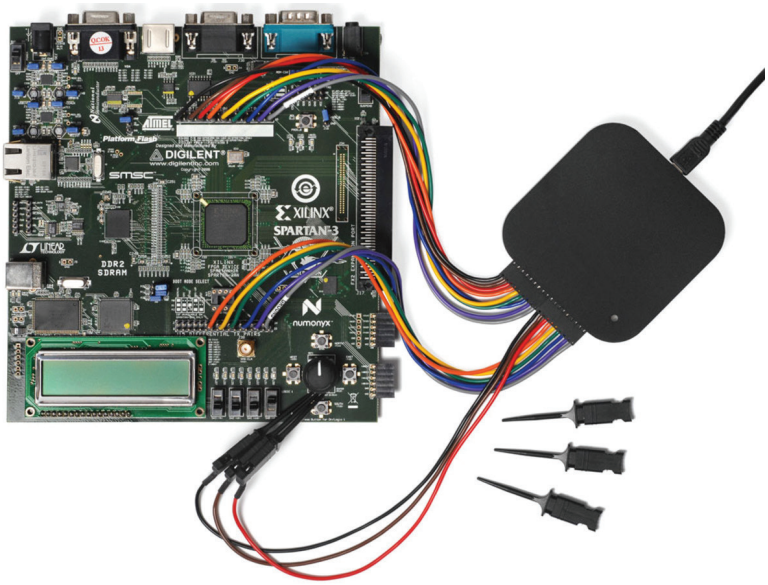


Рис. 1.7. Пример подключения сниффера к плате устройства[1]

Логический анализатор подключается параллельно анализируемому сигналу и позволяет записать его состояние за определенный промежуток времени. Большинство логических анализаторов имеют 8+ параллельно обрабатываемых входов. Они отличаются количеством входов, максимальной частотой и длительностью захватываемых сигналов. Минимальная частота работы анализатора должна быть в 2 раза выше частоты сигнала, а лучше, если она будет в четыре и больше раз выше. Объяснение этих цифр дает теорема Котельникова, доказывающая, что непрерывный сигнал с ограниченным спектром можно точно восстановить по его дискретным отсчетам, если они были взяты с частотой дискретизации, превышающей максимальную частоту сигнала минимум в два раза. То есть для сниффинга интерфейса SD, работающего на частоте 25 МГц, анализатор должен работать на частоте минимум 50 МГц, а лучше 100 МГц.

[1] https://blog.adafruit.com/2012/02/28/new-product-saleae-logic-16-16-channel-usb-logic-analyzer/comment-page-1/.

Анализатор подключается к компьютеру по USB и с помощью специального ПО позволяет просматривать состояние заснифанных сигналов. В ПО большинства анализаторов есть программные модули-декодеры распространенных протоколов. Например, если подключить анализатор к микросхеме SPI-памяти и записать лог загрузки устройства, то можно будет выгрузить дамп считываемой прошивки из ПО анализатора (притом не только дамп, но и последовательность его считывания, что может быть очень полезно для анализа внутреннего устройства прошивки). Пример подобного считывания для микросхемы с интерфейсом SPI показан на рисунке.

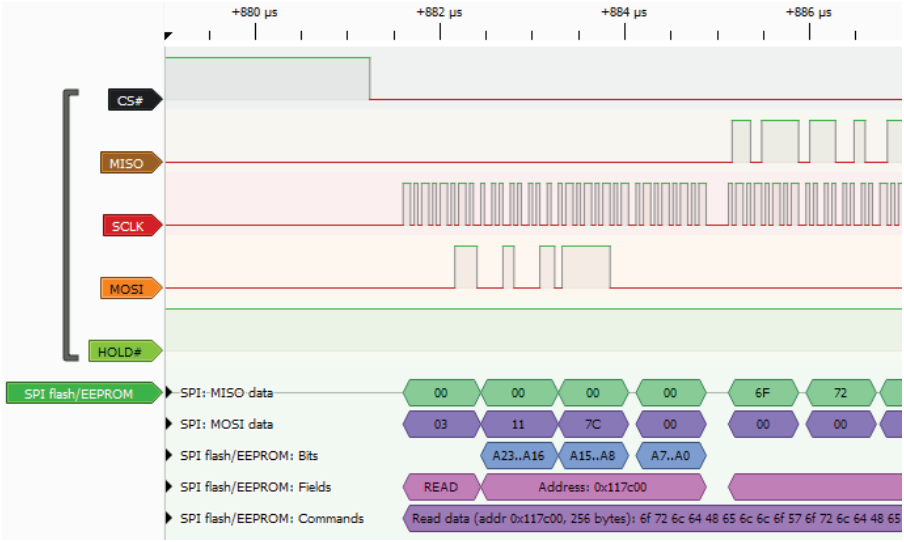


Рис. 1.8. Пример записанных логическим анализатором сигналов от микросхемы SPI ПЗУ

Если интересующего вас протокола нет в списке поддерживаемых, большинство анализаторов позволяют написать парсер протокола самому, используя распространенные языки программирования.

Для начального уровня исследования встраиваемых систем вполне хватит качественных логических анализаторов начального уровня, например фирмы DSLogic или аналогичных. Иногда приходится анализировать интерфейс с количеством линий больше, чем количество входов у используемой модели логического анализатора. Всегда можно попробовать достать еще один экземпляр анализатора и подключить их вместе (в некоторых анализаторах даже есть специальные входы и выходы для синхронизации нескольких устройств). Важное условие – сигнал триггера должен быть заведен на оба анализатора, иначе не получится получить единую точку отсчета. Конечно, декодер протокола в таком случае работать не будет, но, как правило, можно или посмотреть интересующий сигнал «вручную», или сделать экспорт логов и написать свой декодер, например на Python. На фото показаны два анализатора DSLogic, имеющие 16 входов, использующихся одновременно для анализа интерфейса Compact Flash.

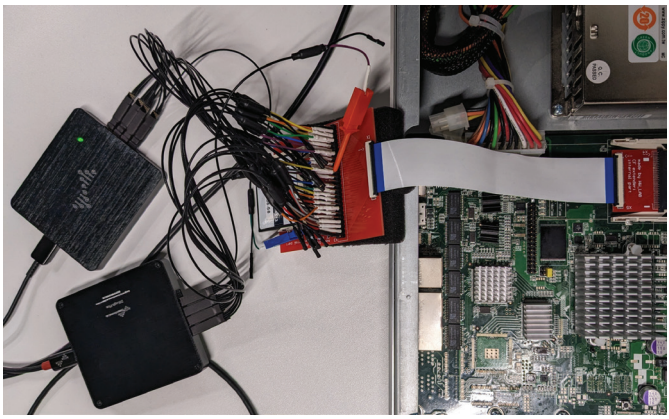


Рис. 1.9. Два логических анализатора, подключенных одновременно

Для анализа высокоскоростных сигналов, работающих на сотнях и тысячах мегагерц, подобные модели уже не подойдут. Для работы с высокоскоростными сигналами и с шинами с большим количеством линий существуют профессиональные модели осциллографов с опцией логического анализатора от известных фирм, например Keysight (Agilent), Rohde&Schwarz или Tektronix. Они могут стоить десятки и даже сотни тысяч долларов, но, например, позволяют анализировать сигналы всех выводов современных x86-64 процессоров. Они позволяют работать с активными пробниками (т. е. требующими собственного внешнего питания), которые обеспечивают измерение сигналов с широкой полосой пропускания. Как правило, такие пробники дороже пассивных и имеют более узкий диапазон входных напряжений, однако благодаря значительно меньшей емкостной нагрузке они обеспечивают более точные измерения быстро изменяющихся сигналов. Внешний вид подобных «монстров» из мира анализаторов показан на фото:

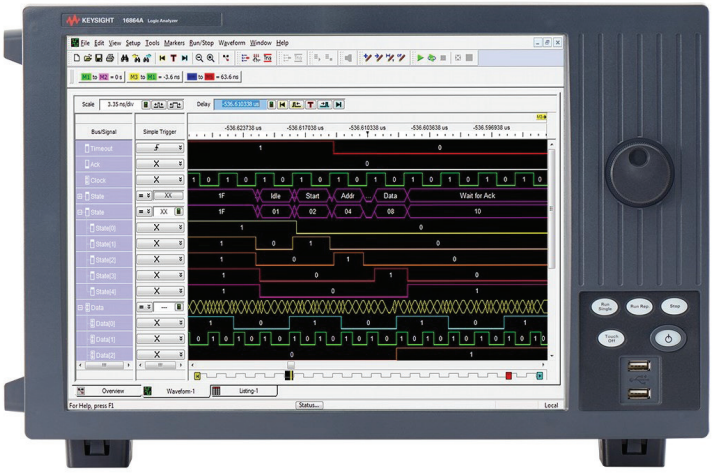


Рис. 1.10. Логический анализатор профессионального уровня[1]

[1] https://www.keysight.com/zz/en/product/16863A/16863a-102-channel-portable-logic-analyzer.html.

Ну и конечно же, применение логических анализаторов не ограничивается получением образа ПЗУ. Это одно из самых часто применяемых устройств при исследовании цифровых устройств. С помощью него можно просматривать состояние практически любых сигналов «в динамике», смотреть трафик на поддерживаемых анализатором интерфейсах и получить множество информации, полезной для понимания логики работы устройства.

Мы рассмотрели два распространенных способа получения дампа внешней микросхемы памяти, в которой может храниться прошивка: считывание программатором и снифинг логическим анализатором. Но в большинстве микроконтроллеров есть внутренняя flash-память, доступ к которой такими способами получить невозможно. В ней может храниться первичный загрузчик (BootROM), часть прошивки или даже вся прошивка целиком. Рассмотрим, каким образом можно получить доступ к содержимому внутренней flash-памяти, и начнем мы с самого очевидного способа – доступа через отладочные интерфейсы.

Считывание через отладочные интерфейсы

Считаем, что нам повезло в ходе первичного анализа устройства: мы обнаружили JTAG/SWD-разъем или хотя бы контактные площадки. В главе 0 мы уже кратко коснулись темы, что такое JTAG и SWD, но для дальнейших исследований нам потребуется более глубокое понимание специфики работы этих интерфейсов, ведь через них можно не только получить доступ к прошивке, но и полноценно отлаживать устройство. Следующие два раздела довольно сложны для восприятия, несмотря на то что я постарался максимально простым языком объяснить базовые концепции. Не стоит разочаровываться, если вы сразу не сможете понять рассматриваемые в них вещи, проводить множество исследований цифровых устройств можно и без глубокого понимания принципов работы JTAG и SWD. Современное ПО, работающее с отладчиками, скрывает особенности реализации отладочных интерфейсов от пользователя, так что, даже не до конца понимая особенности работы этих интерфейсов, можно успешно пользоваться их функциями.

JTAG

В главе 0 мы говорили, что аббревиатура JTAG (Joint Test Action Group) не является правильной для стандарта IEEE Std 1149.1 (IEEE Standard Test Access Port and Boundary-Scan Architecture), описывающего отладочный интерфейс. Однако в обиход для обозначения этого стандарта и отладочного интерфейса вошла именно она. Стандарт описывает возможности, намного более широкие, чем применяются в большинстве устройств. Но для использования или поиска уязвимостей в механизмах безопасности JTAG придется немного погрузиться в его внутреннее устройство.

Изначально JTAG был придуман для периферийного тестирования (boundary-scan) сигналов на выводах микросхем, а именно для определения обрывов, закороченных или неправильно подключенных проводников. Если мы можем генерировать сигнал на выводах одной микросхемы и проверять

его на нужных выводах другой микросхемы, то сможем продиагностировать корректность всех подключений. Пример подобной диагностики соединения двух микросхем показан на рисунке.

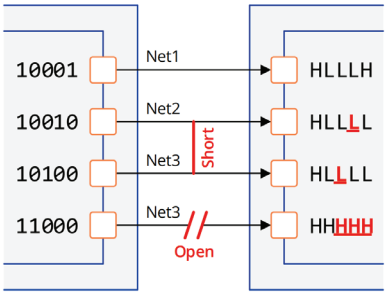


Рис. 1.11. Определение ошибок при периферийном сканировании JTAG[1]

До разработки JTAG подобный анализ выполнялся только за счет подключения к большому количеству тестовых контактов на плате (test points) с помощью специальных пробников, позиционирующихся в нужных местах, или «летающих щупов» (подробнее про функциональное тестирование плат можно прочитать в статье по адресу: **https://habr.com/ru/company/thirdpin/blog/425569/**).

Для реализации периферийного сканирования в микроконтроллере должен быть специальный блок JTAG, включающий в себя Test Access Port контроллер (TAP-контроллер), обеспечивающий доступ к выводам (точнее, к специальным ячейкам ввода-вывода Boundary Scan Cells), а также набор регистров. Схематично набор архитектурных блоков JTAG внутри микросхемы показан на рисунке (обратите внимание, что сигнал данных входит по линии TDI, проходит последовательно сквозь все блоки, выходя по линии TDO).

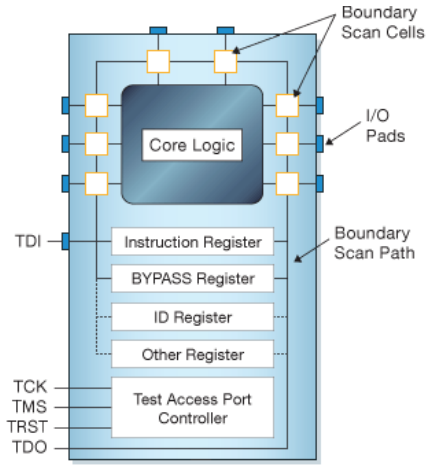


Рис. 1.12. Структурная схема блока JTAG[2]

[1] https://www.corelis.com/education/tutorials/jtag-tutorial/what-is-jtag/.

[2] https://www.xjtag.com/about-jtag/jtag-a-technical-overview/.

Эта архитектура позволяет не только контролировать состояние выводов, но и управлять ими. Таким образом, можно проводить отладку цифровых микросхем или устройств уровня печатной платы, т. к. сигнал проходит все микросхемы поочередно, соединяя их в цепочку (JTAG chain). Несмотря на всю мощь стандарта, как правило, при анализе простых устройств мы имеем дело только с одним микроконтроллером и подключаемся по интерфейсу JTAG только к нему. Пример цепочки JTAG для трех устройств показан на рисунке.

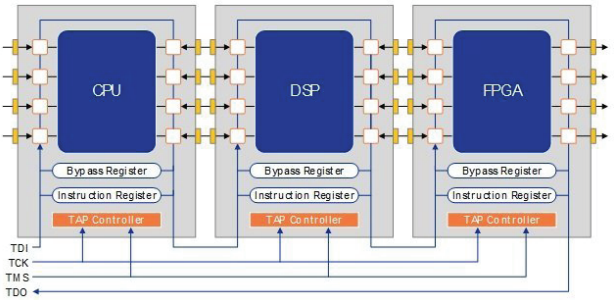


Рис. 1.13. Цепочка JTAG-контроллеров[1]

В каждом блоке JTAG должен присутствовать минимальный набор регистров: один регистр инструкций (Instruction Register, IR) и два регистра данных (Data Register, DR): BSR и BYPASS. В регистре инструкций хранится текущая команда, на основе которой TAP-контроллер может корректно обработать поступающие сигналы. Основные регистры данных выполняют следующие функции:

- BSR (Boundary Scan Register) – основной регистр данных для тестирования устройства. Он используется для взаимодействия с I/O-выводами микросхемы;
- BYPASS – однобитный регистр, передающий данные с сигнала TDI на TDO. Позволяет тестировать другие микросхемы в JTAG-цепочке, пропуская внутреннюю схему периферийного сканирования текущей микросхемы. На рисунке показан пример работы первого устройства (CPU) в JTAG-цепочке в режиме BYPASS, в то время как второе устройство (FPGA) работает в режиме Boundary Scan;

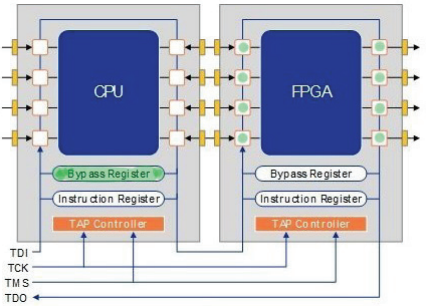


Рис. 1.14. Цепочка JTAG-контроллеров в режиме BYPASS[2]

[1] https://www.corelis.com/education/tutorials/jtag-tutorial/jtag-technical-primer/.
[2] https://www.corelis.com/education/tutorials/jtag-tutorial/jtag-technical-primer/.

- IDCODE – в этом регистре хранятся ID-код и номер ревизии устройства. Эта информация помогает выбрать корректный файл настроек для работы всех функций JTAG. Если вы работаете с чипом неизвестной архитектуры (например, с затертой маркировкой), то по ID часто можно определить тип архитектуры и производителя микроконтроллера, а иногда даже модель.

Управление функциями JTAG происходит посредством TAP-контроллера, внутри которого находится конечный автомат с 16 состояниями, управляемый сигналом TMS. Переходы между состояниями происходят по переднему фронту сигнала TCK. Запись данных на линии TDI происходит по фронту сигнала TCK, а чтение данных на линии TDO – по спаду. Так как сигнал TMS подключен ко всем TAP-контроллерам в цепочке JTAG параллельно, все TAP-контроллеры всегда находятся в одинаковом состоянии. Схема состояний конечного автомата TAP-контроллера показана ниже.

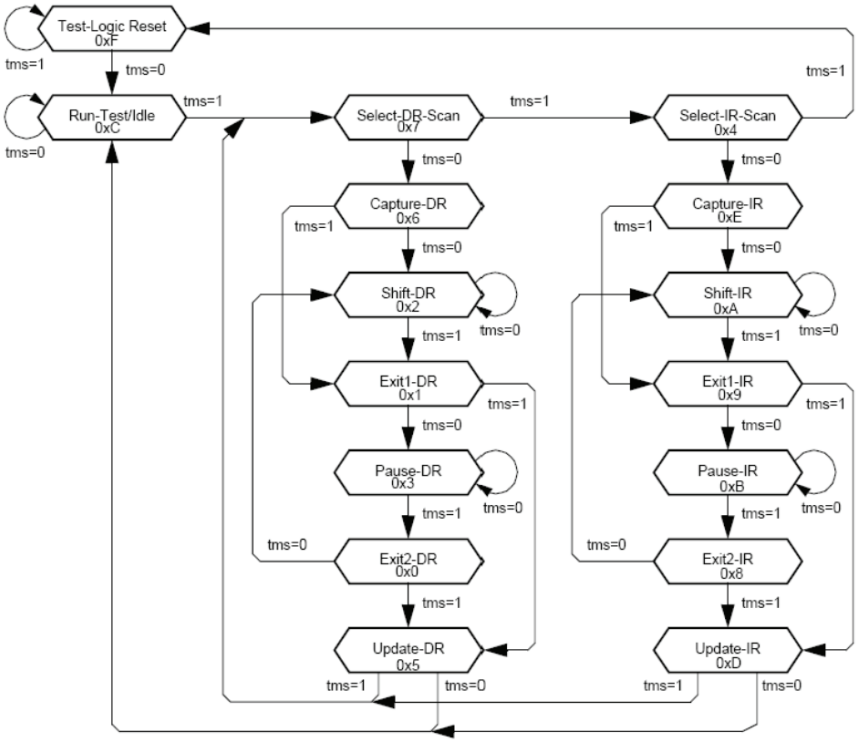


Рис. 1.15. Схема состояний конечного автомата JTAG[1]

Кратко рассмотрим назначение состояний TAP-контроллера:

- Test-Logic-Reset – исходное состояние, в котором переинициализируются внутренние схемы JTAG и в котором автомат находится после включения; TAP-контроллер переходит в это состояние, если приходит сигнал по линии TRST (является опциональным). Независимо от своего

[1] https://developer.arm.com/documentation/ddi0168/a/debug-support/the-jtag-state-machine.

начального состояния TAP-контроллер перейдет в состояние Test-Logic-Reset, если на линии TMS будет сигнал логической единицы 5 циклов сигнала синхронизации TCK. Именно поэтому мы можем переинициализировать схему без сигнала TRST;

- Run-Test/Idle – переходное состояние контроллера при ожидании выполнения тестов или следующей команды;
- Select-IR, Select-DR – временные состояния, после которых возможна инициация загрузки последовательности данных в соответствующие выбранные регистры (IR для состояния Select-IR/Scan и DR в состоянии Select-DR/Scan);
- Capture-IR, Capture-DR – состояния приема команд (в регистр IR) или данных (в регистр DR), в зависимости от ранее выбранного регистра;
- Shift-IR, Shift-DR – состояния сдвига команд, данных. В этом состоянии необходимые данные загружаются (или выгружаются) в соответствующий регистр. TAP-контроллер будет находиться в этом состоянии, пока сигнал на линии TMS = 0. На каждом фронте сигнала синхронизации один бит данных записывается (или считывается) в выбранный регистр (IR/TD) на линии TDI (TDO);
- Exit1-IR, Exit2-IR – выходы из режима работы с командами, все нужные тестовые данные загружены в соответствующие регистры;
- Exit1-DR, Exit2-DR – выходы из режима работы с данными, все нужные тестовые данные загружены в соответствующие регистры;
- Pause-IR, Pause-DR – состояние паузы, необходимое для ожидания выполнения некоторых внешних операций;
- Update-IR, Update-DR – состояние обновления данных внутренних регистров TAP-контроллера.

Рассмотрим универсальные команды, используемые при работе с JTAG-интерфейсом (фактически данные команды представляют собой базовый транспорт, над которым вендоры строят более сложное взаимодействие):

- IDCODE – в результате выполнения этой инструкции линии TDI и TDO подключаются к регистру IDCODE. После ресета TAP-контроллера инструкция IDCODE автоматически записывается в IR, следовательно, значение IDCODE может быть считано с помощью 32 тактов сигнала TCK (т. к. регистр IDCODE имеет размерность 32 бита). Если в цепочке JTAG больше одного устройства, то значения всех регистров IDCODE могут быть считаны за 32*(количество устройств) последовательных такта;
- BYPASS – инструкция соединяет линии TDI и TDO через регистр BYPASS;
- EXTEST – подключает линии TDI и TDO к регистру BSR. Состояние выходов устройства может быть считано с помощью состояния TAP-контроллера Capture-DR, а записано с помощью состояния Shift-DR;
- SAMPLE/PRELOAD – аналогично команде EXTEST, линии TDI и TDO подключаются к регистру BSR, однако устройство продолжает функционировать в обычном режиме. С помощью состояния Capture-DR возможен

«сниффинг» значений на выходах устройства без вмешательства в его работу.

На двух следующих рисунках показаны отправка команды IDCODE (первая диаграмма) и получение ответа на нее в виде 0x3ba00477 (вторая диаграмма). Обратите внимание, что запись данных по линии TDI происходит по фронту сигнала TCK, а обновление данных на линии TDO – по спаду.

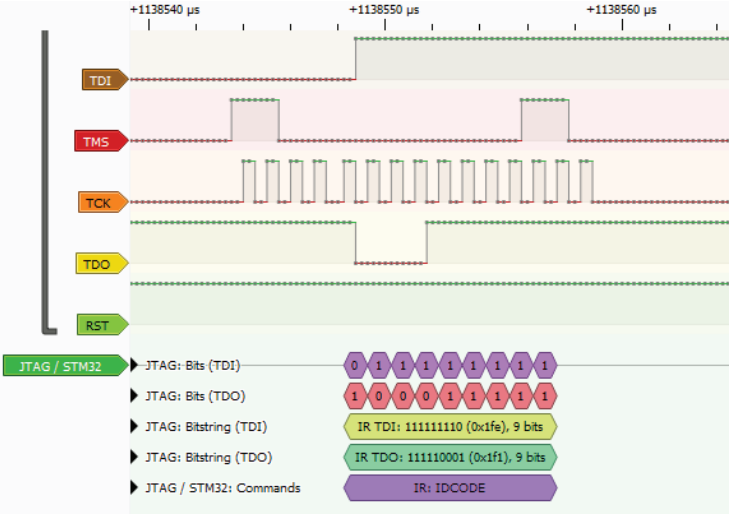


Рис. 1.16. Пример диаграммы сигналов JTAG при отправке команды IDCODE

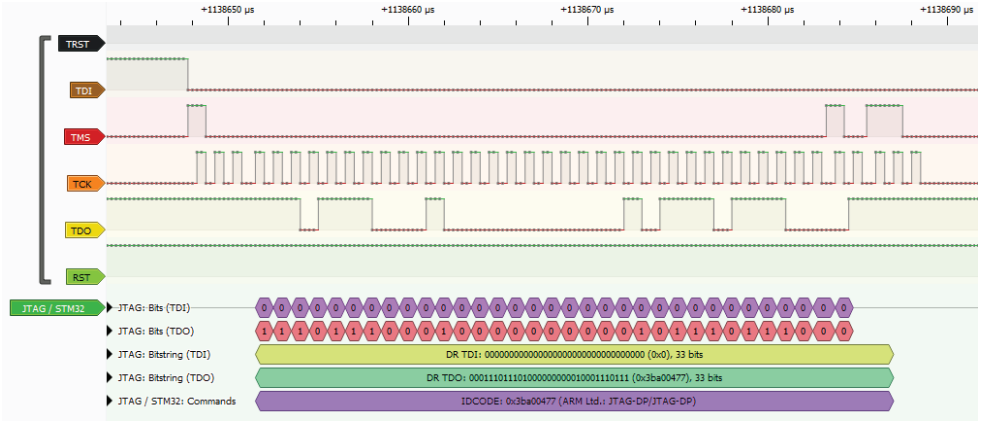


Рис. 1.17. Пример диаграммы сигналов JTAG при получении ответа на команду IDCODE

Мы кратко рассмотрели архитектуру и устройство отладочного интерфейса JTAG. Если есть желание разобраться с особенностями реализации, советую прочитать сам стандарт (**https://standards.ieee.org/ieee/1149.1/4484/**) и провести эксперименты с open source

проектом для периферийного сканирования микроконтроллера STM32F103C8T6 (**https://github.com/jxwleong/jtag-boundary-scan**).

В главе 0 мы говорили о специальном устройстве – JTAGulator'e, позволяющем определить сигналы JTAG интерфейса. Теперь мы разбираемся в устройстве JTAG достаточно, чтобы понять, как работают подобные устройства. Мы уже знаем, что после ресета TAP-контроллер должен выполнить инструкцию IDCODE, т. е. подключить на линии TDO/TDI регистр IDCODE. Следовательно, подав 32 тактовых сигнала на предполагаемую линию TCK и получив валидное значение ID по линии TDO, мы определяем сразу два сигнала. Также мы знаем, что переключение контроллера в состояние BYPASS соединяет линии TDI и TDO с помощью однобитного регистра. В этом режиме достаточно записывать известный паттерн на линию TDI и ожидать его с задержкой в 1 бит на предполагаемой линии TDO. Примерно такой алгоритм с перебором всех возможных комбинаций линий и сигналов применяется в подобных устройствах. Также для определения сигналов используются их электрические характеристики, в основном оценка сопротивления. Для разных сигналов JTAG рекомендуется подтяжка разными номиналами резисторов к VCC или к GND (только TCK). Следовательно, найдя линию, подтянутую к GND, можно сделать предположение, что это сигнал TCK, и пробовать ранее описанные техники.

Несмотря на то что изначально JTAG был разработан для тестирования электронной схемы устройства, во многих устройствах JTAG используется для отладки, т. е. доступа к регистрам процессорного ядра и системной шине микроконтроллера, активации точек остановки, доступа к flash-памяти и т. д. Все эти операции выполняются через отдельный TAP-контроллер, при этом система команд и внутренняя архитектура данного TAP-контроллера стандартизованы на уровне производителя вычислительного ядра (например, ARM) или самого микроконтроллера (например, STM). Однако конечный автомат TAP-контроллера с 16 состояниями остается неизменным. В случае двух и более TAP-контроллеров внутри одного чипа они также объединяются в JTAG-цепочку, пример из документации на микроконтроллер STM32F1xx показан на схеме.

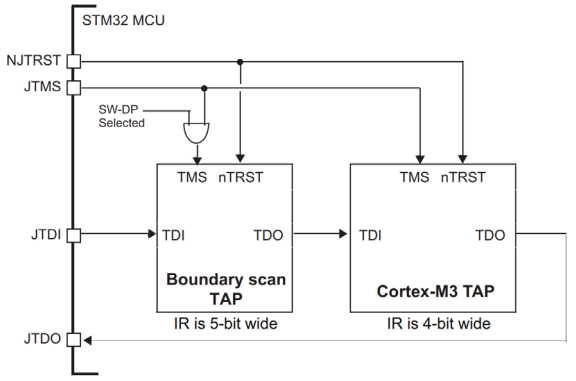


Рис. 1.18. Цепочка JTAG внутри одной микросхемы STM32F1xx[1]

[1] https://www.st.com/content/ccc/resource/technical/document/reference_manual/59/b9/ba/7f/11/af/43/d5/CD00171190.pdf/files/CD00171190.pdf/jcr:content/translations/en.CD00171190.pdf.

В стандарте IEEE 1149.7-2009, появившемся в 2009 году, описывается новый интерфейс 2-wire JTAG (или CompactJTAG, cJTAG), использующий только два сигнала: TCKC (для синхронизации) и TMSC (для передачи данных) вместо четырех. Если говорить упрощенно, в новой версии интерфейса команды «оригинального» JTAG инкапсулируются в пакеты, передаваемые по последовательному интерфейсу cJTAG. Соответственно, требуется как минимум другая реализация TAP-контроллера и машины состояний. Однако на начало 2023 года данная версия интерфейса не получила широкого распространения.

Надеюсь, после этого небольшого ликбеза вам стало понятно, что JTAG – это не просто отладочный интерфейс, а мощный механизм отладки и тестирования всего цифрового устройства. Конкретные функции, доступные через JTAG-интерфейс, зависят от производителя и модели микроконтроллера. Немного позже мы попробуем с помощью JTAG считать прошивку из внутренней flash-памяти микроконтроллера, но вначале нам стоит рассмотреть реализацию отладочного интерфейса для микроконтроллеров, построенных на базе архитектуры ARM (подробнее о различных архитектурах мы поговорим в главе 2). В большинстве цифровых устройств будут находиться микроконтроллеры именно на базе архитектуры ARM, имеющей собственную реализацию отладочного интерфейса, описанную в ARM Debug Interface Architecture Specification (**https://developer.arm.com/documentation/ihi0074/latest/**).

ARM Debug Interface и интерфейс SWD

В микроконтроллерах на базе архитектуры ARM встроен блок отладки и трассировки, построенный по технологии ARM CoreSight (**https://developer.arm.com/documentation/ddi0314/h/**). Для доступа к этому блоку ARM Limited реализовала собственный последовательный интерфейс отладки и трассировки Serial Wire Debug (SWD). Вместо четырех линий JTAG для работы SWD нужно всего две линии – тактирование SWCLK и передача данных SWDIO. Возможно добавление третьей линии SWO для вывода асинхронных данных от модуля трассировки, к которому, например, может быть подключен вывод функции printf(). В отличие от интерфейса JTAG, SWD не поддерживает периферийное сканирование, т. к. изначально создавался именно для отладки и трассировки микропроцессорных ядер. Как и JTAG, интерфейс SWD (правда, только в версии SWDv2) умеет работать в режиме multi-drop (также встречается написание multidrop), в котором несколько устройств подключаются к шине параллельно. Однако этот режим поддерживается лишь в некоторых семействах микроконтроллеров, например в RP2040, стоящем в хорошо известной плате Raspberry Pi Pico. Пример подключения двух микроконтроллеров с помощью multi-drop SWD показан на рис. 1.19.

Спецификация ARM Debug Interface (ADI, **https://developer.arm.com/documentation/ihi0074/latest/**) допускает отладку ядер архитектуры ARM и поддержку периферийного сканирования также и через JTAG-интерфейс, если он реализован в микроконтроллере. Реализация ADI носит название Debug Access Port (DAP), структурная схема которого показана на рис. 1.20.

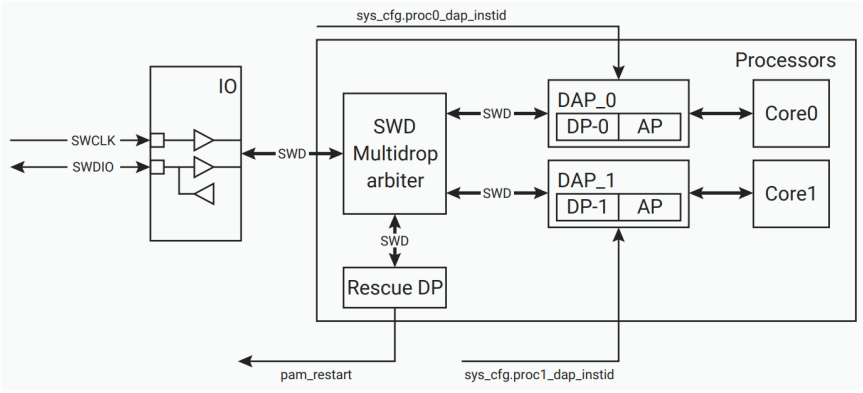


Рис. 1.19. Подключение SWD в режиме multi-drop[1]

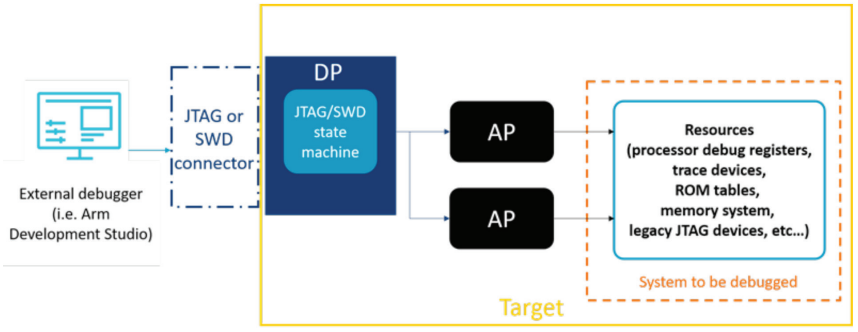


Рис. 1.20. Структура ARM Debug Access Port[2]

DAP состоит из одного или нескольких Access Port (AP) – блоков, позволяющих передавать информацию между отлаживаемым ресурсом (например, ядром) и портом Debug Port (DP), к которому подключается отладчик. Реализация подключения AP к отлаживаемым ресурсам отличается в зависимости от типа ресурса. Стандартным типом подключения является memory-mapping (MEM-AP), в котором необходимые отладочные регистры проецируются на системную память. Также существует JTAG-AP для подключения legacy JTAG-ядра. Приведем примеры некоторых ресурсов:

- отладочные регистры ядра микроконтроллера;
- регистры модуля трассировки;
- таблица отладочных компонентов ROM (хранит ссылки на все отладочные компоненты, подробно мы рассмотрим ее далее);
- подсистема памяти;
- legacy-ядро JTAG.

Существует несколько версий ADI, отличающихся в том числе возможными способами подключения отладочных интерфейсов. Начиная с версии 5.2 поддерживаются три типа подключения к DP:

[1] https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf.
[2] https://developer.arm.com/documentation/102585/0100/.

- JTAG Debug Port (JTAG-DP) – legacy-режим, позволяющий считывать и записывать регистры отладочного блока для функции отладки и периферийного сканирования;
- Serial Wire Debug Port (SW-DP) – реализация пакетного SWD-протокола, также позволяющего считывать и записывать регистры отладочного блока для реализации функций отладки и трассировки;
- Serial Wire / JTAG Debug Port (SWJ-DP) – интерфейс с возможностью динамического переключения между JTAG и SWD. Посредством него можно, например, производить отладку ядра микроконтроллера по интерфейсу SWD и при этом подключиться через цепочку JTAG к другому устройству для периферийного сканирования или отладки.

DP не только поддерживает определенный физический способ подключения отладчика, но также выполняет следующие функции:

- хранит ID-код DP в регистре DPIDR;
- поддерживает необходимый транспорт между DP и AP для разных типов DP и AP;
- реализует автомат (машину) состояний JTAG/SWD.

В зависимости от модели микроконтроллера в нем могут быть реализованы разные DP и AP. Например, на микроконтроллере базового уровня STM32F0xx на ядре ARM Cortex-M0 поддерживается только SW-DP:

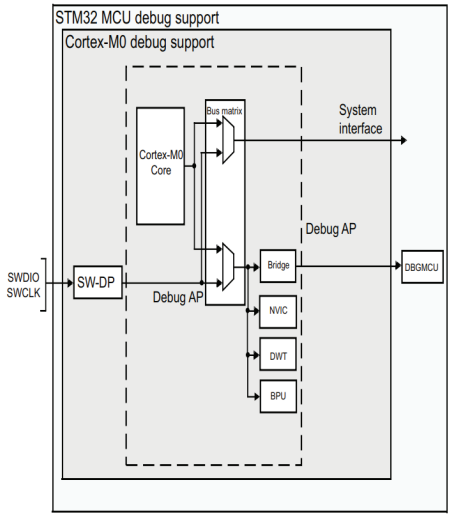


Рис. 1.21. Пример реализации ARM SW-DP[1]

То есть это семейство микроконтроллеров не поддерживает JTAG и не предоставляет возможностей по периферийному сканированию. В данном семействе микроконтроллеров также используется один AP с опционально добавленной таблицей отладочных компонентов ROM:

[1] https://www.st.com/resource/en/reference_manual/dm00031936-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

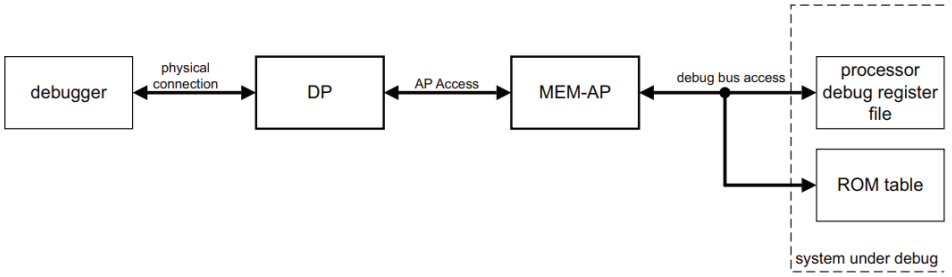


Рис. 1.22. Пример реализации ADI с ARM SW-DP[1]

Если мы посмотрим документацию на более старшее семейство микроконтроллеров того же производителя (STM32F1xx), построенное на базе продвинутого ядра ARM Cortex-M3, то увидим поддержку JTAG и SWD через комбинированный SWJ-DP:

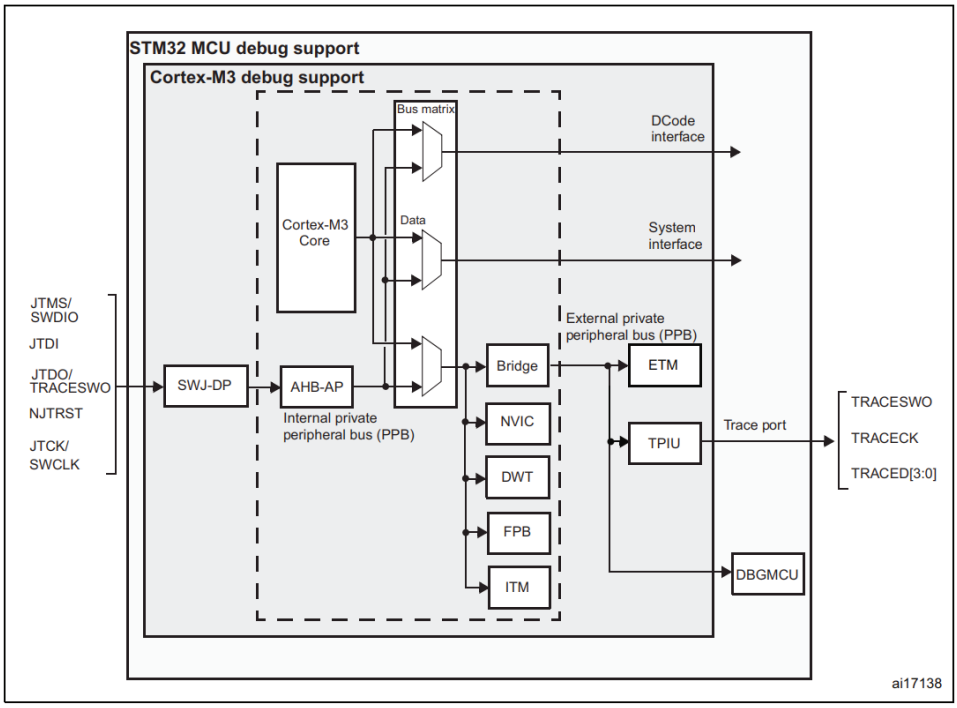


Рис. 1.23. Пример реализации ARM SWJ-DP[2]

Соответственно, AP для реализации нескольких интерфейсов также потребуется несколько:

[1] https://developer.arm.com/documentation/ihi0074/latest/.

[2] https://www.st.com/content/ccc/resource/technical/document/reference_manual/59/b9/ba/7f/11/af/43/d5/CD00171190.pdf/files/CD00171190.pdf/jcr:content/translations/en.CD00171190.pdf.

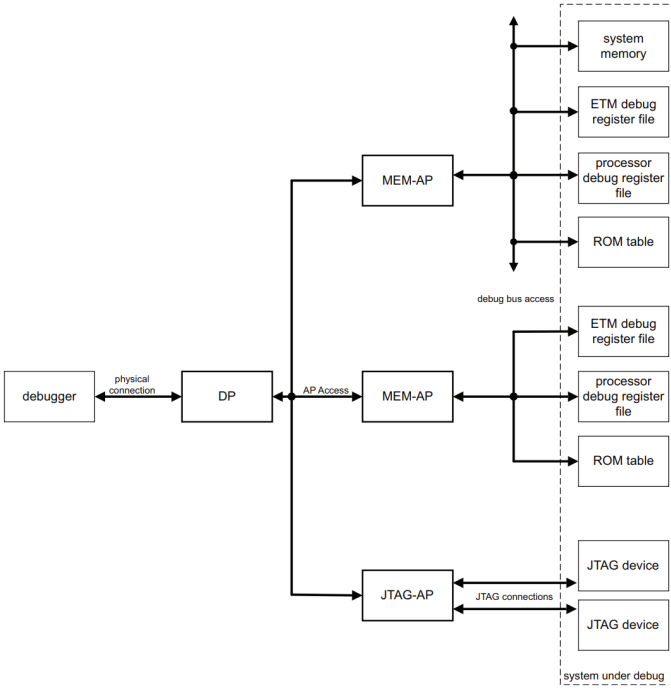


Рис. 1.24. Пример реализации ADI с ARM SWJ-DP[1]

К этому моменту мы разобрались с разными вариантами AP и DP, и для базового понимания принципа работы ADI нам осталось только разобрать таблицу отладочных компонентов ROM, на которую мы уже несколько раз ссылались в этом разделе, и структуру транзакций SWD. SWD – пакетный интерфейс, разберем структуру пакетов на примере транзакции чтения, диаграмма которой показана на рисунке.

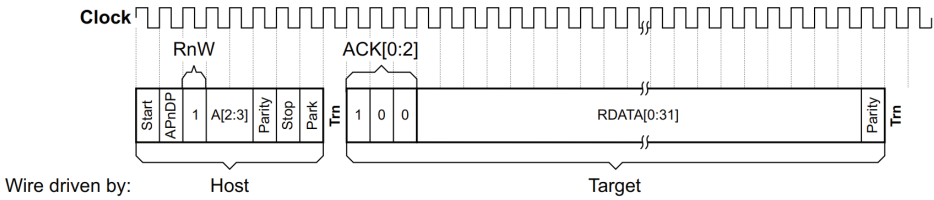


Рис. 1.25. Структура пакета SWD[2]

Каждая успешная транзакция протокола SWD состоит из нескольких частей:

- 8-битный запрос от хоста (пакет header);
- 3-битный пакет ACK хосту;
- опциональный пакет данных длиной 32 бита + 1 бит контроля четности (зависит от параметров, переданных в пакете header).

[1] https://developer.arm.com/documentation/ihi0074/latest/.

[2] https://developer.arm.com/documentation/ihi0074/latest/.

Рассмотрим назначение битов, начнем с пакета header:

- Start (0b1) предназначен для перевода линии в режим ожидания (idle);
- APnDP определяет регистр в зависимости от порта (AP или DP) назначения пакета;
- RnW – бит чтения или записи, определяющий направление передачи данных;
- A[2:3] – образуют 4 варианта адреса для регистра address в DP или AP;
- Parity – бит четности для контроля целостности передачи;
- Stop – бит окончания передачи;
- Park – бит, устанавливаемый хостом в 0b1 (линия подтягивается к уровню логической единицы) для корректного считывания ответной стороной.

После получения пакета header ответное устройство с задержкой в один тактовый сигнал отвечает 3-битовым значением ACK, соответствующим статусу обработки пакета:

- 0b100 – OK (успешно обработанный пакет);
- 0b010 – WAIT (отправляется ответной стороной, если она не может выполнить запрашиваемую операцию чтения или записи);
- 0b001 – FAULT (некорректный запрос, например попытка записи read-only регистра).

В случае получения ответа OK ответная сторона начинает передачу 32 бит запрашиваемых данных и 1 бита четности для контроля целостности переданных данных. Аналогично интерфейсу JTAG, осуществляя чтение или запись в отладочные регистры, соответствующие различным ресурсам, реализуются функции отладки и трассировки.

Для иллюстрации работы интерфейса SWD рассмотрим чтение регистра DPIDR, расположенного в DP по адресу 0x0. Сформируем пакет header в соответствии с ранее описанным форматом:

- Start – 0b1;
- APnDP – 0b0;
- RnW – 0b1;
- A[2:3] – 0b00;
- Parity – 0b1;
- Stop – 0b0;
- Park – 0b1.

Отправив сформированный пакет header, сначала получаем ACK OK, затем значение DPIDR = 0x0bf11477. Диаграмма соответствующих сигналов SWD представлена на рисунке.

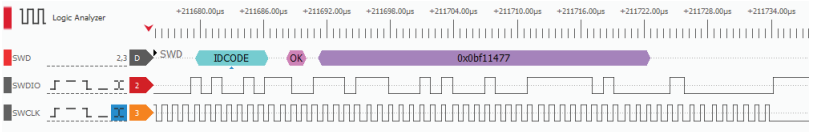


Рис. 1.26. Пример диаграммы сигналов SWD при обработке команды IDCODE

Напоследок разберемся с таблицей ROM. Она хранит в себе указатели на все отладочные компоненты CoreSight (**https://developer.arm.com/Architectures/CoreSight%20Architecture**), доступные в системе. Если в отлаживаемой системе существует только один дебаг-компонент, таблица ROM может отсутствовать, однако в большинстве систем она есть. Таблица ROM подключается к шине MEM-AP, т. е. отображается на участок системной памяти и имеет размер 4 Кбайта. В документации на ядро микроконтроллера содержится информация о записях таблицы ROM конкретного микроконтроллера, например для ранее рассмотренного микроконтроллера STM32F0xx на ядре ARM Cortex-M0, имеющего архитектуру ARMv6-M, таблица имеет вид:

Table C1-4 ARMv6-M DAP accessible ROM table

Offset	Value	Name	Description
0x000	0xFFF0F003	SCS	Points to the SCS at 0xE000E000.
0x004	0xFFF02002 or 0xFFF02003	ROMDWT	Points to the DWT at 0xE0001000. Bit [0] is set to 1 if a DWT is fitted.
0x008	0xFFF03002 or 0xFFF03003	ROMBPU	Points to the BPU at 0xE0002000. Bit [0] is set to 1 if a BPU is fitted.
0x00C	0x00000000	End	End-of-table marker. It is IMPLEMENTATION DEFINED whether the table is extended with pointers to other system debug resources. The table entries must terminate with 0x00000000.
0x010 to 0xFFC	If unused, RAZ		For CoreSight compliance requirements, see Appendix A *ARMv6-M CoreSight Infrastructure IDs*.

Рис. 1.27. Пример таблицы ROM[1]

Каждая запись в таблице занимает 4 байта и имеет следующую структуру:

Table C1-3 ROM table entry format

Bits	Name	Description	
[31:12]	Address offset	Signed base address offset of the component relative to the ROM base address.	
[11:2]	Reserved	UNK/SBZP	
[1]	Format	**0** **1**	8-bit format, not used by ARMv6-M. 32-bit format.
[0]	Entry present	**0** **1**	when bits [31:1] are not all zero, null entry, ignore this table entry. valid table entry.

Рис. 1.28. Структура записи таблицы ROM[2]

Биты [31:12] записи таблицы ROM содержат в себе смещение отладочного компонента в физической памяти микроконтроллера относительно адреса начала ROM таблицы. Рассмотрим, как вычисляется адрес компонента на

[1] https://developer.arm.com/documentation/ddi0419/latest/.
[2] https://developer.arm.com/documentation/ddi0419/latest/.

примере System Control Space (SCS), необходимого для управления и конфигурирования микроконтроллера. Значение SCS offset = 0xFFF0F000, дефолтный адрес расположения таблицы ROM в памяти для ARMv6-M = 0xE00FF000, следовательно, адрес SCS будет равен 0xFFF0F000 + 0xE00FF000 = 0x1E000E000. Так как все адреса имеют размер 32 бита, отбрасываем старший бит и получаем значение 0xE000E000, совпадающее с описанием в таблице выше. При выполнении подобных вычислений стоит учитывать, что значения адресов отрицательные и хранятся в формате дополнительного кода (**https://ru.wikipedia.org/wiki/Дополнительный_код**).

Несмотря на то что дефолтный адрес таблицы ROM написан в документации, в конкретном семействе микроконтроллеров он может быть изменен, поэтому определим его так же, как это делает отладчик. Мы уже знаем, что указатель на таблицу ROM хранится в AP. Посмотрев документацию на CoreSight, мы видим, что в регистре по смещению 0xF8 хранится ссылка на таблицу ROM (Debug Base ROM Pointer):

Table 2-28 APB-AP registers

Offset	Type	Width	Reset value	Name
0x00	R/W	32	0x00000002	Control/Status Word, CSW
0x04	R/W	32	0x00000000	Transfer Address, TAR
0x08	-	-	-	Reserved SBZ
0x0C	R/W	32	-	Data Read/Write, DRW
0x10	R/W	32	-	Banked Data 0, BD0
0x14	R/W	32	-	Banked Data 1, BD1
0x18	R/W	32	-	Banked Data 2, BD2
0x1C	R/W	32	-	Banked Data 3, BD3
0x20-0xF4	-	-	-	Reserved SBZ
0xF8	RO	32	0x80000000	Debug ROM Address, ROM
0xFC	RO	32	0x14770002	Identification Register, IDR

Рис. 1.29. Пример регистров AP[1]

Также интерес представляет регистр IDR по адресу 0xFC, в нем хранится ID AP, по которому можно восстановить архитектуру микроконтроллера.

Таблица ROM содержит список компонентов, подключенных к DP или MEM-AP, позволяя отладчику составить список доступных CoreSight компонентов для отладки. Во время перечисления записей таблицы ROM отладчик смотрит бит Entry Present в записи ROM и проходит всю таблицу, пока не встретит запись, состоящую из всех 0, означающую конец таблицы. Каждая таблица ROM должна включать стандартизованный набор «Component and Peripheral ID Registers», начинающийся со смещения 0xFD0 в таблице.

[1] https://developer.arm.com/documentation/ddi0314/h/.

Offset	Type	Bits	Name	Function
0xFDC	-	[7:0]	Peripheral ID7	Reserved SBZ. Read as 0x00.
0xFD8	-	[7:0]	Peripheral ID6	Reserved SBZ. Read as 0x00.
0xFD4	-	[7:0]	Peripheral ID5	Reserved SBZ. Read as 0x00.
0xFD0	RO	[7:4]	Peripheral ID4	4KB count, set to 0x0.
		[3:0]		JEP106 continuation code, implementation defined.
0xFEC	RO	[7:4]	Peripheral ID3	RevAnd, at top level, implementation defined.
		[3:0]		Customer Modified, implementation defined.
0xFE8	RO	[7:4]	Peripheral ID2	Revision number of Peripheral, implementation defined.
		[3]		1 = indicates that a JEDEC assigned value is used. 0 ⊨ indicates that a JEDEC assigned value is not used.
		[2:0]		JEP106 Identity Code [6:4], implementation defined.
0xFE4	RO	[7:4]	Peripheral ID1	JEP106 Identity Code [3:0], implementation defined.
		[3:0]		PartNumber1, implementation defined.
0xFE0	RO	[7:0]	Peripheral ID0	PartNumber0, implementation defined.
0xFF0	RO	[7:0]	Component ID0	Preamble. Set to 0x0D.
0xFF4	RO	[7:0]	Component ID1	Preamble. Set to 0x10.
0xFF8	RO	[7:0]	Component ID2	Preamble. Set to 0x05.
0xFFC	RO	[7:0]	Component ID3	Preamble. Set to 0xB1.

Рис. 1.30. Стандартные записи таблицы ROM[1]

В них нас также интересует регистр Peripheral ID, хранящий идентификаторы компонентов. Помимо списка компонентов, таблица ROM может включать (и почти всегда включает) в себя ссылки на таблицы ROM более низкого уровня, описывающие конкретные компоненты. Для определения типа записи используется 4 бита в регистре Component ID1:

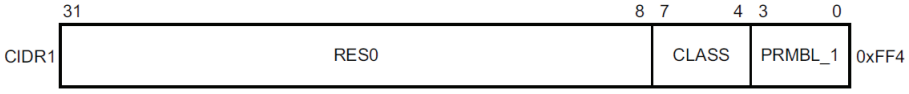


Рис. 1.31. Структура регистра Component ID[2]

Если значение CLASS = 0x1, значит, запись ROM ссылается на ROM-таблицу более низкого уровня. Значение CLASS = 0x9 говорит о том, что запись соответствует описанию компонента CoreSight. Таким образом, возможна многоуровневая иерархия таблиц ROM, пример которой показан на схеме:

[1] https://developer.arm.com/documentation/ddi0314/h/.
[2] https://developer.arm.com/documentation/ddi0314/h/.

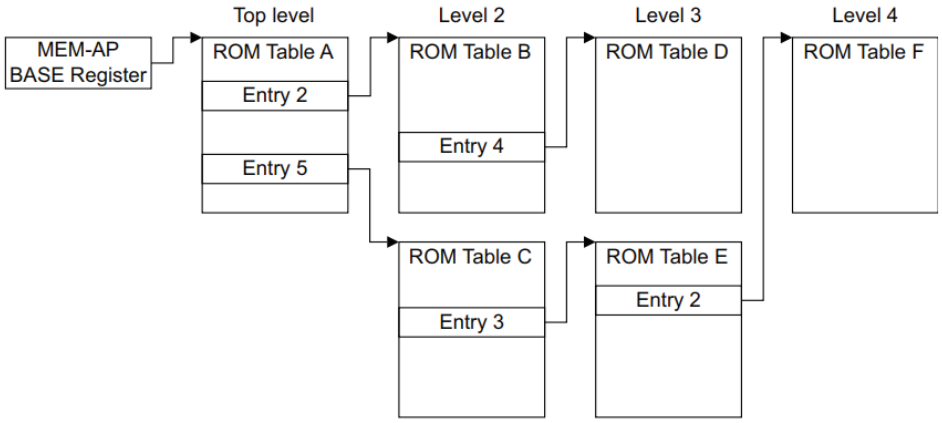


Рис. 1.32. Иерархия таблиц ROM[1]

Обратившись к документации на ядро ARM Cortex-M0, мы найдем структуру таблиц ROM для микроконтроллера STM32F0:

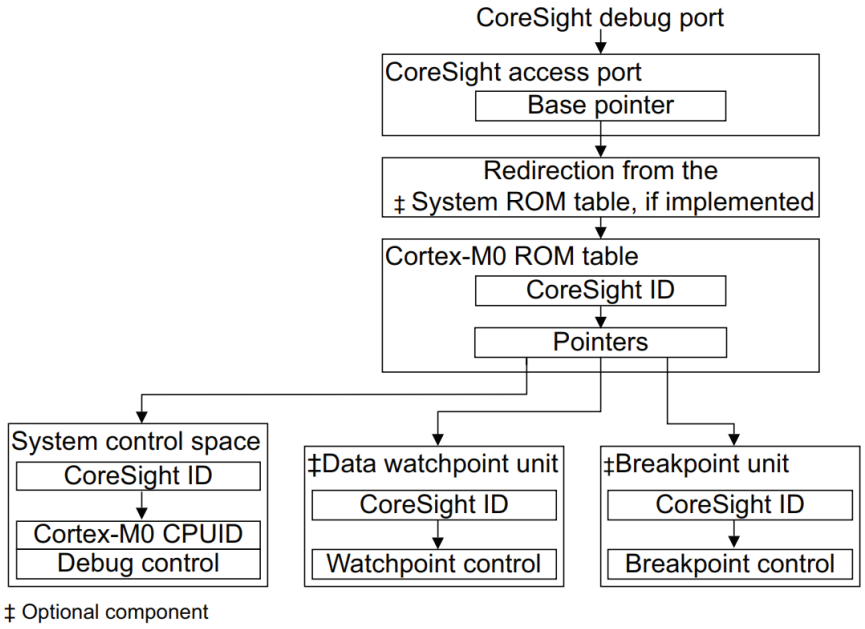


Рис. 1.33. Пример иерархии таблиц ROM для микроконтроллеров семейства STM32F0[2]

Надеюсь, у меня получилось объяснить концепцию ARM Debug Interface и интерфейса SWD. Оригинальная документация на ADI, CoreSight, ядро и архитектуру микроконтроллера, даже самого базового, занимает не одну тысячу страниц. Если вам понадобится глубокое понимание работы отладочных ин-

[1] https://developer.arm.com/documentation/ihi0074/latest/.
[2] https://developer.arm.com/documentation/ddi0432/c.

терфейсов (не важно, ADI, JTAG или какого-то еще), теперь вы должны понимать, где и какую информацию можно найти. Хорошая новость для начинающих исследователей цифровых устройств заключается в том, что отладчик и его ПО скрывают большинство особенностей реализации отладочного интерфейса. Получается, что можно работать с отладочными интерфейсами, не обладая пониманием принципов их работы, однако теперь вы сразу поймете, что означают и делают команды, например dap info, в консоли управления отладчика OpenOCD.

Считывание прошивки с помощью OpenOCD и SWD

Разобравшись с основами JTAG и SWD, попробуем применить эти знания на практике, считав внутреннюю flash-память микроконтроллера с помощью универсального ПО для отладки OpenOCD (**https://openocd.org/**). OpenOCD поддерживает большинство микроконтроллеров, которые могут встретиться в цифровых устройствах, т. к. оно умеет работать с JTAG- и SWD-интерфейсами, предоставляя пользователю удобный командный интерфейс к высокоуровневым примитивам в виде TAP/DAP-контроллеров, интерфейса памяти и т. п. В то же время OpenOCD позволяет опускаться на самый низкий уровень, например установки значений регистра данных и команд в TAP-контроллере.

Это open source проект, поддерживающий скриптовый язык tcl для описания параметров подключения и последовательности действий при работе с микроконтроллерами. Он позволяет описывать устройства внутри микроконтроллера и добавлять поддержку моделей микроконтроллеров или отладчиков, которых еще нет в списке.

Для примера работы с OpenOCD возьмем устройство-конвертер интерфейсов с микроконтроллером GigaDevices GD32E230 на базе архитектуры ARM Cortex-M23 (задуманного быть заменой микроконтроллеров серии ST32F0xx на базе архитектуры ARM Cortex-M0), имеющего интерфейс SWD. На плате устройства присутствуют контакты для подключения к интерфейсу SWD:

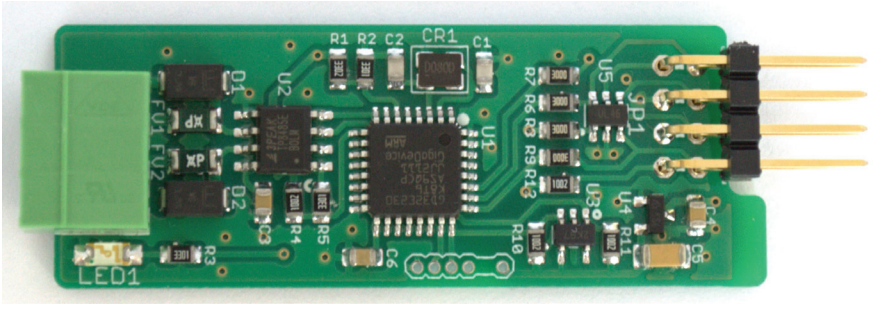


Рис. 1.34. Отладочные контакты интерфейса SWD

В данном случае распиновку контактов легко восстановить, получаем пять сигналов: GND, SWDIO, SWCLK, RESET, VCC. Для удобства работы запаиваем контактную гребенку с шагом 1,27 мм:

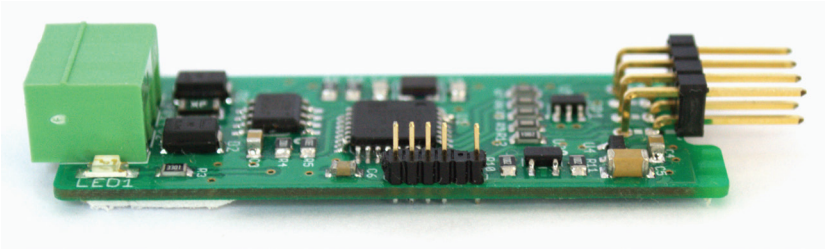


Рис. 1.35. Отладочные контакты с напаянной контактной гребенкой

И подключаем к отладчику Segger J-link:

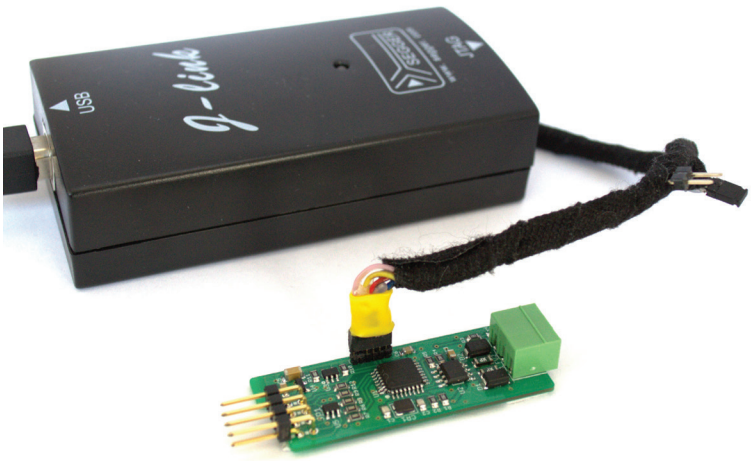


Рис. 1.36. Устройство, подключенное к отладчику J-link

Для работы OpenOCD надо указать конфигурационный файл отладчика (находится в папке «OpenOCD\share\openocd\scripts\interface» по относительному пути) и конфигурационный файл микроконтроллера (из папки «OpenOCD\share\openocd\scripts\target»). Запускаем OpenOCD, указывая конфигурационные файлы ключом «-f» (от слова find), а ключом «-c» вводим команду выбрать в качестве транспорта интерфейс SWD:

```
openocd.exe -f interface/jlink.cfg -c "transport select swd" -f target/
gd32e23x.cfg
```

Получаем следующий вывод:

```
Open On-Chip Debugger 0.12.0-rc2 (2022-10-26-14:02)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
swd
Info : Listening on port 6666 for tcl connections
```

```
Info : Listening on port 4444 for telnet connections
Info : J-Link V9 compiled Sep 1 2016 18:29:50
Info : Hardware version: 9.60
Info : VTarget = 3.306 V
Info : clock speed 1000 kHz
Info : SWD DPIDR 0x0bf11477
Info : [gd32e23x.cpu] Cortex-M23 r1p0 processor detected
Info : [gd32e23x.cpu] target has 4 breakpoints, 2 watchpoints
Info : starting gdb server for gd32e23x.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

OpenOCD корректно считал уже знакомый нам регистр DPIDR=0x0bf11477 из DP (и это значение соответствует разработчику DP – ARM limited) и корректно определил процессор на базе архитектуры Cortex-M23. Также видим, что для управления OpenOCD по умолчанию открыт telnet порт 4444, подключаемся к нему (например, с помощью putty или telnet) и видим строку «Open On-Chip Debugger» и символ начала ввода команды «>»:

```
Open On-Chip Debugger
>
```

Проверим, какие цели видит OpenOCD:

```
> targets
    TargetName         Type       Endian  TapName            State
 -- ------------------ --------- ------- ------------------ ------------
 0* gd32e23x.cpu       cortex_m  little  gd32e23x.cpu       running
```

Ядро найдено корректно и находится в состоянии running, т. е. исполняет код прошивки. Так как у нас загружен конфиг микроконтроллера, в котором содержится описание внутренней flash-памяти, выведем, какую память видит OpenOCD:

```
> flash list
       {name gd32e23x.flash driver stm32f1x base 134217728 size 0 bus_width
0 chip_width 0 target gd32e23x.cpu}
```

Виден один блок памяти, все корректно (пусть вас не смущает название драйвера для STM32F1x, мы говорили, что микроконтроллер специально сделан, чтобы максимально просто заменять представителей семейства STM32Fx). Считываем ее содержимое в файл следующей командой:

```
> flash read_bank 0 bank0.bin
device id = 0x19090410
flash size = 64 KiB
wrote 65536 bytes to file bank0.bin from flash bank 0 at offset 0x00000000
in 1.245701s (51.377 KiB/s)
```

Отлично, мы считали содержимое внутренней flash-памяти микроконтроллера, в которой должна быть прошивка. Конечно, у нас это получилось, т. к. не была активирована защита от считывания (readout protection). Далее мы рассмотрим, какие есть способы предотвратить считывание прошивки через отладочные интерфейсы. Но если вы хотите понять, откуда OpenOCD знает, где и как хранится и интерпретируется информация, получаемая через отладочный интерфейс, у меня для вас есть подробный пример. Если желания вспоминать устройство ARM Debug Interface и погружаться в дебри конфигурации OpenOCD у вас нет, можете переходить к следующему разделу.

Представим, что мы не знаем ничего о модели микроконтроллера, кроме того что для подключения используется интерфейс SWD. Напишем самый простой конфигурационный файл ocd.cfg со следующим содержимым (символ # комментирует строку):

```
#указываем, что в качестве отладчика мы используем J-link
source [find interface/jlink.cfg]

#подключаемся через swd
transport select swd

#частота работы интерфейса 1 МГц
adapter speed 1000

#конфиг сигнала reset (только ресет всего чипа, включить до подключения SWD)
reset_config srst_only srst_nogate connect_assert_srst

# объявляем новый DAP с именем chip и ролью CPU
swd newdap chip cpu -enable

# создаем объявленный DAP
dap create chip.dap -chain-position chip.cpu
```

В нем мы указали, что будем работать с SWD DAP портом через интерфейс SWD отладчика J-link. Запускаем OpenOCD, указывая созданный конфиг «openocd.exe -f ocd.cfg», и получаем следующий вывод:

```
Open On-Chip Debugger 0.12.0-rc2 (2022-10-26-14:02)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : J-Link V9 compiled Sep 1 2016 18:29:50
Info : Hardware version: 9.60
Info : VTarget = 3.308 V
Info : clock speed 1000 kHz
```

```
Info : SWD DPIDR 0x0bf11477
Warn : gdb services need one or more targets defined
```

Видим предупреждение о том, что OpenOCD не нашел в конфиге ни одного отладочного целевого устройства (мы их и не указывали), но корректно считал уже знакомый нам регистр DPIDR=0x0bf11477 из DP (и это значение, как мы уже знаем, соответствует разработчику DP – ARM limited)! Подключаемся к telnet-порту 4444 и видим строку «Open On-Chip Debugger» и символ начала ввода команды «>»:

```
Open On-Chip Debugger
>
```

Проверим, какие цели видит OpenOCD:

```
> targets
  TargetName     Type    Endian TapName      State
```

Никакие, логично, ведь мы их не задавали, и OpenOCD выводил предупреждение при запуске. Давайте попробуем получить значение DPIDR «руками», не зря же вы читали предыдущий раздел про ADI. Сначала вводим команду, показывающую имена созданных DAP, а потом считываем 0-регистр DP в DAP (ведь мы помним, что значение DPIDR хранится в нем):

```
> dap names
chip.dap

> chip.dap dpreg 0
0x0bf11477
```

Отлично, получили корректное значение, и теперь у нас нет вопросов, откуда OpenOCD его считывает. Следующим шагом попробуем узнать адрес таблицы ROM, для этого в OpenOCD есть команда «baseaddr», считывающая значение регистра BASE из выбранного (по умолчанию нулевого) MEM-AP (подробнее на рисунке «Figure C2-1 MEM-AP connecting the DP to debug components» из документации ARM Debug Interface):

```
> chip.dap baseaddr
0xe00ff003
```

Добавим в наш конфиг ocd.cfg строчку, создающую MEM_AP в качестве отладочного целевого устройства:

```
target create chip.ahb mem_ap -dap chip.dap -ap-num 0
```

Перезапускаем OpenOCD, видим изменения в выводе:

```
Open On-Chip Debugger 0.12.0-rc2 (2022-10-26-14:02)
…
```

```
Info : SWD DPIDR 0x0bf11477
Info : gdb port disabled
```

Предупреждение о том, что нет ни одной цели, пропало, но появилось сообщение, что порт для подключения отладчика закрыт. Логично, ведь нельзя отлаживать MEM-AP. Снова подключаемся к telnet-порту 4444 и проверяем список целей:

```
> targets
    TargetName         Type       Endian  TapName            State
 -- ------------------ ---------- ------- ------------------ ------------
 0* chip.ahb           mem_ap     little  chip.cpu           running
```

Отлично, отладчик видит MEM-AP, мы уже знаем адрес расположения таблицы ROM (0xe00ff000), давайте ее считаем:

```
> mdw 0xe00ff000 1024
0xe00ff000: fff0f003 fff02003 fff03003 fff01002 fff41002 fff42002 fff43002 fff44002
0xe00ff020: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
...
0xe00fffa0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0xe00fffc0: 00000000 00000000 00000000 00000001 00000004 00000000 00000000 00000000
0xe00fffe0: 000000cb 000000b4 0000000b 00000000 0000000d 00000010 00000005 000000b1
```

Успех! Корректными являются только первые три записи (т. к. в них установлен бит present). Поле PartNumber = 0x4cb (значения выделены). Можно попробовать поискать его в интернете и сделать предположение о модели процессора. Но мы пойдем более правильным путем, узнаем адрес первого ресурса (ранее в документации мы видели, что для Cortex-M там находится System Control Space (SCS)): 0xE00FF000 + 0xFFF0F000 = 0x1E000E000, т. е. 0xE000E000. Считаем 4 КБ с этого адреса:

```
> mdw 0xe000e000 1024
0xe000e000: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0xe000e020: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
...
0xe000efa0: 00000000 00000000 00000000 00000000 00000000 00000000 0000000f 47702a04
0xe000efc0: 00000000 00000000 00000000 00000000 00000004 00000000 00000000 00000000
0xe000efe0: 00000020 000000bd 0000000b 00000000 0000000d 00000090 00000005 000000b1
```

Тип записи CLASS = 0x9 в регистре Component ID1 говорит нам о том, что это отладочный компонент CoreSight. Device Architecture Register (DEVARCH) по смещению 0xFBC = 0x47702a04, что соответствует архитектуре ARMv8-M, на которой в основном построены ядра ARM Cortex-M23 и Cortex-M33. Мы достоверно определили архитектуру ядра и, возможно, его тип. Чтобы каждый раз не проделывать эти операции вручную, в OpenOCD есть команда считывания и интерпретации таблицы ROM:

```
> chip.dap info
AP # 0x0
        AP ID register 0x04770025
        Type is MEM-AP AHB5
MEM-AP BASE 0xe00ff003
        Valid ROM table present
        Component base address 0xe00ff000
        Peripheral ID 0x04000bb4cb
        Designer is 0x23b, ARM Ltd
        Part is 0x4cb, Unrecognized
        Component class is 0x1, ROM table
        MEMTYPE system memory present on bus
    ROMTABLE[0x0] = 0xfff0f003
        Component base address 0xe000e000
        Peripheral ID 0x04000bbd20
        Designer is 0x23b, ARM Ltd
        Part is 0xd20, Unrecognized
        Component class is 0x9, CoreSight component
        Type is 0x00, Miscellaneous, other
           Dev Arch is 0x47702a04, ARM Ltd "Processor debug architecture
(ARMv8-M)" rev.0
    ROMTABLE[0x4] = 0xfff02003
        Component base address 0xe0001000
        Peripheral ID 0x04000bbd20
        Designer is 0x23b, ARM Ltd
        Part is 0xd20, Unrecognized
        Component class is 0x9, CoreSight component
        Type is 0x00, Miscellaneous, other
           Dev Arch is 0x47701a02, ARM Ltd "DWT architecture" rev.0
    ROMTABLE[0x8] = 0xfff03003
        Component base address 0xe0002000
        Peripheral ID 0x04000bbd20
        Designer is 0x23b, ARM Ltd
        Part is 0xd20, Unrecognized
        Component class is 0x9, CoreSight component
        Type is 0x00, Miscellaneous, other
           Dev Arch is 0x47701a03, ARM Ltd "Flash Patch and Breakpoint unit
(FPB) architecture" rev.0
    ROMTABLE[0xc] = 0xfff01002
        Component not present
    ROMTABLE[0x10] = 0xfff41002
        Component not present
```

```
    ROMTABLE[0x14] = 0xfff42002
        Component not present
    ROMTABLE[0x18] = 0xfff43002
        Component not present
    ROMTABLE[0x1c] = 0xfff44002
        Component not present
    ROMTABLE[0x20] = 0x00000000
        End of ROM table
```

Данные аналогичны тем, что были получены нами вручную. Дописываем строчку с новым отладочным целевым устройством в конфиг (процессор с ядром семейства Cortex-M):

```
target create chip.cpu cortex_m -dap chip.dap
```

Перезапускаем OpenOCD и снова видим изменения в выводе:

```
Open On-Chip Debugger 0.12.0-rc2 (2022-10-26-14:02)
…
Info : SWD DPIDR 0x0bf11477
Info : [chip.cpu] Cortex-M23 r1p0 processor detected
Info : [chip.cpu] target has 4 breakpoints, 2 watchpoints
Info : gdb port disabled
Info : starting gdb server for chip.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

OpenOCD определил модель ядра (Cortex-M23), что опять подтверждает правильность нашего парсинга таблицы ROM вручную. Проверяем цели:

```
> targets
   TargetName            Type       Endian  TapName            State
   -- ------------------ ---------- ------- ------------------ ------------
   0  chip.ahb           mem_ap     little  chip.cpu           running
  1*  chip.cpu           cortex_m   little  chip.cpu           running
```

Ищем в интернете, какие микроконтроллеры основаны на ядре Cortex-M23, находим следующий список в Википедии (**https://en.wikipedia.org/wiki/ARM_Cortex-M#Cortex-M23**):

```
GigaDevice GD32E230;
Microchip SAM L10, L11;
Nuvoton M2351;
Renesas S1JA, RA2A1, RA2L1, RA2E1, RA2E2.
```

На первом месте – как раз семейство исследуемого микроконтроллера. Находим в документации на GD32E230xx расположение flash-памяти в адресном пространстве:

Code		0x1FFF F810 - 0x1FFF FFFF	Reserved
		0x1FFF F800 - 0x1FFF F80F	Option bytes
		0x1FFF EC00 - 0x1FFF F7FF	System memory
		0x0801 0000 - 0x1FFF EBFF	Reserved
		0x0800 0000 - 0x0800 FFFF	Main Flash memory
		0x0001 0000 - 0x07FF FFFF	Reserved

Рис. 1.37. Диапазоны памяти для хранения кода в адресном пространстве микроконтроллеров GD32E230xx[1]

Осталось считать диапазон 0x08000000-0x0800FFFF. Поскольку у нас нет конфигурационного файла с описанием процессора и, следовательно, не загружен драйвер для работы с flash-памятью, попробуем считать диапазон универсальной командой дампа области памяти:

```
> dump_image f.bin 0x08000000 0x10000
dumped 65536 bytes in 1.243405s (51.472 KiB/s)
```

У нас все получилось, данные аналогичны считанным ранее! Данный пример показывает, что даже без конфигурационного файла OpenOCD, но зная внутреннее устройство ADI и CoreSight, можно добиться успеха в считывании прошивки. Даже в этом примере OpenOCD командой dap info существенно упрощает жизнь исследователю. Но далеко не на все случаи есть команды, а главное – чтобы пользоваться ими осознанно, надо понимать, что именно они делают и откуда берут информацию. Если вы прочитали и осознали этот пример, у вас будет гораздо меньше вопросов по тому, как использовать отладочные интерфейсы в исследованиях устройств.

Считывание прошивки с помощью Segger J-Link и JTAG

Подробно разобравшись с OpenOCD и SWD, попробуем схожим образом считать внутреннюю память другого микроконтроллера, используя интерфейс JTAG и, для разнообразия, ПО для отладчика J-Link. При работе с OpenOCD мы использовали этот же отладчик, но устанавливали драйверы на основе libusb, в этом же примере будем использовать оригинальные драйверы Segger вместе с их ПО J-Link Commander для управления отладчиком.

В качестве исследуемого устройства выберем плату высокоскоростного GPS-логгера с микроконтроллером Atmel SAM3S4B, построенного на базе ядра ARM Cortex-M3 и имеющего отладочный порт SWJ-DP. На плате устройства прозвонкой легко обнаружить контактные площадки для отладочного интерфейса JTAG, их расположение показано на рис. 1.38.

Подключившись к ним с помощью J-Link и запустив ПО J-Link Commander, видим сообщение с версией ПО, успешно найденным отладчиком J-Link и символом начала ввода команды «>» с предложением ввести команду connect для подключения к отлаживаемому микроконтроллеру:

[1] https://gd32mcu.com/data/documents/datasheet/GD32E230xx_Datasheet_Rev1.9.pdf.

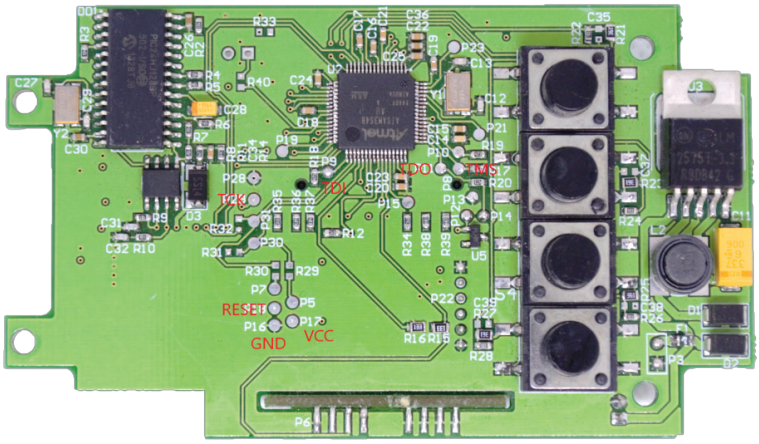


Рис. 1.38. Отладочные контакты интерфейса JTAG

```
SEGGER J-Link Commander V7.82d (Compiled Nov 23 2022 16:10:38)
DLL version V7.82d, compiled Nov 23 2022 16:09:10

Connecting to J-Link via USB...O.K.
Firmware: J-Link V9 compiled Sep 1 2016 18:29:50
Hardware version: V9.60
J-Link uptime (since boot): N/A (Not supported by this model)
S/N: 69657135
License(s): RDI, GDB, FlashDL, FlashBP, JFlash
VTref=3.287V


Type "connect" to establish a target connection, '?' for help
J-Link>
```

Так и поступим, вводим команду `connect`:

```
J-Link>connect
Please specify device / core. <Default>: Unspecified
Type '?' for selection dialog
Device>
```

Видим запрос на ввод модели микроконтроллера, в базе J-Link Commander есть большое количество поддерживаемых моделей, включая Atmel SAM3S4B, поэтому смело указываем его и в следующих запросах выбираем интерфейс JTAG, автоматическое определение положения устройства в JTAG-цепочке и скорость работы интерфейса в 1 МГц:

```
Device>ATSAM3S4B
Please specify target interface:
```

```
 J) JTAG (Default)
 S) SWD
 T) cJTAG
TIF>J
Device position in JTAG chain (IRPre,DRPre) <Default>: -1,-1 => Auto-detect
JTAGConf>
Specify target interface speed [kHz]. <Default>: 4000 kHz
Speed>1000
```

Теперь ПО знает, что мы хотим подключиться к микроконтроллеру Atmel SAM3S4B по интерфейсу JTAG, и после ввода скорости интерфейса видим результат успешной попытки подключения:

```
Device "ATSAM3S4B" selected.

Connecting to target via JTAG
TotalIRLen = 4, IRPrint = 0x01
JTAG chain detection found 1 devices:
 #0 Id: 0x4BA00477, IRLen: 04, CoreSight JTAG-DP
DPv0 detected
CoreSight SoC-400 or earlier
Scanning AP map to find all available APs
AP[1]: Stopped AP scan as end of AP map has been reached
AP[0]: AHB-AP (IDR: 0x24770011)
Iterating through AP map to find AHB-AP to use
AP[0]: Core found
AP[0]: AHB-AP ROM base: 0xE00FF000
CPUID register: 0x412FC230. Implementer code: 0x41 (ARM)
Found Cortex-M3 r2p0, Little endian.
FPUnit: 6 code (BP) slots and 2 literal slots
CoreSight components:
ROMTbl[0] @ E00FF000
[0][0]: E000E000 CID B105E00D PID 002BB000 SCS
[0][1]: E0001000 CID B105E00D PID 002BB002 DWT
[0][2]: E0002000 CID B105E00D PID 002BB003 FPB
[0][3]: E0000000 CID B105E00D PID 002BB001 ITM
[0][4]: E0040000 CID B105900D PID 002BB923 TPIU-Lite
Cortex-M3 identified.
J-Link>
```

Надеюсь, что после прочтения раздела про ARM Debug Interface у вас не вызывает вопросов информация, выданная отладчиком. Пора посмотреть в документации на микроконтроллер Atmel SAM3S4B, где в адресном пространстве находится внутренняя flash-память (и заодно SRAM, адрес которой пригодится нам чуть позже):

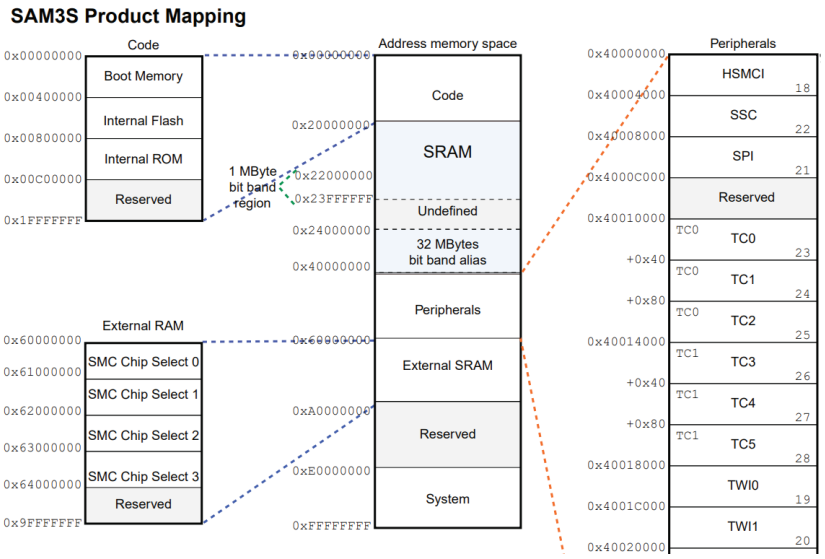


Рис. 1.39. Структура адресного пространства микроконтроллеров семейства SAM3S[1]

Попробуем считать первые 0x100 байт по адресу 0x400000 с помощью команды mem:

```
J-Link>mem 0x00400000 0x100
00400000 = C0 3B 00 20 F9 2A 40 00 F7 2A 40 00 F7 2A 40 00 .;. .*@..*@..*@.
00400010 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 00 00 00 00 .*@..*@..*@.....
00400020 = 00 00 00 00 00 00 00 00 00 00 00 00 F7 2A 40 00 .............*@.
00400030 = F7 2A 40 00 00 00 00 00 F7 2A 40 00 F7 2A 40 00 .*@......*@..*@.
00400040 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 .*@..*@..*@..*@.
00400050 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 .*@..*@..*@..*@.
00400060 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 B5 28 40 00 .*@..*@..*@..(@.
00400070 = C9 28 40 00 F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 .(@..*@..*@..*@.
00400080 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 .*@..*@..*@..*@.
00400090 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 .*@..*@..*@..*@.
004000A0 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 .*@..*@..*@..*@.
004000B0 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 .*@..*@..*@..*@.
004000C0 = F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 F7 2A 40 00 .*@..*@..*@..*@.
004000D0 = 08 B5 05 48 05 4B 19 1A 06 29 00 D8 08 BD 04 4A ...H.K...).....J
004000E0 = 00 2A FB D0 90 47 F9 E7 E8 9B 40 00 EB 9B 40 00 .*...G....@...@.
004000F0 = 00 00 00 00 08 B5 07 48 07 4B 19 1A 8A 10 02 EB .......H.K......
```

И сразу видим, что содержимое действительно представляет собой прошивку, т. к. мы видим таблицу векторов прерываний, характерных для ARM серий M/R, в самом начале:

[1] https://www.keil.com/dd/docs/datashts/atmel/sam3s/sam3s_datasheet.pdf.

- C0 3B 00 20, т. е. значение 0x20003BC0, указывающее на начало стека в SRAM-памяти (что соответствует диапазону SRAM в документации);
- F9 2A 40 00, т. е. значение 0x00402AF9, являющееся Reset Handler и указывающее на начальный код прошивки. Его значение также соответствует диапазону для flash-памяти.

Из документации мы знаем, что для считывания flash-памяти нас интересует диапазон адресов 0x400000-0x800000, однако он слишком большой для большинства внутренних ПЗУ, почти 4 МБ, поэтому опять обратимся к документации:

Table 1-1. Configuration Summary

Device	Flash	SRAM	Timer Counter Channels	GPIOs	UART/ USARTs	ADC	12-bit DAC Output	External Bus Interface	HSMCI	Package
SAM3S4C	256 Kbytes single plane	48 Kbytes	6	79	2/2(1)	15 ch.	2	8-bit data, 4 chip selects, 24-bit address	1 port 4 bits	LQFP100 BGA100
SAM3S4B	256 Kbytes single plane	48 Kbytes	3	47	2/2(1)	10 ch.	2	-	1 port 4 bits	LQFP64 QFN 64
SAM3S4A	256 Kbytes single plane	48 Kbytes	3	34	2/1	8 ch.	-	-	-	LQFP48 QFN 48

Рис. 1.40. Размер памяти микроконтроллеров семейства SAM3S[1]

Видим, что для нашей модели микроконтроллера SAM3S4B объем внутреннего ПЗУ составляет 256 Кб, т. е. 0x40000 байт. Считаем их в файл fw.bin с помощью команды savebin J-Link Commander:

```
J-Link>savebin sam3s_fw.bin 0x400000 0x40000
Opening binary file for writing... [sam3s_fw.bin]
Reading 262144 bytes from addr 0x00400000 into file...O.K.
J-Link>
```

Вот и все, содержимое внутреннего ПЗУ, включающее в себя прошивку, получено. Этот пример снова показывает, что современное ПО для отладки имеет большое количество удобно реализованных функций, позволяющих работать с отладочными интерфейсами без погружения в особенности их реализаций. Конечно, пока производитель устройства не решил позаботиться о его защите и не стал задействовать имеющиеся в микроконтроллерах механизмы защиты от считывания прошивки (readout protection) через отладочные интерфейсы. Поэтому переходим к обзору этих механизмов и известным примерам их преодоления.

Механизмы защиты от считывания

Отладочные интерфейсы удобны при разработке и отладке устройства, но хорошая практика (которой, к счастью для исследователей, не всегда следуют) – их отключать в релизной версии устройства/прошивки. Для отключения некоторых функций отладочных интерфейсов есть настраиваемые биты в энергонезависимой памяти внутри микроконтроллера. Гибкость настройки функций защиты зависит от конкретного ядра и модели микроконтроллера.

[1] https://www.keil.com/dd/docs/datashts/atmel/sam3s/sam3s_datasheet.pdf.

Например, можно отключить возможность доступа к flash-памяти, оставив загрузку из bootloader'a или SRAM. Некоторые модели микроконтроллеров поддерживают полное отключение отладочного интерфейса, в ряде реализаций есть возможность активировать авторизацию (например, пароль доступа, который необходимо ввести для разблокировки функций интерфейса). Точная конфигурация возможных функций защиты может быть найдена в документации на микроконтроллер. Примером является технология Flash Readout Protection (RDP) в микроконтроллерах STM32. Данная технология позволяет защитить содержимое встроенной в микроконтроллер flash-памяти от считывания через отладочный интерфейс.

RDP имеет три уровня защиты (0, 1 и 2):

- RDP level 0: настройка по умолчанию, не предполагающая защиты;
- RDP level 1: отладочный интерфейс активен, доступ к flash-памяти запрещен. Может быть понижен до level 0 (со стиранием содержимого flash-памяти) или повышен до level 2;
- RDP level 2: постоянное отключение отладочного интерфейса без возможности даунгрейда.

Естественно, что если есть механизмы защиты от использования отладочных интерфейсов, то существуют и широко известные уязвимости в этих механизмах защиты. Рассмотрим основные и самые известные из них.

Известные уязвимости отладочных интерфейсов

Первый тип уязвимостей отладочных интерфейсов связан с некорректным конфигурированием устройства в нестандартных режимах работы. Например, на плате устройства могут быть контакты (или, если их нет, выводы микроконтроллера), при замыкании которых микроконтроллер устройства не будет пытаться загружать прошивку из внешней микросхемы памяти, а останется в ROM-mode (т. е. будет исполнять только код из своей ROM-памяти). На рис. 1.41 стрелкой в левом нижнем углу показаны два контакта, переводящих контроллер SSD диска Kingston SKC300S37A в ROM-mode.

Также в него часто можно попасть, помешав устройству загрузиться с внешней микросхемы памяти, испортив (закоротив или отключив) какой-то сигнал ее интерфейса, например CLK. В режиме ROM-mode может быть активен отладочный интерфейс или дополнительный код, позволяющий считывать/записывать содержимое памяти через какой-то интерфейс управления (USB, UART и т. п.).

Второй тип уязвимостей отладочных интерфейсов связан с их некорректной реализацией в микроконтроллере. Одним из самых известных примеров является считывание содержимого внутренней flash-памяти микроконтроллеров STM32F0 с активным режимом RDP level 1 через интерфейс SWD (**https://www.aisec.fraunhofer.de/en/FirmwareProtection.html**). Так как JTAG/SWD и т. п. – это отдельный аппаратный блок внутри микроконтроллера, который имеет машину состояний, существует вероятность перехода между состояниями не в той последовательности, которую предполагал производитель.

Возможно выиграть временную «гонку», когда переключение состояния уже произошло, а доступ к части функций остается в связи с архитектурными особенностями реализации JTAG/SWD или микроконтроллера. Некоторые реализации отладочных интерфейсов могут содержать в себе недокументированные возможности, использование которых позволит получить доступ к прошивке или процессорному ядру в обход механизмов защиты (подробнее можно узнать в документах **https://cyphunk.files.wordpress.com/2010/02/blackbox-jtag-reverse-engineering-tmbinc.pdf** и **https://www.cl.cam.ac.uk/~sps32/Silicon_scan_draft.pdf**). Весьма полезным примером обхода мехнизмов защиты и считывания прошивки за счет возможности работы с регистрами процессора является пример атаки на Nordic Semiconductor nRF51822 (**https://blog.includesecurity.com/2015/11/firmware-dumping-technique-for-an-arm-cortex-m0-soc/**).

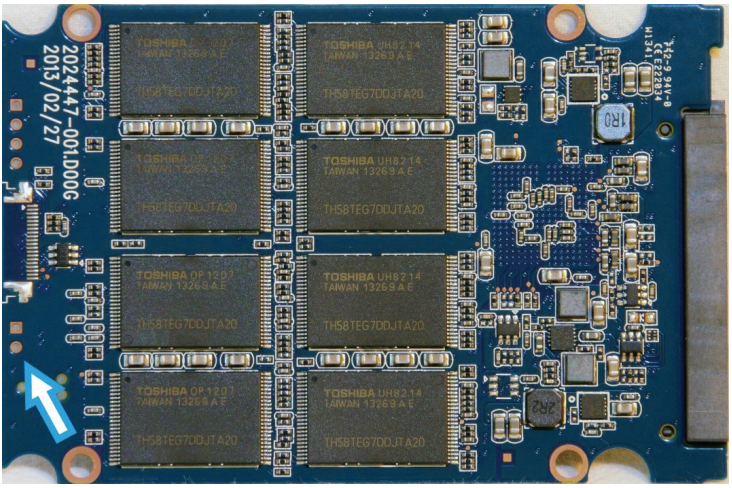


Рис. 1.41. Контакты для активации режима ROM-mode на SSD-диске[1]

В любом случае, заблокированные функции отладочного интерфейса – не всегда непоправимая ситуация. Вполне возможно, что уже существуют успешные атаки для разблокировки части функций, и надо как следует поискать в интернете и адаптировать их для исследуемого микроконтроллера. Отдельным способом обхода защиты отладочных интерфейсов является семейство неинвазивных атак, о которых вы узнаете чуть позже.

Считывание через диагностические интерфейсы

В предыдущей главе мы говорили о широком применении интерфейса UART для вывода лога загрузки устройства или организации CLI. В большинстве

[1] https://greentechreviews.ru/2014/04/22/obzor-ssd-nakopitelya-kingston-skc300s37a240g-240-gbajt/.

случаев в релизных версиях устройств диагностический интерфейс UART может или только выводить данные, или защищен каким-либо способом авторизации (например, паролем) для включения возможности принимать команды управления. Но если нарушить логику работы устройства, например glitch-атакой (о них далее) или физическим вмешательством в схему устройства (отключением интерфейсов связи между компонентами, микросхем памяти и т. п.), то существует шанс получения дампа SRAM микроконтроллера, содержимого стека или любой другой полезной информации. Данный метод направлен на использование ошибок реализации обработки нестандартных ситуаций в прошивке или на ошибку конфигурирования устройства. Например, отключение проверки пароля для CLI-интерфейса в случае входа устройства в режим восстановления прошивки (Recovery или ROM-mode).

Хорошей иллюстрацией примера использования UART для считывания и модификации прошивки устройства (да не простого, а аркадного автомата) является статья «Intro to Embedded RE: UART Discovery and Firmware Extraction via UBoot» (**https://voidstarsec.com/blog/uart-uboot-and-usb**).

Препарирование обновлений

Многие устройства поддерживают обновление прошивки. Механизмы обновления могут быть различные: от скачивания прошивки с сайта производителя на SD-карту с подключением ее к устройству, подключения устройства к компьютеру по UART до обновления через встроенный модуль связи и подключение к серверу производителя через интернет самим устройством. Как правило, разработчики современных устройств используют какие-либо механизмы защиты обновлений. Однако зачастую производитель защищается от модификации прошивки, т. е. использует цифровую подпись прошивки, а не шифрует ее. На данном этапе нас такой вариант вполне устраивает, ведь мы сможем получить прошивку для исследования. А надо ли ее будет модифицировать и как это делать, будем решать позже.

Рассмотрим самые распространенные подходы к получению прошивки в процессе обновления.

1. Скачивание бинарного образа прошивки с сайта производителя, тематических форумов и других источников в интернете. Самый простой способ, однако работающий.
2. Через ПО обновления прошивки. Прошивка может быть в ресурсах ПО обновления или лежать в распакованном архиве рядом с бинарником ПО. А может, ПО скачивает ее из интернета. В таком случае можно попробовать перехватить трафик (как от ПО до сервера производителя, так и от ПО до устройства, например с помощью драйвера-фильтра, Wireshark или подключившись к интерфейсу с помощью логического анализатора). Возможно, нужно будет немного модифицировать ПО обновления, чтобы сохранить скачанный или расшифрованный дамп. Главное – не

забывать, что поиск ПО обновления не ограничивается только сайтом производителя. Форумы ремонтников оборудования, китайские или тематические форумы часто помогают в случае, если производитель не выкладывает в свободный доступ утилиты.

3. Часто «безопасность» протоколов взаимодействия технологического ПО и устройства (ПО обновления прошивки в том числе) строится на их недокументированности. Возможно, вы не смогли найти ПО именно для вашего устройства. Но смогли найти ПО для похожей линейки устройств или для предыдущих поколений исследуемого устройства. С большой долей вероятности технологические протоколы или приватные ключи для подписи прошивки не поменялись и могут подойти для исследуемого устройства.
4. Нельзя забывать о социальной инженерии. Иногда, когда что-то нельзя найти, можно попробовать это попросить у инженера производителя, найдя его контакт на LinkedIn.
5. OTA (Over The Air) обновление. Если устройство имеет модуль связи LTE, Wi-Fi или Bluetooth, оно вполне может обновляться через один из этих интерфейсов. И для определенных сценариев можно попробовать перехватить трафик с помощью Android-телефона с root-правами или модифицированного роутера.

Главное – это понимать принцип построения системы обновления и пытаться понять всю поверхность атаки. На десерт приведу еще один экзотический пример возможного способа получения прошивки. Например, обновление прошивки реализовано через встроенный модуль LTE посредством OTA. Перехватить трафик в сотовой сети мы не можем. Модуль LTE подключен к микроконтроллеру посредством UART. Отлично, подключаемся логическим анализатором и снифаем прошивку. Но вдруг там TLS-шифрование на стороне микроконтроллера или прошивка зашифрована? В таком случае снифинг UART между модулем связи и микроконтроллером не поможет. А если представить, что мы можем подключить логический анализатор к внешней микросхеме RAM-памяти микроконтроллера? Сложно, но если очень захотеть, то можно. Предположим, что образ прошивки перед программированием где-то должен располагаться и, например, расшифровываться. Внутренней RAM-памяти микроконтроллера для этого, скорее всего, не хватит. Так что подход со снифингом интерфейса внешней RAM-памяти пусть и сложен в реализации, но может привести к успешному получению образа прошивки. Конечно, это задача не для начинающих исследователей встраиваемых систем, но именно им надо научиться смотреть максимально широко и искать нестандартные подходы к решению возникающих перед ними задач. Атаки на внешнюю RAM-память, получившие название Cold Boot Attack, широко известны в мире компьютерной криминалистики (**https://www.usenix.org/legacy/event/sec08/tech/full_papers/halderman/halderman.pdf**) и могут быть адаптированы для цифровых устройств (**https://cfp.recon.cx/2023/talk/HCJHBW/**).

Неинвазивные атаки

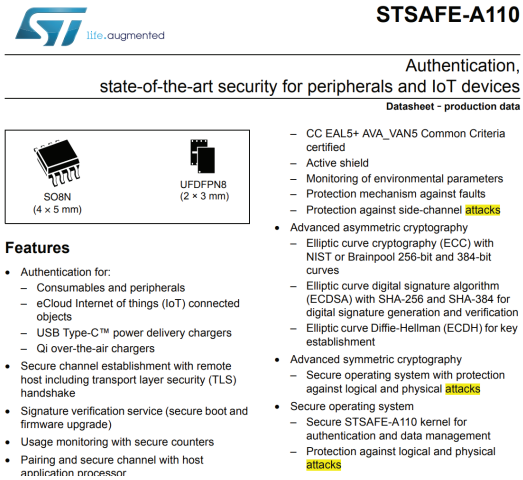
В предыдущих разделах мы рассмотрели стандартные способы получения прошивки (в некоторых случаях содержимого всего ПЗУ, в том числе прошивки). Существуют менее известные методы, которые могут помочь в случае, если другие попытки не увенчались успехом. Одни из них требуют сложного технологического оборудования, материалов и умений – это инвазивные атаки, т. е. атаки, при которых происходит вскрытие корпуса чипа (механическим, лазерным, химическим или комбинированным способом) и осуществляется доступ непосредственно к кристаллу микроконтроллера. Такой техникой можно попытаться изменить электрические сигналы в кристалле, отключив защиту или считав сигнал непосредственно с шины вычислительного ядра микроконтроллера. Большинство подобных атак под силу только крупным лабораториям ввиду своей сложности и требований к необходимому оборудованию (при современных техпроцессах производства микроконтроллеров).

Гораздо практичнее для большинства исследователей безопасности встраиваемых систем неинвазивные атаки, т. е. те, что не требуют физического вмешательства в микроконтроллер. Такие атаки делятся на атаки индуцированным сбоем (fault-injection, FI, также называемые glitch-атаки) и атаки по побочным каналам (Side Channel Attacks, SCA). Атаки по побочным каналам мы рассмотрим чуть позже, а пока сконцентрируемся на glitch-атаках. Glitch-атаки проводятся серией кратковременных импульсов в точный момент времени по определенному методу воздействия. Наиболее популярные методы: по тактовому сигналу (CLK-glitch), питанию (VCC-glitch) и при помощи электромагнитного импульса (EMFI). Все они направлены на то, чтобы с помощью внешнего воздействия изменить логику работы какой-то части микроконтроллера (ядра или аппаратного блока, например JTAG), т. е. внести управляемую ошибку в работу. Glitch-атаки позволяют в том числе обойти какую-либо инструкцию МК, например ожидание ввода пароля или проверку подписи прошивки.

К минусам проведения подобных атак можно отнести:

1) нестабильность процесса проведения атаки. Изменение температуры окружающей среды и малейшие отклонения параметров питания могут существенно понизить вероятность успешного проведения атаки;
2) необходимость в точке отсчета. Так как атака должна произойти в точно определенный момент времени (например, для пропуска конкретной инструкции в прошивке), необходим внешний сигнал, от которого можно отсчитать заданный временной интервал. Подобным сигналом может быть вывод в отладочную консоль, изменение состояния светодиода и т. п. Также можно пробовать отсчитывать время подачи сигнала от момента подачи питания или искать паттерны энергопотребления;
3) необходимость доработки платы устройства или перенос микроконтроллера на специально спроектированную плату для проведения подобных атак. Это поможет повысить вероятность успешной атаки за счет уменьшения влияния внешних компонентов (например, конденсаторов на линиях питания) на работу микроконтроллера.

Существуют специальные методы защиты микроконтроллеров, памяти и прошивок от неинвазивных атак, однако они применяются далеко не в каждой микросхеме, а только там, где неинвазивные атаки – один из потенциальных существенных рисков для компрометации системы. Например, в защите смарт-карт или специальных микросхем безопасности. На рисунке показана выдержка из документации на микросхему типа secure element (которые мы рассмотрим в главе 4) STMicroelectronics STSAFE-A110:



life.augmented

STSAFE-A110

Authentication,
state-of-the-art security for peripherals and IoT devices

Datasheet - production data

SO8N
(4 × 5 mm)

UFDFPN8
(2 × 3 mm)

Features

- Authentication for:
  - Consumables and peripherals
  - eCloud Internet of things (IoT) connected objects
  - USB Type-C™ power delivery chargers
  - Qi over-the-air chargers
- Secure channel establishment with remote host including transport layer security (TLS) handshake
- Signature verification service (secure boot and firmware upgrade)
- Usage monitoring with secure counters
- Pairing and secure channel with host application processor
  - CC EAL5+ AVA_VAN5 Common Criteria certified
  - Active shield
  - Monitoring of environmental parameters
  - Protection mechanism against faults
  - Protection against side-channel attacks
- Advanced asymmetric cryptography
  - Elliptic curve cryptography (ECC) with NIST or Brainpool 256-bit and 384-bit curves
  - Elliptic curve digital signature algorithm (ECDSA) with SHA-256 and SHA-384 for digital signature generation and verification
  - Elliptic curve Diffie-Hellman (ECDH) for key establishment
- Advanced symmetric cryptography
  - Secure operating system with protection against logical and physical attacks
- Secure operating system
  - Secure STSAFE-A110 kernel for authentication and data management
  - Protection against logical and physical attacks

Рис. 1.42. Фрагмент документации на микросхему STM STSAFE-A110[1]

Для проведения различного вида неинвазивных атак существует оборудование полупрофессионального уровня (например, от компании **https://www.riscure.com**), но для проведения некоторых атак достаточно довольно дешевого оборудования, часть из которого можно собрать на базе распространенных отладочных плат. Самым известным доступным универсальным «комбайном» для проведения неинвазивных атак является OpenSource-устройство ChipWhisperer (**https://www.newae.com/chipwhisperer**), базовый кит которого стоит всего 500$.

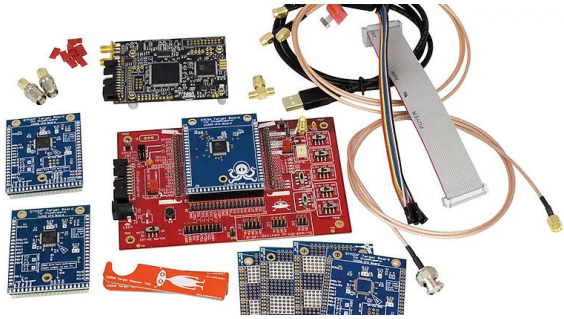


Рис. 1.43. Набор с устройством ChipWhisperer для проведения неинвазивных атак[2]

[1] https://www.farnell.com/datasheets/2917358.pdf.

[2] https://store.newae.com/side-channel-glitching-starter-pack-level-1/.

Далее мы кратко рассмотрим распространенные виды glitch-атак. Подробнее про неинвазивные атаки можно почитать на сайте компании-разработчика ChipWhisperer (**www.newae.com**), а в книге «The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks» основателя компании NewAE Colin O'Flynn данная тема раскрыта максимально подробно.

Атаки на синхронизацию (CLK-glitch)

Создавая краткосрочные импульсы (намного короче, чем период тактового сигнала) на линии внешней синхронизации микроконтроллера, можно заставить его пропустить инструкцию микропрограммы или нарушить последовательность выполнения инструкции за счет сбоя декодера адреса или команд. Пример нормального тактового сигнала и сигнала, модифицированного с помощью CLK-glitch-атаки, показан на рисунке:

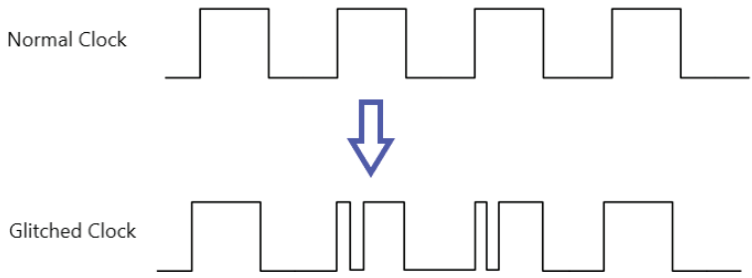


Рис. 1.44. Clock glitch[1]

Представим, что вычислительное ядро микроконтроллера имеет конвейер, позволяющий ускорить процесс выполнения программы. Для простоты возьмем две ступени конвейера, на первой ступени происходит загрузка инструкции из прошивки, а на второй – ее декодирование и исполнение.

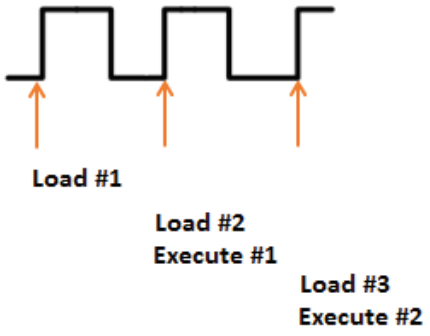


Рис. 1.45. Работа конвейера без clock clitch[2]

[1] https://wiki.newae.com/V3:Tutorial_A2_Introduction_to_Glitch_Attacks_(including_Glitch_Explorer).
[2] https://wiki.newae.com/V3:Tutorial_A2_Introduction_to_Glitch_Attacks_(including_Glitch_Explorer).

Если мы сможем модифицировать сигнал на линии внешней синхронизации микроконтроллера, ядро может попасть в ситуацию, когда ему не хватит времени на декодирование и исполнение инструкции. Если следующий сигнал синхронизации придет в ядро до того момента, как будет выполнена текущая инструкция, микроконтроллер пропустит ее и выполнит следующую. На примере ниже происходит пропуск выполнения инструкции #1:

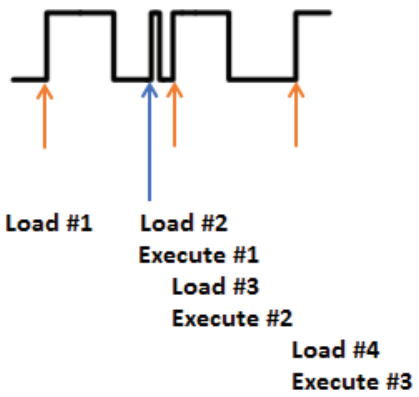


Рис. 1.46. Работа конвейера с clock clitch[1]

В качестве примера рассмотрим возможность использования glitch для пропуска инструкции из модуля Linux PAM – набора библиотек, который позволяет системному администратору Linux настраивать методы аутентификации пользователей (фрагмент взят из файла auth.c уязвимой версии):

```
int auth_pam(const char *service_name, uid_t uid, const char *username)
{
  if (uid != 0) {
    pam_handle_t *pamh = NULL;
    struct pam_conv conv = { misc_conv, NULL };
    int retcode;

    retcode = pam_start(service_name, username, &conv, &pamh);
    if (pam_fail_check(pamh, retcode))
      return FALSE;

    retcode = pam_authenticate(pamh, 0);
    if (pam_fail_check(pamh, retcode))
      return FALSE;

    retcode = pam_acct_mgmt(pamh, 0);
    if (retcode == PAM_NEW_AUTHTOK_REQD)
      retcode =
```

[1] https://wiki.newae.com/V3:Tutorial_A2_Introduction_to_Glitch_Attacks_(including_Glitch_Explorer).

```
        pam_chauthtok(pamh, PAM_CHANGE_EXPIRED_AUTHTOK);
    if (pam_fail_check(pamh, retcode))
      return FALSE;

    retcode = pam_setcred(pamh, 0);
    if (pam_fail_check(pamh, retcode))
      return FALSE;

    pam_end(pamh, 0);
    /* no need to establish a session; this isn't a
     * session-oriented activity... */
  }
  return TRUE;
}
```

В приведенном листинге функции авторизации достаточно пропустить инструкцию проверки if (uid != 0) для успешной авторизации без выполнения каких-либо проверок.

Для повышения вероятности успешной атаки CLK-glitch целесообразно внести изменения в схему устройства. Пример подобных изменений для микроконтроллера STM32F03 показан на схеме ниже. Тактирующий сигнал ChipWhisperer подключается вместо сигнала тактового генератора микроконтроллера, убираются фильтрующие конденсаторы в цепях питания.

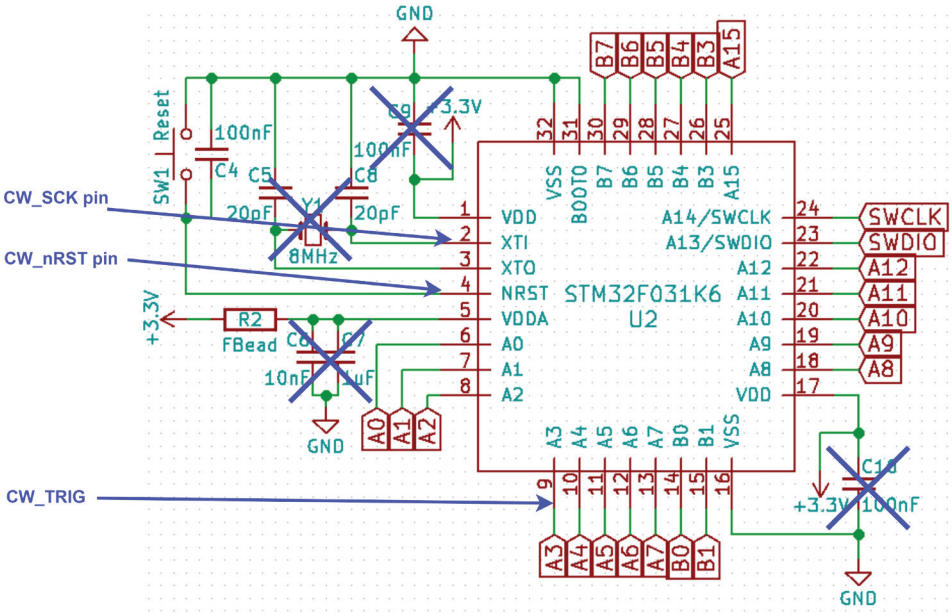


Рис. 1.47. Доработка платы для повышения вероятности атаки clock-glitch

Атаки «по питанию» (VCC-glitch)

Атаки VCC-glitch, исходя из своего названия, направлены на подсистему питания микроконтроллера. Создавая импульсы пониженного (в некоторых случаях повышенного) напряжения, можно изменить логику работы отдельных компонентов микроконтроллера, например заставить вычислительное ядро пропустить текущую инструкцию без ее исполнения или изменить инструкцию, тем самым модифицировав логику работы прошивки, или вызвать сбой при записи данных в память. На рисунке далее показаны два состояния линии питания микроконтроллера: слева – нормальное состояние, справа – просадка питания при импульсе VCC-glitch.

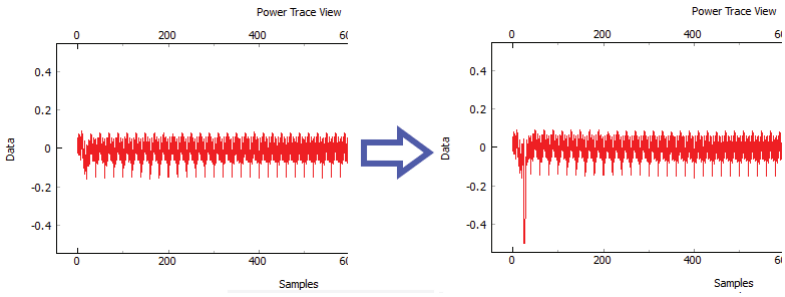


Рис. 1.48. Пример сигнала при атаке VCC-glitch[1]

На схеме ниже показаны необходимые доработки, которые надо выполнить на печатной плате устройства (на примере платы с микроконтроллером STM32F031). Основная их часть так же, как и при CLK-glitch, заключается в отключении фильтров питания устройства, ведь они не дадут выполнить инъекцию импульса на шине питания с требуемыми характеристиками.

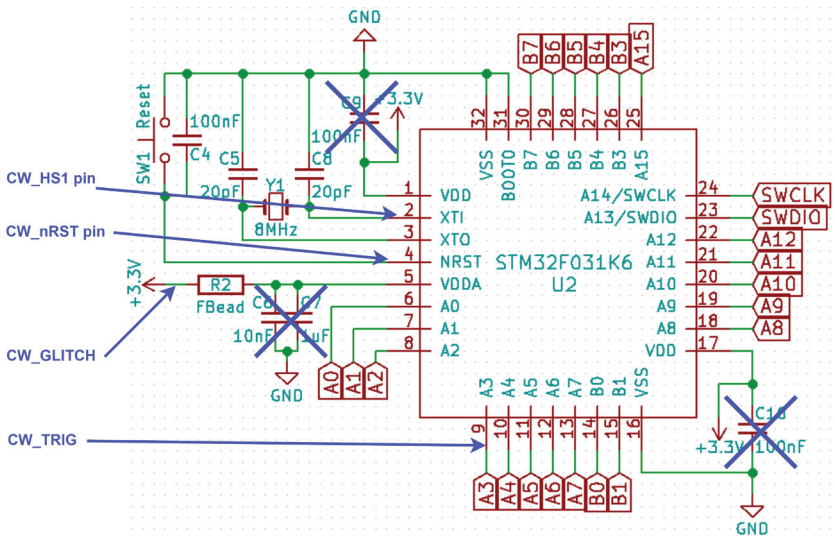


Рис. 1.49. Доработка платы для повышения вероятности атаки VCC-glitch

[1] https://wiki.newae.com/V3:Tutorial_A3_VCC_Glitch_Attacks.

Примеров успешно проведенных VCC-glitch атак в интернете довольно много, зачастую для их выполнения нужно совсем базовое оборудование, собранное на базе распространенных отладочных плат. Хорошими описаниями успешно реализованных VCC-glitch атак являются следующие исследования:

1) Pwn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction (**https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/**);

2) nRF52 Debug Resurrection (APPROTECT Bypass) – **https://limitedresults.com/2020/06/nrf52-debug-resurrection-approtect-bypass/**;

3) Glitched on Earth by Humans: A Black-Box Security Evaluation of the SpaceX Starlink User Terminal (**https://i.blackhat.com/USA-22/Wednesday/US-22-Wouters-Glitched-On-Earth.pdf**).

При проведении glitch-атак необходимо учитывать, что параметры успешного импульса зависят от множества факторов, таких как:

- особенности конструкции печатной платы для МК и наличие разнообразной обвязки (конденсаторов на линиях питания и т. д.);
- длины коаксиального кабеля подключения МК и glitch-порта устройства-глитчера (чем длиннее кабель, тем меньше вероятность успешной атаки, оптимальная длина – не более 10–15 см);
- инструкции, которую необходимо пропустить;
- оптимизации компилятора;
- условий внешней среды.

Поэтому при проведении glitch-атак необходимо следить за тем, чтобы эти параметры не изменялись в процессе повторения эксперимента.

Для защиты от glitch-атак производитель микросхемы и разработчик прошивки могут предпринять дополнительные меры при разработке:

- использовать внутренний генератор тактового сигнала или схему контроля допустимых частот и напряжений, перезагружающую микроконтроллер при выходе параметров за разрешенные пределы;
- проверку целостности данных и кода в прошивке с помощью контрольных сумм и дополнительных проверок, в том числе в случайные моменты времени;
- использование счетчика сбоев, отключающего устройства или стирающего чувствительные данные при достижении заданного количества неудач.

Атаки электромагнитным импульсом (EMFI)

Атаки с помощью электромагнитного импульса производятся за счет создания высоковольтного разряда вблизи определенной области микроконтроллера. Для осуществления простых ЭМИ-атак проще всего использовать устройство ChipSHOUTER.

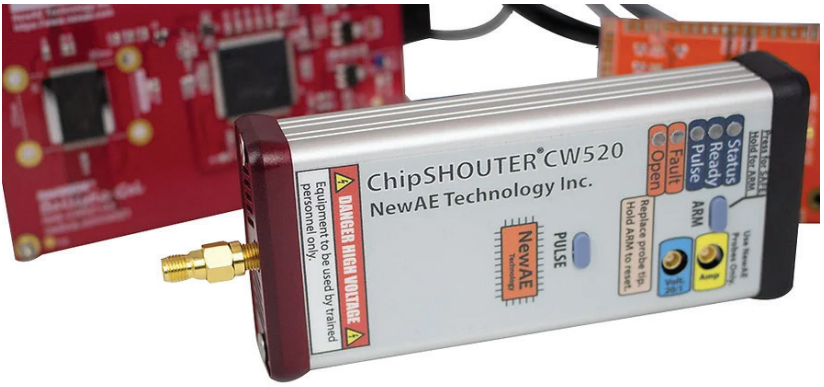


Рис. 1.50. Устройство ChipSHOUTER для проведения атак EMFI[1]

В отличие от VCC-glitch и CLK-glitch атак, EMFI не требует подключения к электрической схеме исследуемого устройства для воздействия. Специальный инжектор, состоящий из ферритового сердечника и индуцирующей электромагнитный импульс проволочной обмотки (показан на фото далее), располагается рядом с исследуемой областью микроконтроллера.

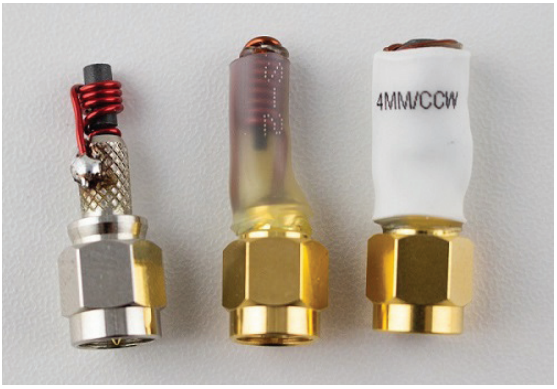


Рис. 1.51. Инжекторы для проведения EMFI-атак[2]

При высоковольтном разряде та часть микросхемы, которая физически находится рядом с пробником, может сработать неправильно вследствие высоковольтного импульса (например, произойдет инвертирование бита в памяти или искажение сигнала в цепи микроконтроллера).

[1] https://store.newae.com/chipshouter-kit/.

[2] https://github.com/newaetech/chipshouter-picoemp/.

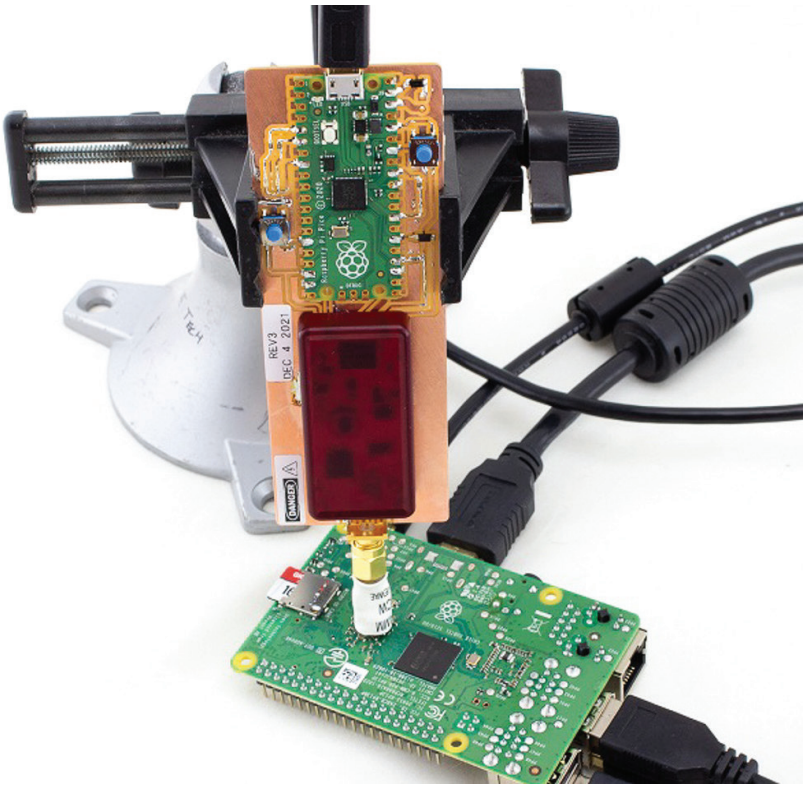


Рис. 1.52. Пример проведения EMFI-атаки[1]

Для защиты от подобных атак могут использоваться как программные способы, такие как контроль целостности чувствительных данных, так и аппаратные схемы, измеряющие время распространения сигнала по наиболее длительному пути. Преимуществом данного типа атак является сложность реализации аппаратной защиты, поэтому большинство микросхем являются уязвимым к EMFI-атакам.

Примером подобной атаки является исследование безопасности аппаратного кошелька от Colin O'Flynn (автора ChipWhisperer & ChipSHOUTER) – Glitching Trezor using EMFI Through The Enclosure (**https://colinoflynn.com/2019/03/glitching-trezor-using-emfi-through-the-enclosure/**).

Еще одним примером применения EMFI-атак может быть изменение содержимого отдельных битов памяти. Для повышения вероятности успеха в подобных атаках зачастую необходимо выполнять процедуру декапсуляции микросхемы, т. е. получения доступа непосредственно к кристаллу (и тогда атаки уже становятся инвазивными, про них мы поговорим далее). Определение уязвимого места на микросхеме (точнее, на кристалле) является одной из наиболее сложных задач при реализации ЭМИ-атак. На фото показан декапсулированный кристалл микроконтроллера STM32F051R8T6.

[1] https://github.com/newaetech/chipshouter-picoemp/.

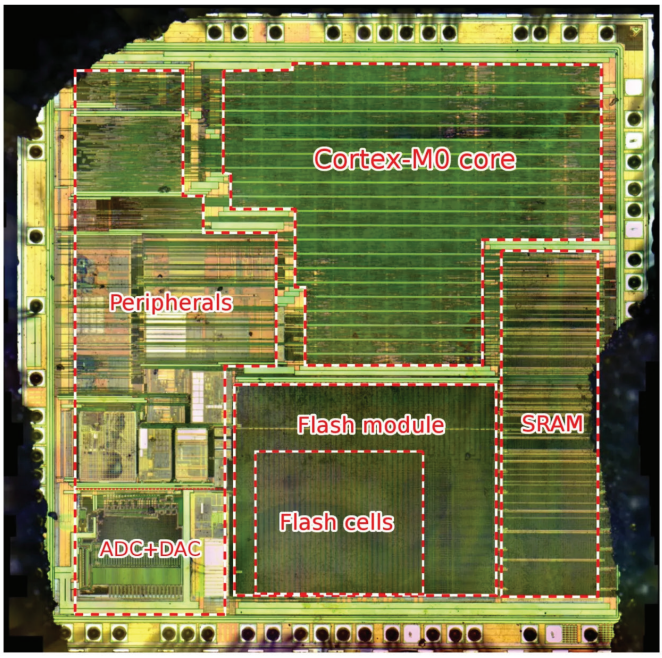


Рис. 1.53. Декапсулированный кристалл микроконтроллера STM32F051R8T6[1]

Для автоматизации перебора положений нужен двухкоординатный стол с высоким разрешением (аналогичный применяемым в 3D-принтерах), пример использования которого показан на фото:

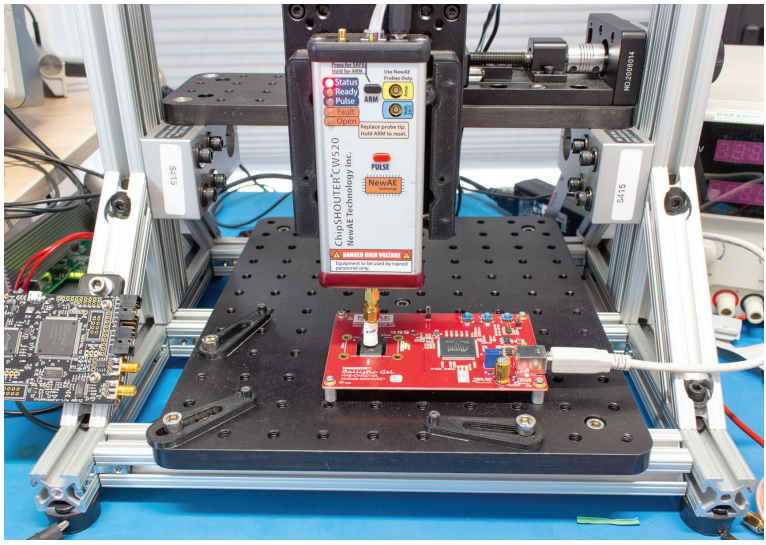


Рис. 1.54. Координатный стол для проведения EMFI-атак[2]

[1] https://www.aisec.fraunhofer.de/content/dam/aisec/ResearchExcellence/woot17-paper-obermaier.pdf.

[2] https://media.newae.com/appnotes/NAE0011_Whitepaper_EMFI_For_Automotive_Safety_Security_Testing.pdf.

В результате проведения множества экспериментов составляется карта кристалла с участками, в которых атака имеет ответную реакцию. Например, в исследовании «Espressif ESP32: Bypassing Secure Boot using EMFI» (**https://www.gushiciku.cn/pl/pKYq**) были проведены 165 000 итераций воздействия, что заняло 8,5 ч. В результате была составлена карта участков микроконтроллера с ответной реакцией на EMFI-атаку в зависимости от положения пробника:

- зеленым цветом отмечены участки без видимой реакции;
- желтым отмечены участки, при воздействии на которые микроконтроллер зависает или перезагружается;
- красным ромбом отмечено место, в котором атака приводит к желаемому изменению потока выполнения прошивки;
- красными крестами отмечены участки, атака в которых приводит к генерации исключения о некорректной инструкции.

Рис. 1.55. Карта кристалла с результатами воздействия EMFI-атакой

На текущий момент EMFI-атаки являются одними из самых перспективных с точки зрения исследователя цифровых устройств, т. к. внедрение аппаратной защиты от них требует существенных усилий от разработчика микросхемы. Стоимость проведения EMFI-атак может быть ниже инвазивных, при этом вероятность повреждения устройства ниже. Однако иногда более эффективно может быть использование другого типа воздействия, например оптическим импульсом.

Атаки оптическим импульсом (LFI)

LFI-атаки (Laser Fault Injection) похожи по своему способу выполнения на EMFI, только вместо электромагнитного излучения используется излучение специального лазера, импульс которого может изменить заряд в транзисторе, лежащем в основе ячейки памяти или логической схемы. Хорошим источником подробной информации про механизм LFI-атак является презентация «Hardware attacks: theory and experimental state-of-the-art of laser fault injection attacks» (**https://jnsecurite2018.sciencesconf.org/data/pages/Jean_Max_DUTERTRE.pdf**).

Установка фирмы Alphanov, позволяющая проводить LFI-атаку сразу двумя лазерами, показана на фото:

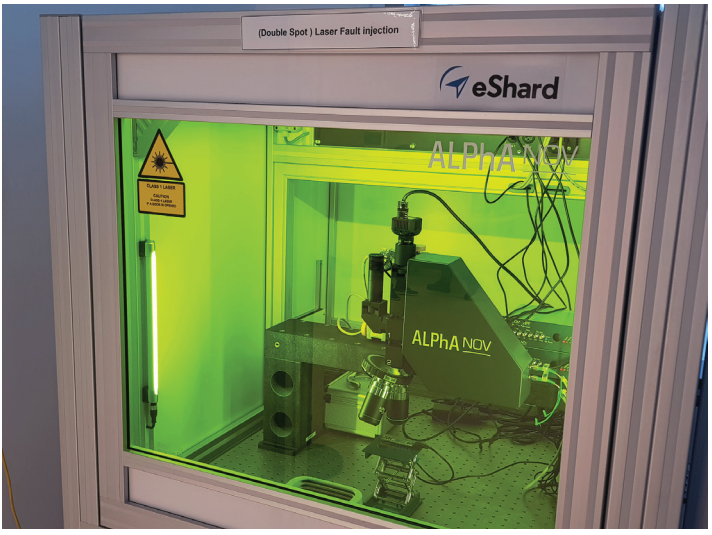


Рис. 1.56. Установка Alphanov для проведения Laser Fault Injection атак[1]

К сожалению, оборудование для LFI-атак стоит от сотен тысяч до миллионов долларов и недоступно для большинства исследователей, а для проведения атаки необходима процедура декапсуляции микросхемы.

Для защиты от LFI-атак, как и для EMFI-атак, актуальны программные методы контроля целостности данных в процессе выполнения прошивки. В качестве аппаратной защиты применяются защитные экраны из металлических слоев на кристалле и оптические датчики, срабатывающие при декапсуляции кристалла или обнаружении излучения лазера.

Каким образом изменение единичного бита в памяти может помочь в исследовании цифровых устройств? Например, для понижения уровня защиты прошивки микроконтроллера от считывания. Ранее мы рассматривали механизм RDP в микроконтроллерах STM32, а именно STM32F051R8T6. Особенностью ее реализации является кодирование уровня защиты двумя восьмибитными регистрами (RDP и nRDP, значение в последнем является инверсией значения в первом). Значения уровней кодируются следующим образом:

[1] https://eshard.com/posts/alphanov-eshard-lab-for-integrated-circuits-testing-against-physical-attacks.

- Level 0 – RDP = 0x55AA;
- Level 2 – RDP = 0x33CC;
- Level 1, RDP принимает любые значения, отличные от 0x55AA и 0x33CC.

Даунгрейд значения с Level 1 до Level 0 реализовать достаточно сложно, т. к. Level 1 может кодироваться различными значениями. Даунгрейд с Level 2 на Level 1, наоборот, выглядит гораздо проще, т. к. для него нужно изменение одного любого бита (показано на рисунке ниже).

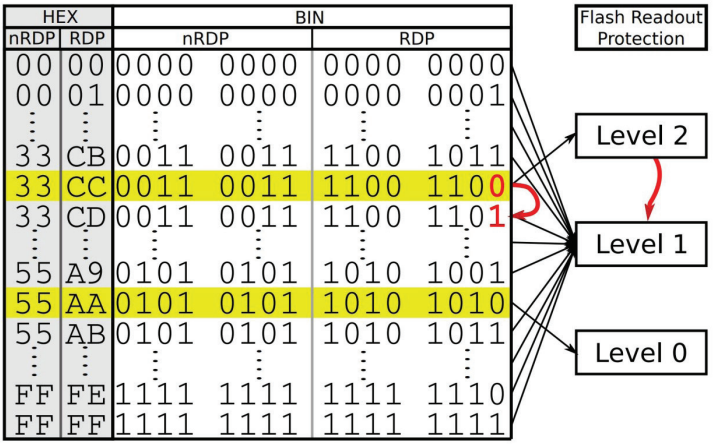


Рис. 1.57. Схема даунгрейда RDP на микроконтроллере STM32F051R8T6[1]

При этом для выбранного микроконтроллера существуют атаки, позволяющие считать прошивку при уровне защиты Level 1 за счет использования уязвимостей отладочного интерфейса. Таким образом, для компрометации системы защиты от считывания содержимого flash-памяти микроконтроллера необходимо выполнить две последовательные атаки:

1) даунгрейд с Level 2 на Level 1 с помощью EMFI/LFI;
2) использование VCC-glitch для атаки на ошибку реализации отладочного интерфейса в режиме работы защиты Level 1.

Важным фактором для реализации подобной атаки является место хранения значения регистра RDP. В случае микроконтроллера STM32F051R8T6 этим местом является flash-память, что делает атаку с помощью электромагнитного излучения возможной, т. к. с помощью него мы можем изменить заряд на затворе транзистора, являющегося основой ячейки flash-памяти. Если значение хранится в электронно-пережигаемых перемычках типа eFUSE, необходимо будет проводить инвазивные атаки (о них мы также поговорим немного позднее) или проводить LFI-атаку на аппаратную схему, отвечающую за чтение значения eFUSE. Подробнее о конкретно этой реализации атаки, с использованием довольно дешевого источника излучения с длиной волны ультрафиолетового спектра (фактически являющего подмножеством электромагнитного

[1] https://www.aisec.fraunhofer.de/content/dam/aisec/ResearchExcellence/woot17-paper-obermaier.pdf.

излучения) для изменения значения RDP с 0x33CC на 0x33CD, вы можете узнать в презентации Shedding too much Light on a Microcontroller's Firmware Protection (**https://www.aisec.fraunhofer.de/en/FirmwareProtection.html**).

Side-Channel атаки

Атаки по побочным каналам (Side Channel Attacks, SCA) являются второй разновидностью неинвазивных атак (хотя зачастую для повышения вероятности некоторых типов SC-атак проводят процедуру декапсуляции чипа). В качестве побочных каналов могут выступать любые параметры, изменение которых может дать информацию об исследуемой величине. Например, минимальное изменение потребления тока или электромагнитного поля при выполнении различных инструкций ядром микроконтроллера. Можно измерять время выполнения веток подпрограммы при проверке пароля в случае успешного или неуспешного сравнения. ChipWhisperer (особенно в версии CW1200 Pro, показан на фото далее) позволяет проводить некоторые SC-атаки, но ничто не мешает сделать аналогичное устройство самостоятельно, ведь главное – это понимать принцип, на котором основывается атака.

Рис. 1.58. Устройство ChipWhisperer CW1200 Pro для проведения SC-атак[1]

Наиболее распространенным сценарием применения SC-атак для исследования цифровых устройств является анализ потребления тока микроконтроллером в зависимости от времени. С помощью этой информации можно даже пытаться определить конкретный тип инструкции, исполняющейся в данный момент. На рисунке показаны различия в графиках потребления тока микроконтроллером Atmel ATXMEGA128D4 для инструкций NOP (no operation) и MUL (multiply). Инструкция MUL перемножает два восьмибитных числа, требует двух тактов для выполнения и вызывает значительное потребление тока микроконтроллером. Инструкция NOP не выполняет операций, требует одного такта для выполнения и имеет гораздо меньшее значение потребления тока.

[1] https://rtfm.newae.com/Starter%20Kits/ChipWhisperer-Pro/.

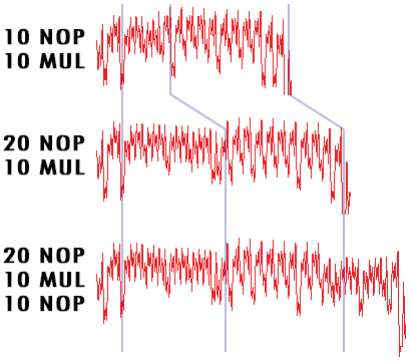


Рис. 1.59. Различие энергопотребления микроконтроллера в зависимости от исполняемых инструкций[1]

Например, если процедура проверки пароля внутри функции микроконтроллера написана без учета возможного проведения SC-атак, то график потребления тока микроконтроллером при проверке пароля может выглядеть следующим образом:

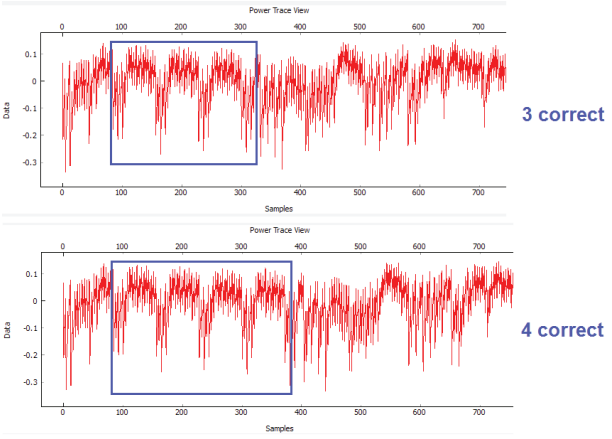


Рис. 1.60 Различие энергопотребления микроконтроллером при разном количестве правильных символов в проверке пароля[2]

На графике сверху мы видим график потребления тока при трех правильных символах в начале пароля, на нижнем графике – при четырех правильных символах. Имея возможность собирать подобную информацию, можно автоматизировать перебор пароля. Примером атак подобного типа является взлом электронных замков на конференции DEF CON 24 (**https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Plore-Side-Channel-Attacks-On-High-Security-Electronic-Safe-Locks.pdf**).

[1] https://wiki.newae.com/V4:Tutorial_B2_Viewing_Instruction_Power_Differences.

[2] https://wiki.newae.com/V4:Tutorial_B3-1_Timing_Analysis_with_Power_for_Password_Bypass.

Анализируя параметры потребления тока и зная устройство алгоритмов шифрования, можно делать предположения об используемых алгоритмах шифрования и даже восстанавливать их параметры, в том числе ключи. Конечно, реализация подобных атак, называемых простым анализом питания (Simple Power Analysis, SPA), дифференциальным анализом питания (Differential Power Analysis, DPA) и корреляционным анализом питания (Correlation Power Analysis, CPA), требует определенной математической подготовки. Для попытки восстановления ключей шифрования необходимо набрать статистику энергопотребления при множестве итераций шифрования/расшифрования. Чтобы определить используемый алгоритм шифрования, большое количество итераций не нужно. На рисунках ниже показаны графики энергопотребления, по которым легко определить 16 раундов шифрования алгоритма DES и 10 раундов шифрования AES.

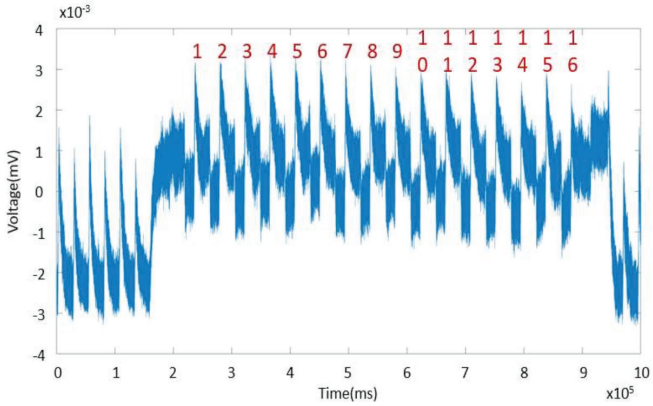


Рис. 1.61. График энергопотребления микроконтроллера при выполнении 16 раундов шифрования DES[1]

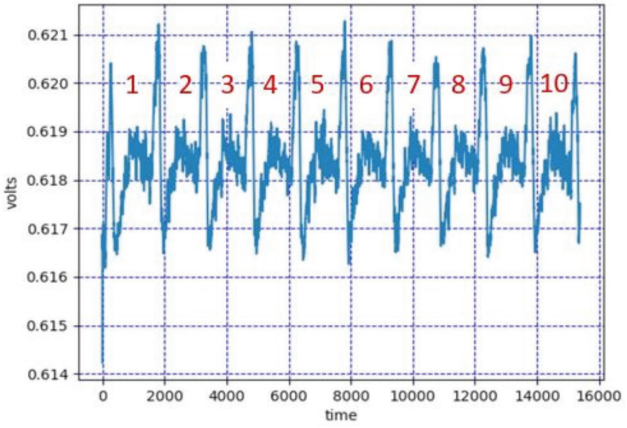


Рис. 1.62. График потребления тока микроконтроллером при выполнении 10 раундов шифрования AES[2]

[1] https://www.mdpi.com/2410-387X/4/2/15#B28-cryptography-04-00015.

[2] https://www.mdpi.com/2410-387X/4/2/15#B28-cryptography-04-00015.

В начале описания SC-атак мы сказали, что источником данных для проведения атаки также может выступать электромагнитное излучение. Подобные атаки и методы интерпретации информации в них похожи на Power Analysis атаки, соответственно, они называются Electromagnetic Analysis атаки. Более подробную информацию можно найти в статье Electromagnetic Techniques and Probes for Side-Channel Analysis on Cryptographic Devices (**https://www.esat.kuleuven.be/cosic/publications/thesis-182.pdf**).

Для съема данных об электромагнитном излучении интересующей части микросхемы используются специальные пробники, которые можно сделать самостоятельно, например как показано на фото из статьи ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels (**https://www.cs.tau.ac.il/~tromer/mobilesc/mobilesc.pdf**):

Рис. 1.63. Самодельный пробник электромагнитного излучения[1]

Существуют готовые недорогие пробники электромагнитного излучения (показаны на фото ниже), которые можно найти на различных маркетплейсах.

Рис. 1.64. Заводские пробники электромагнитного излучения[2]

Для проведения точных измерений электромагнитного излучения существуют профессиональные пробники, например от компании Langer:

[1] https://www.cs.tau.ac.il/~tromer/mobilesc/mobilesc.pdf.

[2] https://teamcmmd.org/Electrical-Testing-itm-105200/Radiation-Test-Antennas-Magnetic-Field-Probe.jsp.

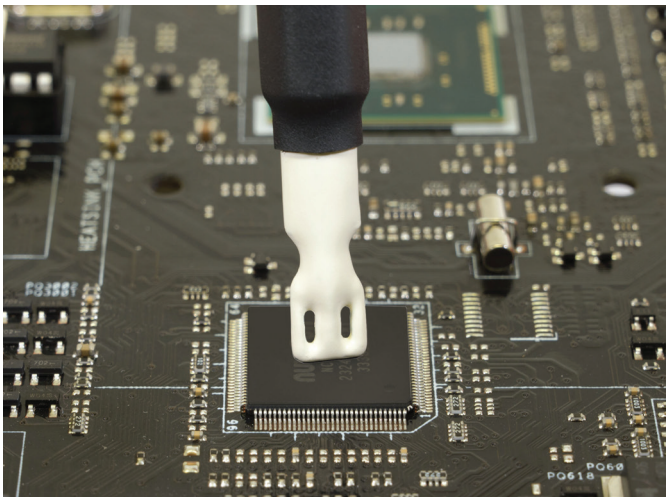


Рис. 1.65. Профессиональный пробник электромагнитного излучения[1]

Как видите, для проведения подобных атак зачастую можно использовать оборудование, доступное практически каждому исследователю встраиваемых систем. Но если есть возможность получить хороший инструмент, лучше ей воспользоваться, т. к. некорректные или сильно зашумленные данные, полученные на входе, существенно снижают вероятность успешной атаки.

Для защиты от SC-атак, так же как и для FI-атак, могут применяться программные и аппаратные методы:

- использование тактирующего генератора с «плавающей» частотой и специальных схем, усредняющих энергопотребление различных частей схемы и инструкций (например, константное время выполнения операций копирования памяти вне зависимости от объема копируемых данных);
- внедрение случайных задержек и прерываний в поток выполнения программы;
- модификация криптографических алгоритмов, например порядка шифрования или использования промежуточных блоков накладывания случайной «маски» на значения вычислений;
- внедрение счетчика криптографических операций, блокирующего устройство при достижении заданного значения. Так как для проведения SC-атак необходимо существенное количество экспериментов, зачастую значительно превышающих количество включений устройства при его времени жизни, большое количество включений является признаком, косвенно указывающим на попытку проведения SC-атаки.

Применение Fault Injection атак и Side Channel атак для исследований безопасности встраиваемых систем является очень перспективным направлением. Они потенциально позволяют получить доступ к секретам устройства

[1] https://www.langer-emv.com/en/product/sx-passive-1ghz-up-to-20-ghz/33/sx-r-20-1-set-near-field-probe-1-ghz-up-to-20-ghz/1246.

тогда, когда другие способы не помогли. При этом большая часть оборудования, необходимого для проведения подобных атак, доступна широкому кругу исследователей. Однако использование профессиональных решений существенно расширяет возможности по проведению атак и увеличивает шансы на успех. Использование профессиональных решений для проведения SC- и FI-атак почти всегда сопровождается вскрытием корпуса микросхемы для доступа непосредственно к кристаллу, т. к. при таком режиме получаются максимально точные результаты и становятся доступны новые техники исследования. Теперь рассмотрим инвазивные атаки на микросхемы.

Хардкор – инвазивные атаки

Техники, описываемые в этом разделе, в основном доступны очень узкому кругу исследователей устройств. В первую очередь из-за стоимости оборудования, составляющей зачастую несколько сотен тысяч долларов, и возможности его купить. Вторым препятствием к проведению подобных исследований являются специфические знания, которыми необходимо обладать. В то же время простые или старые микроконтроллеры, произведенные по нормам в сотни нанометров, могут быть восстановлены с помощью относительно доступного набора оборудования. Пример подобного исследования описан в статье «Pulling Bits From ROM Silicon Die Images: Unknown Architecture» (**https://ryancor.medium.com/pulling-bits-from-rom-silicon-die-images-unknown-architecture-b73b6b0d4e5d**), фото из которой иллюстрируют этот раздел. В данном исследовании использовался оптический микроскоп стоимостью всего 1500 долларов.

Первым делом для проведения инвазивных исследований нужно получить доступ к кристаллу микросхемы. Эта процедура называется декапсуляция и может делаться химическим, механическим или лазерным способом, а также их комбинацией. На фото показан открытый кристалл микросхемы, подключенный к выводам с помощью золотой проволоки.

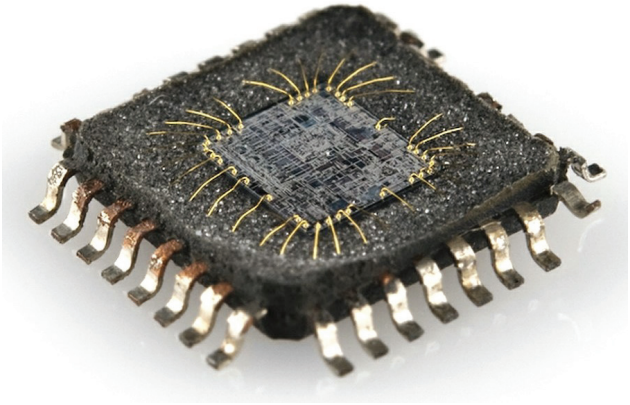


Рис. 1.66. Декапсулированный кристалл микросхемы[1]

[1] https://learn.sparkfun.com/tutorials/integrated-circuits/inside-the-ic.

После получения доступа к кристаллу можно попробовать определить физическое расположение основных функциональных блоков. На фото показан декапсулированный кристалл микроконтроллера STM32F051R8T6.

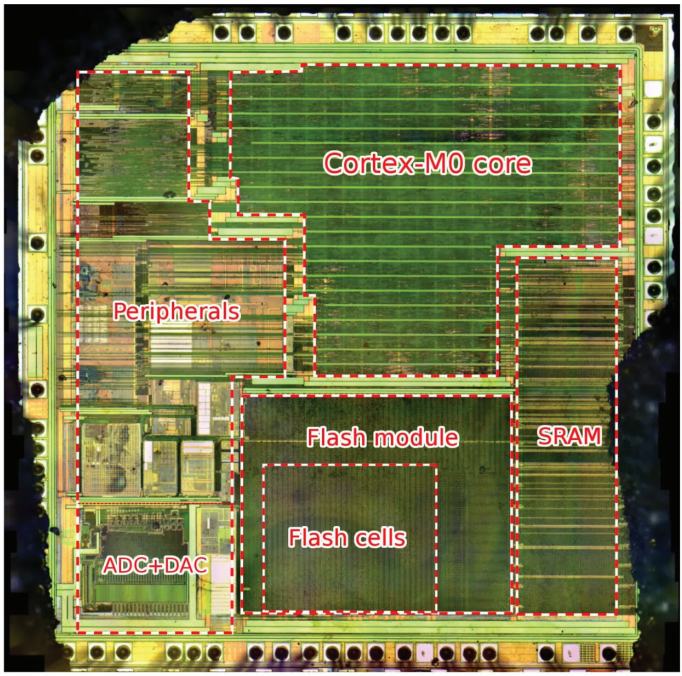


Рис. 1.67. Декапсулированный кристалл микроконтроллера STM32F051R8T6[1]

Для получения подобных фотографий используют оптический микроскоп с высокой разрешающей способностью. Для восстановления топологии кристаллов, сделанных по современным нормам в десятки нанометров, разрешающей способности оптического микроскопа уже не хватает и необходимо использовать сканирующий электронный микроскоп (SEM).

Для защиты от инвазивных атак может быть применено множество методов, однако все из них приводят к существенному удорожанию процесса разработки (а иногда и производства) микросхемы:

- защитный экран из проводников на верхних слоях микросхемы, нарушение которого приводит к срабатыванию схемы защиты;
- физически неклонируемые функции (PUF, Physically Unclonable Functions), подробнее о них мы поговорим в главе 4;
- шифрование ПЗУ и ОЗУ;
- модификация схемы с целью ее обфускации.

Рассмотрим основные виды инвазивных атак, применяющихся при анализе встраиваемых систем.

[1] https://www.aisec.fraunhofer.de/content/dam/aisec/ResearchExcellence/woot17-paper-obermaier.pdf.

Восстановление ROM

В контексте этой книги нас интересует определенная часть топологии кристалла, а именно ROM (также называемая масочной памятью), в которой часто хранится первичный загрузчик, являющийся корнем доверия всей системы. Ячейки ROM, эквивалентные значениям 0 и 1, имеют разную структуру и, соответственно, разный внешний вид. Сканирующий микроскоп позволяет сделать множество снимков определенной области кристалла и с помощью специального ПО «сшить» их в единое изображение. Внешний вид ROM-памяти показан на фото:

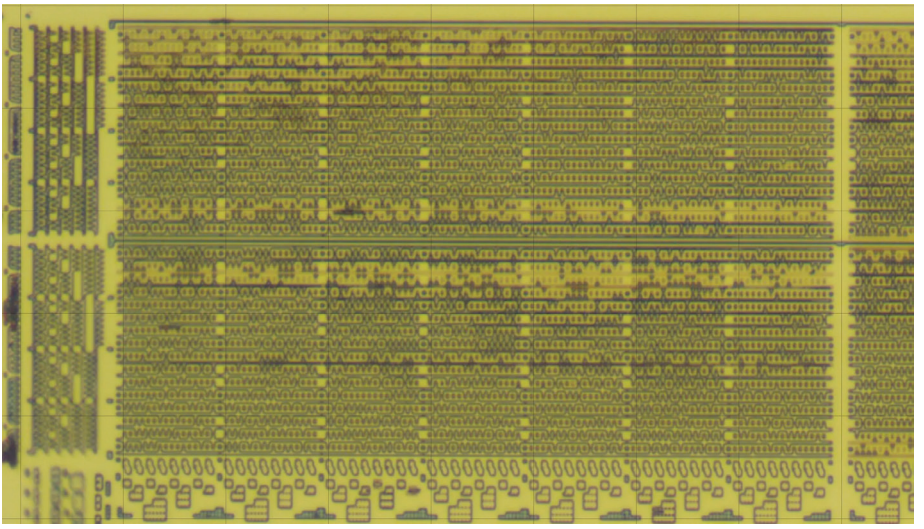


Рис. 1.68. Фотография ROM на кристалле микроконтроллера[1]

Получив такой снимок ROM и зная внешний вид структур, из которых он формируется, можно определить, где расположены ячейки, соответствующие 0 и 1. Пример отмеченных ячеек показан на фото:

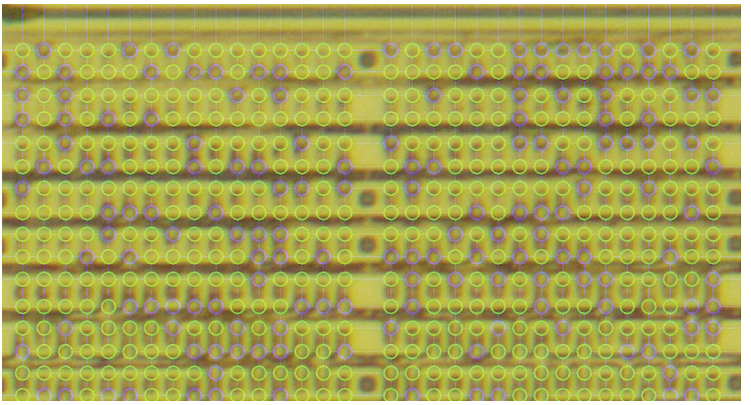


Рис. 1.69. Определение ячеек ROM[2]

[1] https://ryancor.medium.com/pulling-bits-from-rom-silicon-die-images-unknown-architecture-b73b6b0d4e5d.

[2] https://ryancor.medium.com/pulling-bits-from-rom-silicon-die-images-unknown-architecture-b73b6b0d4e5d.

Восстановив местоположение ячеек, необходимо понять соответствие адресов памяти и расположения ячеек. Для этого следует восстановить схему декодера памяти, показанную на фото:

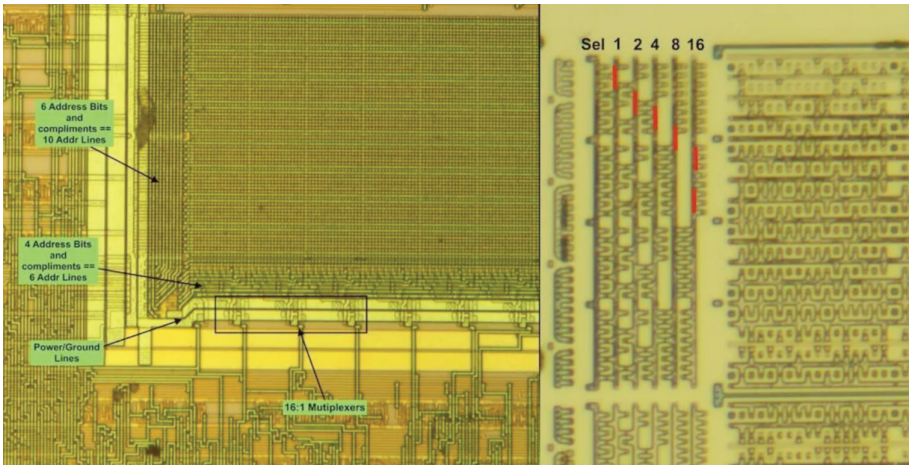


Рис. 1.70. Определение декодера ROM[1]

Восстановив схему декодирования адреса и расположения ячеек, можно загрузить фото в специальное ПО (показано ниже) и, настроив параметры, восстановить содержимое ROM-памяти, в которой помимо первичного загрузчика также могут храниться ключи, пароли и другая информация, являющаяся ценной для исследования.

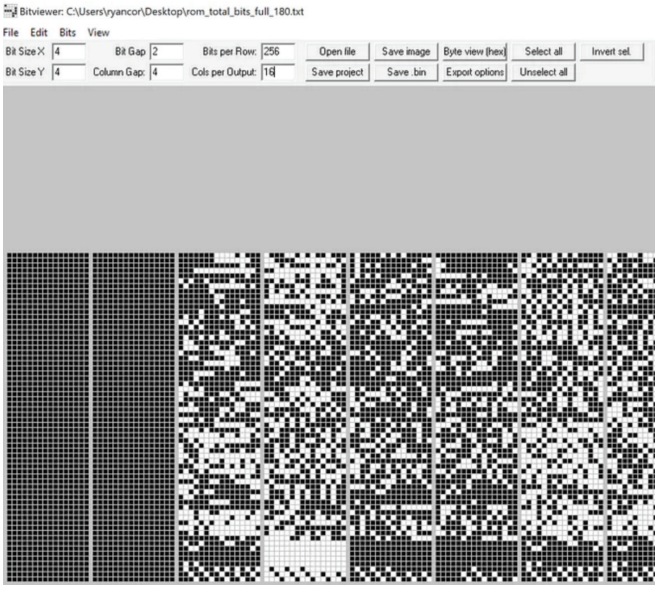


Рис. 1.71. Восстановленные значения ячеек ROM[2]

[1] https://ryancor.medium.com/pulling-bits-from-rom-silicon-die-images-unknown-architecture-b73b6b0d4e5d.

[2] https://ryancor.medium.com/pulling-bits-from-rom-silicon-die-images-unknown-architecture-b73b6b0d4e5d.

Существуют попытки также считывать flash-память (содержимое которой нельзя считать аналогичным подходом, т. к. заряд не видно), например в статье «Reverse engineering Flash EEPROM memories using Scanning Electron Microscopy» (**https://www.cl.cam.ac.uk/~sps32/cardis2016_sem.pdf**). Данная техника возможна только с электронным сканирующим микроскопом и имеет много особенностей, затрудняющих подобное исследование.

Микрозондовый анализ

Следующий класс атак основан на применении микрозондов – специальных пробников (тончайших проводников) с очень точной механикой, позволяющей позиционировать пробники в нужном месте кристалла. С помощью пробников можно как считывать сигнал с определенных участков схемы, так и генерировать сигнал для осуществления активных атак на схему. Внешний вид микрозондов показан на фото:

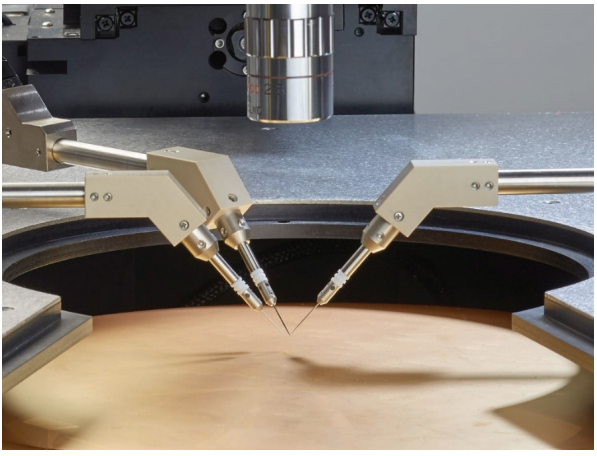


Рис. 1.72. Установка для микрозондового анализа[1]

С помощью микрозондов можно, например, считать данные непосредственно с шины декодера памяти. Физический размер микрозондов является существенным ограничением применения подобной техники исследования, но она поможет при исследовании кристаллов, произведенных по нормам в несколько сотен нанометров (к которым до сих пор относятся многие микроконтроллеры). Для проведения аналогичных исследований с кристаллами, выполненными на современных техпроцессах в десятки нанометров, используется установка с фокусированным ионным пучком, ФИП (Focused Ion Beam, FIB). Разрешающая способность подобных установок достигает 5 нм, в них сфокусированный луч ионов позволяет снять или нанести тончайший слой проводника в нужное место кристалла (например, вывести сигнал для подключения микрозондового пробника). Как вы уже поняли, мы дошли до самой «хардкорной» части инвазивных атак – модификации кристалла.

[1] https://blog.semiprobe.com/high-power-wafer-probe-station-success.

Модификация кристалла микросхемы

С помощью ФИП возможно вносить изменения непосредственно в структуру кристалла микросхемы (удалять существующие проводники или добавлять новые), тем самым изменяя логику работы схемы и получая доступ к сигналу в промежуточных участках схемы. Например, ФИП позволяет вывести сигнал между аппаратным блоком шифрования и процессорным ядром. Таким образом можно считать содержимое зашифрованной памяти, ведь ядро работает с расшифрованными данными. Еще одно применение, интересное исследователям цифровых устройств, – это восстановление структур, отвечающих за блокировку отладочных интерфейсов, т. е. удаление (или восстановление) битов защиты. ФИП является крайне мощным инструментом для проведения исследований, однако для его применения нужно знание структуры кристалла. На фото показаны примеры работы ФИП, два разорванных проводника и новое соединение с помощью напыления платины.

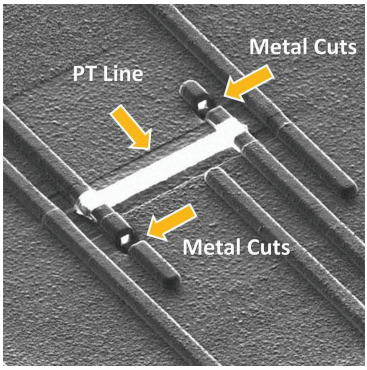


Рис. 1.73. Модификация кристалла микросхемы с помощью ФИП[1]

Хардкор – услуги на китайских форумах

Осталось рассказать про последний из описываемых способов получения прошивки цифровых устройств. На просторах интернета существуют услуги по считыванию прошивки из встроенной в микроконтроллер flash-памяти. Данная услуга предлагается в основном для распространенных моделей микроконтроллеров. Алгоритм выглядит следующим образом: вы демонтируете микросхему микроконтроллера с платы устройства и отправляете ее китайскому «подрядчику». В обмен на определенную сумму вам через несколько дней (или недель) присылают файл с содержимым flash-памяти микроконтроллера. Также могут прислать новый экземпляр микроконтроллера с прошивкой внутри и снятой защитой. Каким именно образом китайцы получают доступ к прошивке, доподлинно неизвестно. Возможно, за счет неинвазивных атак, а может быть, за счет инвазивных. В любом случае, учитывая их колоссальный опыт по клонированию устройств, данная услуга у них получается стабильно хорошо и не стоит значительных денег (от нескольких сотен долларов за 1 микроконтроллер).

[1] https://www.istgroup.com/en/service/ic-fib-circuit-edit/.

Level Up!

Вот мы и добрались до конца главы, посвященной различным способам получения прошивки устройства. Описанные в главе подходы не являются единственно верными, они лишь должны натолкнуть вас на ту цепочку предположений и экспериментов, которая позволит добыть прошивку исследуемого устройства. Естественно, не все показанные методы доступны большому кругу исследователей, но даже знание о теоретической возможности иногда может помочь. Например, найти нужных людей, имеющих подобное оборудование и готовых помочь с проведением исследования.

Исследователям встраиваемых систем, как и разработчикам, стоит получить больше знаний об аппаратных Side Channel и Fault Injection атаках, т. к. эта область постоянно развивается и появляются новые методы атак, например атака смещением базового напряжения на подложке (BBI, Body Bias Injection), имеющая ряд преимуществ перед EMFI/LFI-атаками (**https://www.researchgate.net/publication/311490102_Body_Biasing_Injection_Attacks_in_Practice**).

Некоторые виды аппаратных атак могут быть выполнены с помощью относительно недорого набора оборудования, доступного многим исследователям. Однако ключом к успеху проведения атаки являются понимание принципов работы атаки и повторяемость эксперимента. Производители устройств и регулирующие органы также в курсе подобных атак, поэтому устройства, где компрометация данных недопустима, проходят сертификацию криптографических модулей, например по федеральному стандарту США FIPS 140-2/3 (ISO/IEC 19790) или его аналогам. Среди разработчиков, в том числе встраиваемых систем, все чаще используются подходы безопасной разработки. Поэтому далеко не для всех устройств можно легко получить прошивку и исследователю необходимо постоянно поддерживать актуальность знаний о новых способах атак.

Знания, полученные в ходе добывания прошивки, обязательно должны обогатить тот информационный архив, который мы начали собирать в начале нашего пути – в главе «Level 0. Первичный анализ». Считаю, что у вас все получилось, прошивка имеется. Значит, пришло время приступать к самому интересному – к ее реверс-инжинирингу. Чтобы наконец понять, как работает исследуемое устройство на самом деле. Для начала мы попробуем статический реверс-инжиниринг прошивки. Пора узнать, какие бывают типы прошивок, как прошивка может быть устроена и как это связано с «железом» устройства.

Level 2

Начинаем статический анализ

Мы уже провели первичный анализ исследуемого устройства, собрали необходимую информацию и даже получили бинарный образ прошивки устройства. Конечно, понятно, что с ним делать дальше, – загружать в какой-нибудь дизассемблер. Но, в отличие от реверс-инжиниринга приложений под x86/64 (а я считаю, что с этим у читателя проблем нет), при реверс-инжиниринге прошивок цифровых устройств надо узнать кое-что про аппаратные особенности архитектуры микроконтроллера и про самые распространенные подходы к организации прошивок. Скорее всего, часть этой информации вам так или иначе уже известна, не зря же вы читаете эту книгу. Но что-то новое, я уверен, для себя найдет каждый. Подробнее рассмотрим, как устроен микроконтроллер и из чего он состоит.

В начале книги мы уже рассматривали, чем микроконтроллер отличается от микропроцессора. Микроконтроллер состоит из вычислительного ядра, ОЗУ, скорее всего, ПЗУ и набора интерфейсных контроллеров, позволяющих подключать к ядру внешние устройства посредством стандартных интерфейсов (I2C, SPI и т. д.).

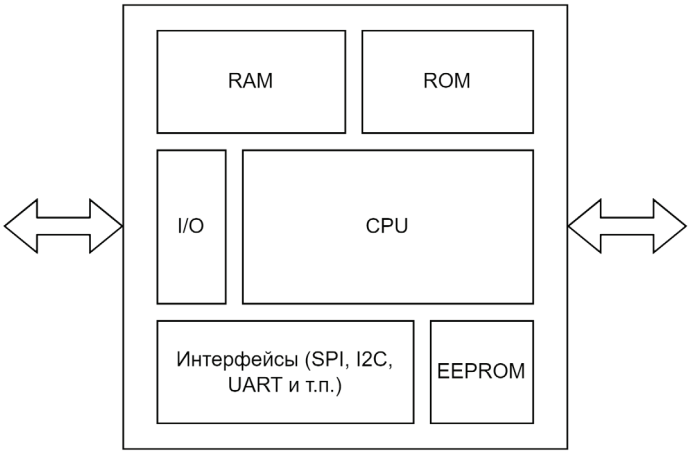


Рис. 2.1. Общая схема микроконтроллера

Перед тем как перейти к распространенным архитектурам ядра микроконтроллера и системам команд, будет полезно узнать немного истории про развитие электронных вычислительных машин. Существуют два классических подхода к построению архитектуры ЭВМ: гарвардский и фон Неймана (пристонский).

Архитектурные подходы проектирования ЭВМ

Гарвардская архитектура

Гарвардская архитектура была разработана в 1930-е годы Говардом Эйкеном в Гарвардском университете для компьютера Марк-1. Гарвардская архитектура подразумевала разные хранилища для данных и инструкций, а также разные каналы их передачи. В компьютере с использованием гарвардской архитектуры процессор может считывать очередную команду и оперировать памятью данных одновременно и без использования кеш-памяти. Исходя из физического разделения шин команд и данных, разрядности этих шин могут различаться.

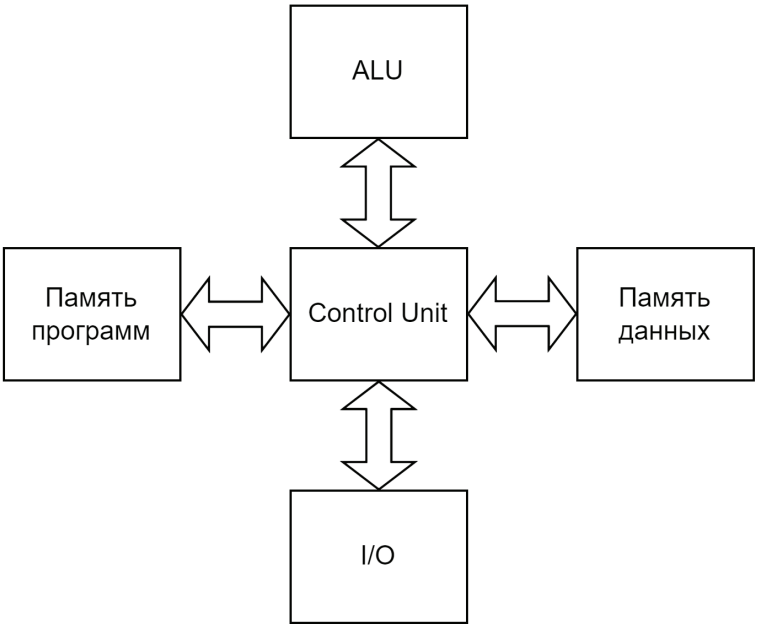


Рис. 2.2. Схема гарвардской архитектуры ЭВМ

Такой подход позволял одновременно считывать команду из программы и данные из памяти, что вело к значительному увеличению общей производительности компьютера. Но в то же время такая схема сильно усложняет реализацию системы и увеличивает ее стоимость, т. к. при разделении каналов передачи команд и данных на кристалле процессора последний должен иметь почти вдвое больше интерфейсных выводов.

Архитектура фон Неймана

Основы учения об архитектуре вычислительных машин заложил Джон фон Нейман в 1944 году, когда подключился к созданию первого в мире лампового компьютера ЭНИАК, разрабатываемого в Институте Мура в Пенсильванском университете.

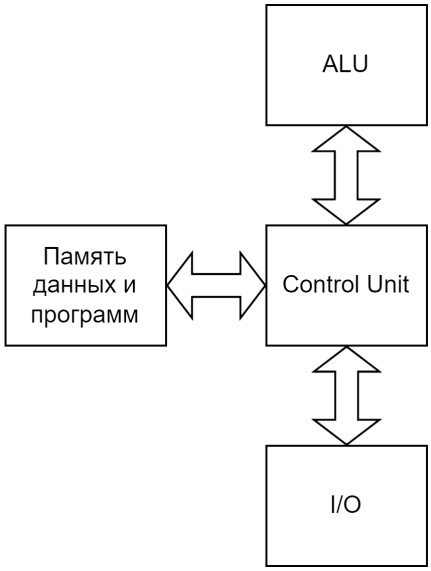


Рис. 2.3. Схема архитектуры ЭВМ фон Неймана

Архитектура фон Неймана строилась на следующих принципах:

- *принцип однородности памяти*. Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Распознать их можно только по способу использования; то есть одно и то же значение в ячейке памяти может использоваться и как данные, и как команда, и как адрес в зависимости лишь от контекста обращения к нему. Это позволяет производить над командами те же операции, что и над числами, и, соответственно, открывает ряд возможностей. Так, циклически изменяя адресную часть команды, можно обеспечить обращение к последовательным элементам массива данных;
- *принцип адресности*. Структурно основная память состоит из пронумерованных ячеек, причем процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые словами, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек – адреса.

Основным недостатком этой архитектуры является ограничение пропускной способности между памятью и процессором. Из-за того, что программа и данные не могут считываться одновременно, пропускная способность между памятью и процессором существенно ограничивает скорость работы про-

цессора. В дальнейшем данную проблему решили с помощью введения кеша, что вызвало другие проблемы (например, ставшие широко известными в последнее время уязвимости спекулятивного выполнения, например Meltdown and Spectre, **https://meltdownattack.com/**).

Модифицированная гарвардская архитектура

Гарвардская схема реализации доступа к памяти имеет один очевидный недостаток – высокую стоимость, т. к. процессор должен иметь почти вдвое больше интерфейсных выводов. Способом разрешения этой проблемы стала идея использовать общую шину данных и шину адреса для всех внешних данных, а внутри процессора применять шину данных, шину команд и две шины адреса. Такую концепцию стали называть модифицированной гарвардской архитектурой.

Подобный подход применяется в современных цифровых сигнальных процессорах (Digital Signal Processor, DSP). Еще дальше по пути уменьшения стоимости пошли при создании многих современных микроконтроллеров. В них одна шина команд и данных применяется и внутри кристалла. Разделение шин в модифицированной гарвардской структуре осуществляется при помощи раздельных управляющих сигналов: чтения, записи или выбора области памяти.

В конце концов, после множества улучшений и убирания «бутылочных горлышек» в архитектурных подходах современные процессоры и микроконтроллеры имеют гибридную архитектуру, в которой есть черты как архитектуры фон Неймана, так и гарвардской.

Системы команд RISC и CISC

Микропроцессоры первых поколений обладали жестко «прошитой» логикой декодирования команд. В то время технология создания запоминающих устройств сильно уступала технологии изготовления процессоров, то есть более скоростным процессорам приходилось долго ожидать считывания следующей команды для декодирования из медленно действующего запоминающего устройства. Поэтому с целью эффективного использования этого периода ожидания возникла идея заполнить время до поступления следующей команды. На помощь пришла идея реализовать вычислительный конвейер, ведь исполнение каждой инструкции можно разделить на несколько частей-операций: получение, декодирование, вычисление, запись результата. И если результаты выполнения инструкций не связаны между собой, можно одновременно выполнять разные операции для разных команд, как бы организуя конвейер и сокращая суммарное время выполнения последовательности инструкций. Естественно, что для поддержки подобных архитектурных решений потребовалось дорабатывать компиляторы, которые также со временем научились генерировать машинный код, максимально эффективно задействующий новые архитектурные разработки.

Следующим шагом в развитии стала разработка комплексных машинных команд, поддерживаемых процессорами CISC (Complex Instruction Set

Computer – компьютер со сложным набором команд). Эти команды выполняли последовательно несколько внутрипроцессорных операций до тех пор, пока не поступала следующая внешняя команда. Развитие компиляторов привело к тому, что компиляторы могли преобразовывать конструкции высокоуровневых языков программирования как в простые машинные инструкции, так и в эффективную последовательность комплексных инструкций, которую было сложно реализовать человеку при разработке на языке низкого уровня. Типичными представителями архитектуры процессоров CISC являются семейства 80x86 компании Intel или семейства 680x0 компании Motorola. Практически все процессоры CISC работают по принципу микрокода, то есть каждая машинная команда обрабатывается микропрограммой, которая выполняется внутри кристалла процессора. Однако в реальной жизни получилась ситуация, при которой из большого набора команд процессоров CISC (у некоторых типов – свыше нескольких сотен команд) используется только небольшая, постоянно повторяющаяся часть в размере около 20 %. В дополнение к этому комплексную команду можно зачастую заменить несколькими эффективными командами, которые в состоянии справиться с поставленной задачей быстрее. В середине 1980-х годов произошел возврат «к истокам» и была разработана архитектура RISC (Reduced Instruction Set Computer, компьютер с набором упрощенных/редуцированных команд) – возникли компьютеры с сокращенным набором команд, у которых команды, как и в процессорах первых поколений, снова декодировались посредством аппаратной логики.

В настоящее время многие архитектуры процессоров являются RISC-подобными, к примеру ARM, ARC, SPARC, AVR, MIPS и PowerPC. Наиболее широко используемые в настольных компьютерах процессоры архитектуры x86 ранее являлись CISC-процессорами, однако новые процессоры, начиная с Intel Pentium Pro, являются CISC-процессорами с RISC-ядром. Они непосредственно перед исполнением преобразуют CISC-инструкции x86-процессоров в более простой набор внутренних инструкций RISC. После того как процессоры архитектуры x86 были переведены на суперскалярную RISC-архитектуру (т. е. способны выполнять несколько инструкций одновременно), можно сказать, что большинство существующих ныне процессоров (и, соответственно, микроконтроллеров) основаны на архитектуре RISC. Для читателей, желающих более глубоко посмотреть в особенности реализации современного микрокода на примере микрокода процессоров Intel, рекомендую посмотреть презентацию CHIP RED PILL: HOW WE ACHIEVED THE ARBITRARY [MICRO]CODE EXECUTION INSIDE INTEL ATOM CPUS на конференции OffensiveCon22 (**https://www.offensivecon.org/speakers/2022/maxim-goryachy.html**).

Архитектура, микроархитектура и система команд

Кратко мы уже рассмотрели, какие существуют архитектурные подходы к проектированию ЭВМ, а также чем виды системы команд CISC отличаются от RISC. В общем виде архитектуру ЭВМ можно представить в виде комбинации микроархитектуры, микрокода и архитектуры системы команд (Instruction Set Architecture, ISA):

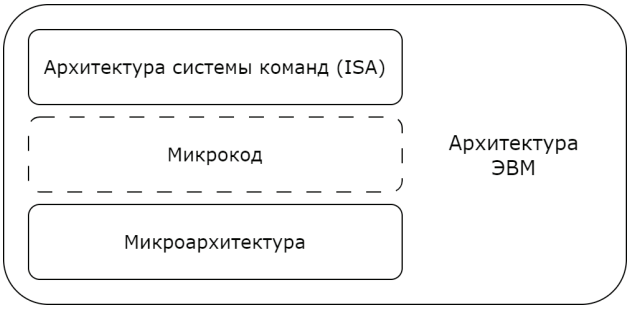


Рис. 2.4. Иерархия архитектуры ЭВМ

ISA определяет программируемую часть ядра процессора, т. е. включает в себя следующие основные части:

- машинные команды;
- модель исполнения;
- архитектуру памяти;
- регистры процессора;
- форматы адресов и данных, режимы адресации;
- обработчики прерываний и исключительных состояний.

На самом нижнем уровне архитектуры располагается микроархитектура – способ, которым данная ISA реализована в процессоре. Микроархитектура включает составные части процессора и способы их взаимосвязи и взаимодействия для реализации ISA. На этом уровне определяются:

- конструкция и взаимосвязь основных блоков процессора;
- структура ядер, исполнительных устройств, АЛУ, а также их взаимодействия
- организация блоков предсказания переходов и конвейеров;
- организация кеш-памяти;
- взаимодействие с внешними устройствами.

В рамках одного семейства микропроцессоров микроархитектура со временем расширяется путем добавления новых усовершенствований и оптимизации существующих команд с целью повышения производительности, энергосбережения и функциональных возможностей микропроцессора. При этом сохраняется совместимость с предыдущей версией ISA. Однако стоит помнить, что микроархитектура не определяет физический уровень микроконтроллера, т. е. структуры на кристалле.

В некоторых случаях работа элементов микроархитектуры контролируется микрокодом, встроенным в процессор. В случае наличия слоя микрокода в архитектуре процессора он выступает своеобразным интерпретатором, преобразуя команды уровня ISA в команды уровня микроархитектуры. Самым известным примером архитектур с микрокодом является архитектура x86-64, лежащая в основе большинства современных процессоров Intel и AMD.

Распространенные архитектуры

ARM

Advanced RISC Machine – усовершенствованная RISC-машина. На данный момент это самая распространенная архитектура, встречающаяся в embeded-устройствах. Впервые представленная в 1985 году, она быстро приобрела популярность за свою простоту и низкое энергопотребление. В настоящее время процессоры и микроконтроллеры с архитектурой ARM стоят практически во всех устройствах интернета вещей, мобильных телефонах, планшетах и даже некоторых серверах. Компания-разработчик ARM Limited является крупнейшей в мире бесфабричной (fabless) компанией, проектирующей и лицензирующей архитектуры 32-разрядных и 64-разрядных ARM-процессоров. То есть сама компания ARM не производит процессоры и микроконтроллеры. Она разрабатывает архитектуру микропроцессорного ядра и продает лицензии на использование своих разработок. Выделяется несколько семейств ядер ARM:

- ARM7 (с тактовой частотой до 100 МГц), предназначенные, например, для встраиваемых решений средней производительности. В настоящее время активно вытесняется новым семейством Cortex;
- ARM9, ARM11 (с частотами до 1 ГГц) для телефонов, карманных компьютеров и встраиваемых решений высокой производительности;
- Cortex A – семейство процессоров на смену ARM9 и ARM11;
- Cortex M – семейство процессоров на смену ARM7, также призванное занять новую для ARM нишу встраиваемых решений низкой производительности;
- Cortex R – семейство процессоров, ядра которых оптимизированы для выполнения приложений реального времени и критически важных применений.

Стандартным для ARM является 32-битный набор команд (ARM Base Instruction Set). Для улучшения плотности кода процессоры, начиная с ARM7TDMI, снабжены режимом «Thumb». В этом режиме процессор поддерживает альтернативный набор 16-битных команд. Большинство из этих 16-разрядных команд переводятся в 32-битные команды ARM. Уменьшение длины команды достигается за счет сокрытия некоторых операндов и ограничения возможностей адресации по сравнению с режимом полного набора команд ARM.

В режиме Thumb коды операций обладают меньшей функциональностью, многие коды операций имеют ограничение в виде доступа только к половине главных регистров процессора. Более короткие коды операций в целом дают большую плотность кода, хотя некоторые операции требуют дополнительных команд. В конце 2011 года была опубликована версия архитектуры ARMv8. В ней появилась поддержка 64-битного набора команд (A64). А в 2021 году появилась самая последняя на момент написания книги архитектура ARMv9.

Важный момент, касающийся не только ARM, но и архитектур в общем, – это порядок байтов (анг. Endianness), применяющийся для хранения последовательности байтов (например, слова) в памяти. Порядок бывает от старшего к младшему (BE, англ. big-endian – с большого конца) или от младшего к стар-

шему (LE, англ. little-endian – с малого конца). При порядке BE самый старший байт последовательности хранится по младшему адресу, а младший байт – по старшему адресу:

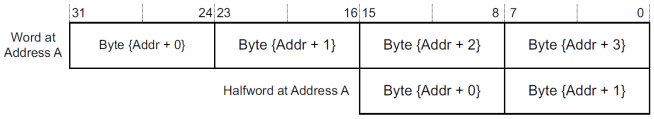


Рис. 2.5. big-endian порядок байтов[1]

При порядке LE ситуация прямо противоположная:

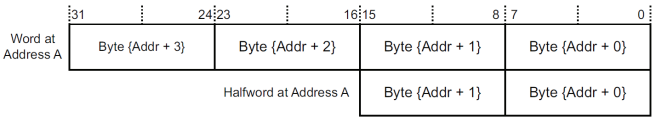


Рис. 2.6. little-endian порядок байтов[2]

Одна и та же архитектура может иметь разный порядок байтов в зависимости от реализации в конкретном микроконтроллере. Неправильное определение порядка байтов является одной из ошибок при попытке загрузки прошивки в дизассемблер.

На примере архитектуры ARM рассмотрим интересный механизм – предикацию, т. е. возможность условного выполнения команд. Под «условным исполнением» здесь понимается то, что команда будет выполнена или проигнорирована в зависимости от текущего состояния флагов состояния процессора. В то время как для других архитектур таким свойством, как правило, обладают только команды условных переходов, в архитектуру ARM была заложена возможность условного исполнения практически любой команды. Это было достигнуто добавлением в коды их инструкций особого 4-битового поля Condition (предиката). Одно из его значений зарезервировано на то, что инструкция должна быть выполнена безусловно, а остальные кодируют то или иное сочетание условий (флагов).

На рисунке ниже показан формат команд ARMv7 в 32-битном режиме.

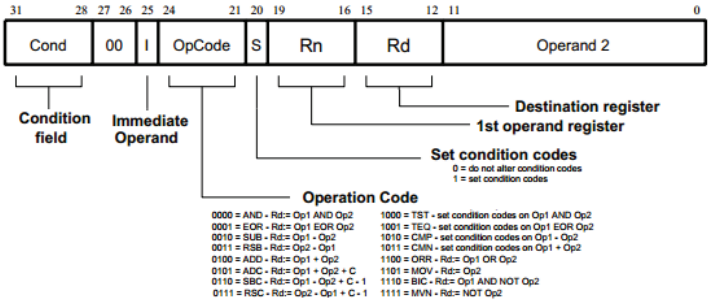


Рис. 2.7. Структура команд ARMv7 в 32-битном режиме[3]

[1] https://developer.arm.com/documentation/ddi0419/c.

[2] https://developer.arm.com/documentation/ddi0419/c.

[3] https://developer.arm.com/documentation/ddi0406/latest/.

С одной стороны, с учетом ограниченности общей длины инструкции реализация условного выполнения сократила число битов, доступных для кодирования смещения в командах обращения к памяти, но с другой – позволила избавиться от инструкций ветвления при генерации кода для небольших if-блоков. В режиме Thumb предикация не используется. Кстати, именно удаление 4-битового предиката из всех инструкций, кроме ветвлений, является одним из способов, которым Thumb-код достигает большей экономии объема кода.

Важной особенностью любой архитектуры является соглашение о вызове (англ. calling convention) – описание технических особенностей вызова подпрограмм, определяющее:

- способы передачи параметров подпрограммам;
- способы вызова (передачи управления) подпрограмм;
- способы передачи результатов вычислений, выполненных подпрограммами, в точку вызова;
- способы возврата (передачи управления) из подпрограмм в точку вызова.

Для архитектуры ARM они описаны в документе «Procedure Call Standard for the Arm® Architecture» (**https://github.com/ARM-software/abi-aa/releases/download/2022Q3/aapcs32.pdf**). Например, в нем указаны 16 регистров процессора, имеющих определенное назначение при вызове подпрограммы, которое должны учитывать компиляторы при создании кода:

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	Frame Pointer or Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Рис. 2.8. Назначение регистров ARM при вызове процедур[1]

[1] https://github.com/ARM-software/abi-aa/releases/download/2022Q3/aapcs32.pdf.

На примере ARM мы рассмотрели некоторые особенности, актуальные и для других архитектур. Несмотря на то что ARM является самой распространенной архитектурой в мире встраиваемых систем, существует множество других архитектур, про которые надо знать.

ARC

Argonaut RISC Core – семейство архитектур 32-разрядных процессоров, первоначально разработанное ARC International (впоследствии выкупленной компанией Synopsys). Как и ARM Limited, компания Synopsys (ARC International) является fabless-компанией, продающей лицензии на использование своих процессорных ядер. Процессоры ARC имеют систему команд RISC и используют 16/32-битный набор инструкций.

Отличительной особенностью процессорных ядер ARC является гибкая среда конфигурирования ядра микропроцессора, чтобы адаптировать каждый экземпляр процессора ARC для удовлетворения конкретных требований к производительности, энергопотреблению и применению. Это стало возможным благодаря расширяемому ядру. В отличие от большинства встраиваемых микропроцессоров, в процессор ARC можно добавить дополнительные инструкции, регистры и функциональные возможности, используя подобие блочной системы. Клиенты могут выбирать соответствующие расширения, чтобы создать собственный микропроцессор, вплоть до расширения системы команд. Отличительной особенностью архитектуры ARC является низкое энергопотребление. А также весьма дешевая лицензия на использование архитектуры.

Процессоры, построенные на базе архитектуры ARC, встречаются во многих потребительских устройствах:

- в сетевых устройствах хранения данных (NAS);
- в твердотельных накопителях (SSD);
- в сетевых устройствах;
- в телеприставках;
- в автомобильных системах управления;
- в устройствах IoT;
- в старых поколениях сопроцессора Intel Management Engine.

Как видно из перечня, спектр применения микропроцессоров с ядром ARC максимально широк (каждый год выпускается почти 2 млрд микросхем).

MIPS

Microprocessor without Interlocked Pipelined Stages – система команд и микропроцессорных RISC-архитектур (32- и 64-битных), разработанных компанией MIPS Computer Systems. Архитектура MIPS проприетарная, но в 2018–2019 годах была предпринята попытка ее свободного распространения, от которого впоследствии отказались. На базе микропроцессорных ядер с архитектурой MIPS за 35 лет было выпущено множество специализированных микроконтроллеров.

В 1990-е годы архитектура MIPS была широко распространена во встраиваемых системах и применялась в телекоммуникационном оборудовании, моде-

мах, игровых консолях (в том числе Sony PlayStation), телевизорах. В настоящее время микроконтроллеры с архитектурой MIPS применяются в сетевых маршрутизаторах, автомобильных блоках управления (например, японской компании Denso), промышленных контроллерах и контроллерах мультимедиаустройств. На базе архитектуры MIPS разработаны российские микропроцессоры Baikal-T1 и КОМДИВ (конвейерный однокристальный микропроцессор для интенсивных вычислений).

Наиболее распространенными ОС, портированными для архитектуры MIPS и встречающимися во встраиваемых системах, являются Linux, QNX и VxWorks, реже встречаются устройства на Windows CE, особенно в современном мире.

PowerPC

Архитектура PowerPC (сокращенно PPC) появилась в начале 1990-х годов в результате работы альянса компаний Apple, IBM и Motorola. В какой-то момент она даже могла потеснить архитектуру Intel x86/64 с рынка персональных компьютеров и суперкомпьютеров. Компьютеры Apple до середины 2000-х годов были построены на базе процессоров с архитектурой PPC. Изначально 32-битная, архитектура со временем стала иметь разрядность 64 бита.

Во встраиваемых системах широко применяются микроконтроллеры фирмы Freescale, построенные на базе архитектуры PPC. Их можно встретить в автомобильных блоках управления и мультимедийных устройствах, промышленных контроллерах и в телеком-оборудовании.

MCS-51 (Intel 8051)

Широко распространенная архитектура 8-битных микроконтроллеров, применяющихся в большом количестве устройств, особенно в различных контроллерах интерфейсов. Одна из немногих CISC-архитектур, получивших распространение в мире встраиваемых систем. Прародителем архитектуры является микроконтроллер Intel 8051, выпущенный в 1980 году. Примечательно, что изначальная частота работы оригинального процессора составляла 12 МГц, в то время как современные реализации микроконтроллеров на данной архитектуре могут работать на частотах в десятки и даже сотни мегагерц.

Периферийные блоки оригинальной архитектуры микроконтроллера были весьма простыми (по современным меркам): несколько портов ввода-вывода и интерфейс UART. Современные реализации могут иметь максимально широкий спектр поддерживаемых интерфейсов (I2C, SPI, LCD, USB), аппаратные блоки ускорения вычислений и DMA. Если микроконтроллер должен часто выполнять какие-то затратные по времени выполнения операции, они могут быть реализованы в виде аппаратного блока. К примеру, операция деления 2 DWORD'ов на 8-битном микроконтроллере будет занимать не один десяток инструкций и выполняться с существенными временными затратами. При ее реализации в виде аппаратного блока она может быть выполнена за несколько тактов работы микроконтроллера.

В большинстве устройств, построенных на базе архитектуры MCS-51, в основе прошивки не используется ОС, а применяется так называемый bare-metal подход (подходы к организации прошивок будут рассмотрены в этой главе немного позднее).

Микроконтроллеры с архитектурой MCS-51 с большой вероятностью можно встретить в следующих классах устройств:

- преобразователи интерфейсов (USB-UART, USB-SATA и т. д.);
- контроллеры запоминающих устройств;
- устройства ввода-вывода информации (клавиатуры, LCD-панели и т. п.);
- сервисные контроллеры, управляющие электропитанием и мониторингом состояния материнских плат различных устройств.

AVR

Изначально восьмибитная RISC-архитектура микроконтроллеров фирмы Atmel (на данный момент принадлежащая компании Microchip). Существует также менее распространенная архитектура 32-битных микроконтроллеров Atmel AVR32. Микроконтроллеры на базе архитектуры AVR часто встречаются во встраиваемых системах на позициях вспомогательных контроллеров управления периферией, взаимодействующих с основным управляющим микроконтроллером устройства, т. е. выполняют роль расширителей портов ввода-вывода или контроллеров интерфейсов.

Широкую известность контроллеры семейства Atmel AVR получили с выходом платформы для DIY-разработчиков электронных устройств Arduino. В основе первых поколений Arduino-плат находились именно микроконтроллеры семейства Atmel AVR, а впоследствии были добавлены микроконтроллеры других производителей, в том числе 32-битные. Для исследователей устройств это может быть важно, т. к. некоторые серийно выпускаемые устройства могут быть основаны на платах Arduino. В том числе существуют даже промышленные контроллеры, построенные на базе Arduino.

PIC

Семейство архитектур компании Microchip. В семейство входят 8-, 16- и 32-битные микроконтроллеры (правда, 32-битные контроллеры на самом деле имеют архитектуру MIPS). Аббревиатура PIC расшифровывается как Peripheral Interface Controller, т. е. контроллер периферии. Большинство применений контроллеров с архитектурой PIC как раз связаны с обработкой периферийных интерфейсов устройства и взаимодействием с основным управляющим микроконтроллером устройства. То есть применение микроконтроллеров PIC схоже с микроконтроллерами Atmel AVR. Однако модельный ряд микроконтроллеров с архитектурой PIC поистине огромен. Фактически существует огромное количество похожих модификаций микроконтроллеров, отличающихся типом поддерживаемых периферийных интерфейсов. За счет некой потери унификации (к которой стремятся другие производители микроконтроллеров) и широкого модельного ряда микроконтроллеры PIC, как правило, имеют более низкую цену по сравнению с аналогами, что делает их хорошими кандидатами на применение в серийно выпускаемых устройствах.

Xtensa

Изначально разработанная компанией Tensilica и впоследствии купленная компанией Cadence 32-битная архитектура, совместимая с 16- и 24-битной

системами команд. Ядро микропроцессора с данной архитектурой широко применяется в процессорах для обработки цифровых сигналов (Digital Signal Processors, DSP). DSP широко используется в современных мультимедиаустройствах, смартфонах, очках виртуальной реальности. Во встраиваемых устройствах очень широкое распространение получили Wi-Fi-SOC Espressif ESP32, построенные на базе архитектуры Xtensa (кроме версий Espressif ESP32-C3, Espressif ESP32-C6 и более новых, построенных на базе архитектуры RISC-V).

RISC-V

Стремительно набирающая распространение Open Source RISC-архитектура и система команд, доступная для свободного и бесплатного использования, включая коммерческие реализации непосредственно в микроконтроллерах или ПЛИС. Ее разработка началась в 2010 году как исследовательский проект в Калифорнийском университете Беркли в США. В отличие от других академических проектов, которые обычно сосредоточены на простоте и образовательных целях, набор команд RISC-V сразу проектируется для широкого круга применений.

С 2017 года были выпущены десятки микропроцессоров и микроконтроллеров на базе архитектуры RISC-V, преимущественно для систем хранения и обработки данных. Скорее всего, в дальнейшем доля микроконтроллеров, построенных на базе RISC-V, может существенно увеличиться. Исследователь встраиваемых систем может встретить эту архитектуру в микроконтроллерах Espressif ESP32-C3, Espressif ESP32-C6 и более новых. Они широко применяются в различных IoT-устройствах, т. к. помимо микропроцессорного ядра содержат в себе модули связи Wi-Fi и Bluetooth LE 5.0. На базе архитектуры RISC-V были разработаны и выпущены микроконтроллеры российских фирм Миландр и Микрон.

Процесс разработки и производства микроконтроллера

Основой любого микроконтроллера является вычислительное ядро, это мы уже выяснили. В большинстве случаев нет смысла разрабатывать новую архитектуру ядра, можно купить лицензию на использование уже готовой технологии. В мире существует несколько компаний, специализирующихся на разработке архитектур процессорных ядер. Как мы упомянули ранее, самой известной является ARM Limited. Хорошо, ядро у нас есть, но этого недостаточно. К ядру надо подключить различные аппаратные блоки для ускорения вычислений, организации информационного обмена и т. д. Разработкой дизайна таких блоков можно заняться самостоятельно, а можно купить у компаний, специализирующихся на подобных работах (например, Synopsys), ведь ими уже разработано большое количество блоков. И дизайн самого микропроцессорного ядра, и дизайн периферийных аппаратных блоков называется IP-ядрами (Intellectual Property). В результате сложной работы по интеграции IP-ядер между собой получается дизайн микроконтроллера, описанный на уровне Register Transfer Level (RTL). Данный дизайн все еще не описывает расположение конкретных структур на кристалле. Нужно как-то превратить RTL в описание физических элементов на кристалле.

Каждая фабрика, производящая кристалл, имеет свою библиотеку стандартных ячеек (Standard Cell Library), описывающих, как реализуются стандартные элементы (логические функции, буферы, триггеры, регистры и т. п.) на кристалле.

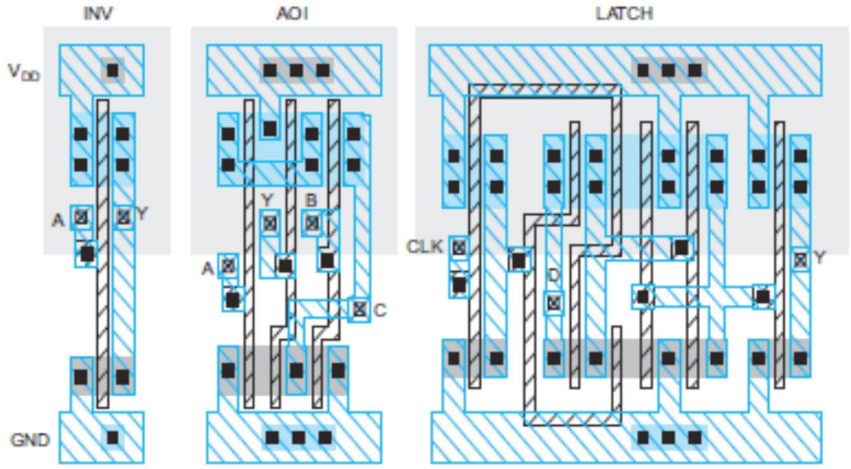


Рис. 2.9. Пример структур Standard Cell Library[1]

Далее следует целая последовательность процессов, получающих на вход RTL, файл с описанием временных характеристик всех сигналов (Design Constraints) и библиотеку стандартных ячеек.

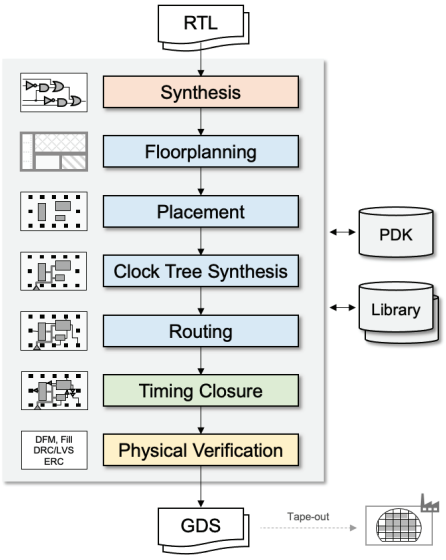


Рис. 2.10. Последовательность процессов разработки описания кристалла[2]

[1] https://slideplayer.com/slide/4473261/.

[2] https://www.anyscale.com/blog/infusing-ai-and-ml-into-integrated-circuit-design-for-faster-chip-delivery.

Выходом этих процессов является описание всех структур кристалла в виде векторных изображений в специальном формате (например, GDS-II). Пример структуры кристалла, описанной с помощью формата GDS-II и открытой в программе-редакторе, показан на рисунке:

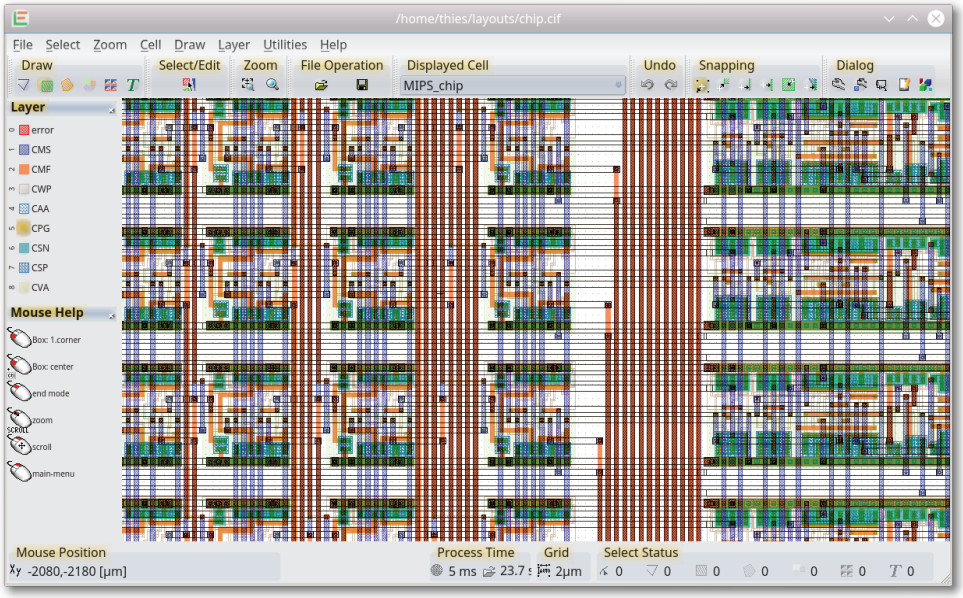


Рис. 2.11. Пример файла со структурой кристалла[1]

Это описание можно отдавать на производство (правда, только на то, что предоставило свою библиотеку стандартных ячеек) и получить готовый кристалл. Производством кристаллов, как правило, занимаются отдельные компании, владеющие фабриками (например, TSMC). После получения кристалла его надо корпусировать, т. е. упаковать в корпус, пригодный для монтажа и защищающий кристалл от воздействий окружающей среды. Часто подобные услуги предлагают на тех же фабриках, где кристалл производится, однако существуют компании, специализирующиеся только на этой операции.

Вот такая интересная и тесная интеграция нескольких компаний требуется для производства даже простого микроконтроллера. Конечно, в мире существует компания, способная разработать все сама – от архитектуры ядра до выпуска готового микроконтроллера (или процессора). Угадали название этой компании? Верно, это Intel. Но тот факт, что у Intel есть ресурсы для полного цикла разработки и производства, не значит, что они не пользуются услугами сторонних компаний на этом рынке. Например, часть выпускаемых Intel микропроцессоров построена на базе архитектур ARC и ARM. На диаграмме ниже показана «пирамида Маслоу» экосистемы производства электронных компонентов.

[1] https://www.layouteditor.org/.

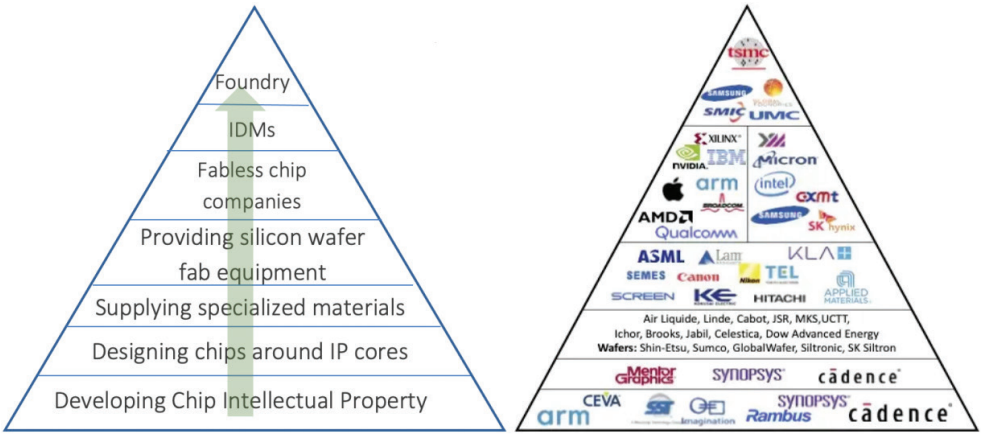


Рис. 2.12. Основные фирмы, занятые в производстве радиоэлектронных компонентов[1]

Более детальное описание процесса производства и участвующих в этом процессе компаний можно найти на странице **www.semiwiki.com/eda/307494-the-semiconductor-ecosystem-explained/**.

Определение архитектуры и системы команд ядра микроконтроллера

Для корректной загрузки исследуемой прошивки в дизассемблер необходимо знать систему команд микропроцессора, соответствующую какому-то вычислительному ядру. Ранее мы рассмотрели, что большинство применяемых в современных цифровых устройствах микроконтроллеров имеют ядро на RISC-архитектуре. Но как можно определить тип ядра и систему команд? Естественно, уже придуманы утилиты, позволяющие определить, где в образе находится код и какой системе команд он соответствует. Самой известной и незаменимой утилитой этого класса является binwalk (**https://github.com/ReFirmLabs/binwalk**). Она обладает обширными возможностями по определению участков кода, известных констант, в том числе криптографических, ключей и т. п. Все возможности не перечислить, лучше один раз прочитать документацию и понять, что binwalk способна существенно упростить жизнь исследователю цифровых устройств. К тому же она может быть интегрирована в дизассемблер IDA Pro в виде плагина, что повышает удобство работы. Пример вывода binwalk в виде плагина для дизассемблера IDA Pro показан на рисунке:

[1] www.semiwiki.com/eda/307494-the-semiconductor-ecosystem-explained/.

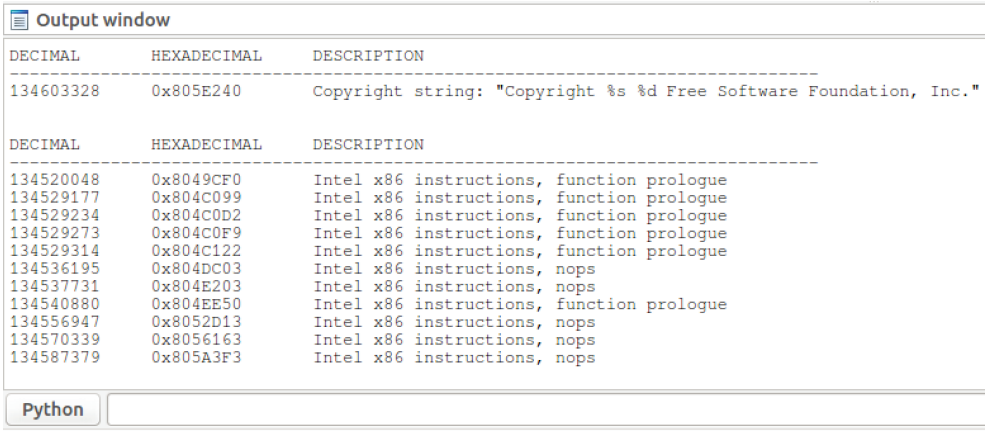


Рис. 2.13. Результат работы плагина binwalk для IDA Pro[1]

Однако еще лучше это делает утилита cpu_req (**https://github.com/airbus-seclab/cpu_rec**), которая может работать отдельно или использоваться как плагин binwalk'a. Она поддерживает почти все распространенные архитектуры и системы команд (более 70!). Пример с определением нескольких архитектур показан на рисунке:

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             None (size=0x1800, entropy=0.156350)
6144          0x1800          PPCeb (size=0x1b800, entropy=0.772708)
118784        0x1D000         None (size=0xd000, entropy=0.588620)
172032        0x2A000         X86 (size=0x2000, entropy=0.594146)
180224        0x2C000         None (size=0x800, entropy=0.758712)
182272        0x2C800         X86-64 (size=0x800, entropy=0.767427)
184320        0x2D000         X86 (size=0x18800, entropy=0.786143)
284672        0x45800         None (size=0xc000, entropy=0.612610)
```

Рис. 2.14. Результат работы плагина cpu_rec[2]

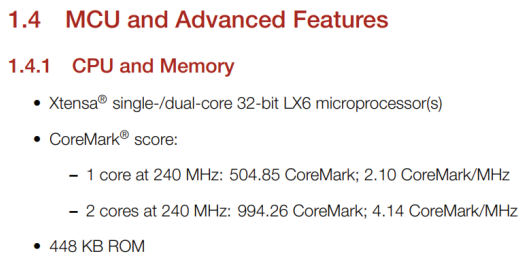
В случае если мы анализируем бинарный файл какого-то стандартного формата (например, ELF), дизассемблер может сам определить тип используемой архитектуры на основе парсинга заголовка. Во многих встраиваемых устройствах под управлением linux на этом проблема с определением архитектуры решена.

Определить архитектуру и систему команд с помощью описанных утилит получается не всегда. Например, если прошивка зашифрована или упакована (хотя binwalk умеет определять стандартные упаковщики и архиваторы). Что делать в этом случае, мы рассмотрим немного позднее. А пока попробуем определить тип архитектуры вручную. Рассмотрим, какие есть подходы и на чем они основаны.

[1] https://github.com/ReFirmLabs/binwalk.
[2] https://github.com/airbus-seclab/cpu_rec.

По документации на чип

Самый очевидный и простой путь – если, конечно, документация доступна. Если в ходе выполнения первичного анализа мы получили информацию о модели микроконтроллера (по найденным даташитам, рекламным материалам или просто по маркировке на корпусе), то вероятность найти информацию о типе ядра сильно повышается. Например, легко найти в документации на микроконтроллер Espressif ESP32, что внутри одно или два процессорных ядра на базе архитектуры Xtensa:

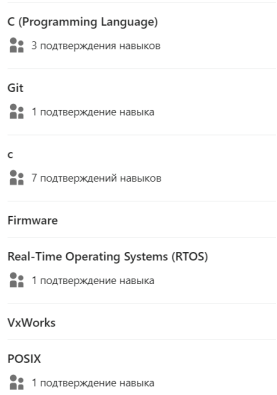

1.4 MCU and Advanced Features

1.4.1 CPU and Memory

- Xtensa® single-/dual-core 32-bit LX6 microprocessor(s)
- CoreMark® score:
  - 1 core at 240 MHz: 504.85 CoreMark; 2.10 CoreMark/MHz
  - 2 cores at 240 MHz: 994.26 CoreMark; 4.14 CoreMark/MHz
- 448 KB ROM

Рис. 2.15. Фрагмент документации на микроконтроллер ESP32[1]

Возможно, информации именно на искомый чип найти не получится, но исходя из контекста поиска станет ясно, к какой серии чипов он относится. Опять напомню про широту подходов при поиске информации: если быстро нужную информацию найти не удалось, стоит посмотреть под другим углом. Например, посмотреть, какие навыки указывают сотрудники производителя исследуемого устройства на своих страницах в LinkedIn. Так вы имеете шанс не только найти информацию о типе ядра микроконтроллера, но и получить полезную информацию о применяющихся в компании библиотеках и операционных системах (ведь прошивка устройства почти никогда не пишется с «нуля»). На рисунке ниже показана часть страницы профиля LinkedIn инженера в одной из известных компаний-разработчиков систем хранения данных. Как думаете, возможно ли, что в прошивке этих устройств будет ОС VxWorks?


C (Programming Language)
3 подтверждения навыков

Git
1 подтверждение навыка

c
7 подтверждений навыков

Firmware

Real-Time Operating Systems (RTOS)
1 подтверждение навыка

VxWorks

POSIX
1 подтверждение навыка

Рис. 2.16. Фрагмент страницы профиля на LinkedIn с описанием навыков

[1] https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

По кодам основных инструкций

Допустим, чип микроконтроллера попался проприетарный, т. е. сделан на заказ для производителя устройства и никакой информации о нем, кроме маркировки, у вас нет. Или даже маркировка не нанесена, это ничего не меняет. Как же тогда выбрать правильную архитектуру и систему команд при загрузке в дизассемблер? Конечно, можно перебрать все возможные варианты, но количество поддерживаемых процессорных ядер в той же IDA Pro довольно велико и на эту процедуру может уйти не один час. Можно воспользоваться утилитами binwalk или cpu_req, а можно попробовать сделать анализ вручную, заодно разобравшись, что делают указанные утилиты. Если знать значение кода основных машинных инструкций для распространенных архитектур, можно сделать предположение о типе набора инструкции и архитектуре за несколько секунд.

Коды каких ассемблерных инструкций нужно знать? Как минимум команд передачи и возврата управления, записи в регистры. Осталось только посмотреть значения байтов в дампе прошивок и попытаться соотнести логику их расположения с предполагаемыми инструкциями. Проще всего проиллюстрировать это на примере простой 8-битной архитектуры MCS-51. Значения кодов интересующих нас инструкций:

- 0x02 – JMP XX XX;
- 0x12 – CALL XX XX;
- 0x22 – RET.

В дампе обнаруживается много участков кода, заканчивающихся байтом 0x22. Похоже на конец процедуры и возврат из нее.

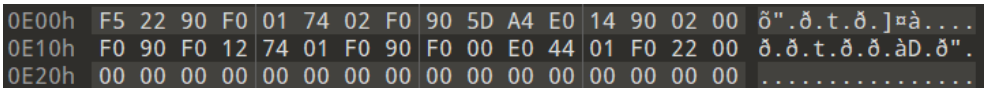


Рис. 2.16. Возврат из процедуры с помощью команды RET

Есть участки, оканчивающиеся предположительно инструкцией JMP с двумя байтами адреса. Тоже подходит для конца процедуры.

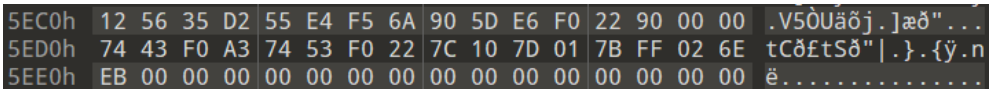


Рис. 2.17. Возврат из процедуры с помощью команды JMP

Предполагая архитектуру MCS-51, зная код инструкции CALL (0x12) и что она имеет аргумент из 2 байт адреса, можно посмотреть использование байта 0x12 в дампе. Опять не видно противоречий нашему предположению.

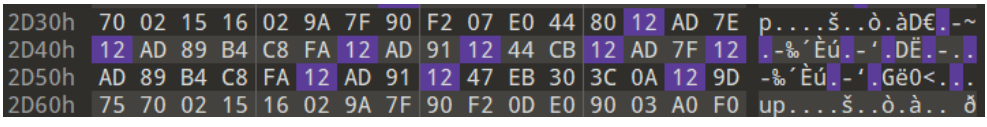


Рис. 2.18. Вызовы команды CALL

Значит, можно попробовать выбрать архитектуру MCS-51 при загрузке в дизассемблер. На весь анализ ушло не более 30 секунд, если знать, что искать.

Некоторые закономерности для определенных наборов инструкций видны невооруженным взглядом при просмотре содержимого прошивки в шестнадцатеричном редакторе. Помните 4-битовое поле Condition в 32-битном режиме архитектуры ARM? На всякий случай напомню структуру команды.

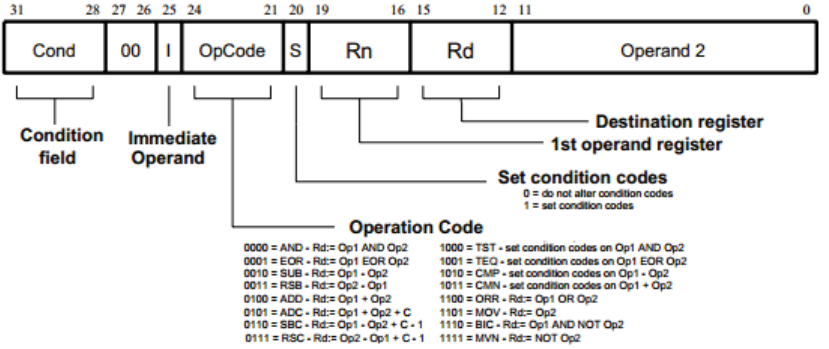


Рис. 2.19. Структура команд ARMv7 в 32-битном режиме[1]

Получается, старшие 4 бита в 32-битном коде команды – это предикат, значения которого представлены в таблице:

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Рис. 2.20. Предикат команд ARMv7[2]

Так как статистически большинство команд будут выполняться, то наиболее встречающееся значение этого предиката будет AL, т. е. 1110b, т. е. 0xE. Следо-

[1] https://developer.arm.com/documentation/ddi0406/latest/.

[2] https://developer.arm.com/documentation/ddi0406/latest/.

вательно, в большинстве старших нибблов (полубайт) в кодах 32-битных команд архитектуры ARM мы увидим значение 0xE. Как думаете, какая система команд в дампе прошивки в следующем примере?

```
6D 00 A0 E3 8B 00 00 EB  0D 00 A0 E3 89 00 00 EB  m...............
00 30 A0 E3 20 30 43 E3  14 30 0B E5 18 00 00 EA  .0....C.........
14 30 1B E5 00 30 93 E5  08 30 0B E5 08 30 1B E5  .0.....0.....0..
23 3C A0 E1 73 30 EF E6  03 00 A0 E1 7D 00 00 EB  #<..s0......}...
08 30 1B E5 23 38 A0 E1  73 30 EF E6 03 00 A0 E1  .0..#8..s0......
78 00 00 EB 08 30 1B E5  23 34 A0 E1 73 30 EF E6  x.....0..#4..s0..
03 00 A0 E1 73 00 00 EB  08 30 1B E5 73 30 EF E6  ....s....0..s0..
03 00 A0 E1 6F 00 00 EB  14 30 1B E5 04 30 83 E2  ....o....0.....0.
14 30 0B E5 14 20 1B E5  FF 3F 0F E3 2F 30 43 E3  .0.....?../0C.
03 00 52 E1 E1 FF FF 9A  66 00 A0 E3 65 00 00 EB  ..R.....f...e...
75 00 A0 E3 63 00 00 EB  73 00 A0 E3 61 00 00 EB  u...c...s...a...
65 00 A0 E3 5F 00 00 EB  73 00 A0 E3 5D 00 00 EB  e..._...s...]...
0D 00 A0 E3 5B 00 00 EB  89 35 A0 E3 10 30 0B E5  ....[....5...0..
18 00 00 EA 10 30 1B E5  00 30 93 E5 0C 30 0B E5  .....0...0.....0
0C 30 1B E5 23 3C A0 E1  73 30 EF E6 03 00 A0 E1  .0..#<..s0......
50 00 00 EB 0C 30 1B E5  23 38 A0 E1 73 30 EF E6  P....0..#8..s0..
03 00 A0 E1 4B 00 00 EB  0C 30 1B E5 23 34 A0 E1  ....K....0..#4..
73 30 EF E6 03 00 A0 E1  46 00 00 EB 0C 30 1B E5  s0......F....0..
73 30 EF E6 03 00 A0 E1  42 00 00 EB 10 30 1B E5  s0......B....0..
```

Рис. 2.21. Фрагмент прошивки для архитектуры ARM в 32-битном режиме команд

Мы уже сказали, что система команд ARM в 16-битном режиме Thumb не имеет предиката, следовательно, и такого признака. Однако код в режиме Thumb можно найти по началу большинства функций в виде инструкции «PUSH {R4,LR}», имеющей значение 0x10 0xB5. Посмотрев, как часто это сочетание байтов встречается в коде, можно сделать предположение, что перед нами ISA ARM в режиме Thumb:

```
10 30 23 F0 DD FC 70 BD  10 D0 01 40 10 B5 00 20  .0#.....p..@... 
88 49 08 70 88 49 08 70  88 49 08 70 81 21 88 48  .I.p.I.p.I.p.!.H
39 F0 5D FB 81 21 86 48  81 30 39 F0 58 FB 81 21  9.]..!.H.09.X..!
84 48 39 F0 54 FB 10 BD  10 B5 FF F7 E7 FF FE F7  .H9.T...........
D7 FE FE F7 C9 FE FE F7  F8 FE 10 BD 79 48 00 78  ............yH.x
70 47 10 B5 77 48 00 78  80 28 0F D1 00 20 76 49  pG..wH.x.(... vI
08 70 81 21 76 48 39 F0  3A FB 81 21 74 48 81 30  .p.!vH9.:..!tH.0
39 F0 35 FB 81 21 73 48  39 F0 31 FB 6D 48 00 78  9.5..!sH9.1.mH.x
20 F0 80 00 6B 49 08 70  10 BD 10 B5 FF F7 BE FF   ...kI.p........
39 F0 65 FA 0A 20 22 F0  BB FA 39 F0 6E FA 00 20  9.e.. "...9.n.. 
10 BD 10 B5 FF F7 B2 FF  39 F0 4B FA 00 20 10 BD  ........9.K.. ..
10 B5 FF F7 AB FF FE F7  C0 FE 10 BD 10 B5 0A 20  ...............
```

Рис. 2.22. Фрагмент прошивки для архитектуры ARM в 16-битном режиме команд

Дополнительным признаком, подтверждающим нашу догадку, будет являться встречающаяся до искомого значения последовательность 0x10 0xBD, соответветствующая инструкции «POP {R4,LR}» в конце предыдущей функции (естественно, с поправкой на используемый компилятор). Конечно, какие конкретно команды сохранения и восстановления контекста используются, зависит от компилятора. Приведенный пример иллюстрирует результат работы компиляторов из набора gcc, использующегося при разработке большинства

прошивок. Поэтому если вы видите начало прошивки с подобным признаком, задача определения архитектуры и системы команд успешно решена.

Хардкор – восстановление неизвестной системы команд

Представим, что документации на чип микроконтроллера нет, статистика не помогает и даже все варианты процессорных модулей из популярных дизассемблеров не помогли. Такая ситуация случается крайне редко, и эта задача явно не для начинающих исследователей устройств. Тем не менее и в этой ситуации остается шанс получить успешный результат. Первым делом потребуется определить длину инструкции. Найти, где в поле инструкции лежат код команды, аргументы, флаги и т. п. После – на основе статистики понять, какими значениями кодируются наиболее часто встречающиеся команды (как правило, это всегда mov и различные варианты команд передачи управления). Действуя подобным способом, методом проб и ошибок иногда можно восстановить систему команд и написать свой дизассемблер (или плагин для существующего). Для большего погружения в тему можно посмотреть доклад «Reverse engineering of binary programs for custom virtual machines» с конференции Recon 2012 (**https://recon.cx/2012/schedule/events/236.en.html**) или доклад об исследовании микрокода процессоров Intel (**https://www.youtube.com/watch?v=V1nJeV0Uq0M**).

Реверс-инжиниринг прошивки FPGA

В главе про первичный анализ устройства рассматривался вариант построения устройства на базе FPGA, без микроконтроллера. Для исследователя встраиваемых систем этот вариант более сложен, но не всегда бесперспективен. В этом разделе кратко опишу, какие есть варианты для исследования подобных устройств.

В большинстве микросхем FPGA-конфигурация загружается из внешней микросхемы ПЗУ с интерфейсом SPI. Соответственно, считать конфигурацию (также называемую bitstream, битстрим) не представляет сложностей (не всегда все просто, некоторые микросхемы FPGA поддерживают шифрование содержимого конфигурации). Задача реверс-инжиниринга конфигурации FPGA решается весьма сложно. Но есть и хорошая новость – т. к. внутри FPGA часто используется один из распространенных программных микропроцессоров (т. е. ядро реализовано на базе логических элементов FPGA), можно выделить из всей конфигурации FPGA часть, хранящую прошивку данного программного микропроцессора. И задача исследования устройства может сводиться к реверс-инжинирингу прошивки программно-аппаратного процессора. Отличие будет заключаться в недокументированных выводах микросхемы (т. к. выводы FPGA являются конфигурируемыми, т. е. разработчик в большинстве случаев сам решает, какой сигнал внутренней схемы на какой вывод микросхемы коммутировать) и неизвестной периферии микропроцессора (т. к. при использовании FPGA, как правило, разработчики реализуют какие-то функциональные блоки в виде аппаратных блоков внутри FPGA).

Альтернативные варианты исследования устройств, использующих FPGA, по своей сложности явно лежат за границами повествования этой книги, поэтому остается только пожелать начинающему исследователю, столкнувшемуся с принципиальной необходимостью получить битстрим и восстановить из него что-то осмысленное, удачи.

Подготовка прошивки к дизассемблированию

На данный момент мы знаем тип ядра и систему команд ядра микроконтроллера. С этой информацией мы уже можем попробовать загрузить имеющийся у нас бинарный образ прошивки в дизассемблер. В зависимости от того, каким способом образ добыли (на этапе первичного анализа), у нас может быть разная информация о составе образа. Для примера рассмотрим распространенный случай, когда у нас есть образ считанной микросхемы ПЗУ. Размер образа равен объему микросхемы ПЗУ. При этом полезной информации в нем может быть намного меньше. Первым делом нам нужно определить, где в образе микросхемы ПЗУ находится часть, соответствующая образу прошивки. А во-вторых, понять, с какой части прошивки стоит начать загрузку, ведь прошивка не всегда представляет собой линейный образ.

Выделение образа прошивки из образа ПЗУ

Для решения этой задачи как нельзя лучше нам снова может помочь утилита binwalk. Она успешно определяет в анализируемом двоичном образе участки кода, сжатые данные, распространенные типы данных, такие как изображения и т. п. Пример вывода binwalk для прошивки роутера показан на рисунке:

```
$ binwalk --signature --term archer-c7.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
21876         0x5574          U-Boot version string, "U-Boot 1.1.4-g4480d5f9-dirty (May
                              20 2019 - 18:45:16)"
21940         0x55B4          CRC32 polynomial table, big endian
23232         0x5AC0          uImage header, header size: 64 bytes, header CRC:
                              0x386C2BD5, created: 2019-05-20 10:45:17, image size:
                              41162 bytes, Data Address: 0x80010000, Entry Point:
                              0x80010000, data CRC: 0xC9CD1E38, OS: Linux, CPU: MIPS,
                              image type: Firmware Image, compression type: lzma, image
                              name: "u-boot image"
23296         0x5B00          LZMA compressed data, properties: 0x5D, dictionary size:
                              8388608 bytes, uncompressed size: 97476 bytes
64968         0xFDC8          XML document, version: "1.0"
78448         0x13270         uImage header, header size: 64 bytes, header CRC:
                              0x78A267FF, created: 2019-07-26 07:46:14, image size:
                              1088500 bytes, Data Address: 0x80060000, Entry Point:
                              0x80060000, data CRC: 0xBB9D4F94, OS: Linux, CPU: MIPS,
                              image type: Multi-File Image, compression type: lzma,
                              image name: "MIPS OpenWrt Linux-3.3.8"
78520         0x132B8         LZMA compressed data, properties: 0x6D, dictionary size:
                              8388608 bytes, uncompressed size: 3164228 bytes
1167013       0x11CEA5        Squashfs filesystem, little endian, version 4.0,
                              compression:xz, size: 14388306 bytes, 2541 inodes,
                              blocksize: 65536 bytes, created: 2019-07-26 07:51:38
15555328      0xED5B00        gzip compressed data, from Unix, last modified: 2019-07-26
                              07:51:41
```

Рис. 2.23. Пример анализа образа прошивки с помощью binwalk[1]

[1] https://habr.com/ru/post/487406/.

Естественно, не стоит упускать возможность выполнить часть задач по анализу бинарного образа с помощью binwalk. Но представим, что мы хотим сделать аналогичные действия самостоятельно (или результаты работы binwalk'a нас не устраивают). Значит, начнем просматривать образ ПЗУ в шестнадцатеричном редакторе, надеясь увидеть:

1) блоки информации со значениями, похожими на команды определенной архитектуры и данными. Скорее всего, это и есть прошивка, надо корректно определить ее начало и конец. В ней могут встречаться человекочитаемые строковые данные (отличный вариант, проще будет при реверс-инжиниринге за что-то «зацепиться»).

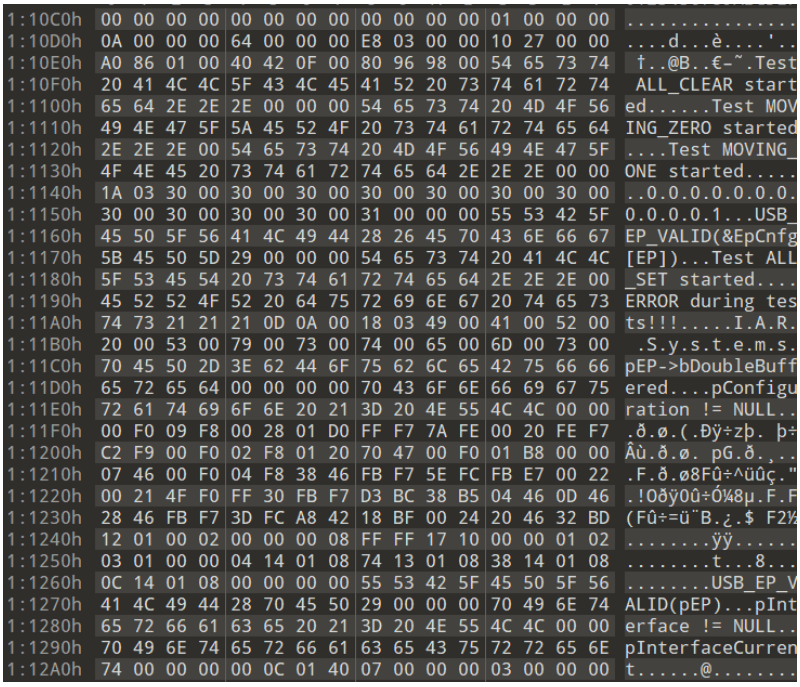
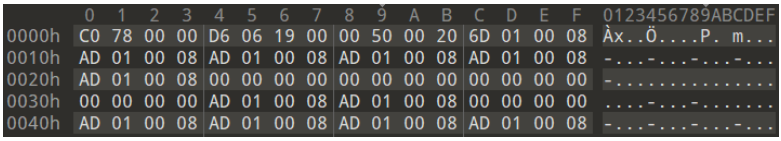


Рис. 2.24. Строки в дампе прошивки

Границы расположения прошивки в образе можно определить несколькими способами:

- на глаз. Если после конца последнего блока находится множество байтов 0x00 или 0xFF, возможно, мы нашли конец прошивки. Главное – не поторопиться и не отрезать прошивку в середине, ведь подобные блоки данных размером до нескольких килобайтов могут быть внутри определенных частей прошивки. Если вы пересмотрите структуру даже нескольких десятков образов, ваш опыт позволит довольно быстро определять границы прошивки;
- с помощью средств контроля целостности прошивки. Часто прошивка начинается или заканчивается блоком с контрольной суммой

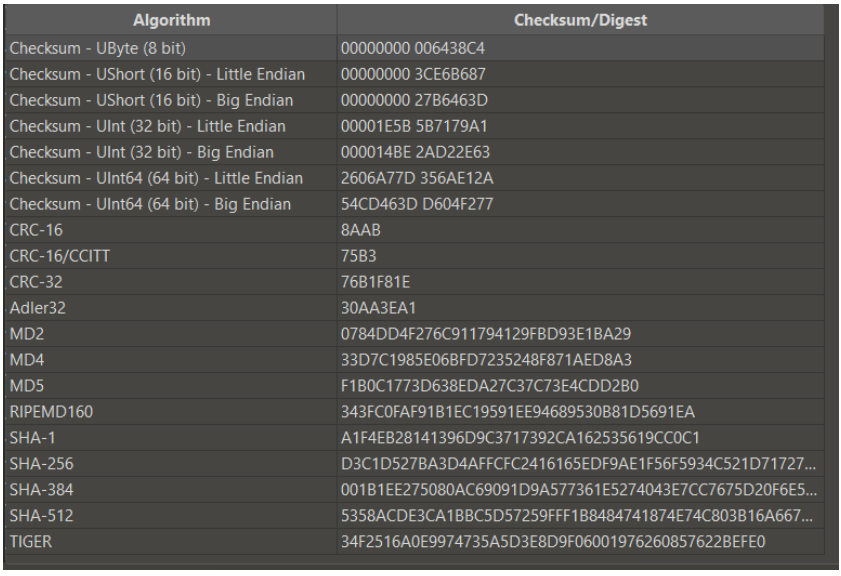
прошивки и/или значением ее размера. Например, это может быть 8-байтовый блок, в котором первые 4 байта – это размер прошивки (0x78C0 на картинке далее), а вторые 4 байта – значение контрольной суммы (0x1906D6).



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h	C0	78	00	00	D6	06	19	00	00	50	00	20	6D	01	00	08	Àx..Ö....P. m...
0010h	AD	01	00	08	AD	01	00	08	AD	01	00	08	AD	01	00	08	-...-...-...-...
0020h	AD	01	00	08	00	00	00	00	00	00	00	00	00	00	00	00	-...............
0030h	00	00	00	00	AD	01	00	08	AD	01	00	08	00	00	00	00	....-...-.......
0040h	AD	01	00	08	AD	01	00	08	AD	01	00	08	AD	01	00	08	-...-...-...-...

Рис. 2.25. Значения размера и контрольной суммы в дампе прошивки

Размер может считаться в байтах, dword'ах или даже в килобайтах. Нет никакого стандарта, но чаще всего встречается именно подсчет в байтах. Контрольная сумма может считаться по разным алгоритмам, самыми используемыми являются Checksum 32 и CRC32. Хотя могут встречаться и другие варианты битности и даже нераспространенные полиномы. Для поиска подобных мест удобно пользоваться встроенными в шестнадцатеричный редактор средствами подсчета значений контрольных сумм по разным алгоритмам.



Algorithm	Checksum/Digest
Checksum - UByte (8 bit)	00000000 006438C4
Checksum - UShort (16 bit) - Little Endian	00000000 3CE6B687
Checksum - UShort (16 bit) - Big Endian	00000000 27B6463D
Checksum - UInt (32 bit) - Little Endian	00001E5B 5B7179A1
Checksum - UInt (32 bit) - Big Endian	000014BE 2AD22E63
Checksum - UInt64 (64 bit) - Little Endian	2606A77D 356AE12A
Checksum - UInt64 (64 bit) - Big Endian	54CD463D D604F277
CRC-16	8AAB
CRC-16/CCITT	75B3
CRC-32	76B1F81E
Adler32	30AA3EA1
MD2	0784DD4F276C911794129FBD93E1BA29
MD4	33D7C1985E06BFD7235248F871AED8A3
MD5	F1B0C1773D638EDA27C37C73E4CDD2B0
RIPEMD160	343FC0FAF91B1EC19591EE94689530B81D5691EA
SHA-1	A1F4EB28141396D9C3717392CA162535619CC0C1
SHA-256	D3C1D527BA3D4AFFCFC2416165EDF9AE1F56F5934C521D71727...
SHA-384	001B1EE275080AC69091D9A577361E5274043E7CC7675D20F6E5...
SHA-512	5358ACDE3CA1BBC5D57259FFF1B8484741874E74C803B16A667...
TIGER	34F2516A0E9974735A5D3E8D9F06001976260857622BEFE0

Рис. 2.26. Результат подсчета контрольных сумм и хешей разными алгоритмами в шестнадцатеричном редакторе

Еще один момент, на который стоит обратить внимание, – это порядок байтов значений контрольной суммы и размера прошивки. Он может быть как Little Endian, так и Big Endian;

- с помощью логического анализатора. Помните, я говорил, что логический анализатор используется очень часто при анализе устройств? Вот отличный пример применения. Если подключить логический

анализатор к микросхеме ПЗУ с интерфейсом SPI и записать лог обращений микроконтроллера, то можно определить начало и конец считываемых из ПЗУ данных, соответствующих прошивке. Пример подобного сниффинга данных показан на рисунке:

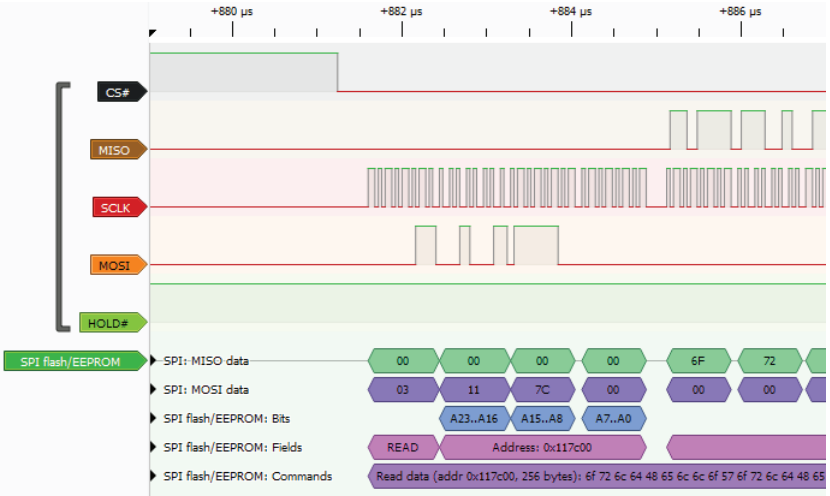


Рис. 2.27. Пример записанных логическим анализатором сигналов от микросхемы SPI ПЗУ

- из логов UART. Например, часто загрузчики выводят в консоль размер загружаемого кода и его смещение, пример такого вывода для лога загрузки микроконтроллера ESP32 представлен на рисунке:

```
boot: SPI Flash Size : 4MB[0m
boot: Partition Table:[0m
boot: ## Label            Usage          Type ST Offset   Length[0m
boot:  0 nvs              WiFi data        01 02 00009000 00006000[0m
boot:  1 phy_init         RF data          01 01 0000f000 00001000[0m
boot:  2 factory          factory app      00 00 00010000 00100000[0m
boot: End of partition table[0m
```

Рис. 2.28. Лог загрузки микроконтроллера ESP32[1]

Смущающий новичков момент – блоков с прошивкой в образе ПЗУ может быть несколько. Это норма. Часто загрузчик умеет искать несколько копий прошивки, основную и резервные. Резервные копии нужны в случае, если содержимое основной прошивки будет повреждено. Например, в ходе прерванного обновления;

2) данные, необходимые для работы прошивки, но не включенные в код прошивки. Например, сертификаты или конфигурационные данные для настройки специфичных аппаратных блоков;

3) Padding, т. е. заполнение неиспользуемых адресов в ПЗУ каким-либо значением (как правило, байтом 0x00 или 0xFF).

[1] https://esp32.com/viewtopic.php?t=14285.

В результате такого поверхностного анализа можно представить структуру образа считанного ПЗУ и изобразить ее схематически, например как на следующем рисунке.

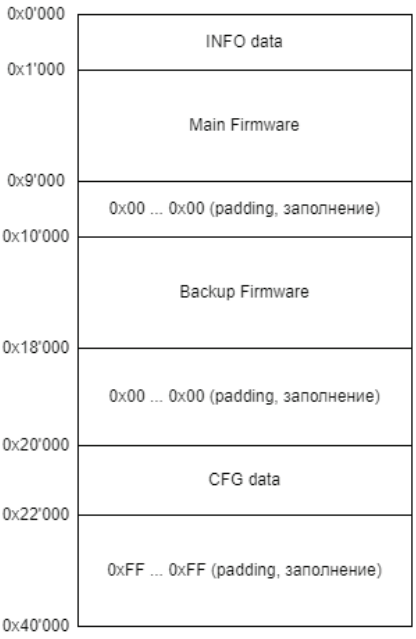


Рис. 2.29. Пример структуры ПЗУ с прошивкой

Определение структуры прошивки

Вроде бы мы уже выделили образ прошивки из образа ПЗУ и можно загружать его в дизассемблер. На самом деле это правда, но лучше потратить еще немного времени и внимательно посмотреть на сам образ.

Прошивки в устройствах далеко не всегда представляют собой линейный образ, загружаемый единым блоком, все зависит от способа организации прошивки и количества ресурсов микроконтроллера. У микроконтроллера может быть мало внутренней памяти, а функциональные возможности устройства могут требовать много кода и большого объема прошивки, следовательно, часть прошивки придется хранить во внешней микросхеме ПЗУ. Или у микроконтроллера достаточный объем внутренней памяти, в который помещаются все нужные функции, но для удешевления этот код находится в ROM-памяти и является частью дизайна микросхемы. Тогда внешняя память может хранить исправленные участки кода, ведь при работе любого устройства рано или поздно находятся ошибки и их надо как-то исправлять.

В реальных встраиваемых системах можно встретить различные варианты организации прошивки во внешнем ПЗУ. В идеальном случае в ПЗУ лежит линейный образ прошивки, выделяем его по описанному выше алгоритму и можем смело загружать в дизассемблер. Пример структуры подобного образа показан на схеме:

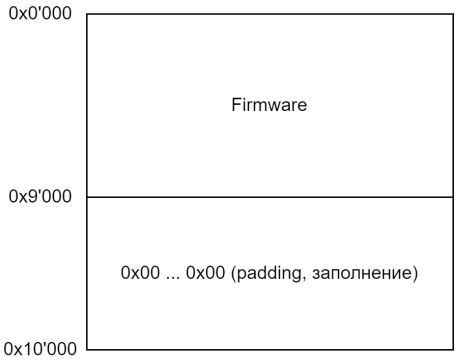


Рис. 2.30. Пример линейного образа прошивки

Следующий вариант организации прошивки устройства может быть чуть хуже для исследователя: во внешнем ПЗУ лежит вся прошивка, но ее структура не линейна. Есть некоторая «базовая» часть, которая подгружает отдельные модули прошивки по разным адресам ОЗУ внутри адресного пространства микроконтроллера, откуда их код исполняется. В таком случае при анализе будет немного сложнее, но т. к. у нас все равно есть вся кодовая база, задача точно решаемая. В разделе «Определение адреса загрузки прошивки» мы рассмотрим, как корректно загружать такие типы прошивок. Структура образа с модулями показана на схеме:

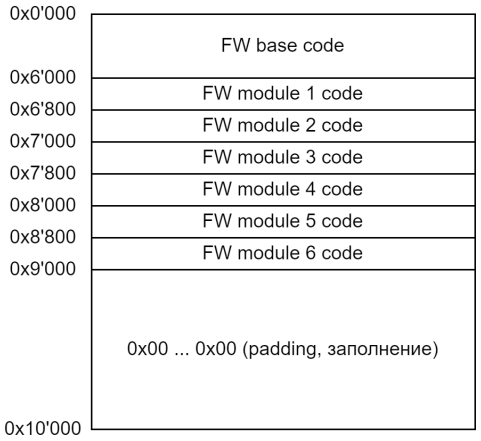


Рис. 2.31. Пример образа прошивки с подгружаемыми модулями

Наконец, существуют еще два варианта хранения кода во внешнем ПЗУ. Первый похож на описанный ранее подход с модулями, за исключением того, что базовая часть прошивки расположена во внутреннем ПЗУ микроконтроллера (в ROM или flash-памяти), а во внешней части хранится только код загружаемых модулей. С одной стороны, без базовой части прошивки будет сложно разобраться в логике работы. С другой – модули, скорее всего, выполняют какой-то законченный функционал (например, содержат в себе обработчик про-

токола или команды) и по нему можно понять часть логики работы базовой прошивки или даже вычитать недостающую часть.

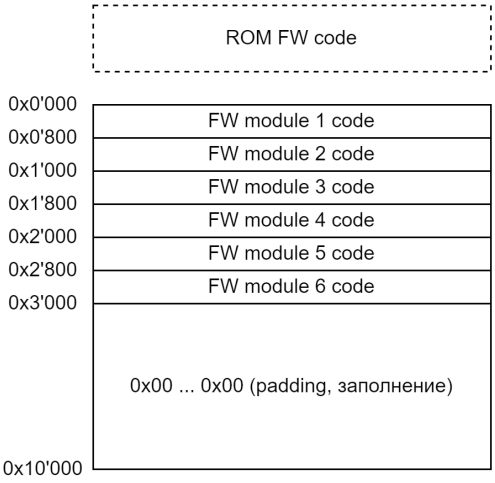


Рис. 2.32. Пример образа прошивки с частью кода в ROM и подгружаемыми модулями

Второй (и самый плохой для исследователя, за исключением отсутствия доступа к прошивке вовсе) вариант встречается редко. В нем во внешнем ПЗУ лежат атомарные куски кода (патчи), которые могут загружаться в ОЗУ «поверх» аналогичных блоков кода из основной прошивки. Подобный механизм используется для исправления ошибок в основной прошивке. «Применением» подобных патчей занимается либо сама базовая прошивка (т. к. этот механизм заранее закладывается разработчиком на этапе проектирования прошивки), либо даже специальный аппаратный блок внутри микроконтроллера. В таком варианте основная прошивка вообще не участвует в процессе патчинга.

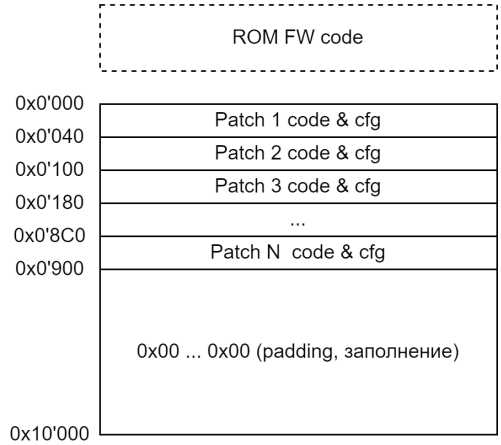


Рис. 2.33. Пример образа прошивки кодом в ROM и патчами

Сжатая или зашифрованная прошивка

Если производитель хочет сохранить свою интеллектуальную собственность, к которой также относится прошивка, в секрете, он может использовать шифрование прошивки. Для шифрования прошивки используются блочные (DES, 3DES, AES, RC5 и т. п.) или потоковые шифры (RC4 и т. п.). Ключ должен храниться внутри микроконтроллера (например, в One Time Programmable памяти (OTP), BootROM или хотя бы во внутренней flash-памяти) и не покидать его ни при каких условиях. При наличии ошибок (в том числе архитектурных) в реализации механизмов безопасности можно получить значение ключей и расшифровать прошивку. Например, получив доступ к содержимому OTP или, что еще лучше, получив возможность исполнять свой код с помощью аппаратной отладки.

В случае использования сжатия, при наличии дополнительных данных для анализа, можно попытаться восстановить требуемые параметры и распаковать прошивку, например как было сделано экспертами компании Positive Technologies при исследовании прошивки Intel Managment Engine (**https://www.blackhat.com/eu-17/briefings/schedule/index.html#intel-me-flash-file-system-explained-8484**).

Довольно часто для упаковки прошивки или отдельных ее модулей используется алгоритм LZMA. С определением частей образа, упакованных LZMA, отлично справляется binawalk, пример подобного вывода показан на рисунке:

```
23296         0x5B00          LZMA compressed data, properties: 0x5D, dictionary size:
                              8388608 bytes, uncompressed size: 97476 bytes
64968         0xFDC8          XML document, version: "1.0"
78448         0x13270         uImage header, header size: 64 bytes, header CRC:
                              0x78A267FF, created: 2019-07-26 07:46:14, image size:
                              1088500 bytes, Data Address: 0x80060000, Entry Point:
                              0x80060000, data CRC: 0xBB9D4F94, OS: Linux, CPU: MIPS,
                              image type: Multi-File Image, compression type: lzma,
                              image name: "MIPS OpenWrt Linux-3.3.8"
78520         0x132B8         LZMA compressed data, properties: 0x6D, dictionary size:
                              8388608 bytes, uncompressed size: 3164228 bytes
```

Рис. 2.34. Поиск сжатых фрагментов в образе с помощью binwalk[1]

Представим, что binwalk не смог определить используемый алгоритм упаковки. Можно ли как-то попытаться отличить упакованные данные от зашифрованных без дополнительной информации? На помощь приходит анализ энтропии данных. Энтропия в последовательности байтов – это количественно выраженная непредсказуемость появления какого-либо байта. Энтропия зашифрованных и сжатых данных может немного отличаться, т. к. при сжатии могут быть периоды снижения энтропии на определенных участках, например на служебных данных, используемых алгоритмами сжатия. На следующих графиках представлен вывод анализа энтропии с помощью Binwalk (слева – запакованный образ, справа – зашифрованный):

[1] https://habr.com/ru/post/487406/.

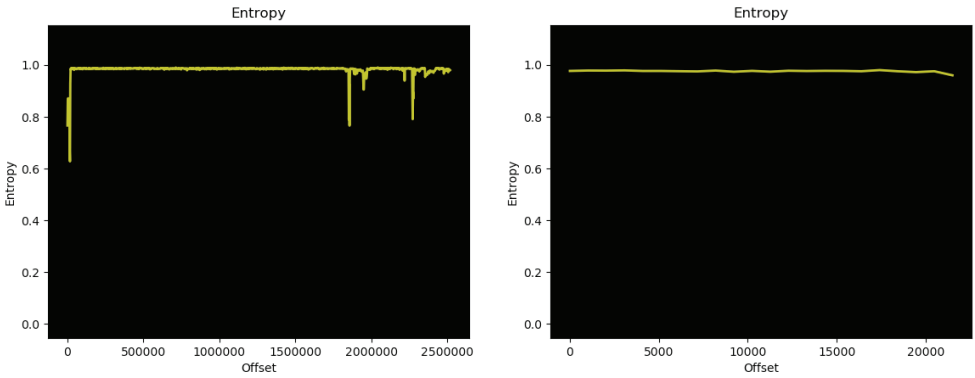


Рис. 2.35. Анализ энтропии образа с помощью Binwalk

Если посмотреть на графики энтропии этих же файлов с большей детализацией, то можно явно увидеть разницу между зашифрованными и сжатыми данными. Аналогично предыдущим, на следующих графиках представлен вывод анализа энтропии с помощью Binwalk, но с гораздо большим приближением (слева – запакованный образ, справа – зашифрованный):

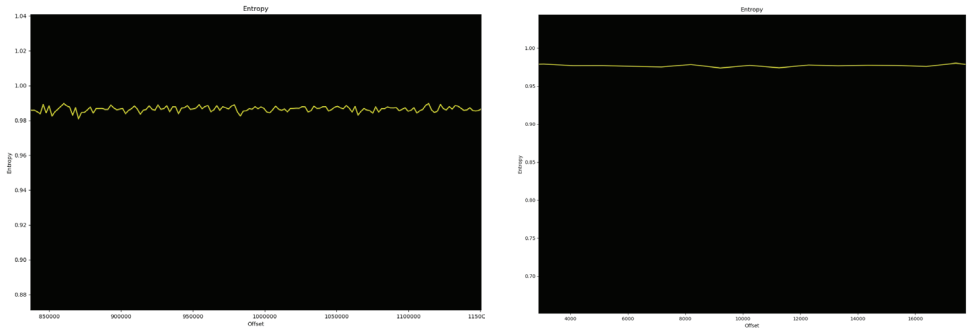


Рис. 2.36. Анализ энтропии небольшого фрагмента образа с помощью Binwalk

Так как одним из важнейших свойств криптографических алгоритмов шифрования является максимально равномерное значение зашифрованных данных, не зависящее от входных данных, то график энтропии зашифрованных данных должен максимально напоминать прямую линию.

Разобравшись с распространенными способами хранения и структурами прошивки, можем приступать к определению ее адреса загрузки, но сначала надо понять, как устроено адресное пространство микроконтроллеров.

Структура адресного пространства микроконтроллера

Мы уже знаем, что микроконтроллер состоит из вычислительного ядра и набора аппаратных периферийных модулей, выполняющих какие-либо функции и/или обеспечивающих взаимодействие по различным интерфейсам. В доку-

ментации на микроконтроллер можно найти структурную схему, на которой отражены все функциональные блоки. Пример схемы для микроконтроллера STM32F051K8 на базе процессорного ядра ARM Cortex-M0 показан на рисунке далее.

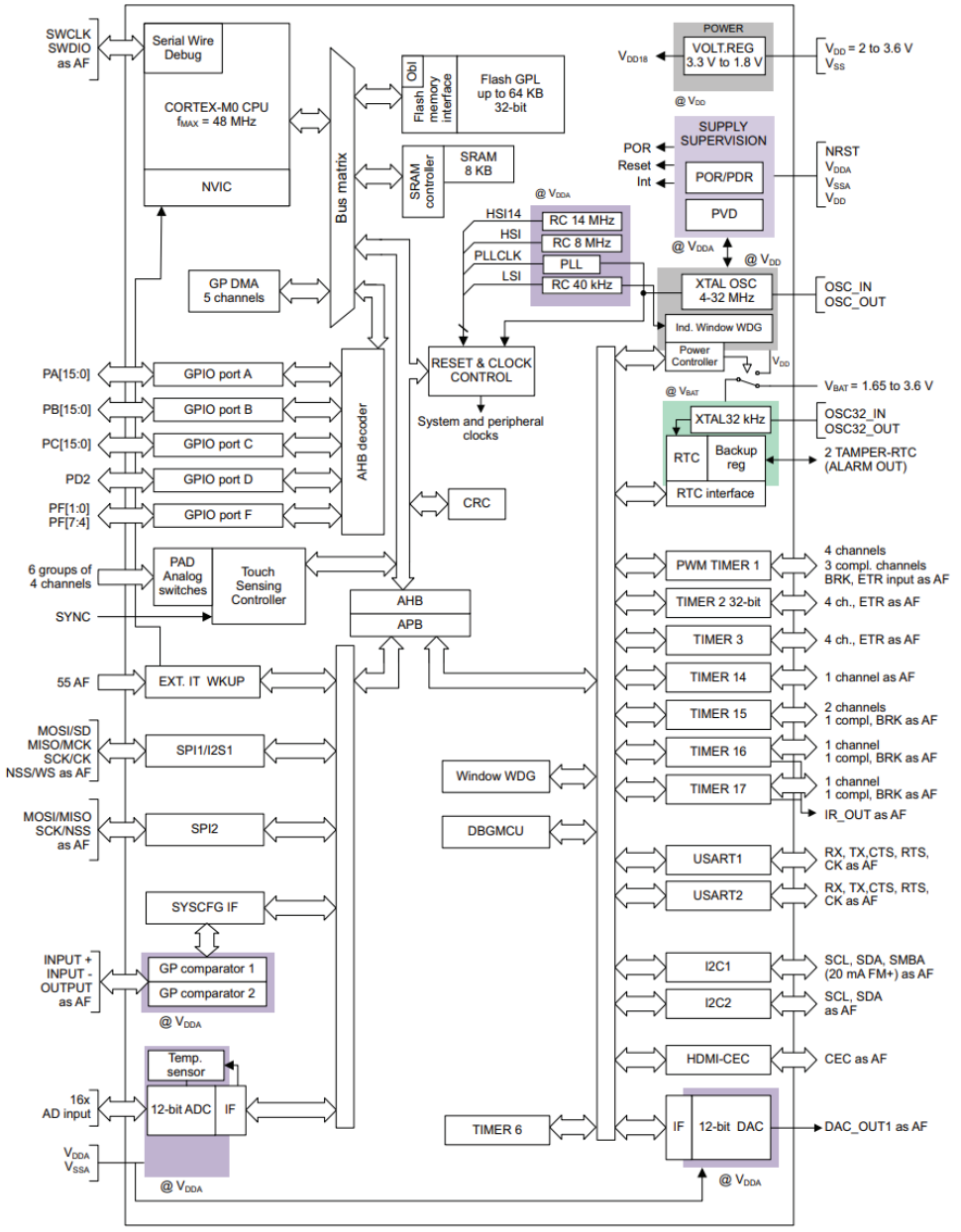


Рис. 2.37. Структура простого микроконтроллера на базе ядра ARM Cortex-M0[1]

[1] https://www.st.com/resource/en/datasheet/stm32f051k8.pdf.

Мы уже знаем, что в большинстве случаев используется подход MMIO (Memory Mapped Input Output) для работы с аппаратными периферийными блоками, заключающийся в отражении конфигурационных регистров в адресное пространство микроконтроллера. Но как выглядит это адресное пространство? Базовую структуру адресного пространства задает процессорное ядро. Оно определяет разрядность адресного пространства и диапазоны используемых адресов, как показано в примере для ядра ARM Cortex-M0:

Table 3-2 Memory map usage

Address range	Code	Data	Device
0xF0000000 - 0xFFFFFFFF	No	No	Yes
0xE0000000 – 0xEFFFFFFF	No	No	No[a]
0xA0000000 – 0xDFFFFFFF	No	No	Yes
0x60000000 – 0x9FFFFFFF	Yes	Yes	No
0x40000000 – 0x5FFFFFFF	No	No	Yes
0x20000000 - 0x3FFFFFFF	Yes[b]	Yes	No
0x00000000 - 0x1FFFFFFF	Yes	Yes	No

Рис. 2.38. Базовая структура адресного пространства для ядра ARM Cortex-M0. Cortex-M0 Technical Reference Manual

Периферийные блоки подключаются к шине микроконтроллера и предоставляют набор регистров для конфигурирования и передачи данных ядру. Самый распространенный вариант организации доступа к периферийным регистрам – отображение в определенную область адресного пространства микроконтроллера, однако есть ядра с поддержкой механизмов доступа к периферийным регистрам за счет использования специального набора команд.

Какая наиболее часто встречающаяся разрядность микроконтроллеров? В современном мире это 32 бита (хотя есть и 64, и 8, 16, и даже 24 бита). Следовательно, размер адресного пространства равен 4 гигабайта (2 в 32-й степени). Конечно, такой большой диапазон адресов (от 0x0 до 0xFFFFFFFF) для микроконтроллера избыточен, но позволяет выделить участки для регистров всех аппаратных блоков и памяти без пересечения адресов, что очень удобно для разработчика. Рассмотрим пример организации адресного пространства на 32-битном микроконтроллере STM32F0.

Карта памяти микроконтроллера

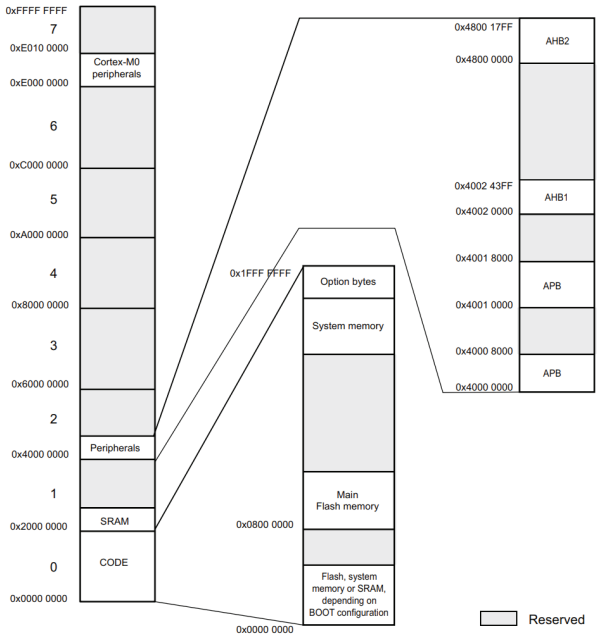


Рис. 2.39. Карта памяти микроконтроллера STM32F051K8[1]

Данная схема карты памяти приведена в документации на микроконтроллер. Мы видим значения диапазонов адресов в адресном пространстве, в том числе соответствующие шинам AHB (Advanced High-performance Bus) и APB (Advanced Peripheral Bus) микроконтроллера, к которым подключены периферийные контроллеры. Архитектура микроконтроллера STM32F0, включающая шины AHB и APB, показана на схеме:

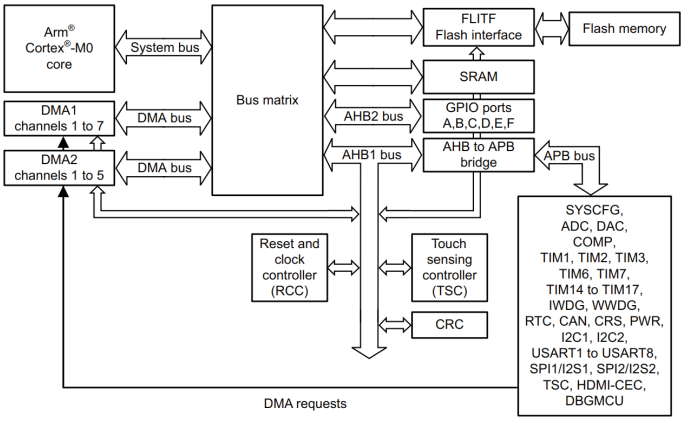


Рис. 2.40. Шины AHB, APB и подключенные к ним устройства[2]

[1] https://www.st.com/resource/en/datasheet/stm32f051k8.pdf.

[2] https://www.st.com/resource/en/reference_manual/dm00031936-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

Большинство периферийных контроллеров подключены к шине APB и отображаются (замаплены, англ. mapped) на определенные диапазоны адресов, выделенные для этой шины. В документации можно найти описание основных диапазонов системной памяти, соответствующих регистрам периферийных аппаратных модулей (хотя могут быть и недокументированные):

Table 1. STM32F0xx peripheral register boundary addresses (continued)

Bus	Boundary address	Size	Peripheral	Peripheral register map
APB	0x4000 7C00 - 0x4000 7FFF	1 KB	Reserved	-
	0x4000 7800 - 0x4000 7BFF	1 KB	CEC	*Section 31.7.7 on page 919*
	0x4000 7400 - 0x4000 77FF	1 KB	DAC	*Section 14.10.15 on page 299*
	0x4000 7000 - 0x4000 73FF	1 KB	PWR	*Section 5.4.3 on page 94*
	0x4000 6C00 - 0x4000 6FFF	1 KB	CRS	*Section 7.7.5 on page 147*
	0x4000 6800 - 0x4000 6BFF	1 KB	Reserved	-
	0x4000 6400 - 0x4000 67FF	1 KB	CAN	*Section 29.9.5 on page 864*
	0x4000 6000 - 0x4000 63FF	1 KB	USB/CAN SRAM	*Section 30.6.3 on page 900*
	0x4000 5C00 - 0x4000 5FFF	1 KB	USB	*Section 30.6.3 on page 900*
	0x4000 5800 - 0x4000 5BFF	1 KB	I2C2	*Section 26.7.12 on page 698*
	0x4000 5400 - 0x4000 57FF	1 KB	I2C1	*Section 26.7.12 on page 698*
	0x4000 5000 - 0x4000 53FF	1 KB	USART5	*Section 27.8.12 on page 765*
	0x4000 4C00 - 0x4000 4FFF	1 KB	USART4	*Section 27.8.12 on page 765*
	0x4000 4800 - 0x4000 4BFF	1 KB	USART3	*Section 27.8.12 on page 765*
	0x4000 4400 - 0x4000 47FF	1 KB	USART2	*Section 27.8.12 on page 765*
	0x4000 3C00 - 0x4000 43FF	2 KB	Reserved	-
	0x4000 3800 - 0x4000 3BFF	1 KB	SPI2	*Section 28.9.10 on page 824*
	0x4000 3400 - 0x4000 37FF	1 KB	Reserved	-
	0x4000 3000 - 0x4000 33FF	1 KB	IWDG	*Section 23.4.6 on page 582*
	0x4000 2C00 - 0x4000 2FFF	1 KB	WWDG	*Section 24.5.4 on page 588*
	0x4000 2800 - 0x4000 2BFF	1 KB	RTC	*Section 25.7.18 on page 628*
	0x4000 2400 - 0x4000 27FF	1 KB	Reserved	-
	0x4000 2000 - 0x4000 23FF	1 KB	TIM14	*Section 19.4.13 on page 490*
	0x4000 1800 - 0x4000 1FFF	2 KB	Reserved	-
	0x4000 1400 - 0x4000 17FF	1 KB	TIM7	*Section 21.4.9 on page 572*
	0x4000 1000 - 0x4000 13FF	1 KB	TIM6	*Section 21.4.9 on page 572*
	0x4000 0800 - 0x4000 0FFF	2 KB	Reserved	-
	0x4000 0400 - 0x4000 07FF	1 KB	TIM3	*Section 18.4.19 on page 469*
	0x4000 0000 - 0x4000 03FF	1 KB	TIM2	*Section 18.4.19 on page 469*

Рис. 2.41. Фрагмент описания расположения периферийных контроллеров в адресном пространстве микроконтроллера[1]

То есть, чтобы принять или передать данные по, например, встроенному в микроконтроллер интерфейсу UART2, нам надо обратиться к регистрам USART-контроллера, расположенным по адресам (0x40004400–0x400047FF) в адресном пространстве микроконтроллера.

Однако не всегда адресное пространство со стороны прошивки может однозначно трактоваться как физические адреса. Некоторые семейства микроконтроллеров поддерживают виртуальную память, трансляцией адресов которой

[1] https://www.st.com/resource/en/reference_manual/dm00031936-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

в физическую занимается блок MMU (Memory Management Unit). Хорошее описание MMU приведено в статье «MMU в картинках (часть 1)» (**https://habr.com/ru/articles/211150/**), а статья «Секреты кеш-памяти, или Как потратить 1000 тактов на 10 команд» (**https://habr.com/ru/articles/187654/**) послужит для нее хорошим дополнением, т. к. кеш процессора – это еще один аппаратный блок, «удлиняющий», казалось бы, простую цепочку запроса данных из памяти.

Во многих семействах микроконтроллеров есть блок управления памятью Memory Controller (MC). Он управляет запросами к памяти со стороны ядра и периферийных контроллеров, например контроллера DMA. Рассмотрим пример реализации подобного контроллера на примере семейства микроконтроллеров AT91SAM фирмы Atmel на базе ядра ARM7TDMI. Схема Memory Controller для данного семейства микроконтроллеров показана на рисунке:

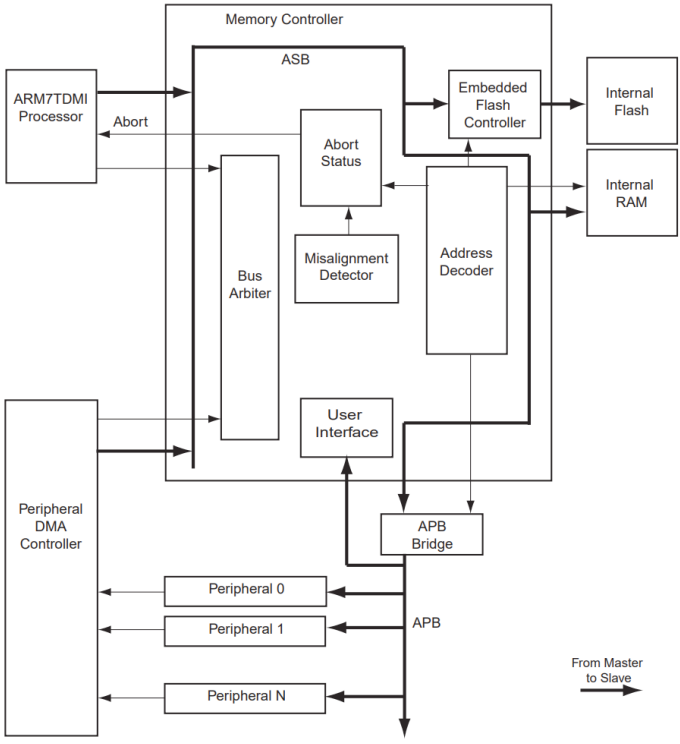


Рис. 2.42. Схема Memory Controller в микроконтроллере семейства AT91SAM[1]

В состав MC входят арбитр шины, декодер адреса, система обслуживания состояния abort, детектор ошибки выравнивания и контроллер встроенной flash-памяти. Данный MC поддерживает механизмы Memory Mapping и Remap, позволяющие менять отображаемые на определенные участки физического адресного пространства диапазоны памяти.

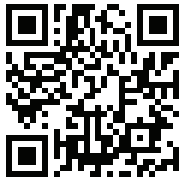
[1] https://www.microchip.com/content/dam/mchp/documents/OTH/ProductDocuments/DataSheets/doc6175.pdf.

Часто в документации на микроконтроллер можно встретить блок MPU (Memory Protection Unit). В зависимости от конкретной реализации блок MPU может обладать разными функциональными возможностями, начиная от простой защиты от случайной записи памяти (Write Protect) до гибкой установки прав доступа для различных регионов адресного пространства. Грамотная настройка MPU может затруднить процесс динамического анализа устройства. Подробнее про механизмы работы MPU можно узнать на примере документации для распространенных микроконтроллеров STM32 «Introduction to memory protection unit management on STM32 MCUs» (**https://www.st.com/resource/en/application_note/dm00272912-managing-memory-protection-unit-in-stm32-mcus-stmicroelectronics.pdf**).

Какие еще распространенные аппаратные блоки, помимо интерфейсных, часто применяются в микроконтроллерах?

- Таймеры. Через периферийные регистры может настраиваться период (в тиках ядра микроконтроллера) вызова специальной подпрограммы – обработчика прерывания таймера. То есть независимо от основного кода прошивки можно настроить вызов определенной подпрограммы в прошивке через точный промежуток времени.
- Аппаратные блоки ускорения математических операций. К ним же отнесем криптографические функции и генераторы случайных значений. Некоторые математические операции при реализации в коде прошивки могут выполняться слишком долго, а при реализации их «в железе» – на порядки быстрее. Поэтому для ускорения некоторые из подобных функций реализуются в виде аппаратного модуля микроконтроллера.
- DMA-блок. Direct Memory Access – технология прямого доступа к памяти с игнорированием ядер микроконтроллера. Представим, что нам нужно передавать большие объемы информации с высокой скоростью из RAM-памяти в какой-то интерфейс. Если делать это программно, то придется каждый байт (или несколько байтов) считывать в регистры ядра микроконтроллера, а потом записывать в периферийные регистры аппаратного блока соответствующего интерфейса. Мы будем сильно ограничены вычислительной мощностью ядра, ведь для передачи каждых нескольких байтов информации надо будет выполнить целый набор инструкций. Для ускорения подобных операций была придумана технология DMA. Очень упрощенно можно представить ее как аппаратно реализованную функцию memcpy. В периферийных регистрах блока DMA настраиваются адреса источника и приемника, размер передаваемых данных, а также инициируется запуск передачи данных.

Для распространенных моделей микроконтроллеров существуют плагины для дизассемблеров, автоматически создающие конфигурацию адресного пространства в проекте дизассемблера. Например, плагин FirmLoader (**https://github.com/Accenture/FirmLoader**) для IDA Pro имеет большой список поддерживаемых семейств микроконтроллеров, что показано на рисунке:



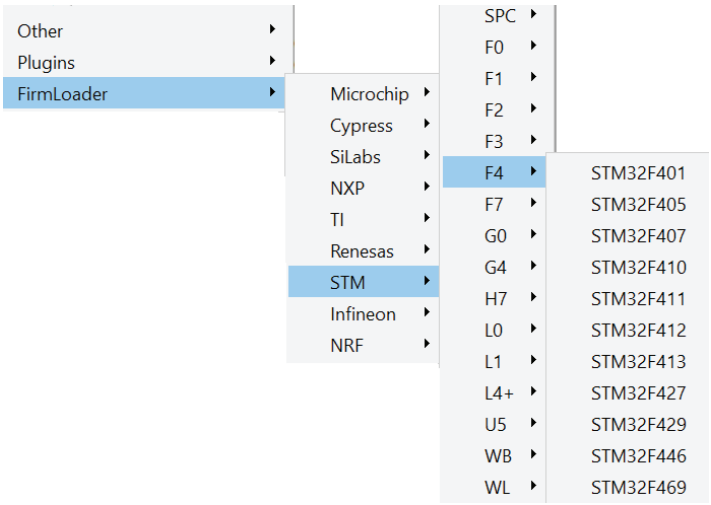


Рис. 2.43. Поддерживаемые плагином FirmLoader микроконтроллеры семейства STM32F4

В результате его работы для каждого периферийного блока в адресном пространстве дизассемблера создается отдельный сегмент, с нужными смещениями и названиями. Это существенно ускоряет процесс реверс-инжиниринга прошивок, т. к. сразу видны обращения к аппаратным регистрам. Пример созданных плагином сегментов для семейства микроконтроллеров STM32F0x1 показан на рисунке и соответствует адресам периферийных регистров, которые мы рассматривали пару страниц назад:

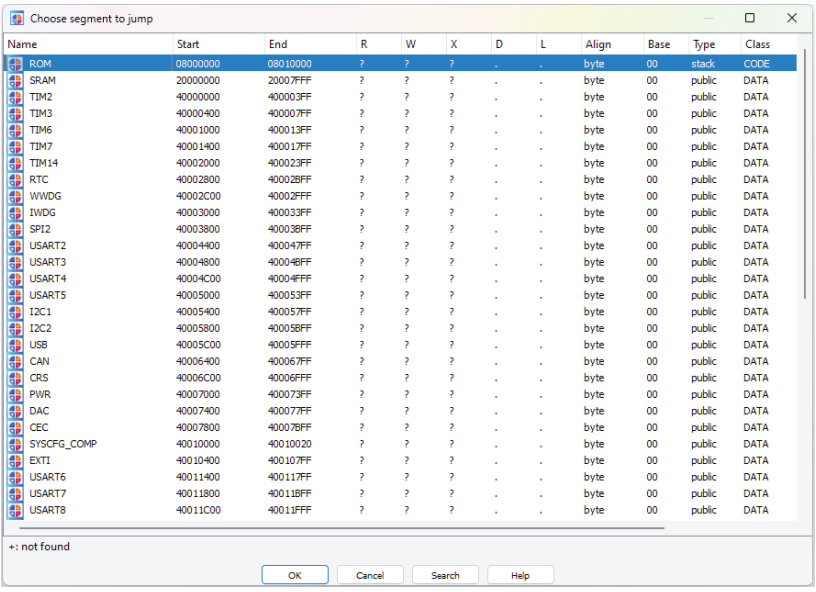


Рис. 2.44. Сегменты в дизассемблере IDA Pro, созданные плагином FirmLoader для микроконтроллера семейства STM32F0x1

Декодеры адресного пространства и банки памяти

Если для микроконтроллера требуется больше памяти, чем можно адресовать через ядро, или если подключение аппаратного блока памяти нельзя выполнить целиком напрямую в адресное пространство ядра, применяется аппаратный декодер адресного пространства. За счет его настройки (через периферийные регистры) можно отображать разные участки памяти на один и тот же диапазон адресного пространства микроконтроллера без перезагрузки кода в ОЗУ. Например, на рисунке представлена схема памяти, позволяющая отобразить 4 блока (bank или overlay) по 4 Кб на один участок адресного пространства микроконтроллера по адресу 0x6000–0x7000. То есть если у микроконтроллера адресное пространство размером 32 Кб (0x8000 байт), подобный подход позволит загрузить прошивку объемом до 44 Кб (0x8000 + 3 * 0x1000).

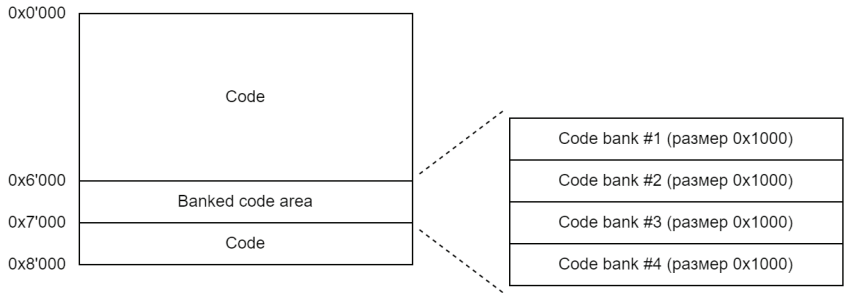


Рис. 2.45. Пример адресного пространства микроконтроллера с декодером

Аппаратный декодер адресного пространства часто применяется для отключения ROM-памяти микроконтроллера при загрузке внешней прошивки. После загрузки внешней прошивки запись в специальный конфигурационный регистр декодера приводит к переключению областей памяти. В адресах, где ранее располагался код ROM, будет находиться код загруженной прошивки. Это один из плохих сценариев для исследователя устройств. Ведь при такой организации памяти внутри микроконтроллера не всегда получится считать содержимое ROM-памяти, если доступ к микроконтроллеру получен уже в процессе исполнения внешней прошивки.

При реверс-инжиниринге прошивки, применяющей банки памяти, удобнее всего использовать для каждого банка отдельный сегмент. Использование сегментов – мощный инструмент, применимый также и при анализе прошивок многоядерных микроконтроллеров.

Межъядерное взаимодействие

В некоторых микроконтроллерах может быть несколько процессорных ядер. Такая архитектура позволяет ускорить выполнение кода за счет разнесения различных функций прошивки по разным исполнительным ядрам. Ядра могут иметь как одинаковую архитектуру, так и различную, например низкопроизводительное (и энергоэффективное) ядро для опроса датчиков и высокопроизводительное для обработки информационных потоков. На разных ядрах даже могут исполняться совсем разные прошивки. Для обмена информацией между

ядрами (межъядерное взаимодействие) в микроконтроллере может быть выделен специальный блок общей памяти (shared memory). Этот блок памяти одновременно отображается в адресное пространство всех ядер микроконтроллера. Пример коммуникации с помощью механизма IPC и shared memory из документации на микроконтроллер STM32H7 (включающий в себя по одному ядру ARM Cortex-M7 и ARM Cortex-M4) показан на рисунке:

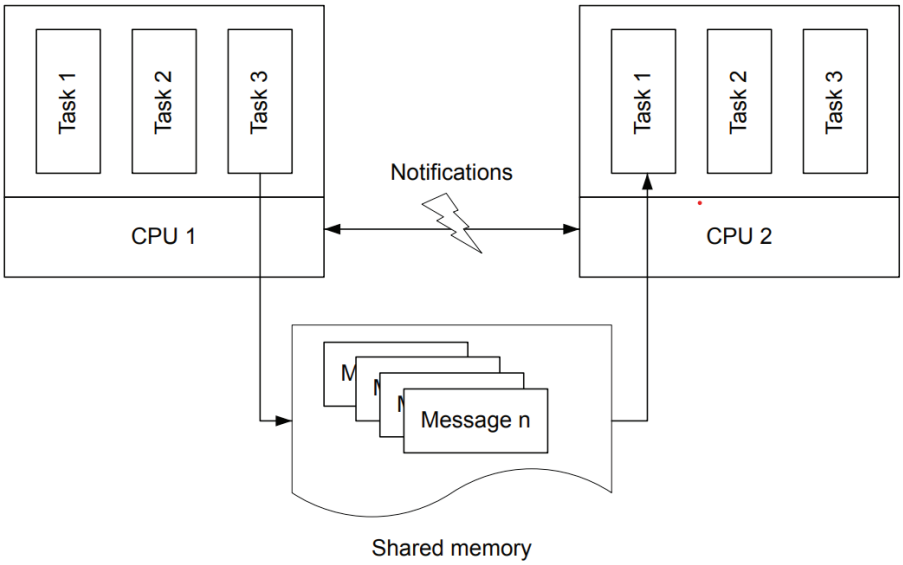


Рис. 2.46. Использование shared memory для межъядерного взаимодействия[1]

Определение адреса загрузки прошивки

Теперь мы знаем, что прошивка должна быть загружена по определенному адресу в адресном пространстве микроконтроллера. Допустим, мы нашли документацию на микроконтроллер и можем попробовать указать соответствующий адрес загрузки в дизассемблере. А если микроконтроллер не документирован или сильно кастомизирован на этапе проектирования? Попробуем посмотреть, за что можно «зацепиться» для определения адреса загрузки. Первым делом прошивку все же нужно загрузить в дизассемблер с адресом по умолчанию (как правило, это 0x00).

Статические ссылки

В листинге дизассемблера с большой долей вероятности встретятся статические ссылки. Это могут быть ссылки на код или данные (например, строки), расположенные по фиксированным адресам. Это самый простой путь для определения корректности адреса загрузки прошивки. В листинге показана загрузка строки из фиксированного адреса:

[1] https://www.st.com/resource/en/application_note/dm00771441-stm32h745755-and-stm32h747757-lines-interprocessor-communications-stmicroelectronics.pdf.

```
R3, #0x40005A20
R2, #0
R2, [R3]
R3, #aAllTestsPassed
loc_393A
R3, #0x200000E8  aAllTestsPassed DCB "All tests PASSED SUCCESSFULLY!!!",0xD,0xA,0
R3, [R3]                                          ; DATA XREF: sub_544+42↑o
```

Рис. 2.47. Пример статической ссылки

Прерывания

Прерывания представляют собой механизм, который позволяет микроконтроллеру реагировать на внешние или внутренние события. При наступлении некоторого события в процессоре возникает сигнал, заставляющий процессор прервать выполнение текущей процедуры и передать управление на специальный адрес в памяти, в котором хранится адрес процедуры обработки прерывания. После окончания процедуры обработки прерывания (если это не прерывание критической ошибки) управление возвращается на прерванное место в основной процедуре.

В современных микроконтроллерах существует множество источников прерываний: таймеры, различные интерфейсные блоки и исключения. Для корректного вызова соответствующего обработчика прерывания ссылка на него (вектор прерывания) должна быть расположена по фиксированному физическому адресу. Ссылки формируют таблицу векторов прерываний, структура которой зависит от используемого ядра микроконтроллера и особенностей реализации микроконтроллера производителем. Например, для микроконтроллера Atmel SAM3S4B, построенного на базе архитектуры ARM Cortex-M3, таблица должна иметь вид:

Exception number	IRQ number	Offset	Vector
45	29	0x00B4	IRQ29
.		.	.
.		.	.
.		.	.
		0x004C	
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	Reserved
1		0x0004	Reset
		0x0000	Initial SP value

Рис. 2.48. Описание таблицы векторов микроконтроллера Atmel SAM3S4B[1]

[1] https://www.keil.com/dd/docs/datashts/atmel/sam3s/sam3s_datasheet.pdf.

Ранее, в примере главы 1, мы уже считывали прошивку для данного микроконтроллера, еще раз приведем ее начало:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h	C0	3B	00	20	F9	2A	40	00	F7	2A	40	00	F7	2A	40	00	À;. ù*@.÷*@.÷*@.
0010h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	00	00	00	00	÷*@.÷*@.÷*@.....
0020h	00	00	00	00	00	00	00	00	00	00	00	00	F7	2A	40	00	............÷*@.
0030h	F7	2A	40	00	00	00	00	00	F7	2A	40	00	F7	2A	40	00	÷*@.....÷*@.÷*@.
0040h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	÷*@.÷*@.÷*@.÷*@.
0050h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	÷*@.÷*@.÷*@.÷*@.
0060h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	B5	28	40	00	÷*@.÷*@.÷*@.µ(@.
0070h	C9	28	40	00	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	É(@.÷*@.÷*@.÷*@.
0080h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	÷*@.÷*@.÷*@.÷*@.
0090h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	÷*@.÷*@.÷*@.÷*@.
00A0h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	÷*@.÷*@.÷*@.÷*@.
00B0h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	÷*@.÷*@.÷*@.÷*@.
00C0h	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	F7	2A	40	00	÷*@.÷*@.÷*@.÷*@.
00D0h	08	B5	05	48	05	4B	19	1A	06	29	00	D8	08	BD	04	4A	.µ.H.K...).Ø.½.J
00E0h	00	2A	FB	D0	90	47	F9	E7	E8	9B	40	00	EB	9B	40	00	.*ûÐ.GùçèŒ@.ëŒ@.
00F0h	00	00	00	00	08	B5	07	48	07	4B	19	1A	8A	10	02	EB	.....µ.H.K..Š..ë

Рис. 2.49. Таблица векторов микроконтроллера Atmel SAM3S4B в начале прошивки

Соотнеся адрес расположения обработчика прерывания с фактическим адресом обработчика в загруженном коде прошивки, можно понять корректность загрузки в дизассемблер. Как проще всего найти обработчик прерывания? Скорее всего, он будет располагаться где-то в начале прошивки, и его хорошо видно по стандартным кускам кода с сохранением и восстановлением контекста исполнения (в общем случае последовательно идущих команд push в начале обработчика и pop в конце для всех изменяемых в обработчике регистров микроконтроллера), а также по работе с регистрами периферийных контроллеров.

```
;-----------[ int_08: System timer ]------------;
int08_handler proc near                          ;
        push    ax                               ;
        push    es                               ;
                                                 ;
        mov     ax, 40h                          ;
        mov     es, ax                           ;
        inc     dword ptr es:[TCOUNT]            ;

        EndOfInterrupt()                         ;
        pop     es                               ;
        pop     ax                               ;
        iret                                     ;
int08_handler endp                               ;
;------------------------------------------------;

TCOUNT  equ     6Ch;                             ;
```

Рис. 2.50. Инструкция возврата из прерывания

Не всегда обработчики прерываний могут располагаться в начале прошивки. Некоторые микроконтроллеры позволяют гибко настраивать контроллер прерываний, и в таком случае адрес обработчика может быть любым.

Второй способ быстро найти обработчик прерывания – поискать архитектурно специфичные команды сохранения/восстановления контекста и возврата из прерывания для соответствующего микропроцессорного ядра (если они есть). В примере выше это команда IRET для x86.

Ошибки при загрузке в дизассемблер

С опытом вы поймете, когда загруженный код дизассемблируется неправильно. Вроде команды корректно распознаются, но код не имеет смысла. Например, осуществляется загрузка значений в одни регистры, а последующая операция выполняется с другими регистрами.

```
ROM:00003C2C        LSLS        R7, R7, #3
ROM:00003C2E        SUBS        R4, R0, R0
ROM:00003C30        MOVS        R1, R0
ROM:00003C32        MOVS        R0, R0
ROM:00003C34        LSLS        R0, R6, #2
ROM:00003C36        MOVS        R0, #0
ROM:00003C38        MOVS        R1, R1
ROM:00003C3A        MOVS        R0, R0
ROM:00003C3C        LSLS        R1, R1, #8
ROM:00003C3E        MOVS        R4, R4
ROM:00003C40        LSLS        R1, R0, #4
ROM:00003C42        STRH        R0, [R0]
ROM:00003C44        LSRS        R2, R6, #4
ROM:00003C46        MOVS        R4, R0
ROM:00003C48        MOVS        R0, R0
ROM:00003C4A        LSLS        R6, R7, #7
ROM:00003C4C        LSLS        R2, R0, #0x10
ROM:00003C4E        LSLS        R1, R1, #0x10
ROM:00003C50        LSLS        R0, R0, #4
ROM:00003C52        CDP         p2, 0, c0,c0,c1, 0
ROM:00003C56        LSRS        R5, R0, #4
ROM:00003C58        LSLS        R1, R4, #4
```

Рис. 2.51. Ошибка парсинга в интерфейсе дизассемблера

Значит, дизассемблер пытается сделать что-то некорректно (и ему довольно часто удаются странные вещи), например интерпретировать данные как код. Или распространенная ошибка со сменой битности системы команд в ARM: дизассемблер начал разбор кода для 32-битной системы команд, не смог автоматически определить часть кода, где происходит смена режима на Thumb (или мы сами принудительно стали интерпретировать код в ручном режиме). В итоге с какого-то момента в листинге код превратился в логическую бессмыслицу, т. к. дизассемблер пытается превратить 16-битные инструкции в 32-битные.

Что стоит искать в дизассемблированном коде прошивки в первую очередь

Мы уже значительно продвинулись в нашей подготовке к дизассемблированию прошивки устройства. Скорее всего, она уже корректно загружена (по правильным адресам и с правильным распределением на модули, если они есть).

Дизассемблер немного «подумал» и выдал нам листинг. На что же стоит обратить внимание в первую очередь?

Строки (если они есть)

Логично, что в первую очередь мы будем искать строки. Конечно, их может и не быть, ведь у многих устройств нет явно выраженного канала взаимодействия с человеком (например, есть только кнопки и светодиодные индикаторы статуса), а протоколы общения в большинстве своем бинарные. Но если устройство имеет довольно сложную логику работы и построено на базе ОС, строки вы с большой вероятностью увидите. Из них вы можете узнать тип ОС и версию ее ядра, список используемых при разработке прошивки библиотек или даже найти отладочные сообщения. Все – как и в обычном реверс-инжиниринге ПО.

Константы

Как и строки, известные константы стоить искать при любом реверс-инжиниринге одними из первых. Но набор искомых констант в ПО и в мире встраиваемых систем сильно отличается. Конечно, первым делом стоит поискать известные константы, применяющиеся в криптографии, например 0x10001 (значение публичной экспоненты алгоритма RSA), значения полиномов (0x04C11DB7 для CRC32) и т. п. Для этих целей удобнее всего использовать плагины для популярных дизассемблеров, например findcrypt (**https://github.com/polymorf/findcrypt-yara**) для IDA Pro, пример работы которого показан на рисунке:

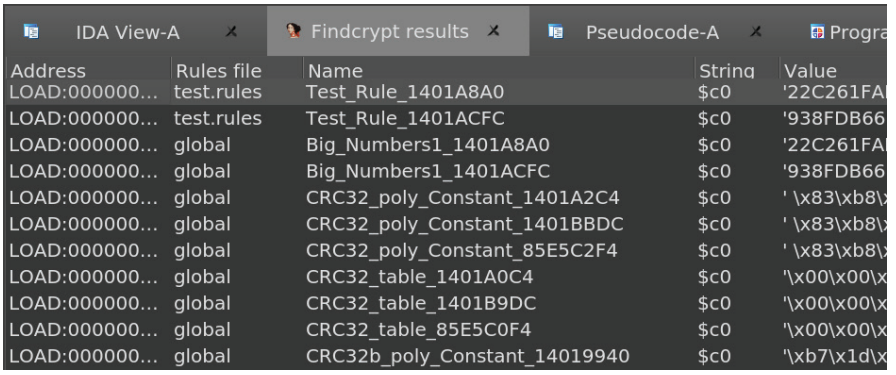


Рис. 2.52. Результат работы плагина findcrypt-yara[1]

В дальнейших поисках нам поможет информация, собранная на нулевом этапе – в ходе первичного анализа устройства. Из него мы знаем, какие интерфейсы применяются в нашем устройстве, и можем зацепиться за коды команд или магические сигнатуры, которые должны использоваться при работе различных протоколов и интерфейсов.

Например, если устройство общается командами SCSI по протоколу USB Mass Storage, то в начале любого пакета с командой от хоста к устройству будет

[1] https://github.com/polymorf/findcrypt-yara.

сигнатура 0x43425355 (если перевести эти байты в значения из таблицы ASCII, то получим строку «USBC» – заголовок пакета USB Command). Эта информация есть в документации на USB Mass Storage Bulk Only Transport:

Table 5.1 - Command Block Wrapper

bit / Byte	7	6	5	4	3	2	1	0
0-3	*dCBWSignature*							
4-7	*dCBWTag*							
8-11 (08h-0Bh)	*dCBWDataTransferLength*							
12 (0Ch)	*bmCBWFlags*							
13 (0Dh)	Reserved (0)				*bCBWLUN*			
14 (0Eh)	Reserved (0)			*bCBWCBLength*				
15-30 (0Fh-1Eh)	*CBWCB*							

dCBWSignature:

Signature that helps identify this data packet as a CBW. The signature field shall contain the value 43425355h (little endian), indicating a CBW.

Рис. 2.53. Структура пакета CBW[1]

А в начале пакета ответа на команду должна быть сигнатура 0x53425355 («USBS» – заголовок пакета USB Status):

Table 5.2 - Command Status Wrapper

bit / Byte	7	6	5	4	3	2	1	0
0-3	*dCSWSignature*							
4-7	*dCSWTag*							
8-11 (8-Bh)	*dCSWDataResidue*							
12 (Ch)	*bCSWStatus*							

dCSWSignature:

Signature that helps identify this data packet as a CSW. The signature field shall contain the value 53425355h (little endian), indicating CSW.

Рис. 2.54. Структура пакета CSW[2]

А вот как это выглядит в листинге дизассемблера:

[1] https://www.usb.org/sites/default/files/usbmassbulk_10.pdf.

[2] https://www.usb.org/sites/default/files/usbmassbulk_10.pdf.

```
P1:0000B818                 WEAK ScsiCbwValid
P1:0000B818 ScsiCbwValid                            ; CODE XREF: ScsiCommImpl+2↑p
P1:0000B818                 PUSH    {R4,LR}
P1:0000B81A                 LDR     R4, =Cbw
P1:0000B81C                 MOVS    R0, R4
P1:0000B81E                 BL      _Veneer_5__3__for___aeabi_uread4
P1:0000B822                 LDR     R1, =0x43425355
P1:0000B824                 CMP     R0, R1
P1:0000B826                 BNE     __ScsiCbwValid_1
P1:0000B828                 LDRB    R0, [R4,#(byte_7FD00AB5 - 0x7FD00AA8)]
P1:0000B82A                 CMP     R0, #0
P1:0000B82C                 BNE     __ScsiCbwValid_1
P1:0000B82E                 LDR     R0, =BotStatus
P1:0000B830                 LDRB    R0, [R0]
P1:0000B832                 LSLS    R0, R0, #0x18
P1:0000B834                 LSRS    R0, R0, #0x18
P1:0000B836                 LSRS    R0, R0, #2
P1:0000B838                 LSLS    R0, R0, #0x1F
P1:0000B83A                 BPL     __ScsiCbwValid_2
P1:0000B83A ; End of function ScsiCbwValid
P1:0000B83A
```

Рис. 2.55. Пример проверки сигнатуры «USBC» в обработчике USB прошивки микроконтроллера с архитектурой ARM

При поиске различных констант обязательно надо помнить, что для разных архитектур и систем команд внешний вид кода, использующего константу, может выглядеть совершенно по-разному. На рисунке выше (в листинге кода прошивки для архитектуры ARM) константа сигнатуры пакета Command Block Wrapper протокола USB Mass Storage присутствует в неизменном виде, т. к. регистры и команды ARM могут хранить и обрабатывать 32-битные числа. Для примера возьмем аналогичный код для архитектуры MCS-51, регистры в которой 8-битные.

```
code:0000D786 check_usbc:                           ; CODE XREF
code:0000D786                 mov     DPTR, #RAM_9518
code:0000D789                 movx    A, @DPTR
code:0000D78A                 mov     R6, A
code:0000D78B                 inc     DPTR
code:0000D78C                 movx    A, @DPTR
code:0000D78D                 xrl     A, #0x1F
code:0000D78F                 orl     A, R6
code:0000D790                 jnz     code_D7AC
code:0000D792                 mov     DPTR, #RAM_9580_USBBUF
code:0000D795                 movx    A, @DPTR
code:0000D796                 xrl     A, #'U
code:0000D798                 jnz     code_D7AC
code:0000D79A                 inc     DPTR
code:0000D79B                 movx    A, @DPTR
code:0000D79C                 cjne    A, #'S, code_D7AC
code:0000D79F                 inc     DPTR
code:0000D7A0                 movx    A, @DPTR
code:0000D7A1                 cjne    A, #'B, code_D7AC
code:0000D7A4                 inc     DPTR
code:0000D7A5                 movx    A, @DPTR
code:0000D7A6                 cjne    A, #'C, code_D7AC
code:0000D7A9                 mov     R7, #1
code:0000D7AB                 ret
```

Рис. 2.56. Пример проверки сигнатуры «USBC» в обработчике USB прошивки микроконтроллера с архитектурой MCS-51

Видно, что значение сигнатуры 0x43425355 проверяется побайтно, т. к. микроконтроллер не может загрузить 32-битное число в регистры. В таком случае бесполезно искать всю контанту целиком. Также не стоит забывать про различный порядок байтов (Little Endian и Big Endian) в разных архитектурах. Эта особенность также может затруднить поиск констант более чем из одного байта.

Какие еще константы могут быть в прошивке и, главное, как узнать, где можно поискать информацию по ним? Так как в устройствах почти всегда используются какие-то стандартные интерфейсы, стоит внимательно изучить документацию на них и выписать в шпаргалку список основных команд/сигнатур. Например:

- если устройство работает с SD-картой, то оно должно считывать и записывать на нее информацию. Значения кодов команд чтения информации с SD-карты – это 0x11 (CMD17 Single Read), 0x12 – CMD18 Multiple Read. Записи – 0x18 (CMD24 Single Write), 0x19 (CMD25 Multiple Write). Эти значения должны быть в прошивке, иначе взаимодействовать с SD-картой устройство не сможет. Даже в простейшей библиотеке SD Card для Arduino, поддерживающей только работу с SD-картами по интерфейсу SPI, есть эти команды:

```
// www.sdcard.org/developers/tech/sdcard/pls/Simplified_Physical_Layer_Spec.pdf
//------------------------------------------------------------------------------
// SD card commands
/** GO_IDLE_STATE - init card in spi mode if CS low */
uint8_t const CMD0 = 0X00;
/** SEND_IF_COND - verify SD Memory Card interface operating condition.*/
uint8_t const CMD8 = 0X08;
/** SEND_CSD - read the Card Specific Data (CSD register) */
uint8_t const CMD9 = 0X09;
/** SEND_CID - read the card identification information (CID register) */
uint8_t const CMD10 = 0X0A;
/** SEND_STATUS - read the card status register */
uint8_t const CMD13 = 0X0D;
/** READ_BLOCK - read a single data block from the card */
uint8_t const CMD17 = 0X11;
/** WRITE_BLOCK - write a single data block to the card */
uint8_t const CMD24 = 0X18;
/** WRITE_MULTIPLE_BLOCK - write blocks of data until a STOP_TRANSMISSION */
uint8_t const CMD25 = 0X19;
```

Рис. 2.57. Определение константных значений команды SD

```
//------------------------------------------------------------------------------
/**
   Read part of a 512 byte block from an SD card.

   \param[in] block Logical block to be read.
   \param[in] offset Number of bytes to skip at start of block
   \param[out] dst Pointer to the location that will receive the data.
   \param[in] count Number of bytes to read
   \return The value one, true, is returned for success and
   the value zero, false, is returned for failure.
*/
uint8_t Sd2Card::readData(uint32_t block,
                          uint16_t offset, uint16_t count, uint8_t* dst) {
  if (count == 0) {
    return true;
  }
  if ((count + offset) > 512) {
    goto fail;
  }
  if (!inBlock_ || block != block_ || offset < offset_) {
    block_ = block;
    // use address if not SDHC card
    if (type() != SD_CARD_TYPE_SDHC) {
      block <<= 9;
    }
    if (cardCommand(CMD17, block)) {
      error(SD_CARD_ERROR_CMD17);
      goto fail;
    }
    if (!waitStartBlock()) {
      goto fail;
    }
    offset_ = 0;
    inBlock_ = 1;
  }
```

Рис. 2.58. Использование константных значений команд SD

- если у устройства есть микросхема ПЗУ с интерфейсом NAND, то есть набор стандартных команд, значения которых также должны быть в прошивке. Их коды можно посмотреть как в документации на микросхему NAND-памяти (притом практически на любую, т. к. протокол на уровне базовых команд стандартизован), так и в спецификации ONFI (Open NAND Flash Interface);

Command	O/M	1st Cycle	2nd Cycle	Acceptable while Accessed LUN is Busy	Acceptable while Other LUNs are Busy	Target level commands
Read	M	00h	30h		Y	
Multi-plane	O	00h	32h		Y	
Copyback Read	O	00h	35h		Y	
Change Read Column	M	05h	E0h		Y	
Change Read Column Enhanced	O	06h	E0h		Y	
Read Cache Random	O	00h	31h		Y	
ODT Disable	O	1Bh		Y	Y	Y
ODT Enable	O	1Ch		Y	Y	Y
Read Cache Sequential	O	31h			Y	
Read Cache End	O	3Fh			Y	
Block Erase	M	60h	D0h		Y	
Multi-plane	O	60h	D1h		Y	
Read Status	M	70h		Y	Y	
Read Status Enhanced	O	78h		Y	Y	
Page Program	M	80h	10h		Y	
Multi-plane	O	80h	11h		Y	
Page Cache Program	O	80h	15h		Y	
Copyback Program	O	85h	10h		Y	
Multi-plane	O	85h	11h		Y	
Small Data Move[2]	O	85h	11h		Y	
Change Write Column[1]	M	85h			Y	
Change Row Address[1]	O	85h			Y	
Read ID	M	90h				Y
Volume Select[3]	O	E1h		Y	Y	
ODT Configure[3]	O	E2h				
Read Parameter Page	M	ECh				Y
Read Unique ID	O	EDh				Y
Get Features	O	EEh				Y
Set Features	O	EFh				Y
Command Based DCC Training	O	18h				
Read DQ Training	M	62h				
Write TX DQ Training Pattern	M	63h				
Write TX DQ Training Readback	M	64h				
Write RX DQ Training	O	76h				
LUN Get Features	O	D4h			Y	
LUN Set Features	O	D5h			Y	
ZQ Calibration Short	O	D9h			Y	
ZQ Calibration Long	O	F9h			Y	
Reset LUN	O	FAh		Y	Y	
Synchronous Reset	O	FCh		Y	Y	Y
Reset	M	FFh		Y	Y	Y

Рис. 2.59. Стандартные команды NAND из спецификации ONFI[1]

- а если в ходе первичного анализа мы нашли акселерометр ADXL345, подключенный по шине I2C, то, посмотрев на его документацию, можно найти значения адресов регистров, использующихся для управления и считывания данных:

[1] https://media-www.micron.com/-/media/client/onfi/specs/onfi_5_0_gold.pdf?la=en&rev=b9d79143b14143a7a8253c1ae20b247c.

Address					
Hex	Dec	Name	Type	Reset Value	Description
0x00	0	DEVID	R	11100101	Device ID
0x01 to 0x1C	1 to 28	Reserved			Reserved; do not access
0x1D	29	THRESH_TAP	R/W	00000000	Tap threshold
0x1E	30	OFSX	R/W	00000000	X-axis offset
0x1F	31	OFSY	R/W	00000000	Y-axis offset
0x20	32	OFSZ	R/W	00000000	Z-axis offset
0x21	33	DUR	R/W	00000000	Tap duration
0x22	34	Latent	R/W	00000000	Tap latency
0x23	35	Window	R/W	00000000	Tap window
0x24	36	THRESH_ACT	R/W	00000000	Activity threshold
0x25	37	THRESH_INACT	R/W	00000000	Inactivity threshold
0x26	38	TIME_INACT	R/W	00000000	Inactivity time
0x27	39	ACT_INACT_CTL	R/W	00000000	Axis enable control for activity and inactivity detection
0x28	40	THRESH_FF	R/W	00000000	Free-fall threshold
0x29	41	TIME_FF	R/W	00000000	Free-fall time
0x2A	42	TAP_AXES	R/W	00000000	Axis control for single tap/double tap
0x2B	43	ACT_TAP_STATUS	R	00000000	Source of single tap/double tap
0x2C	44	BW_RATE	R/W	00001010	Data rate and power mode control
0x2D	45	POWER_CTL	R/W	00000000	Power-saving features control
0x2E	46	INT_ENABLE	R/W	00000000	Interrupt enable control
0x2F	47	INT_MAP	R/W	00000000	Interrupt mapping control
0x30	48	INT_SOURCE	R	00000010	Source of interrupts
0x31	49	DATA_FORMAT	R/W	00000000	Data format control
0x32	50	DATAX0	R	00000000	X-Axis Data 0
0x33	51	DATAX1	R	00000000	X-Axis Data 1
0x34	52	DATAY0	R	00000000	Y-Axis Data 0
0x35	53	DATAY1	R	00000000	Y-Axis Data 1
0x36	54	DATAZ0	R	00000000	Z-Axis Data 0
0x37	55	DATAZ1	R	00000000	Z-Axis Data 1
0x38	56	FIFO_CTL	R/W	00000000	FIFO control
0x39	57	FIFO_STATUS	R	00000000	FIFO status

Рис. 2.60. Описание управляющих регистров акселерометра ADXL345[1]

То есть если из акселерометра идет считывание значений ускорения (из регистров по адресам 0x32-0x37), то в прошивке где-то должны быть такие константы. Вот как эти константы выглядят в исходном коде прошивки (для примера возьмем библиотеку SparkFun ADXL345 Arduino Library):

```
46  #define ADXL345_INT_SOURCE      0x30    // Source of Interrupts
47  #define ADXL345_DATA_FORMAT     0x31    // Data Format Control
48  #define ADXL345_DATAX0          0x32    // X-Axis Data 0
49  #define ADXL345_DATAX1          0x33    // X-Axis Data 1
50  #define ADXL345_DATAY0          0x34    // Y-Axis Data 0
51  #define ADXL345_DATAY1          0x35    // Y-Axis Data 1
52  #define ADXL345_DATAZ0          0x36    // Z-Axis Data 0
53  #define ADXL345_DATAZ1          0x37    // Z-Axis Data 1
54  #define ADXL345_FIFO_CTL        0x38    // FIFO Control
55  #define ADXL345_FIFO_STATUS     0x39    // FIFO Status
```

Рис. 2.61. Определение адресов регистров акселерометра ADXL345 в исходном коде библиотеки

Функция, считывающая значения ускорения, считывает сразу 6 байт из подряд расположенных регистров, начинающихся с адреса 0x32 (DATAX0):

[1] https://www.analog.com/ADXL345.

```
73  void ADXL345::readAccel(int *x, int *y, int *z) {
74      readFrom(ADXL345_DATAX0, ADXL345_TO_READ, _buff);   // Read Accel Data from ADXL345
75
76      // Each Axis @ All g Ranges: 10 Bit Resolution (2 Bytes)
77      *x = (int16_t)((((int)_buff[1]) << 8) | _buff[0]);
78      *y = (int16_t)((((int)_buff[3]) << 8) | _buff[2]);
79      *z = (int16_t)((((int)_buff[5]) << 8) | _buff[4]);
80  }
```

Рис. 2.62. Обращение к регистрам акселерометра ADXL345 в исходном коде библиотеки

И как этот же участок кода, включенный в прошивку, выглядит в листинге дизассемблера (для архитектуры ARM)? И в данной прошивке это единственный участок кода, который ищется по значению константы 0x32.

```
                        PUSH    {z-R7,LR}
                        MOV     R4, this
                        MOV     R7, x
                        MOV     R6, y
                        MOV     R5, z
                        MOVS    y, #6           ; num
y = R6                                          ; int *
                        ADD.W   z, this, #0x20 ; ' ' ; _buff
z = R5                                          ; int *
                        MOVS    x, #0x32 ; '2'  ; address
x = R7                                          ; int *
                        BL      _ZN7ADXL3458readFromEhiPh ; .
                        LDRB.W  R2, [this,#0x21]
                        LDRB.W  R3, [this,#0x20]
                        ORR.W   R3, R3, R2,LSL#8
                        SXTH    R3, R3
                        STR     R3, [x]
                        LDRB.W  R2, [this,#0x23]
                        LDRB.W  R3, [this,#0x22]
                        ORR.W   R3, R3, R2,LSL#8
                        SXTH    R3, R3
                        STR     R3, [y]
                        LDRB.W  R2, [this,#0x25]
                        LDRB.W  R3, [this,#0x24]
                        ORR.W   R3, R3, R2,LSL#8
                        SXTH    R3, R3
                        STR     R3, [z]
                        POP     {R3-x,PC}
```

Рис. 2.63. Обращение к регистрам акселерометра ADXL345 в коде прошивки

Надеюсь, эти примеры отлично показывают основную идею: в мире исследования встраиваемых систем существует множество констант, знание которых существенно помогает и ускоряет реверс-инжиниринг прошивки устройства.

Обработчики интерфейсов и взаимодействие с аппаратными регистрами

Этот пункт тесно связан с предыдущим, ведь большинство констант из прошлого примера как раз связаны со взаимодействием по каким-либо интерфейсам микроконтроллера. Например, найденные константы 0x0D и 0x0A (возврат каретки и перевод строки) наталкивают нас на мысль, что они могут быть встречены в обработчике вывода строк в диагностический интерфейс,

например UART. Взаимодействие по интерфейсам в микроконтроллере связано с чтением и записью информации из каких-то периферийных регистров. Например, в микроконтроллерах серии STM32F0 регистры управления первого USART-контроллера располагаются в физическом адресном пространстве в диапазоне 0x40013800-0x40013BFF:

Table 1. STM32F0xx peripheral register boundary addresses (continued)

Bus	Boundary address	Size	Peripheral	Peripheral register map
	0x4001 5C00 - 0x4001 7FFF	9 KB	Reserved	-
	0x4001 5800 - 0x4001 5BFF	1 KB	DBGMCU	*Section 32.9.6 on page 934*
	0x4001 4C00 - 0x4001 57FF	3 KB	Reserved	-
	0x4001 4800 - 0x4001 4BFF	1 KB	TIM17	*Section 20.6.17 on page 558*
	0x4001 4400 - 0x4001 47FF	1 KB	TIM16	*Section 20.6.17 on page 558*
	0x4001 4000 - 0x4001 43FF	1 KB	TIM15	*Section 20.5.19 on page 541*
	0x4001 3C00 - 0x4001 3FFF	1 KB	Reserved	-
	0x4001 3800 - 0x4001 3BFF	1 KB	USART1	*Section 27.8.12 on page 765*

Рис. 2.64. Адреса регистров периферийного контроллера USART1 микроконтроллера семейства STM32F0[1]

По адресу 0x40013800 расположен 32-битный регистр USART_CR1 (USART Control Register), поля которого представлены на рисунке:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Рис. 2.65. Структура полей регистра USART_CR1 периферийного контроллера USART1[2]

Из приведенной таблицы и описания ее полей видим, что для включения контроля четности (Parity Control) интерфейса USART1 необходимо записать бит 1 по смещению 10 (PCE, Parity Control Enable) в регистре USART_CR1. То есть все довольно логично и просто, главное – понимать принцип: работа с периферийными интерфейсами построена через взаимодействие с регистрами, отображенными в адресное пространство микроконтроллера.

Есть несколько особенностей, из-за чего я решил рассказать про обработку интерфейсов отдельно, а не ограничиться только предыдущим разделом со значениями различных констант. Первая особенность заключается в определенном фиксированном порядке записи значений в регистры микроконтроллера для организации правильного общения по интерфейсу. Поясню на примере. При работе по интерфейсу NAND для считывания содержимого страницы (NAND Page) из микросхемы памяти используется последовательность из двух команд со значениями 0x00 и 0x30 соответственно.

[1] https://www.st.com/resource/en/reference_manual/dm00031936-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

[2] https://www.st.com/resource/en/reference_manual/dm00031936-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

Command	O/M	1st Cycle	2nd Cycle	Acceptable while Accessed LUN is Busy	Acceptable while Other LUNs are Busy	Target level commands
Read	M	00h	30h		Y	
Multi-plane	O	00h	32h		Y	
Copyback Read	O	00h	35h		Y	
Change Read Column	M	05h	E0h		Y	

Рис. 2.66. Команда NAND Read[1]

Между ними должны быть записаны 5 байт адреса источника для считывания данных.

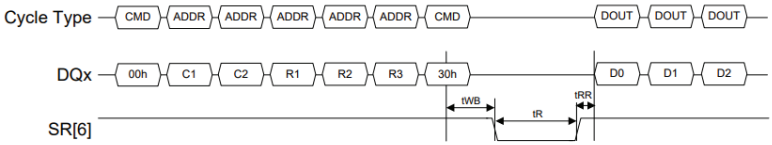


Рис. 2.67. Диаграмма команды NAND Read[2]

В исходном коде прошивки данная команда, скорее всего, выглядит очень похоже на последовательность из документации и легко читается. Для демонстрации рассмотрим код примера работы с NAND-памятью отладочной платы Olimex LPC-H11C14, построенной на базе микроконтроллера NXP LPC11C14FBD48 с ядром ARM Cortex-M0:

```
/*****************************************************************************
 * Function Name: NandReadPage
 * Parameters: Page - NAND Flash Page Number. It is Block*NAND_PG_PER_BLK + Page_in_Block
 *             Buffer_Index - Index to NFC Buffer (0-3)
 * Return: FLASH_OK - No Read Error or 1bit Error
 *         FLASH_ERROR - Non Correcable Read Error
 * Description: Reads one page (512 main+16 spera) from NAND Flash into RBA
 *
 *****************************************************************************/
unsigned int NandReadPage(unsigned char * dest, unsigned int Page,unsigned int Buff_Index)
{
    /*Read commad*/
    NandSetCmd = 0x00;
    /*Set Address*/
    AddrInOperation(528*(Page%4),Page/4);
    /*Clear Status Flags*/
    NandIRQStatusRaw = 0xffffffff;
    /*Read Command*/
    NandSetCmd = 0x30;
    // wait for device ready
    while (!NandIRQStatusRaw_bit.INT28R);
    /*Clear Status flags*/
    NandIRQStatusRaw = 0xffffffff;
    /*Start Reading*/
    NandControlFlow = 1;
    /*Wait reading and ECC to complete*/
    while(!NandIRQStatusRaw_bit.INT21R);
    /*Non Correctable error*/
    if( !NandIRQStatusRaw_bit.INT26R && NandIRQStatusRaw_bit.INT11R) return FLASH_ERROR;

    unsigned char * src = (unsigned char *) 0x70000000;

    for(int i = 0 ; i<516; i+=sizeof(unsigned char))
    {
      *dest++ = *src++;
    }

    return FLASH_OK;
}
```

Рис. 2.68. Использование команды NAND Read в исходном коде прошивки

[1] https://media-www.micron.com/-/media/client/onfi/specs/onfi_5_0_gold.pdf?la=en&rev=b9d79143b14143a7a8253c1ae20b247c.

[2] https://media-www.micron.com/-/media/client/onfi/specs/onfi_5_0_gold.pdf?la=en&rev=b9d79143b14143a7a8253c1ae20b247c.

А в листинге дизассемблера она уже может выглядеть по-другому:

```
                PUSH    {R4-R6,LR}      ; Alternative name is '.text_17'
                MOVS    R4, R0
                MOVS    R0, R1
                MOV     R5, #0
                MOV     R6, #0x17000820
                STR     R5, [R6]
                MOVS    R1, R0,LSR#2
                ANDS    R0, R0, #3
                ADD     R2, R0, R0,LSL#5
                MOVS    R0, R2,LSL#4
                BL      AddrInOperation
                LDR     R0, =_A_NandIRQStatusRaw
                MOV     R1, #0xFFFFFFFF
                STR     R1, [R0]
                MOV     R2, #0x30 ; '0'
                STR     R2, [R6]

__NandReadPage_0                        ; CODE XREF: __NandGetStatus_0+74↓j
                LDR     R2, [R0]
                MOV     R3, #1
                TST     R3, R2,LSR#28
                BEQ     __NandReadPage_0
                STR     R1, [R0]
                MOV     R1, #1
                LDR     R2, =_A_NandControlFlow
                STR     R1, [R2]
```

Рис. 2.69. Использование команды NAND Read в прошивке

Если не знать, что искать, очень сложно рассмотреть в этом фрагменте кода одну из самых важных операций при работе с NAND-памятью (команду Page Read). Ведь значения команд 0x0 и 0x30 не являются уникальными, и эти байты будут встречаться в прошивке множество раз, поиск значения констант не даст быстрого результата. Знание структуры команды NAND Page Read позволяет нам быстро идентифицировать этот фрагмент кода и уже через него найти регистры аппаратного NAND-контроллера в микроконтроллере. Ведь коды NAND-команд и адреса должны записываться в какие-то регистры для работы по интерфейсу NAND, и даже если нет документации на этот блок микроконтроллера, с помощью цепочки логических рассуждений карту регистров можно восстановить.

Вторая особенность, характерная для работы с регистрами управления какими-то аппаратными блоками микроконтроллера, – это бесконечные циклы с проверкой значения в каком-либо регистре (адресе памяти). Пример подобного участка кода показан на графе:

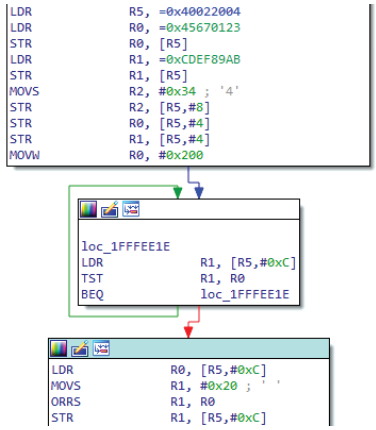


Рис. 2.70. Ожидание изменения значения в регистре периферийного контроллера

С точки зрения обычной программы, исполняющейся на прикладном уровне, такие циклы (без вызова каких-то дополнительных процедур), скорее всего, приведут к зависанию программы. В мире прошивок такие циклы – абсолютно нормальное явление, ведь значения в регистрах периферийных контроллеров могут изменяться аппаратно независимо от выполняющегося кода в ядре микроконтроллера. Подобные циклы характерны для ожидания каких-то событий, например конца отправки или начала получения данных из FIFO USART-контроллера. Или появления сигнала нужного уровня на проверяемой «ножке» микроконтроллера.

Главный цикл

Главный цикл программы или, в нашем случае, микропрограммы – то, что стоит найти в первую очередь, ведь тогда вы сориентируетесь в структуре кода и сможете получить точку отсчета при проведении исследований прошивки.

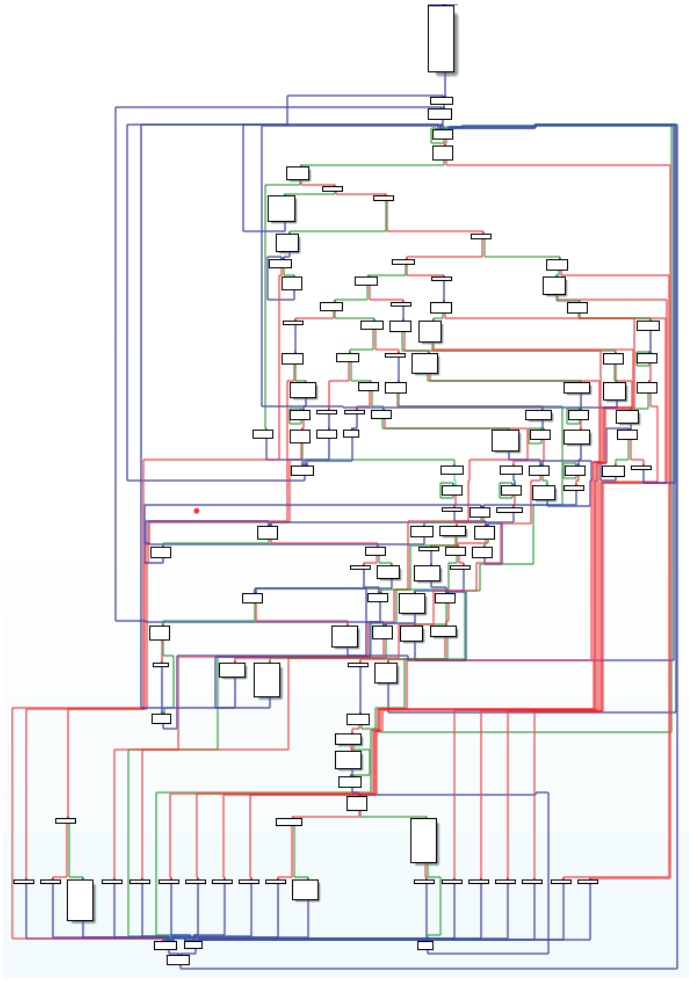


Рис. 2.71. Пример главного цикла прошивки

Главный цикл в каком-то виде будет в любой прошивке, хотя иногда большая часть логики работы устройства может быть реализована в обработчиках прерываний. Пора посмотреть, какие виды организации прошивок бывают и какие известные операционные системы могут быть в анализируемом устройстве.

Виды организаций прошивок и embedded ОС

Мир встраиваемых систем максимально разнообразен. В них могут применяться как низкопроизводительные и энергоэффективные микроконтроллеры, так и мощные микропроцессоры с несколькими ядрами и гигабайтами ОЗУ. В зависимости от вычислительной мощности устройства, его назначения и предпочтения разработчика в устройстве могут быть применены разные подходы к построению прошивки. Первый подход (bare-metal) применяется в основном в простых устройствах. Встречается также название Super Loop, отражающее его суть – большой бесконечный цикл с вызовами процедур, выставлением флагов в памяти и т. п. Второй подход – это использование ОС, которых в мире встраиваемых систем применяется множество, мы рассмотрим только основные и самые часто встречающиеся.

Загрузчик (Bootloader)

Ранее мы рассмотрели, что при старте микроконтроллера управление передается по специальному адресу в физической памяти – reset-вектору, указывающему, в свою очередь, на начало прошивки. Также мы говорили, что иногда часть прошивки реализуется в неизменяемой памяти (ROM) внутри микроконтроллера. И получить ее содержимое – одна из задач исследователя устройства, ведь именно она выполняет первичную настройку оборудования и может хранить в себе приватные данные. Существуют устройства, в которых код в ROM – это единственный исполняемый код. Но такие устройства крайне редки, т. к. обладают рядом недостатков, таких как ограниченный размер кода в ROM и отсутствие возможности обновления прошивки, поскольку код прошивки заложен в дизайн микроконтроллера на заводе, изменить его впоследствии нельзя. Следовательно, новые функции, обновления и исправления прошивки реализовать без производства на фабрике исправленной версии микроконтроллера не получится.

Именно поэтому широко используется концепция загрузчика, получающего управление при старте микроконтроллера и выполняющего ряд функций, в том числе:

1) проверку целостности основной прошивки (с помощью контрольных сумм или криптографической подписи, об этом подробнее в главе «Level 4. Механизмы защиты встраиваемых систем»);
2) инициализацию необходимой периферии и контекста, загрузку основной прошивки и передачу потока выполнения на нее;
3) активацию и реализацию режима восстановления (recovery) в случае повреждения основной прошивки;
4) обновление прошивки.

Код загрузчика может располагаться в ROM (тогда он называется BootROM), находиться во внутренней flash-памяти микроконтроллера или даже во внешней микросхеме памяти. При этом возможны любые комбинации этих вариантов хранения загрузчика. Это зависит от конкретной модели микроконтроллера. Функции загрузчика могут быть разделены на два или даже больше фрагментов кода, загружающихся последовательно. Как правило, загрузчик первого уровня (Primary Bootloader, PBL) располагается в BootROM и не может быть изменен. Его основная задача – загрузить вторичный загрузчик (Secondary Bootloader, SBL) или активировать режим восстановления при повреждении SBL. Вторичный загрузчик уже располагается в изменяемой памяти и обладает более широкими функциональными возможностями. Например, может включать в себя драйверы для инициализации периферии или драйвер файловой системы для загрузки основной ОС с карты памяти или чипа eMMC.

Например, схема загрузки устройства может выглядеть следующим образом:

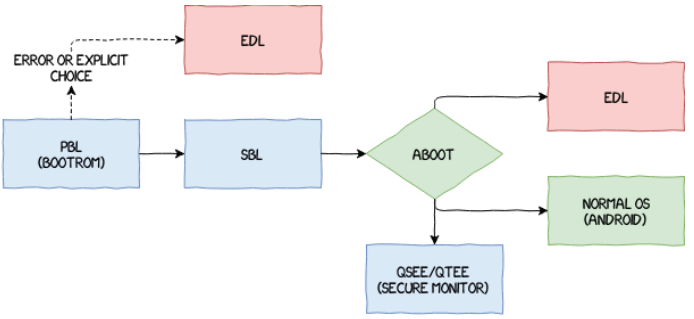


Рис. 2.72. Пример цепочки из нескольких загрузчиков[1]

Несмотря на гибкость и распространенность показанного подхода, у него есть существенный недостаток. В случае обнаружения уязвимости в PBL ее будет невозможно исправить без разработки и производства новой версии микроконтроллера. Именно таким типом уязвимости является checkm8 для мобильных процессоров Apple (**https://twitter.com/axi0mX/status/1177542201670168576**).

В простых устройствах, построенных на базе микроконтроллеров общего назначения начального уровня (например, серий STM32F0-STM32F1) или в проприетарных решениях, загрузчик может быть реализован разработчиком самостоятельно. Но существуют открытые реализации загрузчиков. Для микроконтроллеров базового уровня в качестве примера можно упомянуть OpenBLT (www.feaser.com/openblt/doku.php) и MCUBoot (**https://www.trustedfirmware.org/projects/mcuboot/index.html**). Код этих загрузчиков легко адаптируется под конкретные задачи разработчика и может быть портирован на различные архитектуры.

В устройствах с более мощными микроконтроллерами или процессорами почти стандартом является использование open-source загрузчика U-boot (**www.denx.de/wiki/U-Boot/**). Этот загрузчик

[1] https://blog.quarkslab.com/analysis-of-qualcomm-secure-boot-chains.html.

может быть очень гибко сконфигурирован для загрузки с любого типа памяти и поддерживает все распространенные типы файловых систем. Он может быть разделен на PBL и SBL, что делает его максимально универсальным способом организации загрузки ОС.

Выше мы рассмотрели концепцию простого загрузчика. В современных устройствах на базе высокопроизводительных микроконтроллеров (например, старших семейств ARM) уровней загрузчиков может быть гораздо больше двух. Например, в документации ARM Trusted Firmware (**https://chromium.googlesource.com/external/github.com/ARM-software/arm-trusted-firmware/+/v1.4-rc0/docs/user-guide.md**) содержится схема с пятью разными загрузчиками, выполняющими различные функции в процессе своей работы.

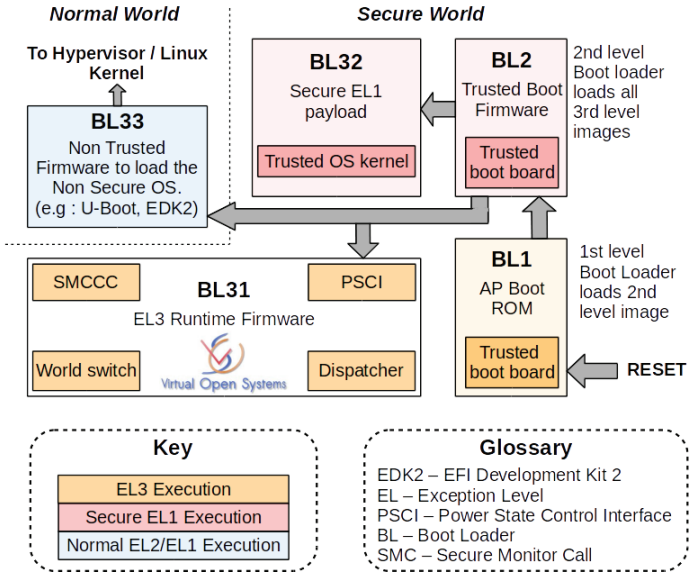


Рис. 2.73. Пример возможной цепочки загрузки микроконтроллеров с архитектурой ARM[1]

Bare-metal

Прошивки с bare-metal подходом активно используются, особенно в микроконтроллерах начального уровня и в простых устройствах. Как правило, используется архитектура прошивки с главным циклом Super Loop, хотя существуют варианты построения основной логики работы устройства в обработчиках прерываний. Какие преимущества у подхода bare-metal?

1. Простота и сравнительно небольшой размер кода для устройств с минимальным набором функций. Да, ранее мы сказали, что подход сложнее, чем разработка для ОС. Но для большинства микроконтроллеров производитель уже сделал SDK, реализующую самую сложную часть – взаимодействие с железом через периферийные регистры. Если функ-

[1] http://community.arm.com/docs/DOC-9306.

циональный код прошивки мал, микроконтроллер относится к классу начального уровня и имеет хороший SDK, то подход будет оправдан.

2. Меньше требования к ресурсам. Для ОС в общем случае нужен больший объем ПЗУ для хранения кода и ОЗУ для хранения контекстов задач. Для ОС также нужна большая производительность из-за трат времени на переключение задач.
3. Возможность разрабатывать код без погружения в особенности реализации железа. Ну почти. Для определенных применений, например связанных с обработкой большого количества состояний, может быть применен подход разработки математической модели прошивки на основе конечного автомата в специальном ПО, например MATLAB. Такой разработчик не взаимодействует с «железом», а описывает конечный автомат (FSM, Finite State Machine), т. е. состояния и правила перехода между ними. Далее этот проект компилируется в бинарный образ прошивки с помощью другого специализированного ПО. Таким образом достигается разделение труда высокоуровнего разработчика-математика, хорошо ориентирующегося в предметной области, и низкоуровневого embedded-разработчика, хорошо знакомого с «железом». По такому принципу строится прошивка некоторых блоков управления в современных автомобилях.

Real-time OS (RTOS)

Работа многих устройств связана с обработкой информации от каких-либо периферийных интерфейсов. Зачастую на такую обработку накладываются определенные временные ограничения. Например, промышленный контроллер, использующийся на заводе и управляющий технологическим процессом, должен успевать обрабатывать поступающую информацию с датчиков и в соответствии со своей программой выдавать управляющие сигналы на различные устройства, непосредственно задействованные в технологическом процессе: манипуляторы, клапаны и т. д. Так как любой технологический процесс имеет допустимые временные параметры операций (тайминги), контроллер должен гарантированно успеть выдать соответствующие управляющие сигналы. Следовательно, разработчик прошивки должен учитывать время выполнения отдельных участков кода, чтобы попасть в нужные тайминги.

Бывают устройства с прошивкой типа bare-metal, в которой нужные тайминги выдерживаются с помощью прерываний. Но существует и другой, более распространенный подход – использование операционных систем реального времени (ОСРВ, по-английски Real Time Operating Systems, RTOS). В RTOS каждому процессу выделяется фиксированный квант процессорного времени, т. е. ситуации, когда определенная процедура прошивки будет исполняться недетерминированное количество времени, архитектурно минимизированы (при правильно организованном коде, даже RTOS можно «сломать» неправильными действиями).

RTOS широко используются в различных устройствах, т. к. снимают с разработчика необходимость проектирования диспетчера задач и предоставляют

инструменты для решения задач синхронизации. Кратко рассмотрим самые распространенные OpenSource и проприетарные RTOS.

1. FreeRTOS (**freertos.org**). Одна из самых распространенных RTOS, поддерживающая более 40 процессорных архитектур и способная исполняться на микроконтроллерах самого базового уровня с минимальными ресурсными требованиями. Первый релиз был выпущен в 2003 году, и за 20 лет FreeRTOS стала широко использоваться во множестве цифровых устройств совершенно разных видов.

2. Zephyr (**www.zephyrproject.org**). Новая Open Source RTOS, первый релиз которой вышел в 2016 году, но уже получившая признание и популярность в среде embedded-разработчиков. Широко применяется в устройствах связи, мультимедиаустройствах и IoT. В списке поддерживаемых процессорных архитектур присутствуют все самые распространенные, включая ARM, ARC, MIPS, Xtensa и даже RICS-V.

3. VxWorks компании Wind River Systems (**www.windriver.com/products/vxworks**). Проприетарная ОС, использующаяся уже почти 40 лет. В списке поддерживаемых архитектур x86-64, ARM, PowerPC, RISC-V и MIPS. Широко используется в промышленных контроллерах, электронных блоках управления транспортных средств (автомобилей, самолетов, катеров и т. д.), медицинских устройствах и других применениях, где требуются надежность и высокая отказоустойчивость. В том числе на ОС VxWorks работает марсоход Curiosity.

4. QNX (**blackberry.qnx.com**). Изначально разработанная компанией Blackberry, эта RTOS получила широкое распространение в автомобильных блоках управления (включая системы беспилотного вождения) и промышленных контроллерах. Сферы применения и список поддерживаемых архитектур схож с ОС VxWorks.

5. ThreadX (**azure.microsoft.com/en-us/products/rtos/**) изначально была разработана компанией Express Logic и впоследствии выкуплена компанией Microsoft. Текущее официальное название Azure RTOS ThreadX. ThreadX поддерживает более 20 архитектур и лежит в основе микроконтроллеров, широко применяющихся во встраиваемых системах, включая носимые устройства, IoT, сетевые устройства (роутеры, маршрутизаторы) и модули связи.

6. Integrity (**www.ghs.com/products/rtos/integrity.html**) компании Green Hills Software. Аналогично VxWorks и QNX, эта RTOS используется в устройствах, где требуется повышенный уровень ответственности применения: медицинская отрасль, военная и аэрокосмическая отрасль, различные виды транспорта.

7. embOS (**www.segger.com/products/rtos/embos/**) компании SEGGER. Семейство RTOS с 30-летней историей. Бесплатная для некоммерческого использования, embOS поддерживает более 20 распространенных процессорных архитектур. Используется в IoT-устройствах, автомобильной промышленности, потребительских и сетевых устройствах.

Embedded Linux

Множество устройств работают под управлением ОС на основе Linux. Естественно, что использование полноценной ОС требует значительных ресурсов микроконтроллера, поэтому embedded Linux можно чаще увидеть на устройствах, обрабатывающих большие объемы информации, взаимодействующих с пользователем и имеющих сенсорные экраны. Фактически нет ни одного типа устройства, где нельзя встретить embedded Linux. Применение embedded Linux в устройствах для ответственного использования также оправдано тем, что инструментарий разработчика и код ОС открыт, в отличие от проприетарных ОС.

Архитектуры, на которых могут работать ОС типа embedded Linux, также лежат в основе соответствующих мощных микроконтроллеров: ARM, ARC, x86-64, AVR32, MIPS, PowerPC, Xtensa и RISC-V.

Известными примерами embedded Linux являются ОС для роутеров DD-WRT и OpenWrt, Debian для Raspberry Pi или Tizen, разрабатываемая и использующаяся Samsung в своих мультимедиаустройствах и smart-телевизорах. Ну и конечно же, нельзя не вспомнить Android, который также содержит в себе ядро Linux. На базе Android построено множество современных встраиваемых систем, от смартфонов и телевизоров до головных устройств автомобилей и модемов сотовых сетей.

Windows CE, Embedded и IoT

Несмотря на то что последний релиз WinCE был в 2013 году, ее все еще можно встретить в устройствах, имеющих длительный цикл разработки и время жизни, таких как промышленные контроллеры и мультимедиасистемы автомобилей. WinCE является ОС типа RTOS, была популярна во встраиваемых системах 2000-х годов и поддерживала архитектуры x86, MIPS, ARM, SuperH и PowerPC.

На смену WinCE компания Microsoft разработала семейство ОС Windows Embedded, в основе которых лежат ОС от Windows XP до Windows 8.1. Список поддерживаемых архитектур унаследовался от WinCE.

Последняя версия embedded ОС Microsoft носит название Windows IoT и основана на Windows 10. Она предназначена для шлюзов интернет-вещей, устройств умных домов, мобильных POS-терминалов, промышленных контроллеров, киосков, банкоматов, медицинского оборудования и многих других типов встраиваемых систем. Поддерживаются только архитектуры ARM и x86-64.

Рассмотрев, какие ОС и архитектуры могут быть внутри цифрового устройства, у читателя может возникнуть справедливый вопрос: а можно ли эмулировать прошивку устройства? Ответ во многих случаях: да, но есть особенности, о которых надо помнить. О них – в следующей главе.

Эмуляция

Несмотря на то что логично было бы отнести эмуляцию к динамическому анализу, я решил сделать ее переходным мостиком между чисто статическим реверс-инжинирингом прошивки и динамическим реверс-инжинирингом с использованием «железа» устройства, про которое мы поговорим в следующей главе. Эмуляция в мире исследований цифровых устройств может осложняться сразу несколькими факторами:

- отсутствием описания регистров периферии микроконтроллера;
- отсутствием эмулятора для нераспространенных архитектур/микроконтроллеров;
- отсутствием части прошивки или кода инициализации, например BootROM.

Несмотря на эти ограничения, эмуляция является мощным инструментом, позволяющим существенно ускорить процесс проведения исследований устройств. Рассмотрим два наиболее часто используемых эмулятора.

QEMU

QEMU (Quick Emulator) – open source эмулятор различных устройств, позволяющий запускать операционные системы, предназначенные под одну архитектуру, на другой. Кроме процессора, QEMU эмулирует множество различных периферийных устройств: сетевые карты, HDD, видеокарты, PCI, USB, DMA-контроллеры и пр.

QEMU может работать в нескольких режимах, основными являются User-mode emulation и System emulation. В режиме User-mode emulation QEMU эмулирует приложения для ОС Linux или Darwin/macOS, скомпилированные для различных архитектур (например, ARM). В режиме System emulation QEMU эмулирует ЭВМ полностью, включая различные контроллеры периферии. В этом режиме бинарный код инструкций эмулируемой архитектуры (например, ARM) конвертируется в промежуточный платформонезависимый код при помощи конвертора TCG (Tiny Code Generator), и затем этот платформонезависимый бинарный код конвертируется уже в код инструкций хостовой ЭВМ (например, x86). Список поддерживаемых эмулятором наборов инструкций включает в себя x86, MIPS, 32-bit ARMv7, ARMv8, PowerPC, RISC-V и MicroBlaze.

Unicorn Engine

Unicorn Engine – open source мультиплатформенный, легковесный и мультиархитектурный эмулятор процессора, основанный на Tiny Code Generator из QEMU. В отличие от QEMU, Unicorn не эмулирует периферию. При его использовании необходимо каждый раз реализовывать разметку памяти, загрузку данных и т. д. Зачем тогда использовать Unicorn Engine? При исследовании какого-либо кода часто не нужно эмулировать работу всего процесса. Достаточно смоделировать работу какой-то конкретной функции. И в этом Unicorn сильно помогает за счет своей простоты. Существуют плагины для популярных дизассемблеров, позволяющие использовать Unicorn в качестве отладчика для

отдельных процедур, что может существенно ускорить понимание их логики работы. Unicorn Engine используется более чем в 100 различных фреймворках, утилитах и эмуляторах.

Level Up!

В этой главе мы разобрали, как знание архитектуры микроконтроллера помогает при статическом реверс-инжиниринге прошивки, как прошивка может быть устроена и как она хранится в ПЗУ. Разобрались, что адресное пространство микроконтроллера может быть организовано весьма непросто и что даже определение корректного адреса загрузки прошивки в дизассемблер иногда является не самой простой задачей.

Мы уже успели начать реверс-инжиниринг с поиска базовых сущностей, за которые можно «зацепиться» для начала анализа. Фактически на данном этапе уже ничто не мешает полноценно заниматься исследованием прошивок, вы уже понимаете, где и как искать ответы на возникающие вопросы. Но процесс исследования цифрового устройства можно сделать более приятным и увлекательным – за счет получения дополнительной информации в процессе работы устройства. Именно этому и посвящена следующая глава – настало время проводить динамический анализ.

Level 3

Настраиваем связь с внешним миром (динамический анализ)

К этому моменту мы уже смогли загрузить прошивку в дизассемблер, найти значащие константы и даже определить тип ОС, на базе которой построена прошивка. Возможно, даже получилось что-то проэмулировать. Можно дальше смотреть в дизассемблер, но в большинстве случаев гораздо эффективнее заставить устройство что-то нам рассказать в процессе своей работы.

Динамический анализ

Динамический анализ, т. е. анализ устройства в процессе работы (в динамике), позволяет получить множество информации, недоступной или трудно получаемой с помощью статического анализа. Например, если устройство расшифровывает часть своей прошивки в ОЗУ или получает какие-то данные с сервера производителя. Далеко не все устройства можно эмулировать, поэтому в этой главе мы рассмотрим именно аппаратные подходы к динамическому анализу. Значит, мы будем взаимодействовать с различными компонентами и интерфейсами устройства для получения какой-то информации. И в совокупности с информацией в дизассемблере можем получить ответ на вопрос «как работает устройство» гораздо быстрее.

Собираем стенд для отладки

Как бы не хотелось побыстрее приступить к динамическому анализу, стоит потратить немного времени на организацию стенда. Он должен удовлетворять двум простым правилам: быть удобным и надежным. Ведь если работать со стендом будет неудобно, удовольствия от исследования можно и не получить. Надежность важна еще больше, ведь вы можете потратить время на проведение экспериментов, которые были бесполезны. Например, если вовремя не заметить, что какой-то проводник отвалился или, что еще хуже, замкнул соседнюю линию, можно не только впустую потратить время, но и убить устройство. По-

старайтесь освободить место на столе для стенда и расположить его так, чтобы он не мешал работе, его не надо было двигать/переносить и чтобы имеющиеся на столе металлические предметы (скрепки, отвертки и т. п.) случайно не привели к замыканию. Конечно, во всем нужен баланс и не надо воспринимать все советы как догму, но хотя бы немного задуматься о подготовке своего рабочего места для исследования устройств однозначно стоит.

Оснастки и 3D-печать

3D-принтер может сильно упростить задачу подготовки стенда, ведь на нем можно распечатать множество полезных элементов:

- защитный экран или корпус, закрывающий доступ к не используемым в процессе исследований частям печатной платы устройства. Если устройство в единичном экземпляре и/или очень дорогое, т. е. права на ошибку нет, – эта мера защиты будет точно не лишней. К тому же пользоваться платой в таком виде зачастую удобнее. Альтернативный вариант – резка защитных экранов из оргстекла с помощью лазерного станка. В большинстве городов найти компании, готовые распечатать модель или вырезать нужный шаблон за небольшие деньги, не составляет труда. Например, именно такими экранами защищает некоторые отладочные платы (development board, devboard) ряд производителей;
- крепления для фиксации проводов. Для подключения к выводам компонентов или к контактным площадкам на печатной плате устройства можно использовать специальные пробники (клипсы/крючки). Они бывают нескольких размеров и удобны быстротой подключения.

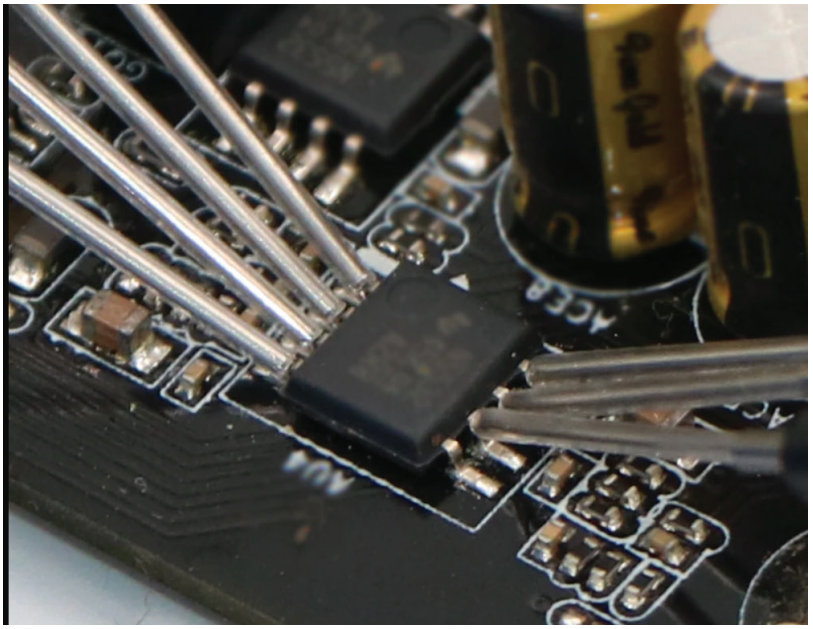


Рис. 3.1. Пробники для подключения к выводам микросхемы[1]

Но у них есть и один большой минус – они держатся очень ненадежно и часто отваливаются при малейшем движении проводника, особенно если прищепка самых малых размеров.

Можно купить готовые клипсы для крепления проводов к печатной плате устройства, однако их не так легко найти и сложно подобрать к конкретной плате. Поэтому можно придумать и реализовать крепление проводников, идущих к прищепке, с помощью 3D-принтера;

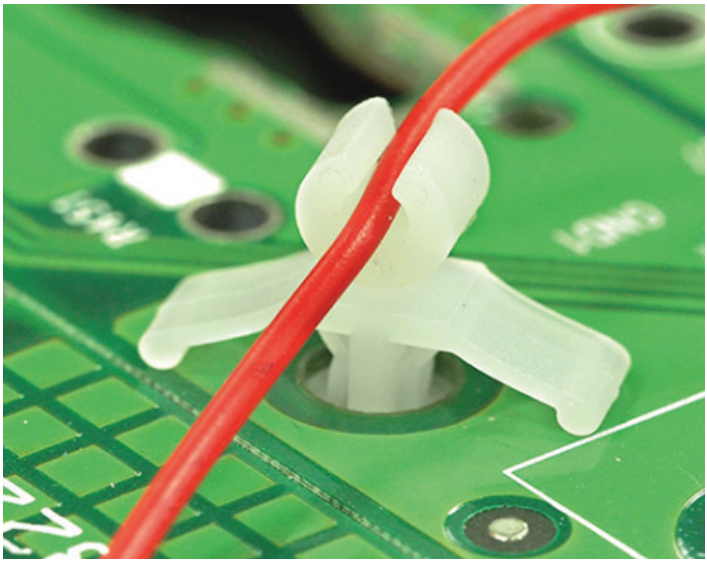


Рис. 3.2. Фиксация отладочного провода с помощью пластикового крепления

- основание оснастки для подключения к контактам без пайки. Например, если по каким-то причинам к контактам на печатной плате устройства нельзя припаиваться (например, если припаянные провода мешают его сборке, а эту операцию надо производить часто). Существуют специальные подпружиненные контакты – пого-пины (pogo-pin). Они бывают разных видов и размеров и позволяют надежно подключаться к контактам на печатной плате без пайки. Альтернативный вариант подключения предполагает использование тонких игл, принимающихся за счет упругой оснастки. Оснастку для использования пого-пинов или игл надо разрабатывать (но можно и взять готовый проект, например с thingiverse.com) или покупать готовую. На рисунке показан проект оснастки с иглами, доступный для скачивания с сайта **thingiverse.com**:

[1] https://aliexpress.ru/item/1005003972033577.html.

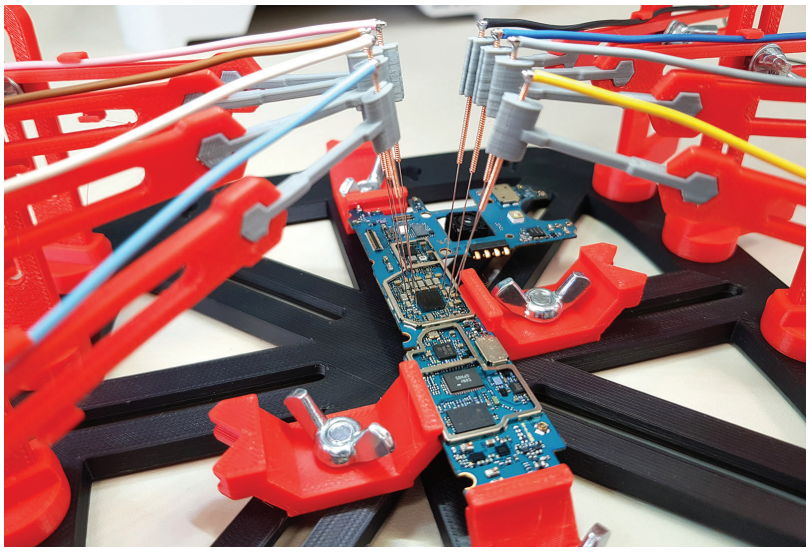


Рис. 3.3. Отладочная оснастка, распечатанная на 3D-принтере[1]

Автоматизация рутинных действий

Наверняка вы знаете шутку про количество итераций для подключения USB-разъема:

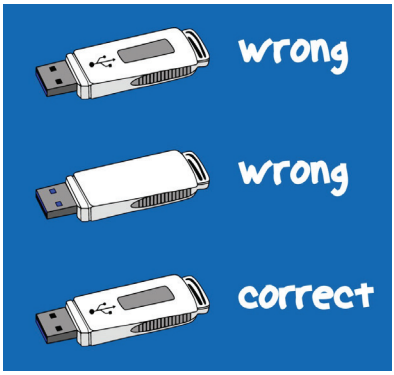


Рис. 3.4. Шутка про разъем USB[2]

И ведь эта шутка появилась не на пустом месте! А еще стандартные разъемы имеют срок службы, как правило, исчисляемый парой тысяч подключений (а дешевые варианты могут начать терять контакт уже после пары сотен). Несколько тысяч подключений разъема могут быть рядовой ситуацией в процессе анализа устройства. То есть мы имеем все шансы рано или поздно получить нестабильный контакт, искажающий наши эксперименты. Наиболее часто приходится отключать питание устройства и/или распространенные интерфейсы (USB, UART).

[1] https://www.thingiverse.com/thing:3615910.

[2] https://www.hackster.io/news/implementing-usb-c-pd-doesn-t-have-to-be-scary-4369eaa4955d.

Для отключения интерфейсов тоже давно придуманы специальные устройства – разрыватели интерфейсов. Они бывают как потребительского уровня, например Yepkit YKUSH XS для отключения одного USB-порта (показан на фото) с ценой пару десятков евро, так и до индустриальных решений ценой в сотни и даже тысячи евро.

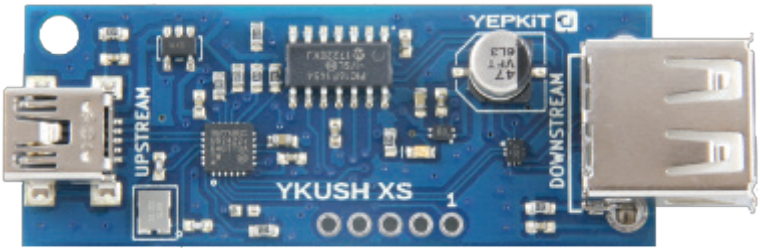


Рис. 3.5. Устройство Yepkit YKUSH XS для отключения USB-порта[1]

Практически каждый интерфейс можно с помощью конвертера подключить к компьютеру через USB, т. е. разрыватель USB позволяет решить проблему управления подключением других интерфейсов (UART, CAN, SPI и т. д.).

Для управления питанием можно использовать управляемые коммутаторы питания. Они управляются по USB или LAN и позволяют с помощью ПО на компьютере отключать/подключать питание и даже делать специальные временные профили с помощью скриптов (хотя можно использовать и обычную «умную» розетку). Также существуют управляемые лабораторные источники питания, которые можно использовать для этой же цели.

Рис. 3.6. Устройство удаленного управления питанием 220 В по сети Ethernet[2]

Еще одно часто требующее автоматизации действие – замыкание контактов или эмуляция ввода пользователем. Для этих действий также существуют разработанные устройства, позволяющие управлять по USB или LAN группой реле или эмулировать USB-клавиатуру и мышь (например, **https://usb2kbd.ru**).

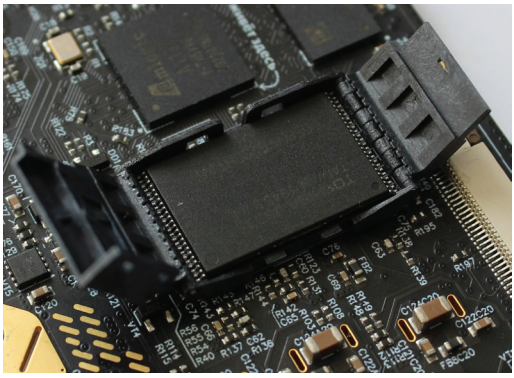
[1] https://www.yepkit.com/product/300115/YKUSHXS.

[2] https://netping.ru/products/netping-2-pwr-220/.

Адаптеры и эмуляция ПЗУ

Представим, что у вас есть микросхема ПЗУ и надо ее постоянно считывать или записывать, например вы пытаетесь изменить прошивку и она почему-то не запускается. Количество экспериментов может исчисляться сотнями и тысячами. После какой-то итерации записи flash-память может деградировать, и появятся битовые ошибки (несмотря на то что производитель обещает несколько десятков тысяч перезаписей). Вы же не будете надеяться, что устройство долго сохранит работоспособность, если каждая итерация будет сопровождаться операциями выпаивания и запаивания микросхемы, правда?

Для решения этой проблемы есть два пути. Первый – использование специальных адаптеров для микросхем ПЗУ. Про адаптеры мы говорили в разделе считывания ПЗУ, но тогда они использовались для подключения различных микросхем к программатору. Существуют адаптеры, позволяющие установить их на печатную плату устройства с помощью подсоединяемого или припаянного разъема. Пример адаптера для микросхемы NAND в корпусе TSOP48 показан на рисунках (первое фото – адаптер открыт и позволяет достать микросхему, второе – закрыт, и микросхема подключена к плате устройства):



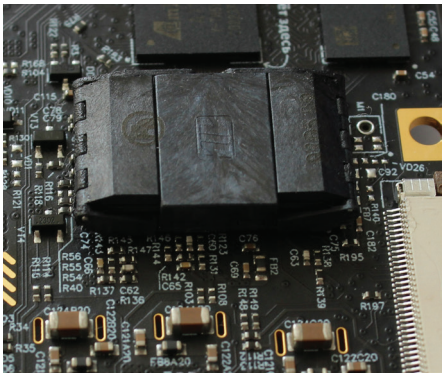


Рис. 3.7. Адаптер для микросхемы NAND в корпусе TSOP48

Конечно, стандартные решения есть только для самых распространенных типов корпусов – SOIC 8/16 и для некоторых видов BGA/TSOP. Возможно, какой-то переходник придется разработать и изготовить самостоятельно, но перед этим хорошо бы поискать в интернете, для большинства самых распространенных типов микросхем существуют готовые переходники. Не стоит забывать, что любые переходники и соединения, особенно большой длины, ухудшают параметры сигнала, и на высоких скоростях работы могут возникнуть ошибки, поэтому учитывайте этот факт при выборе переходника.

Допустим, вы запаяли адаптер на плату устройства, и теперь он позволяет обойтись без пайки микросхемы. Остается последовательность:

1) записали данные в микросхему программатором;
2) вынули микросхему из программатора;
3) установили микросхему в адаптер на плате устройства;
4) устройство выполнило нужные действия;

5) вынули микросхему из адаптера на плате устройства;
6) установили микросхему в программатор.

Получается как минимум 6 шагов, на четыре из которых есть шанс что-то сломать (физически), согнуть и отломать ножку адаптера или микросхемы, лишний раз тронуть стенд и что-то повредить. Есть вариант упростить процесс, используя специальные устройства – эмуляторы ПЗУ. Во многих цифровых устройствах используется flash-память с интерфейсом SPI, для эмуляции подобных микросхем можно использовать DediProg EM100Pro-G2.

Рис. 3.8. Эмулятор SPI DediProg EM100Pro-G2[1]

Он позволяет эмулировать множество микросхем с интерфейсом SPI. Процесс выглядит следующим образом.

1. На печатную плату устройства запаивается разъем или сам шлейф для подключения к эмулятору, например как на фото:

Рис. 3.9. Эмулятор SPI DediProg EM100Pro-G2, подключенный к материнской плате компьютера

[1] https://www.dediprog.com/product/EM100Pro-G2.

2. Эмулятор подключается к компьютеру с ПО по USB и к устройству через шлейф.
3. В ПО выбираются эмулируемый образ, эмулируемая микросхема памяти и запускается процесс эмуляции.

Как видите, операция подключения эмулятора разовая, шанс повреждения устройства из-за многократных манипуляций становится минимальным.

У эмулятора EM100Pro-G2 есть еще одна очень полезная функция для получения дополнительной информации о работе устройства – трассировщик SPI-шины. ПО эмулятора позволяет просмотреть лог всех обращений к содержимому эмулируемой микросхемы памяти, к тому же привязанный ко времени. Такая информация может быть весьма полезна при динамическом анализе устройства, и для ее получения не придется параллельно подключать к шине SPI логический анализатор.

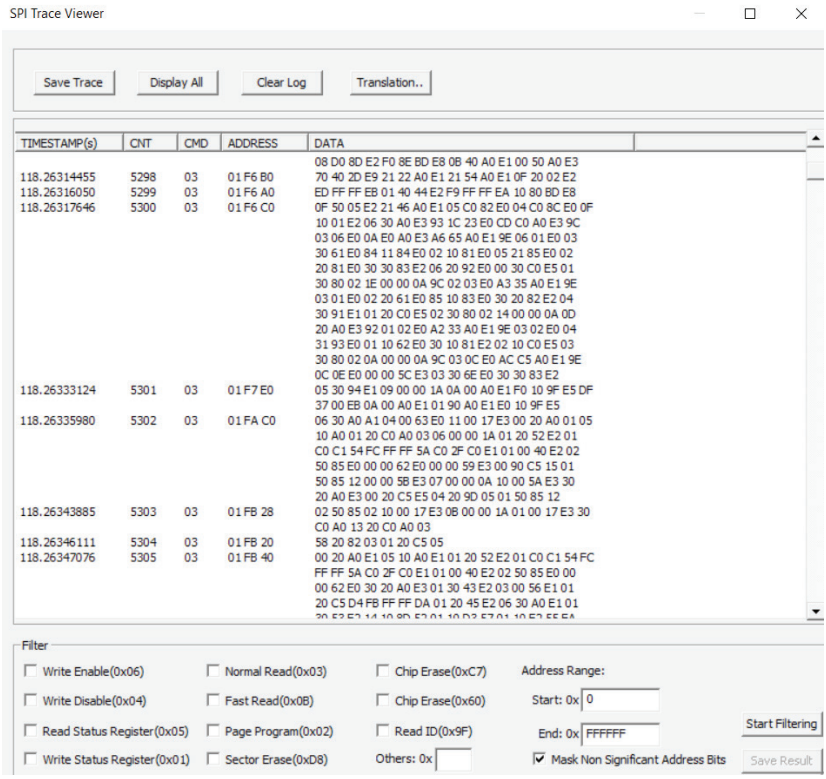


Рис. 3.10. Лог обращений к эмулируемой микросхеме SPI в ПО управления DediProg EM100Pro-G2

Используем отладочные интерфейсы

Кратко мы рассматривали JTAG/SWD в разделах про отладочные интерфейсы и их механизмы защиты. Теперь разберем, какие функции отладочных интерфейсов (в общем случае) могут быть полезны при динамическом анализе

устройства. Доступность функций зависит от режима работы, механизма защиты интерфейса, модели микроконтроллера и даже функций, реализованных в прошивке устройства.

В первую очередь это, конечно же, возможность управлять потоком выполнения прошивки. Отладочные интерфейсы позволяют читать и писать значения управляющих регистров микроконтроллера, в том числе регистры процессорного ядра. То есть можно изменять регистры указателя текущей инструкции прямо на работающем устройстве. Можно ставить аппаратные точки останова процессорного ядра, считывать или записывать значение в ОЗУ или flash-памяти микроконтроллера.

Во-вторых, через отладочные интерфейсы можно делать трассировку потока выполнения программы. Прошивки, как и обычные прикладные программы, даже в готовых устройствах могут быть скомпилированы в режиме debug, ведь embedded-разработчики тоже бывают забывчивыми.

Для взаимодействия по отладочным интерфейсам можно использовать OpenOCD или проприетарные утилиты производителя отладчика. Чтобы лучше понять возможности отладочных интерфейсов, рекомендую купить отладочную плату на распространенном микроконтроллере (например, на каком-нибудь из модульного ряда STM32) и попрактиковаться с работой с отладочными интерфейсами.

В главе 1 в разделе «Считывание прошивки с помощью Segger J-Link и JTAG» мы получали прошивку высокоскоростного GPS-логгера (плата показана на фото) из внутренней памяти микроконтроллера Atmel SAM3S4B, построенного на базе ядра ARM Cortex-M3.

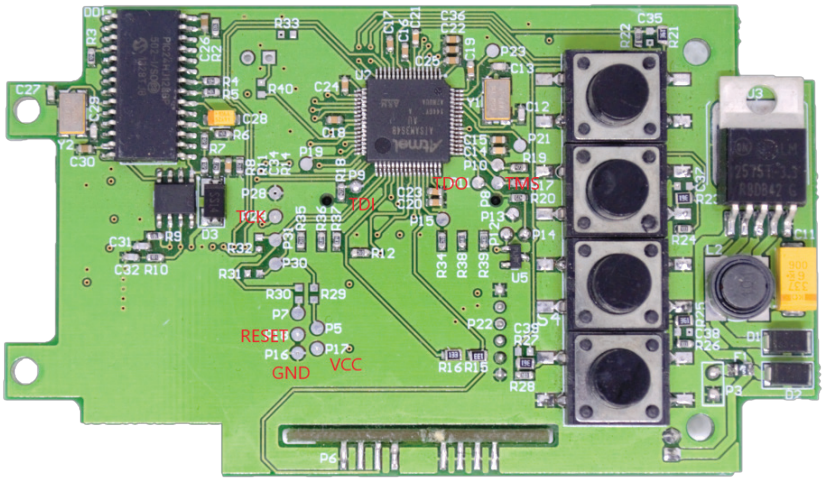


Рис. 3.11. Внешний вид платы устройства
с отмеченными контактами интерфейса JTAG

Для иллюстрации возможностей отладочных интерфейсов подключимся к нему по интерфейсу JTAG с помощью отладчика J-Link и ПО OpenOCD:

```
openocd.exe -f interface/jlink.cfg -f target/at91sam3sxx.cfg
```

Получаем уже знакомый нам вывод об успешном подключении:

```
Open On-Chip Debugger 0.12.0-rc2 (2022-10-26-14:02)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "jtag". To override use
'transport select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : J-Link V9 compiled Sep 1 2016 18:29:50
Info : Hardware version: 9.60
Info : VTarget = 3.250 V
Info : clock speed 500 kHz
Info : JTAG tap: sam3.cpu tap/device found: 0x4ba00477 (mfg: 0x23b (ARM Ltd),
part: 0xba00, ver: 0x4)
Info : [sam3.cpu] Cortex-M3 r2p0 processor detected
Info : [sam3.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for sam3.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

Снова посмотрим в документацию и найдем описание адресного пространства микроконтроллера:

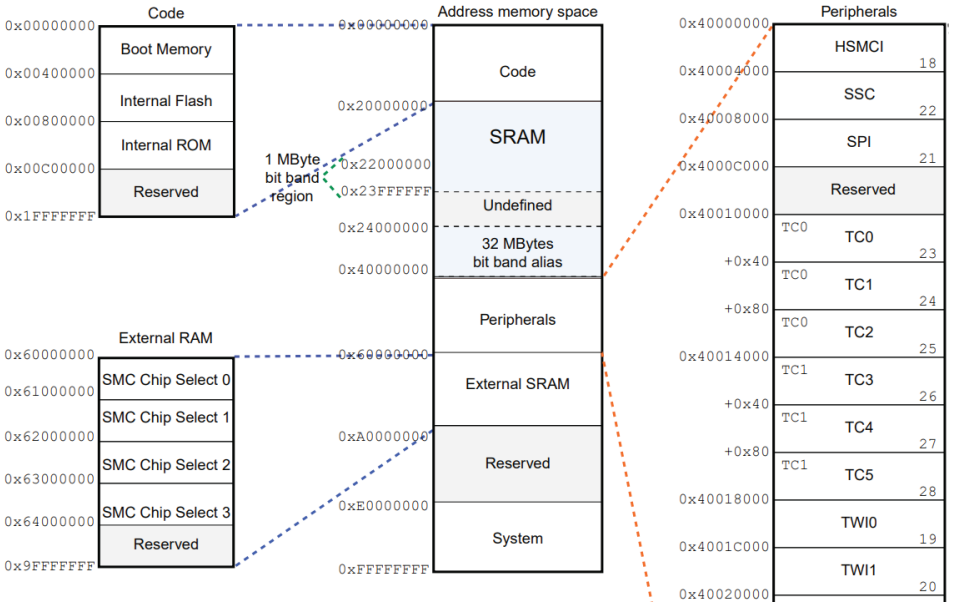


Рис. 3.12. Структура адресного пространства микроконтроллеров семейства SAM3S[1]

[1] https://www.keil.com/dd/docs/datashts/atmel/sam3s/sam3s_datasheet.pdf.

Прошивка и содержимое flash-памяти у нас уже есть. А вот содержимого RAM – нет. Из документации мы помним, что объем ОЗУ у микроконтроллера – 48 Кб (т. е. 0xC000 байт):

1.1 Configuration Summary

The SAM3S microcontrollers differ in memory size, package and features list. Table 1-1 below summarizes the configurations of the device family

Table 1-1. Configuration Summary

Device	Flash	SRAM	Timer Counter Channels	GPIOs	UART/ USARTs	ADC	12-bit DAC Output	External Bus Interface	HSMCI	Package
SAM3S4C	256 Kbytes single plane	48 Kbytes	6	79	2/2(1)	15 ch.	2	8-bit data, 4 chip selects, 24-bit address	1 port 4 bits	LQFP100 BGA100
SAM3S4B	256 Kbytes single plane	48 Kbytes	3	47	2/2(1)	10 ch.	2	-	1 port 4 bits	LQFP64 QFN 64
SAM3S4A	256 Kbytes single plane	48 Kbytes	3	34	2/1	8 ch.	-	-	-	LQFP48 QFN 48

Рис. 3.13. Характеристики микроконтроллеров семейства SAM3S[1]

Следовательно, RAM занимает в адресном пространстве микроконтроллера диапазон 0x20000000–0x2000C000, попробуем считать ее начало уже известной нам командой mdw, предварительно подключившись с помощью telnet к командному порту 4444 OpenOCD:

```
> mdw 0x20000000 0x40
0x20000000: 8f5ff3bf d1fb3801 bf004770 00000000 00000000 00000019 00000020 00040000
0x20000020: 00000000 00000000 00000000 00000000 00000000 00000000 80000000 00000001
0x20000040: f0000000 0000000f 38000000 0000001c 1c000000 00000038 0c000000 00000031
0x20000060: 06000000 00000061 06000000 00000061 06000000 00000061 0603f800 001fc060
0x20000080: 06073c00 0038e060 0c0c0e00 00603030 1c184300 00c21838 38185300 00ca181c
0x200000a0: f0306300 0086080f 80304100 00820c01 00300100 00800c00 00100300 00c01800
0x200000c0: 00180300 00c01800 001c0600 00603800 000e0c00 00707000 0007f800 003fe000
0x200000e0: 0000e000 00070000 00000000 00000000 00000000 00000000 00000000 00000000
```

Видим значения, похожие на содержимое RAM, значит, команда выполнена успешно и можно считать всю память в файл:

```
> dump_image ram.bin 0x20000000 0xC000
dumped 49152 bytes in 1.629386s (29.459 KiB/s)
```

Однако есть одна особенность, ведь мы считываем RAM, т. е. изменяемую память. А микроконтроллер в этот же момент исполняет прошивку, из-за чего содержимое памяти может меняться. Попробуем считать RAM еще раз и сравним образы с помощью hex-редактора:

[1] https://www.keil.com/dd/docs/datashts/atmel/sam3s/sam3s_datasheet.pdf.

```
> dump_image ram2.bin 0x20000000 0xC000
dumped 49152 bytes in 1.627029s (29.502 KiB/s)
```

Естественно, содержимое RAM-памяти изменилось, нашлось более 50 различающихся областей размером от 1 до 0x54 байт.

```
Result      Address A   Size
Difference  9DF8h       7h
Difference  9DF0h       6h
Difference  9DE8h       6h
Difference  9DB5h       2h
....
Difference  54D0h       54h
Difference  2987h       9h
Difference  248Ch       1h
Difference  23B8h       3h
Difference  23A8h       2h
Difference  2374h       2h
Difference  236Eh       1h
Difference  207Ch       3h
```

Выполнение программы процессором можно остановить с помощью команды `halt`.

Для начала посмотрим состояние процессора:

```
> targets
   TargetName         Type       Endian   TapName            State
-- ------------------ ---------- -------- ------------------ ------------
   0* sam3.cpu        cortex_m   little   sam3.cpu           running
```

После чего остановим процессор:

```
> halt
[sam3.cpu] halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x0040c88e msp: 0x20009818
```

В выводе результата работы команды `halt` мы видим значение трех хорошо знакомых вам регистров:

- `xPSR` – содержит флаги результатов выполнения арифметических и логических операций, состояние выполнения программы и номер обрабатываемого в данный момент прерывания;
- `pc` (Program Counter) – хранит адрес текущей инструкции;
- `msp` (Main Stack Pointer) – хранит указатель на основной стек.

Давайте еще раз посмотрим на состояние процессора:

```
> targets
   TargetName         Type       Endian  TapName            State
-- ------------------ ---------- ------- ------------------ ------------
   0* sam3.cpu        cortex_m   little  sam3.cpu            halted
```

Теперь, сколько бы мы раз ни считывали содержимое RAM, оно будет оставаться неизменным, т. к. процессор остановил исполнение программы. Давайте посмотрим на содержимое всех регистров процессора с помощью команды `reg` без аргументов:

```
> reg
===== arm v7m registers
(0) r0 (/32): 0x00000001
(1) r1 (/32): 0x0000000e
(2) r2 (/32): 0x20005d08
(3) r3 (/32): 0x00000000
(4) r4 (/32): 0x00000000
(5) r5 (/32): 0x00000001
(6) r6 (/32): 0x00000001
(7) r7 (/32): 0x20005d08
(8) r8 (/32): 0x200054d0
(9) r9 (/32): 0x200023d9
(10) r10 (/32): 0x03d09000
(11) r11 (/32): 0x20005ce8
(12) r12 (/32): 0x00000000
(13) sp (/32): 0x20009818
(14) lr (/32): 0x0041b695
(15) pc (/32): 0x0040c88e
(16) xPSR (/32): 0x61000000
(17) msp (/32): 0x20009818
(18) psp (/32): 0x00000000
(20) primask (/1): 0x00
(21) basepri (/8): 0x00
(22) faultmask (/1): 0x00
(23) control (/3): 0x00
===== Cortex-M DWT registers
```

Видим значения основных регистров, в том числе уже известных нам `pc` и `msp`, а в регистре lr хранится адрес возврата из функции. Зная, что в нашем микроконтроллере исполнение идет из flash-памяти, и имея ее дамп, можем легко понять, в какой части прошивки мы находимся. Посмотрим в дизассемблере, какому участку кода он соответствует, для этого перейдем на адрес 0x40C88E:

```
ROM:0040C88C sub_40C88C                                 ; CODE XREF: sub_412AF8+5A↓p
ROM:0040C88C                                            ; sub_413730:loc_41375A↓p ...
ROM:0040C88C                PUSH        {R4-R6,LR}
ROM:0040C88E                LDR         R5, =0x200023D8
ROM:0040C890                LDRB        R3, [R5]
ROM:0040C892                CMP         R3, #0
ROM:0040C894                BEQ         locret_40C8E6
ROM:0040C896                LDR         R4, =0x20005CD0
ROM:0040C898                LDR         R6, =0x20006C28
ROM:0040C89A                MOVS        R0, #1
ROM:0040C89C                BL          sub_40C594
ROM:0040C8A0                LDRD.W      R2, R3, [R6]
ROM:0040C8A4                LDRD.W      R0, R1, [R4]
ROM:0040C8A8                BL          sub_41D5F8
ROM:0040C8AC                LDRD.W      R2, R3, [R6,#8]
ROM:0040C8B0                STRD.W      R0, R1, [R4]
ROM:0040C8B4                LDRD.W      R0, R1, [R4,#8]
ROM:0040C8B8                BL          sub_41D5F8
ROM:0040C8BC                LDRD.W      R2, R3, [R6,#0x10]
ROM:0040C8C0                STRD.W      R0, R1, [R4,#8]
ROM:0040C8C4                LDRD.W      R0, R1, [R4,#0x10]
ROM:0040C8C8                BL          sub_41D5F8
ROM:0040C8CC                LDR         R6, =0x20002368
ROM:0040C8CE                STRD.W      R0, R1, [R4,#0x10]
ROM:0040C8D2                LDRD.W      R0, R1, [R6]
ROM:0040C8D6                LDR         R3, =0x3FF00000
ROM:0040C8D8                MOVS        R2, #0
ROM:0040C8DA                BL          sub_41D5F8
ROM:0040C8DE                MOVS        R3, #0
ROM:0040C8E0                STRD.W      R0, R1, [R6]
ROM:0040C8E4                STRB        R3, [R5]
ROM:0040C8E6
ROM:0040C8E6 locret_40C8E6                              ; CODE XREF: sub_40C88C+8↑j
ROM:0040C8E6                POP         {R4-R6,PC}
ROM:0040C8E6 ; End of function sub_40C88C
```

Рис. 3.14. Фрагмент прошивки

Видим, что в данный момент должна исполняться вторая от начала функции sub_40C88C инструкция LDR R5, =0x200023D8. Заодно давайте посмотрим, есть ли вызов нашей подпрограммы по адресу 0x41b695 из регистра lr:

```
ROM:0041B68C loc_41B68C                                 ; CODE XREF: sub_41AEE8+8C4↓j
ROM:0041B68C                                            ; sub_41AEE8+8D2↓j ...
ROM:0041B68C                BL          sub_40C88C
ROM:0041B690                BL          sub_40C88C
ROM:0041B694                BL          sub_4117BC
ROM:0041B698                BL          sub_40C88C
ROM:0041B69C                BL          sub_41121C
ROM:0041B6A0                B           loc_41B556
```

Рис. 3.15. Фрагмент прошивки

Все правильно, т. к. lr содержит адрес, куда будет возвращено управление, то вызов функции sub_40C88C находится по адресу 0x41b690. Вы заметили, что адрес в регистре lr нечетный? Надеюсь, вы помните, что переключение из 32-битного набора команд в 16-битный режим Thumb осуществляется как раз с помощью передачи управления на нечетный адрес. Вот его компилятор и вставил.

Естественно, мы можем не только читать содержимое RAM и регистров, но и писать в них свои значения. Вернемся к нашей первоначальной подпрограмме. Допустим, мы хотим передать управление на код в прошивке, расположенный по адресу 0x40C896, т. е. пропустить проверку значения в RAM по адресу 0x200023D8 на 0:

```
ROM:0040C88C sub_40C88C                              ; CODE XREF: sub_412AF8+5A↓p
ROM:0040C88C                                         ; sub_413730:loc_41375A↓p ...
ROM:0040C88C                 PUSH            {R4-R6,LR}
ROM:0040C88E                 LDR             R5, =0x200023D8
ROM:0040C890                 LDRB            R3, [R5]
ROM:0040C892                 CMP             R3, #0
ROM:0040C894                 BEQ             locret_40C8E6
ROM:0040C896                 LDR             R4, =0x20005CD0
ROM:0040C898                 LDR             R6, =0x20006C28
ROM:0040C89A                 MOVS            R0, #1
ROM:0040C89C                 BL              sub_40C594
ROM:0040C8A0                 LDRD.W          R2, R3, [R6]
ROM:0040C8A4                 LDRD.W          R0, R1, [R4]
ROM:0040C8A8                 BL              sub_41D5F8
ROM:0040C8AC                 LDRD.W          R2, R3, [R6,#8]
ROM:0040C8B0                 STRD.W          R0, R1, [R4]
ROM:0040C8B4                 LDRD.W          R0, R1, [R4,#8]
ROM:0040C8B8                 BL              sub_41D5F8
ROM:0040C8BC                 LDRD.W          R2, R3, [R6,#0x10]
ROM:0040C8C0                 STRD.W          R0, R1, [R4,#8]
ROM:0040C8C4                 LDRD.W          R0, R1, [R4,#0x10]
ROM:0040C8C8                 BL              sub_41D5F8
ROM:0040C8CC                 LDR             R6, =0x20002368
ROM:0040C8CE                 STRD.W          R0, R1, [R4,#0x10]
ROM:0040C8D2                 LDRD.W          R0, R1, [R6]
ROM:0040C8D6                 LDR             R3, =0x3FF00000
ROM:0040C8D8                 MOVS            R2, #0
ROM:0040C8DA                 BL              sub_41D5F8
ROM:0040C8DE                 MOVS            R3, #0
ROM:0040C8E0                 STRD.W          R0, R1, [R6]
ROM:0040C8E4                 STRB            R3, [R5]
ROM:0040C8E6
ROM:0040C8E6 locret_40C8E6                           ; CODE XREF: sub_40C88C+8↑j
ROM:0040C8E6                 POP             {R4-R6,PC}
ROM:0040C8E6 ; End of function sub_40C88C
```

Рис. 3.16. Фрагмент прошивки

Мы можем сделать это несколькими способами, например:

1) записав значение в RAM по адресу 0x200023D8, отличное от 0. Для начала считаем значение байта:

```
> mdb 0x200023D8 1
0x200023d8: 00
```

Теперь запишем новое:

```
> mwb 0x200023D8 0xFF
```

И отменим действие команды `halt` с помощью команды `resume`, разрешив процессору дальше исполнять программу:

```
> resume
```

2) записав нужное нам значение в регистр PC, мы заставим процессор сразу перейти на нужную нам подпрограмму в прошивке:

```
> reg pc 0x40C896
pc (/32): 0x00040C896
> resume
```

Кстати, вместо этого можно просто указать требуемый адрес в качестве аргумента `resume`:

```
> resume 0x40C896
```

3) выполнив всего одну инструкцию с помощью команды `step` (имеющей опциональный аргумент адреса, с которого нужно выполнить инструкцию, иначе выполнение произойдет с текущего). Мы хотим выполнить одну инструкцию LDR R4, =0x20005CD0 из интересующей нас подпрограммы по адресу 0x40C896, загружающую значение в регистр R4:

```
> step 0x40C896
[sam3.cpu] halted due to single-step, current mode: Thread
xPSR: 0x61000000 pc: 0x0040c898 msp: 0x20009828
```

Видим, что значение регистра `pc` изменилось на адрес следующей инструкции. Посмотрим содержимое регистров:

```
 > reg
===== arm v7m registers
(0) r0 (/32): 0x00000001
(1) r1 (/32): 0x0000000e
(2) r2 (/32): 0x20005d08
(3) r3 (/32): 0x00000000
(4) r4 (/32): 0x20005cd0
(5) r5 (/32): 0x00000001
(6) r6 (/32): 0x00000001
(7) r7 (/32): 0x20005d08
(8) r8 (/32): 0x200054d0
(9) r9 (/32): 0x200023d9
(10) r10 (/32): 0x03d09000
(11) r11 (/32): 0x20005ce8
(12) r12 (/32): 0x00000000
(13) sp (/32): 0x20009818
(14) lr (/32): 0x0041b695
(15) pc (/32): 0x0040c898
(16) xPSR (/32): 0x61000000
(17) msp (/32): 0x20009818
(18) psp (/32): 0x00000000
(20) primask (/1): 0x00
```

```
(21) basepri (/8): 0x00
(22) faultmask (/1): 0x00
(23) control (/3): 0x00
===== Cortex-M DWT registers
```

Содержимое регистра R4 соответствует загруженному (0x20005CD0). Запомним значение регистра R6 (0x00000001), ведь после выполнения следующей команды `step` (уже без адреса) оно должно поменяться на 0x20006C28:

```
> step
[sam3.cpu] halted due to single-step, current mode: Thread
xPSR: 0x61000000 pc: 0x0040c89a msp: 0x20009828
```

Команда `reg` позволяет считать один регистр:

```
> reg r6
r6 (/32): 0x20006c28
```

Видим, что значение в регистре R6 изменилось в соответствии с выполненной процессором инструкцией.

Включенный отладочный интерфейс позволяет исследователю получить почти любую информацию от работающего устройства, поэтому необходимо хорошо представлять возможности ПО отладчика и ориентироваться в основных особенностях архитектуры процессора анализируемого устройства.

UART и трассировка

Последовательный интерфейс UART встречается на многих встраиваемых системах, и довольно часто в него выводится лог загрузки или выполнения прошивки (трасса). В большинстве случаев в UART выводятся человекочитаемые данные, легко интерпретируемые через обычный терминал (см. рис. 3.17).

Определить параметры UART-интерфейса в устройстве можно с помощью осциллографа или логического анализатора, а при их отсутствии – перебором стандартных значений скоростей, количества стоп-битов и четности.

Для трассировки могут использоваться бинарные данные, т. к. нужно минимизировать время передачи данных. То есть в таком режиме по UART приходят кодовые значения состояний (например, размером несколько байтов), для интерпретации которых уже потребуется написать специальную программу или скрипт. Некоторые устройства могут реализовывать по UART Command Line Interface (CLI), в том числе даже полноценную системную консоль ОС. В таком случае вполне возможен двусторонний обмен данным при наличии у устройства активного сигнала RX. В большинстве подобных случаев в прошивке устройства организуется дополнительная авторизация для доступа к консоли. Например, устройство просит ввести пароль доступа или принимает на вход сертификат.

```
U-Boot 2010.06 (May 17 2014 - 15:03:14)

Check spi flash controller v350... Found
Spi(cs1) ID: 0xC2 0x20 0x18 0xC2 0x20 0x18
Spi(cs1): Block:64KB Chip:16MB Name:"MX25L128XX"
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
16384 KiB hi_sfc at 0:0 is now current device

## Booting kernel from Legacy Image at 82000000 ...
   Image Name:   Linux-3.0.8
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2134796 Bytes = 2 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Loading Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
```

Рис. 3.17. Фрагмент лога загрузки IP-камеры[1]

Используем логический анализатор и осциллограф

Мы уже не раз говорили про логический анализатор, особенно в главе про способы получения прошивки устройства. Существует бесчисленное множество применений логического анализатора для исследования цифровых устройств и динамического анализа в частности. Приведу лишь некоторые из них:

1) уже знакомый нам способ снифинга прошивки при чтении из SPI/I2C/eMMC микросхем памяти;
2) сниффинг обмена данными по интерфейсам (UART, SPI и т. п.), в том числе для получения паролей (например, для SD- или CF-карт памяти);
3) запись изменения состояния выводов микроконтроллера;
4) определение сигнальных линий на интерфейсе за счет декодирования протокола (например, записали все линии протокола NAND, определили соответствие линий данных D0–D7 за счет перебора возможных комбинаций в декодере протокола);
5) измерение времени между различными состояниями устройства (за счет установки логических фильтров на условие начала захвата сигналов).

Осциллограф – не менее важный инструмент в арсенале исследователя цифровых устройств, ведь он позволяет анализировать аналоговую форму сигнала. Для ряда задач он может заменить логический анализатор, но некоторые задачи решаются только с помощью осциллографа.

[1] https://www.jedge.com/code/ESCAM_QF100_IP_camera_boot_log.txt.

1. Анализ формы сигнала и определение искажений сигнала, например «заваливания» фронтов, как показано на схеме далее:

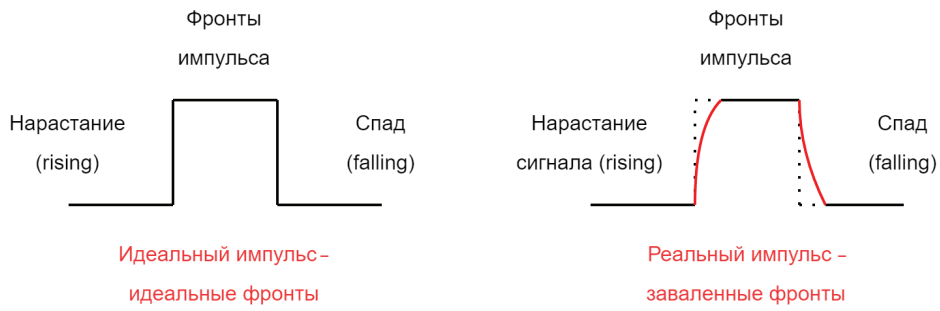


Рис. 3.18. Искажение сигнала

   Эта информация может быть весьма полезной при выявлении причин нестабильной работы каких-либо интерфейсов, например при подключении снифферов UART или SPI.

2. Анализ различных сигналов при SC-атаках.

Очень редко получается провести нормальное исследование устройства без помощи логического анализатора или осциллографа. А иногда и сам осциллограф становится объектом исследования (**http://blog.weinigel.se/2016/05/01/sds7102-hacking.html**).

Хардкор – нестандартные подходы к получению информации

До этого момента мы не рассматривали вариант внесения изменений в прошивку для получения какой-то дополнительной информации. Проблемам, которые могут возникнуть в процессе модификации прошивки, посвящена глава «Level 4. Механизмы защиты встраиваемых систем». Тем не менее, опустив сложности, с которыми можно столкнуться в процессе такой модификации, поговорим о пользе. Очевидно, что, модифицировав прошивку, можно активировать отключенный отладочный интерфейс или UART. Но что делать, если JTAG/SWD отключен полностью без возможности восстановления, а UART не используется для вывода отладочной информации? Искать другие способы заставить устройство что-то рассказать!

Мигаем светодиодом

В большинстве цифровых устройств есть статусные светодиоды или любые другие индикаторы (включая дисплеи различных типов). Рассмотрим светодиод как самый простой пример. Если светодиод управляется микроконтроллером (как на примере схемы далее), значит, модифицировав прошивку, мы можем управлять состоянием светодиода.

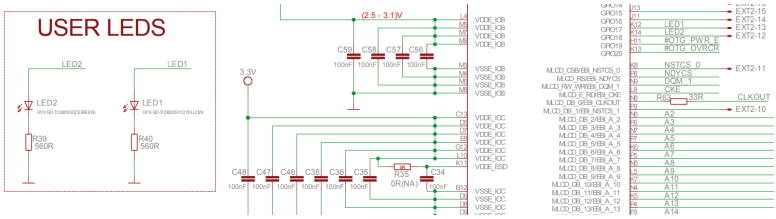


Рис. 3.19. Схема подключения светодиодов[1]

Данный подход справедлив не только для светодиода, но и для любой другой периферии микроконтроллера. Но светодиод визуально понятнее и, скорее всего, не несет функциональной роли с точки зрения работы устройства (например, если вместо светодиода мы будем управлять сигналом управления каким-то реле на плате устройства, это может сильно повлиять на функции исследуемого устройства). Можно смотреть за изменением его состояния с помощью логического анализатора.

Если мы можем управлять светодиодом из модифицированной прошивки – значит, мы можем и написать свою реализацию трассировщика! А чтобы максимально упростить задачу приема подобной трассы, можно использовать программную реализацию UART на GPIO-пине микроконтроллера, управляющего светодиодом. Звучит сложно, но во многих случаях, когда стандартные отладочные интерфейсы недоступны, светодиод (или вывод GPIO) – это мощное оружие в руках исследователя устройств.

Отладка через шину SPI

Несколькими страницами ранее мы рассматривали эмулятор ПЗУ с интерфейсом SPI – DediProg EM100Pro-G2. Помимо эмуляции ПЗУ и постройки трассы обращений к микросхеме памяти, эмулятор позволяет организовать собственный отладочный протокол поверх шины SPI. Это может быть полезным в двух случаях:

1) трассировка выполнения программы. Из прошивки через SPI-контроллер отправляются специальные команды, отличные от набора команд, применяющихся для взаимодействия с SPI-памятью. Эмулятор принимает такие команды и интерпретирует их как сообщения трассировки. На рисунке показано окно ПО DediProg с логом трассировки по SPI;

Рис. 3.20. Лог трассировки через SPI с помощью эмулятора DediProg EM100Pro-G2[2]

[1] https://www.olimex.com/Products/ARM/NXP/LPC-H3131/resources/LPC-H3131-schematic.pdf.

[2] https://www.dediprog.com/download/save/79.pdf.

2) полноценная реализация отладочного интерфейса. В таком режиме обмен специальными командами идет в обе стороны, т. е. в прошивке потребуется сделать обработчик отладочных команд. В этом режиме уже возможна полноценная отладка с программными точками остановки, модификацией значений в памяти и регистрах микроконтроллера.

С готовым устройством типа эмулятора DediProg EM100Pro-G2 сделать канал взаимодействия с исследуемым устройством через нестандартный интерфейс очень просто, но ничего не мешает реализовать подход с помощью собственноручно разработанного устройства, в том числе для интерфейсов, отличных от SPI. Подобный подход применим и для других периферийных интерфейсов – I2C, UART или того же SPI, но использующихся для связи различных компонентов внутри устройства.

Отладка задержками и зависанием

При анализе встраиваемых систем может возникнуть ситуация, когда нет никаких отладочных интерфейсов и даже светодиода, например при эксплуатации уязвимости. Поэтому те подходы к получению отладочной информации, что мы рассматривали ранее, будут неприменимы. Если модификация прошивки возможна и устройство меняет свое состояние в процессе работы, часть информации о потоке выполнения программы можно получить за счет внесения дополнительных задержек в различные ветви исполнения прошивки. На схеме показан пример добавления измеримой задержки в одну ветвь исполнения и вечный цикл, т. е. зависание подпрограммы, в другую.

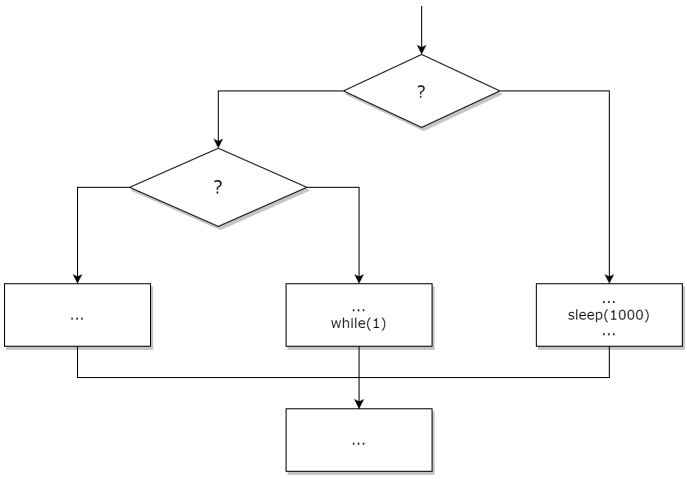


Рис. 3.21. Алгоритм отладки с помощью задержек

Естественно, использовать задержку можно не только для определения ветви выполнения (т. е. трассировки), но и для получения какой-то информации о внутреннем состоянии микроконтроллера. Например, зная ветку исполнения, можно добавить в нее условие с проверкой значения регистра и зависанием в случае успешного совпадения предполагаемого значения. Если существует

возможность с измеримой точностью замерить время зависания, можно даже пытаться считывать значение регистра за счет установки величины задержки, кратной значению в регистре.

Анализ содержимого ОЗУ

Существует несколько векторов атак на ОЗУ в микроконтроллерах. Первый вектор основывается на том, что ОЗУ может не очищаться при перезагрузке микроконтроллера. Следовательно, заменив прошивку на свою без отключения питания микроконтроллера, можно считать оставшиеся в ОЗУ данные.

Второй вектор является одним из самых сложных в реализации способов получения информации для анализа, зачастую требуя изготовления специальной технологической оснастки – интерпозера памяти и высокоскоростных логических анализаторов. Метод заключается в применении платы-переходника для осуществления атаки MITM на ОЗУ (только для чтения данных). В отличие от первого вектора, данный метод работает только для внешней ОЗУ, к которой можно получить физический доступ. Существуют готовые переходники для распространенных типов ОЗУ (ищутся по словам «RAM Interposer»), например фирмы Nexus.

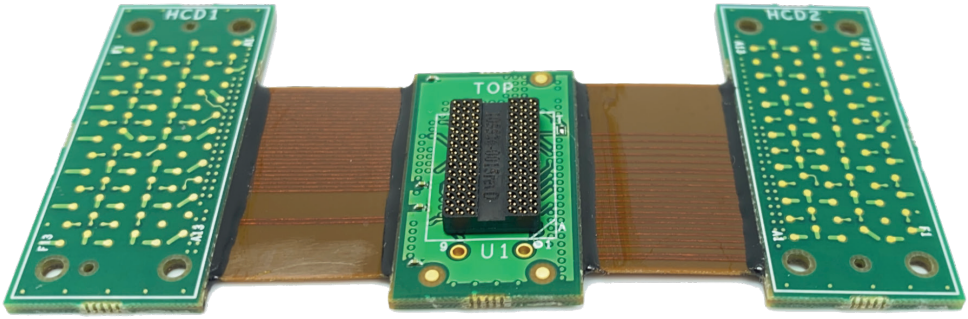


Рис. 3.22. Интерпозер для микросхемы RAM DDR4[1]

На картинке показан интерпозер для микросхемы DDR4-памяти. Центральная часть этого интерпозера (сокет) устанавливается на печатную плату исследуемого устройства вместо микросхемы ОЗУ. Микросхема ОЗУ устанавливается на интерпозер. Ответвление всех сигналов уходит на части платы слева и справа. Естественно, использование подобных интерпозеров негативно сказывается на качестве сигналов и может привести к неработоспособности устройства.

Пример установленного интерпозера RAM на устройство показан на следующей картинке. Тип и внешний вид интерпозера отличаются от рассмотренного выше, но общая концепция использования остается такой же.

[1] https://www.nexustechnology.com/products/memory-interposers/ddr4-memory-interposers/ddr4-96-ball-logiccompliance-interposer/.

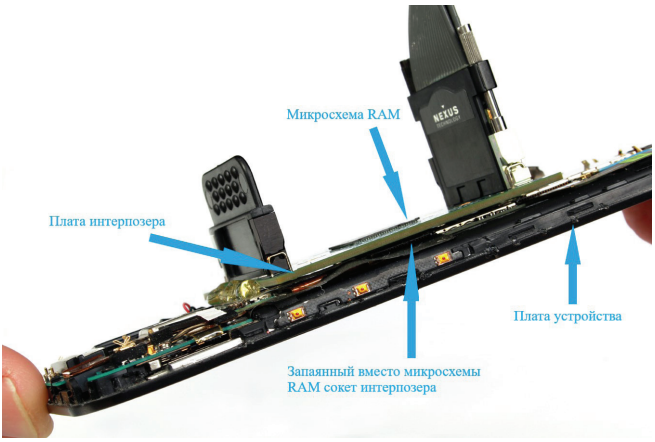


Рис. 3.23. Пример установки интерпозера RAM[1]

Перехват и анализ сигналов DDR-памяти являются сложной задачей, однако иногда это единственный вариант получить дополнительную информацию для анализа устройства. Не во всех цифровых устройствах используется высокоскоростная DDR-память последних поколений (DDR3+). Вполне возможно использование как SDRAM-памяти, так и DDR первых поколений. Такая память отличается гораздо более низкими частотами работы и передачи данных и даже может быть выполнена в виде микросхем с планарными выводами, к которым легко подключить щупы логического анализатора. Соответственно, требования к логическому анализатору также существенно понижаются, что дает возможность проведения подобного класса исследований даже в хорошо оснащенных лабораториях домашнего уровня. Для предотвращения подобных атак производители могут применять шифрование памяти.

Изготовление интерпозера – не единственный способ получения содержимого ОЗУ. Для встраиваемых систем, использующих шину PCI Express, могут быть применимы техники атак на основе прямого доступа к памяти (DMA, Direct Memory Access). В большинстве случаев доступ к ОЗУ со стороны периферийных устройств предоставляется через механизмы ОС. Однако такой подход несет в себе высокие накладные расходы на дополнительные проверки и организацию передачи данных через цепочку драйверов и т. п. Для некоторых высокоскоростных устройств (HDD, SSD и т. п.) был реализован прямой доступ к ОЗУ, позволяющий убрать лишние операции при копировании больших объемов данных. Побочным эффектом стало появление нового класса атак – основанных на чтении и записи содержимого ОЗУ с помощью специальных устройств, способных генерировать запросы на прямой доступ к памяти. Впервые появившись для интерфейса Firewire (**https://papers.put.as/papers/macosx/2005/2005-firewire-cansecwest.pdf**), техники DMA-атак были реализованы для PCI Express и Thunderbolt (**https://thunderspy.io/**).

[1] https://www.nexustechnology.com/products/memory-interposers/ddr4-memory-interposers/ddr4-96-ball-logiccompliance-interposer/.

Существуют Open Source реализации устройств и утилит для проведения DMA-атак (https://github.com/ufrisk/pcileech), позволяющие модифицировать содержимое ОЗУ и запускать код в ядре ОС. Однако в большинстве случаев подобные алгоритмы имеются только для архитектур x86-64. В качестве устройств в основном применяются платы на базе FPGA, например Screamer PCIe Squirrel от Lambda Concept, показанный на фото:

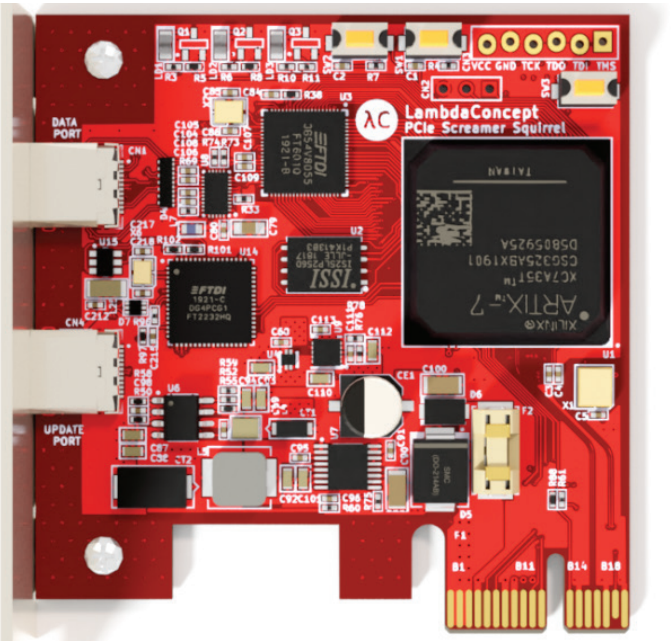


Рис. 3.24. Устройство Screamer PCIe Squirrel[1]

Шина PCI Express широко применяется в современных высокопроизводительных устройствах, таких как мобильные телефоны, планшеты и т. п. Поэтому DMA-атаки могут быть реализованы и для них, например широкую известность получила атака на чип ПЗУ iPhone, подключенный к процесору по PCI Express (**http://ramtin-amin.fr/#nvmepcie** и **http://ramtin-amin.fr/#nvmedma**). Еще одним примером успешной DMA-атаки на устройство является взлом NVidia Jetson Nano: PCIe DMA Attack against a secured Jetson Nano (**https://www.thegoodpenguin.co.uk/blog/pcie-dma-attack-against-a-secured-jetson-nano-cve-2022-21819/**).

Транзакции DMA в большинстве современных процессоров идут через блок IOMMU (англ. input/output memory management unit – блок управления памятью для операций ввода-вывода), транслирующий виртуальные адреса, видимые аппаратным устройством, в физические адреса. Для защиты от DMA-атак производители железа и ОС используют различные технологии, имеющие общее название IOMMU protection. Например, реализация IOMMU фирмы Intel,

[1] https://shop.lambdaconcept.com/home/50-screamer-pcie-squirrel.html.

имеющая название VT-d, может блокировать неразрешенные DMA-транзакции при соответствующей настройке со стороны OS или BIOS. ОС Windows начиная с Windows 10 имеет механизм Kernel DMA Protection (**https://learn.microsoft.com/en-us/windows/security/information-protection/kernel-dma-protection-for-thunderbolt**), позволяющий блокировать внешние устройства и не разрешать им осуществлять транзакции DMA, если драйверы этих устройств не поддерживают изоляцию памяти (DMA-remapping). Транзакции DMA разрешаются только в выделенный драйвером устройства диапазон адресов. Как и любые другие механизмы защиты, механизмы IOMMU protection являются объектом интереса исследователей безопасности и бывают скомпрометированы. Одно из самых известных исследований описано в статье «Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals» (**https://thunderclap.io/thunderclap-paper-ndss2019.pdf**).

MITM

Одной из распространенных техник при проведении динамического анализа является Man-in-the-Middle (MITM). Самые частые применения MITM при анализе встраиваемых систем – это атаки на беспроводные протоколы (Bluetooth, Wi-Fi и т. п.), а также на ПЗУ. Например, задача обхода ограничения на количество попыток подбора пин-кода в мобильных устройствах была решена с помощью вариации MITM-атаки, получившей название NAND mirroring: The bumpy road towards iPhone 5c NAND mirroring (**https://arxiv.org/ftp/arxiv/papers/1609/1609.04327.pdf**). На фото показана плата iPhone с демонтированной микросхемой ПЗУ и адаптером, позволяющим подключать микросхему без пайки:

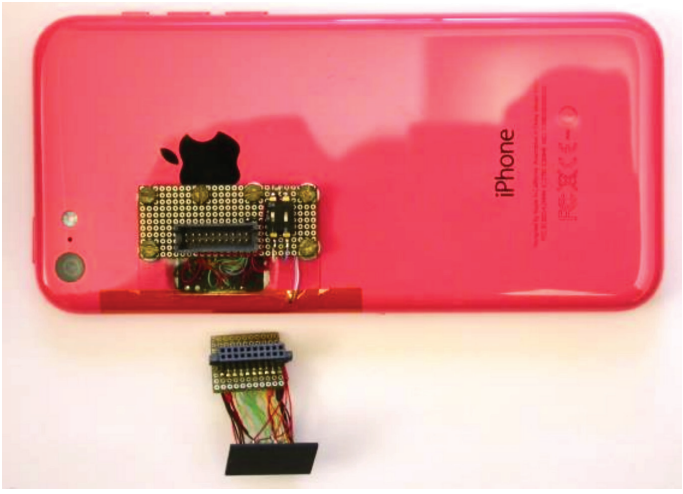


Рис. 3.25. Модификация платы iPhone для возможности отсоединения микросхемы ПЗУ[1]

[1] https://arxiv.org/ftp/arxiv/papers/1609/1609.04327.pdf.

Счетчик неверных попыток ввода пин-кода сохранялся в ПЗУ; сохранив состояние ПЗУ и восстанавливая его до исходного после нескольких попыток подбора пин-кода, можно обойти блокировку устройства и реализовать атаку перебором. При этом модификация кода, проверяющего счетчик, невозможна, т. к. используется механизм secure boot.

Получение информации по беспроводным протоколам

До этого момента мы говорили о физическом подключении к исследуемому устройству для получения какой-либо дополнительной информации для проведения динамического анализа. Многие современные устройства имеют беспроводные интерфейсы взаимодействия. Наиболее распространенными являются Bluetooth, Wi-Fi, сотовые сети стандартов 2G-5G, LoRa и т. п. Не забывайте, что не стоит включать устройство с беспроводными модулями связи без антенн. Это может привести к выходу радиочасти устройства из строя.

Обмен информацией по многим распространенным беспроводным интерфейсам может быть проанализирован доступными большинству исследователей средствами – SDR или универсальными приемопередатчиками. С помощью SDR можно проводить множество экспериментов, позволяющих получить больше информации об исследуемом устройстве. Например, существуют реализации мобильных базовых станций сотовой сети или проекты имитации сигнала GPS.

SDR

Software Defined Radio – программно определяемое (или реализованное) радио. Класс устройств, позволяющих с помощью ПО менять радиочастотные параметры, включая, в частности, диапазон частот, тип модуляции или выходную мощность. SDR позволяет принимать и передавать любые радиосигналы из своего рабочего диапазона, оцифровывать их и передавать на компьютер для анализа и обработки. SDR – обязательный инструмент в лаборатории исследователя встраиваемых систем с беспроводными интерфейсами. Ключевые возможности SDR – это работа в очень широком диапазоне частот, от десятков мегагерц до нескольких гигагерц, и получение необработанного сигнала, т. е. не прошедшего декодирование.

Современные SDR представляют из себя плату с чувствительным радиотрактом, сверхбыстрым АЦП высокой разрядности, ПЛИС и микроконтроллером. Исключением является проект RTL-SDR, работающий на базе TV-тюнера RTL2832U и позволяющий получить SDR с базовыми характеристиками за минимальную цену.

На рис. 3.26 для примера показана структурная схема полноценного SDR bladeRF 2.0 micro. Рассмотрим принцип работы SDR на его примере.

По USB3 плата подключается к компьютеру, на котором запускается ПО анализа радиосигнала. Интерфейс USB3 реализован на базе микроконтроллера Cypress FX3. Микроконтроллер FX3 подключается к FPGA Intel Cyclone V, которая, в свою очередь, подключена к трансиверу с интегрированным цифро-

аналоговым (ЦАП, DAC) и аналогово-цифровым преобразователем (АЦП, ADC) Analog Devices.

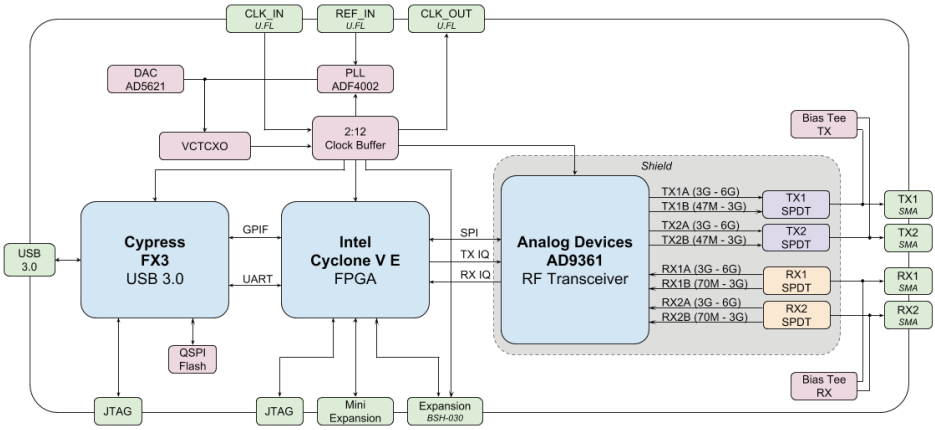


Рис. 3.26. Структурная схема SDR-радио bladeRF 2.0 micro[1]

Рассмотрим сценарий работы SDR при приеме данных из радиоэфира. Электромагнитные волны поступают на вход радиотракта трансивера, где с помощью аналого-цифрового преобразователя (АЦП, он же Analog to Digital Converter, ADC) преобразуются в цифровой сигнал. Цифровой сигнал по шине передается в микросхему FPGA, где происходит его обработка. Обработанный сигнал из FPGA передается в микроконтроллер, обрабатывающий команды USB и упаковывающий обработанный сигнал в пакеты данных, передающихся по шине USB. В случае передачи данных схема работы меняется на обратную.

Самыми распространенными SDR, применяющимися при анализе устройств, являются bladeRF, HackRF, LimeSDR, USRP и Red Pitaya. Внешний вид рассмотренной ранее SDR bladeRF 2.0 micro показан на рисунке, на нем хорошо видны ключевые компоненты, описанные ранее.

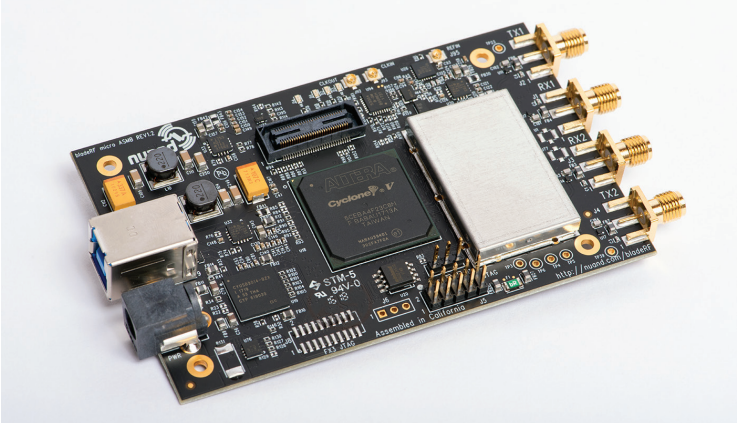


Рис. 3.27. Внешний вид SDR bladeRF 2.0[2]

[1] https://www.nuand.com/product/bladerf-xa9/.

[2] https://www.nuand.com/product/bladerf-xa9/.

Для работы с SDR существует множество open source утилит, осуществляющих обработку и визуализацию принимаемого сигнала, например SDR Concole или GNU Radio. На скриншоте представлен внешний вид распространенной программы SDRSharp (SDR#).

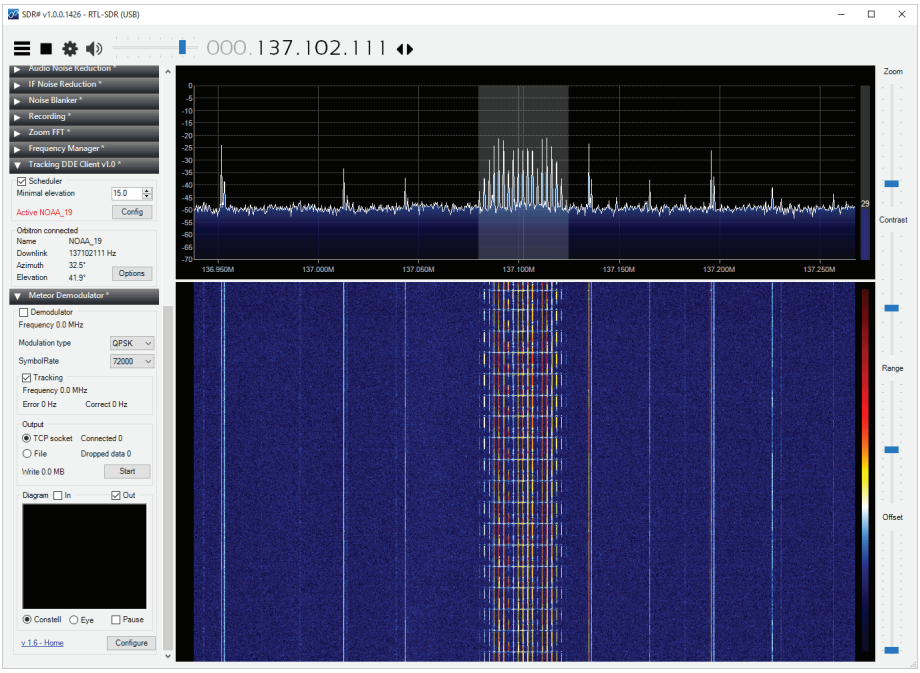


Рис. 3.28. Внешний вид главного окна программы SDRSharp[1]

Flipper Zero

Для исследования беспроводных протоколов устройств, отличных от Wi-Fi и Bluetooth, не всегда получается найти готовый инструмент. А количество протоколов, работающих в диапазоне частот до 1 ГГц, весьма значительно, как и количество использующих их устройств. В 2022 году появилось отличное устройство-комбайн, позволяющее работать с сигналами до 1 ГГц и имеющее open source прошивку, – flipper zero. Внешний вид устройства показан на рисунке.

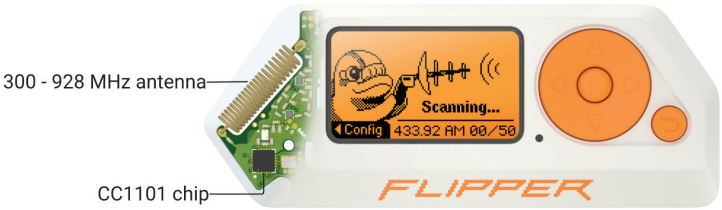


Рис. 3.29. Устройство flipper zero в режиме сканирования радиоэфира[2]

[1] https://www.rtl-sdr.com/setting-up-an-rtl-sdr-based-aptmeteor-satellite-weather-stations/.

[2] https://docs.flipperzero.one/sub-ghz.

Флиппер позволяет записывать, декодировать (при наличии соответствующего скрипта) и воспроизводить сигналы на стандартных частотах 315, 433, 868 и 915 МГц (а также на расширенном диапазоне частот при установке модифицированной прошивки). Также он может выполнять роль NFC-считывателя и эмулятора NFC-метки.

Устройство активно развивается, появляются модули для декодирования новых протоколов. Так как прошивка устройства доступна в исходных кодах, можно легко дописать требуемый модуль самостоятельно, используя флиппер как стабильно работающую аппаратную платформу. Конечно, не составляет труда собрать устройство с аналогичным функционалом из готовых модулей, но флиппер работает «из коробки» и позволяет убрать из исследования долгий процесс отладки самого инструмента.

Разрабатываем патч прошивки

Рассмотрев основные подходы к получению информации от работающего устройства, закончим эту главу небольшим примером разработки патча прошивки. Наш патч будет представлять из себя RWE (Read-Write-Execute) код, получаемый по одному из доступных интерфейсов, ведь мы будем знать, что отладочный интерфейс заблокирован. В качестве подопытного устройства возьмем уже знакомый нам по главе 1 простой конвертер интерфейса RS-485 Modbus в проприетарный протокол.

Конвертер основан на микроконтроллере GigaDevices GD32E230K8T6 на базе архитектуры ARM Cortex-M23. Для приближения к реальным условиям введем следующие ограничения.

1. Микроконтроллер имеет выставленный наивысший уровень защиты Read Out Protection, т. е. отладочный интерфейс SWD недоступен.
2. Обновление прошивки доступно, но зашифровано.
3. У нас есть утилиты обновления прошивки по протоколу Modbus через RS-485, но они не содержат в себе полезной информации и передают прошивку в зашифрованном виде.
4. Мы знаем, что в микроконтроллере есть загрузчик размером 4 Кб (не путать с BootROM), выполняющий обновление прошивки, расшифровывающий и проверяющий ее целостность. То есть ключи и функции для расшифровки обновления находятся в загрузчике.
5. Мы смогли получить ключи шифрования прошивки, знаем алгоритм шифрования и можем расшифровать код обновления, модифицировать его и зашифровать снова таким образом, чтобы загрузчик успешно его прошил.

Мы хотим написать RWE-патч для прошивки, позволяющий считывать и записывать любой участок адресного пространства микроконтроллера, а также передавать управление на любой адрес в коде. Это позволит проводить динамический анализ при отключенном отладочном интерфейсе. Используя разработанный патч, считаем содержимое загрузчика, ведь в нем могут быть какие-то дополнительные функции, про которые мы не узнаем, анализируя только прошивку. Тогда предполагаемая полная последовательность действий примет вид:

1) расшифровать обновление прошивки;
2) провести статический анализ, определить структуру прошивки и найти свободное место для RWE-патча;
3) найти обработчики интерфейса, через который мы будем осуществлять обмен данными с RWE-патчем (выберем RS-485, через который происходит обмен с ПК);
4) разработать RWE-патч, модифицировать образ прошивки;
5) зашифровать модифицированный образ;
6) обновить прошивку на модифицированную;
7) считать содержимое внутренней flash-памяти микроконтроллера и найти код загрузчика.

В контексте разработки патча нам не важны пункты 1, 5–7, их мы перечислили для лучшего понимания всего процесса. Мы сконцентрируемся на пунктах 2–4, непосредственно имеющих отношение к разработке патча.

1. Размер файла прошивки составляет 0x8880 байт, причем в начале есть заголовок длиной 0x10 байт. Находим в документации на GD32E230xx расположение flash-памяти в адресном пространстве:

Code		0x1FFF F810 - 0x1FFF FFFF	Reserved
		0x1FFF F800 - 0x1FFF F80F	Option bytes
		0x1FFF EC00 - 0x1FFF F7FF	System memory
		0x0801 0000 - 0x1FFF EBFF	Reserved
		0x0800 0000 - 0x0800 FFFF	Main Flash memory
		0x0001 0000 - 0x07FF FFFF	Reserved

Рис. 3.30. Диапазоны памяти для хранения кода в адресном пространстве микроконтроллеров GD32E230xx[1]

Получается, что прошивка должна быть загружена в дизассемблер по адресу 0x08000000, соответствующий Main Flash memory. А ее размер больше половины всей flash-памяти, т. е., скорее всего, обновление представляет собой полноценную прошивку. Пробуем загрузить в дизассемблер – и получаем переход из entry point в середину какой-то функции:

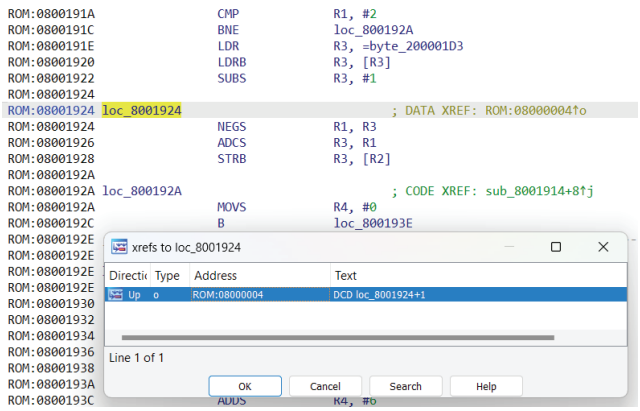


Рис. 3.31. Некорректное определение адреса загрузки прошивки

[1] https://gd32mcu.com/data/documents/datasheet/GD32E230xx_Datasheet_Rev1.9.pdf.

Значит, мы сделали некорректное предположение об адресе загрузки прошивки. Немного посмотрев в листинг дизассемблера и вспомнив, что где-то должен располагаться загрузчик размером 4 Кб (0x1000 байт), а в начале файла с прошивкой была сигнатура длиной 0x10 байт, становится понятно, что корректный адрес для начала кода прошивки 0x08001010, а по адресу 0x08000000, скорее всего, будет располагаться загрузчик. Перезагружаем прошивку по адресу 0x08001010 и получаем корректный код инициализации:

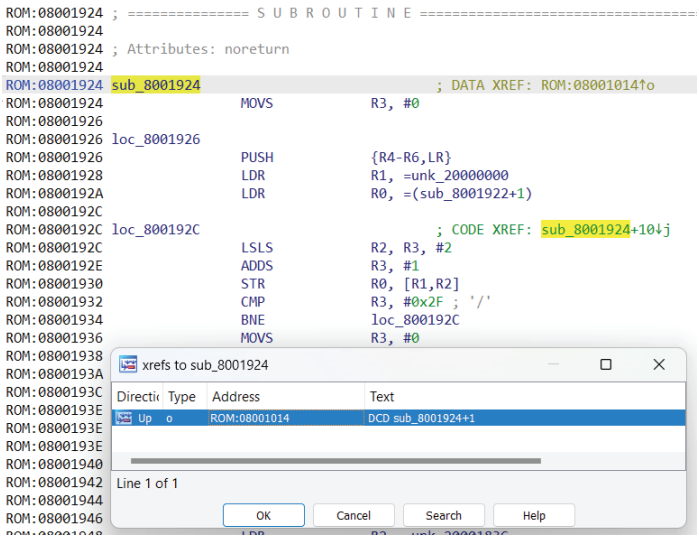


Рис. 3.32. Корректное определение адреса загрузки прошивки

Анализируем прошивку в статике и подтверждаем свое предположение, что обновление представляет собой полнофункциональную прошивку, полностью заменяющую предыдущую версию. Не обновляется только загрузчик, его кода у нас нет, а именно он занимается обновлением прошивки. Структура прошивки самая простая: линейно лежащий блок информации, без «банков» и других механизмов подгрузки кода. Прошивка не имеет ОС, т. е. является `bare-metal` кодом. Главный цикл прошивки весьма компактен, что неудивительно, учитывая простоту функций устройства (рис. 3.33).

Самый простой способ разместить дополнительный код RWE-патча – дописать его в свободное место после конца прошивки. В данном случае у нас остается еще 0x10000 – 0x1000 – 0x8880 = 0x6780 байт, которые мы можем использовать для размещения своего патча.

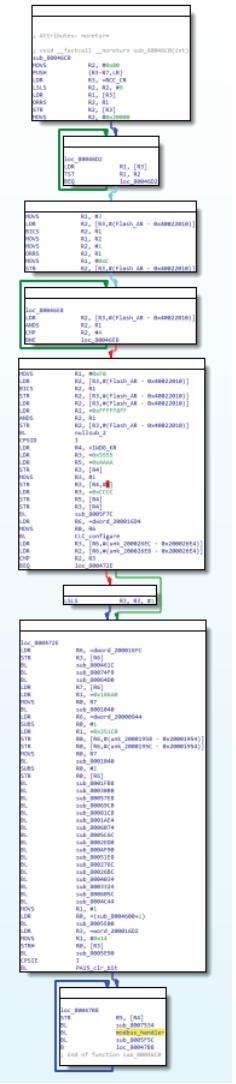


Рис. 3.33. Главный цикл прошивки

2. Теперь нам необходимо найти обработчики протокола Modbus, работающего через интерфейс RS-485. Их можно найти несколькими способами, но мы пойдем самым простым. Мы уже прозвонили плату устройства и знаем, что RS-485 от клеммной колодки идет на RS-485/RS-422 трансивер TP8485E, который преобразовывает сигнал UART с выводов PA2–PA3 микроконтроллера.

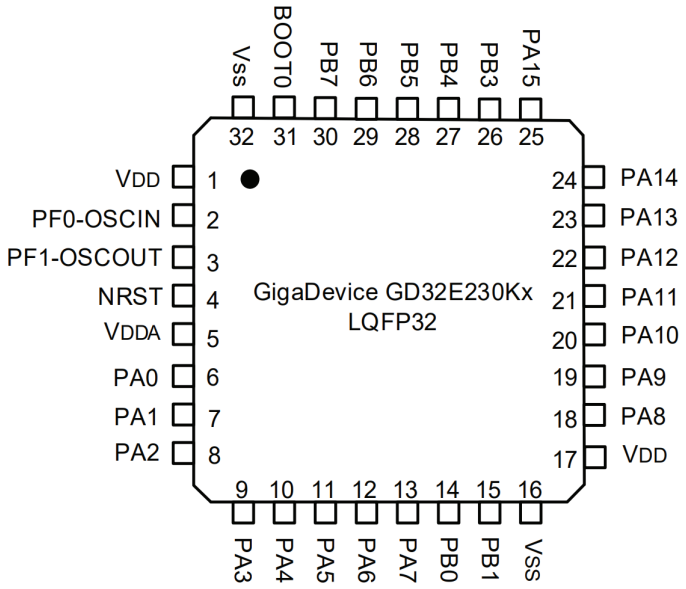


Рис. 3.34. Назначение выводов микроконтролера GD32E230Kx[1]

Посмотрим в документации, сигналы какого из портов U(S)ART могут быть на этих выводах:

PA2	12	I/O		Default: PA2 Alternate: USART0_TX(3), USART1_TX(4), TIMER14_CH0(5) Additional: ADC_IN2, CMP_IM7
PA3	13	I/O		Default: PA3 Alternate: USART0_RX(3), USART1_RX(4), TIMER14_CH1(5) Additional: ADC_IN3

Рис. 3.35. Варианты функций выводов PA2 и PA3 микроконтроллера GD32E230Kx[2]

Видим, что это может быть USART0 или USART1, но, внимательно посмотрев на сноски 3 и 4, видим следующую информацию: (3) Functions are available on GD32E230K4 devices only, (4) Functions are available on GD32E230K8/6 devices. Модель нашего микроконтроллера

[1] https://gd32mcu.com/data/documents/datasheet/GD32E230xx_Datasheet_Rev1.7.pdf.
[2] https://gd32mcu.com/data/documents/datasheet/GD32E230xx_Datasheet_Rev1.7.pdf.

GD32E230K8T6, значит, у нас остается только вариант USART1. Находим в документации, что диапазон 0x40004400-0x400047FF адресного пространства зарезервирован под регистры USART1. Значит, нам надо найти в прошивке обращения к этому диапазону адресного пространства, что позволит нам найти обработчик UART-интерфейса (вариант с использованием DMA мы не рассматриваем). Находим нужный код и с помощью него выходим на функцию в главном цикле, являющуюся обработчиком протокола Modbus:

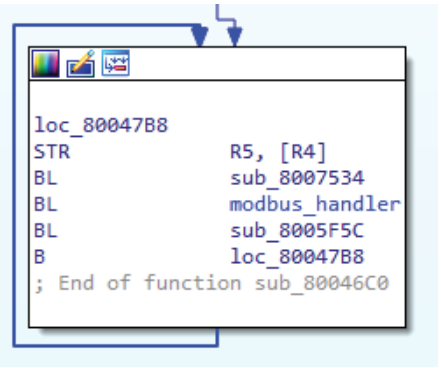


Рис. 3.36. Вызов обработчика modbus в главном цикле прошивки

3. Теперь напишем сам RWE-патч. Для этого скачаем пакет gcc-arm-none-eabi и напишем простой код в файл cstart.c, проверяющий значения в приходящем по Modbus запросе и выполняющий требуемые действия:

```
#include <stdint.h>

typedef enum {
  READ = (uint8_t)'R',
  WRITE = 'W',
  EXEC = 'E'
} COMMANDS;

typedef struct
{
  uint32_t * address;
} read_cmd_t;

typedef struct
{
  uint32_t * address;
  uint32_t data;
} write_cmd_t;
```

```
typedef struct
{
  void (*code)(void);
} exec_cmd_t;

typedef struct
{
  COMMANDS cmd;
  union
  {
   read_cmd_t readCmd;
   write_cmd_t writeCmd;
   exec_cmd_t execCmd;
  };
} cmd_t;

typedef struct
{
  uint32_t data;
  uint32_t size;
} response_t;

volatile cmd_t cmd __attribute__((section(".inBuffer")));
volatile response_t response __attribute__((section(".outBuffer")));

void __attribute__((section(".hook_handler"))) _Hook()
{
  if(cmd.cmd == READ)
  {
   response.data = *(cmd.readCmd.address);
   response.size = sizeof(uint32_t);
  }
  else if(cmd.cmd == WRITE)
  {
   *(cmd.writeCmd.address) = cmd.writeCmd.data;
  }
  else if(cmd.cmd == EXEC)
  {
   cmd.execCmd.code();
  }
  return;
}
```

Исходный код RWE-патча

Нам надо передать управление на наш RWE-патч из оригинального обработчика Modbus, для этого пропатчим вызов функции подсчета контрольной суммы Modbus по адресу 0x08006DC8, заодно убрав ее проверку, т. к. наш протокол ее не пройдет. Для этого создадим файл startup.s всего с двумя строчками:

```
.section .hook, "x"
        bl 0x08009880
```

Код перехвата потока выполнения для RWE-патча

Мы хотим расположить наш RWE-патч после оригинальной прошивки, т. е. по адресу 0x08009880. Хук должен располагаться по адресу 0x08006DC8. Также наш код использует области памяти, выделенные для хранения входных и выходных буферов. Создадим файл скрипта линкера linkerscript.ld со следующим содержимым секций:

```
SECTIONS
{
      .hook 0x08006DC8 : { startup.o }
      .hook_handler 0x08009880 : { KEEP(*(.hook_handler)) }
      .ram_in_buffer 0x20001710 : {KEEP(*(.inBuffer))}
      .ram_out_buffer 0x20001810 : {KEEP(*(.outBuffer))}
}
```

Код настроек линкера для сборки RWE-патча

Компилируем и линкуем исходники следующими командами и получаем патч для модификации образа прошивки:

```
arm-none-eabi-as.exe -o GD32\startup.o GD32\startup.s
arm-none-eabi-gcc.exe -c -nostdlib -nostartfiles -mthumb -lgcc -o GD32\
cstart.o GD32\cstart.c
arm-none-eabi-ld.exe -T GD32\linkerscript.ld -o GD32\patch.elf GD32\
startup.o GD32\cstart.o
arm-none-eabi-objcopy.exe -O binary GD32\patch.elf GD32\patch.bin
```

Если вы не до конца понимаете, что делают эти команды, рекомендую прочитать книжку «Bare-metal programming for ARM» (**http://umanovskis.se/files/arm-baremetal-ebook.pdf**).

Загрузим скомпилированный код патча в дизассемблер, чтобы проверить, что мы все сделали правильно:

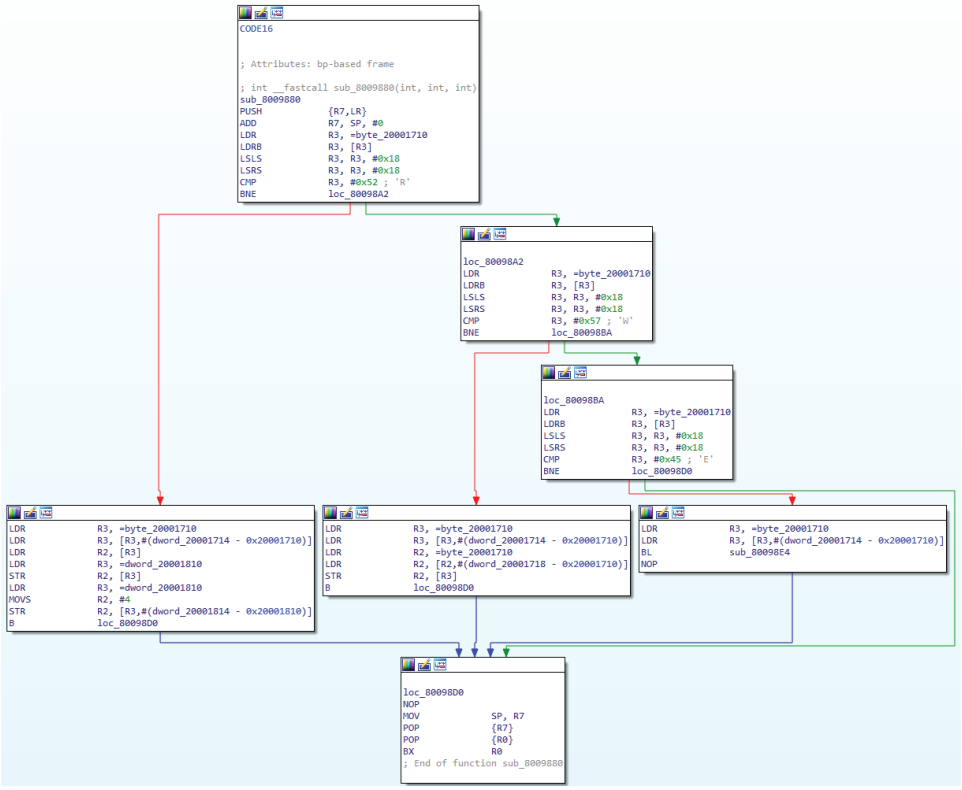


Рис. 3.37. Листинг дизассемблированного кода RWE-патча

Остается только правильно пропатчить оригинальный образ прошивки, не забыв команду перехвата управления. Теперь ничто не мешает нам выполнить шаги 5–7 и получить образ загрузчика из встроенной flash-памяти микроконтроллера, расположенного в диапазоне 0x08000000–0x08001000.

Мы рассмотрели вариант с разработкой патча для прошивки, построенной по принципу bare-metal. Большинство исследований, опубликованных в интернете, посвящены цифровым устройствам, построенным на базе какой-либо ОС. Часто в подобных системах можно внести изменения за счет правки текстовых конфигов или скриптов. Например, хорошим материалом для изучения станет статья по исследованию роутеров d-link (**https://www.greynoise.io/blog/debugging-d-link-emulating-firmware-and-hacking-hardware**).

Level Up!

В этой главе были рассмотрены практики, позволяющие получить дополнительную информацию для проведения исследования в процессе работы устройства. Как и в предыдущих главах, не стоит следовать подходам, предлагаемым в этой главе, буквально. Ваша основная задача – понять варианты

получения дополнительной информации, спроецировать их реализуемость на конкретно ваше устройство и возможности, которые вам доступны. Скорее всего, в вашем исследовании понадобится адаптация описанных методов или их комбинация. Разработка патчей, расширяющих возможности получения дополнительной информации от устройства, также является интересной задачей, которую придется решать регулярно.

Знания, как использовать различные аппаратные интерфейсы, в том числе беспроводные, для получения дополнительной информации от работающего устройства, могут существенно ускорить ход исследования или даже кардинально повлиять на полученный результат.

Level 4

Механизмы защиты встраиваемых систем

Производители встраиваемых систем все чаще начинают защищать устройства и прошивки от реверс-инжиниринга и модификации. Они преследуют несколько целей – усложнить клонирование устройства и затруднить извлечение обрабатываемых на устройстве данных. Применяемые подходы могут сильно усложнить процесс исследования устройств или даже сделать его невозможным ввиду отсутствия необходимых ресурсов (как временных, так и технологических).

Конечно, исследование логики работы устройства можно провести без запуска своего кода. Но всегда приятно (по моему мнению) полностью взять устройство под свой контроль и преодолеть те механизмы, которыми производители защищают устройства. В этой главе мы рассмотрим архитектурные особенности цифровых устройств, с которыми можно столкнуться в процессе разработки патча прошивки, какие существуют способы защиты устройств от исследования, модификации прошивок и как их можно обойти.

Контрольные суммы прошивки

Довольно часто используются для контроля целостности прошивки. Контрольные суммы не служат для защиты прошивки от модификации (хотя иногда кажется, что некоторые производители думают именно так), они лишь показывают, что прошивка не была повреждена в процессе передачи данных или просто хранения в памяти. Подсчет и проверка контрольных сумм могут производиться как в ПО обновления прошивки, так и в прошивке микроконтроллера. Например, при старте загрузчик прошивки может считывать из ПЗУ (внешнего или внутреннего) основную прошивку и считать контрольную сумму. В случае расхождения (например, из-за повреждения прошивки) он может попытаться загрузить резервную копию прошивки (если она есть) или включить отладочный интерфейс для восстановления устройства с помощью программатора/CLI. На рисунке представлен пример алгоритма работы загрузчика с проверкой контрольной суммы и восстановлением поврежденной основной прошивки из резервной.

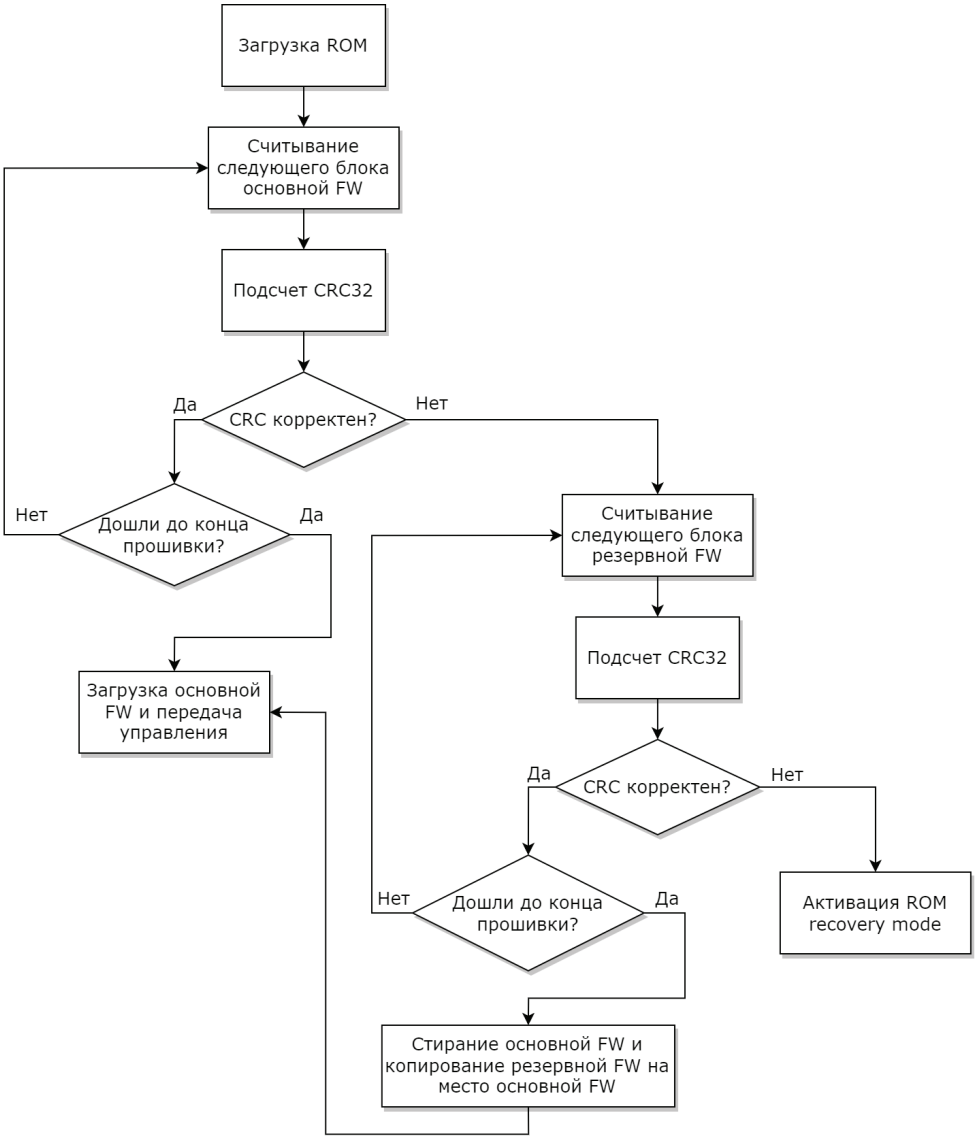


Рис. 4.1. Блок-схема возможного варианта проверки целостности прошивки

Контрольная сумма может считаться по разным алгоритмам, самыми часто используемыми являются Checksum 32 и CRC32. Хотя могут встречаться и другие варианты разрядности и даже нераспространенные полиномы для стандартных алгоритмов. В случае применения различных вариантов Checksum может использоваться значение, дополняющее общую сумму проверяемого блока данных до 0. Например, значение контрольной суммы для проверяемого блока данных получилось 0x1234, тогда значение дополнения будет 0x10000 – 0x1234 = 0xEDCC. Именно такой алгоритм использовался для проверки целост-

ности модулей расширения PCI Option ROM в Legacy-ядрах BIOS. Для проверки значений контрольных сумм удобно пользоваться встроенными в различные шестнадцатеричные редакторы средствами подсчета значений контрольных сумм по разным алгоритмам. Некоторые редакторы умеют выводить результат подсчета всех поддерживаемых контрольных сумм, что очень удобно:

Algorithm	Checksum/Digest
Checksum - UByte (8 bit)	00000000 006438C4
Checksum - UShort (16 bit) - Little Endian	00000000 3CE6B687
Checksum - UShort (16 bit) - Big Endian	00000000 27B6463D
Checksum - UInt (32 bit) - Little Endian	00001E5B 5B7179A1
Checksum - UInt (32 bit) - Big Endian	000014BE 2AD22E63
Checksum - UInt64 (64 bit) - Little Endian	2606A77D 356AE12A
Checksum - UInt64 (64 bit) - Big Endian	54CD463D D604F277
CRC-16	8AAB
CRC-16/CCITT	75B3
CRC-32	76B1F81E
Adler32	30AA3EA1
MD2	0784DD4F276C911794129FBD93E1BA29
MD4	33D7C1985E06BFD7235248F871AED8A3
MD5	F1B0C1773D638EDA27C37C73E4CDD2B0
RIPEMD160	343FC0FAF91B1EC19591EE94689530B81D5691EA
SHA-1	A1F4EB28141396D9C3717392CA162535619CC0C1
SHA-256	D3C1D527BA3D4AFFCFC2416165EDF9AE1F56F5934C521D71727...
SHA-384	001B1EE275080AC69091D9A577361E5274043E7CC7675D20F6E5...
SHA-512	5358ACDE3CA1BBC5D57259FFF1B8484741874E74C803B16A667...
TIGER	34F2516A0E9974735A5D3E8D9F06001976260857622BEFE0

Рис. 4.2. Результат подсчета контрольных сумм и хешей разными алгоритмами в шестнадцатеричном редакторе

Криптографическая подпись прошивки

В современных встраиваемых системах все чаще встречается криптографическая подпись прошивки для гарантии целостности при передаче и отсутствия модификаций. Чаще всего встречаются два варианта использования подписи закрытым ключом значения хеша от содержимого прошивки:

- проверка подписи открытым ключом в процессе обновления прошивки;
- проверка целостности прошивки при загрузке устройства, так называемый механизм доверенной загрузки (secure boot), который будет рассмотрен далее.

Типовой механизм проверки подписи при обновлении выглядит следующим образом.

Рис. 4.3. Типовой алгоритм проверки подписи прошивки при обновлении

В качестве хеш-функций широко применяются MD5, SHA1 и SHA256. Использовать MD5 в настоящее время считается небезопасным ввиду возможного наличия коллизий, однако сгенерировать требуемое значение хеш-функции с учетом употребления модифицированных данных прошивки все еще довольно сложно.

Для подписи хеша в большинстве случаев применяется алгоритм RSA, при этом длина ключа в современных устройствах составляет до 4096 бит, т. е. взломать данный алгоритм шифрования перебором не представляется возможным на текущем уровне вычислительной мощности.

Тем не менее обойти механизм подписи прошивки для запуска модифицированного кода иногда удается за счет использования архитектурных ошибок реализации механизма. Рассмотрим три наиболее часто встречающиеся ошибки (вариант, когда подпись прошивки проверяет только ПО обновления на компьютере, мы не рассматриваем, т. к. при нем можно модифицировать прошивку в обход ПК).

1. Ошибки при реализации алгоритмов проверки данных. Например, отсутствие подписи размера передаваемых данных или адреса начального вектора для передачи управления.
2. Состояние «гонки», т. е. возможность изменения состояния данных между моментом проверки и моментом использования (Time-of-check to time-of-use, TOCTOU).

   Вернемся к рассмотренному ранее типовому варианту проверки подписи. Представим, что размер обновления больше, чем размер доступной ОЗУ микроконтроллера. Получается, невозможно целиком передать содержимое обновления в прошивку микроконтроллера, посчитать от нее значение хеша и сравнить с полученным подписанным значением. В таком случае придется передавать обновление для подсчета значения хеша блоками какого-то размера, перемещающимися в выделенный

участок ОЗУ. В случае успешной проверки значения хеша прошивка может быть обновлена, но т. к. микроконтроллеру негде хранить все обновление целиком, данные передаются повторно. Алгоритм обновления примет следующий вид.

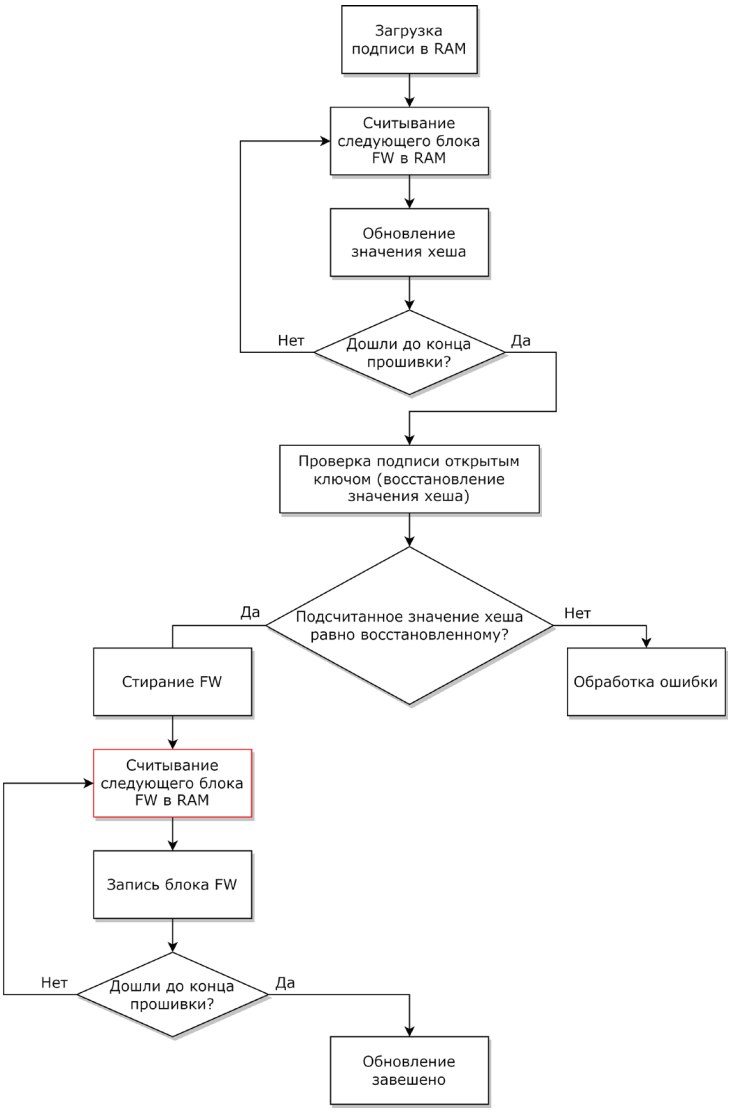


Рис. 4.4. Вариант реализации алгоритма обновления прошивки

Как вы уже поняли, при таком алгоритме проверки совершенно не обязательно оба раза передавать одну и ту же прошивку. На этом и строится атака. В первый раз передаются оригинальные данные обновления, от них успешно проверяется подпись, а во второй раз передаются уже модифицированные данные для обновления.

Как может выглядеть правильная реализация алгоритма обновления прошивки при ограничениях микроконтроллера по свободному ОЗУ? Можно подписывать хеш каждого передаваемого блока или сохранить обновление на встроенной flash-памяти микроконтроллера и проверить уже оттуда. Правда, при таком механизме также можно попытаться изменить данные в ПЗУ, т. е. разорвать цепочку проверки обновления.

3. Использование альтернативных механизмов доступа к ПЗУ.

   Представим, что механизм проверки подписи прошивки реализован абсолютно корректно и нельзя подменить данные в процессе обновления. Казалось бы, ситуация безнадежная, ведь взламывать современные криптографические алгоритмы фактически бессмысленно. Но разработчики могли не задумываться о том, чтобы сделать систему реально безопасной, а могли лишь реализовывать техническое задание (ТЗ). Если в ТЗ было написано «реализовать процедуру безопасного обновления прошивки», то они все сделали правильно. Но в ТЗ могло не быть пункта «отключить диагностические функции прошивки». Иногда при абсолютно корректной реализации обновления прошивки с проверкой криптографической подписи в прошивке могут быть недокументированные функции записи содержимого ПЗУ без каких-либо проверок. То есть не надо пытаться обойти процедуру обновления прошивки, надо просто пойти другим путем.

Доверенная загрузка устройства

Механизм доверенной загрузки (он же trusted boot, или secure boot) встречается во встраиваемых системах, где компрометация обрабатываемых данных может привести к критичным последствиям для пользователя или производителя устройства. Он часто встречается в современных мобильных устройствах, компьютерах и сетевых устройствах. Ранее мы рассмотрели один из вариантов использования криптографической подписи прошивки в процессе обновления. При механизме доверенной загрузки подпись прошивки проверяется каждый раз при включении устройства. Так как современное устройство, в котором используется доверенная загрузка, как правило, включает в себя несколько уровней загрузчиков (как минимум первичный, вторичный и ОС), для реализации механизма доверенной загрузки используется цепочка доверия, когда каждый следующий загрузчик проверяется на целостность предыдущим перед передачей потока выполнения. На рисунке представлена типовая схема цепочки доверия.

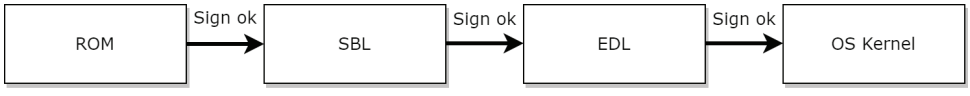


Рис. 4.5. Типовая цепочка доверия

Но кто проверит самый первый загрузчик? Ведь если он скомпрометирован, то и остальная система тоже. Первичный загрузчик является корнем доверия

(Root of Trust, RoT). Как правило, он расположен в ROM микроконтроллера и содержит в себе функции для проверки целостности загружаемого в дальнейшем кода. Он должен хранить значения хеша проверяемого модуля и ключи для расшифровки (в случае если требуется не только проверка, но и расшифровка загружаемого кода), например во фьюзах микроконтроллера. Для организации корня доверия могут использоваться специальные микросхемы – secure element и криптопамять, которые мы рассмотрим далее в разделе «Аппаратная поддержка механизмов защиты». Использование ROM-кода в качестве первичного загрузчика имеет как положительные, так и отрицательные стороны. К положительным относятся гарантия неизменности кода и простота реализации RoT. К отрицательным – невозможность внесения изменений в большинстве микроконтроллеров при обнаружении ошибок без выпуска новой версии чипа (существуют микроконтроллеры, позволяющие выполнять патч ROM-кода с помощью специального аппаратного механизма).

Для обхода цепочки доверенной загрузки необходимо эту цепочку разорвать. И тут снова исследователям стоит обратить внимание на особенности архитектурных реализаций механизма доверенной загрузки. Бывает, что цепочка разрывается в какой-то момент, например как представлено на рисунке. Для разрыва цепочки могут быть использованы в том числе и методы, описанные в разделе про криптографическую подпись прошивки.

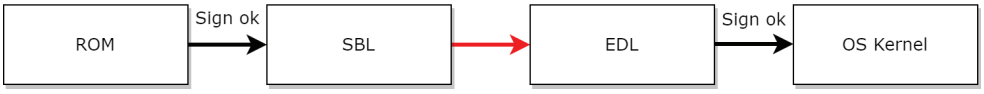


Рис. 4.6. Вариант разрыва цепочки доверия

Хороший пример использования разрыва цепочки доверенной загрузки показан в исследовании Amlogic S905 SoC: bypassing the (not so) Secure Boot to dump the BootROM (**https://fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html**). Больше информации о доверенной загрузке устройств можно узнать из описаний ARM Trusted Firmware и Apple Platform Security (**https://support.apple.com/guide/security/welcome/web**). В устройствах Apple подход к построению безопасного устройства реализован на хорошем уровне, поэтому можно рассматривать их как пример раскрываемой вендором архитектуры безопасности своего решения.

Шифрованные обновления

Производители встраиваемых систем не хотят, чтобы их устройства кто-то пытался анализировать или копировать. Так как для большинства устройств скопировать схемотехнику гораздо проще, чем написать и отладить прошивку, защита прошивки встречается все чаще и чаще. И, говоря про защиту прошивки в этом разделе, я имею в виду, что шифрованные обновления расшифровываются внутри микроконтроллера.

Для шифрования прошивки используются блочные или потоковые шифры. Ключ должен храниться внутри микроконтроллера и не покидать его ни при каких условиях. В случае использования нестойких алгоритмов шифрования (например, DES) можно пытаться перебрать пространство ключей. Для остальных вариантов необходимо искать ошибки в реализации или использовать неинвазивные атаки (например, дифференциальный анализ по питанию) для извлечения ключей шифрования. Про очевидные вещи типа перевода устройства в ROM-mode (recovery), использования JTAG/SWD, анализа утилит обновления и т. п. для получения информации мы уже говорили.

Аппаратная поддержка механизмов защиты

Мы рассмотрели основные механизмы защиты, применяющиеся во встраиваемых системах. Существуют отдельные микросхемы и программно-аппаратно реализованные функции в микроконтроллере, позволяющие упростить реализацию механизмов защиты и существенно повысить их стойкость. Ранее мы уже рассматривали такие механизмы, как Flash Readout Protection, Memory Protection Unit, IOMMU protection и т. д. В этом разделе мы не будем повторяться и сконцентрируемся на технологиях, которые еще не рассматривали.

Датчики на вскрытие корпуса (тамперы)

В некоторых встраиваемых системах, хранящих чувствительные для бизнеса данные (банкоматы, пин-пады, промышленные станки и т. п.), могут применяться техники защиты от вскрытия корпуса или попыток проведения исследований. Их реализация представляет из себя различные датчики (тамперы, от англ. tamper – подделывать), срабатывающие при несанкционированном доступе к внутренностям устройства. После срабатывания датчика, как правило, происходит блокировка устройства, а также стирание чувствительных данных, например ключей, использующихся для шифрования или подписи. Внутри подобных устройств всегда есть элемент резервного питания (аккумулятор или батарея), а для защиты от его несанкционированного отключения используется хранение ключей в RAM-памяти микроконтроллера. То есть при отключении резервного питания ключи стираются из памяти (но атаки типа cold boot иногда провести можно).

Существует несколько основных типов тамперов.

1. *Контактные*. Принцип работы заключается в нарушении электрического контакта при вскрытии корпуса или попытке доступа к элементам печатной платы. Могут быть реализованы в виде микрокнопки или специальных контактных площадок на плате устройства, в нормальном состоянии замкнутых элементами корпуса, пример – на фото:

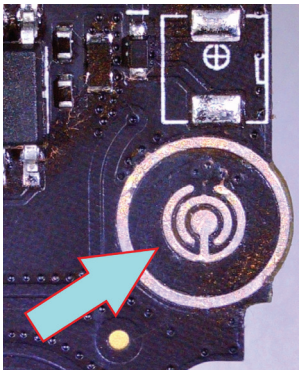


Рис. 4.7. Контактные площадки на плате устройства для защиты от вскрытия

Часто используется защитная пленка со специальным рисунком проводников, сделанная на основе гибкой печатной платы:

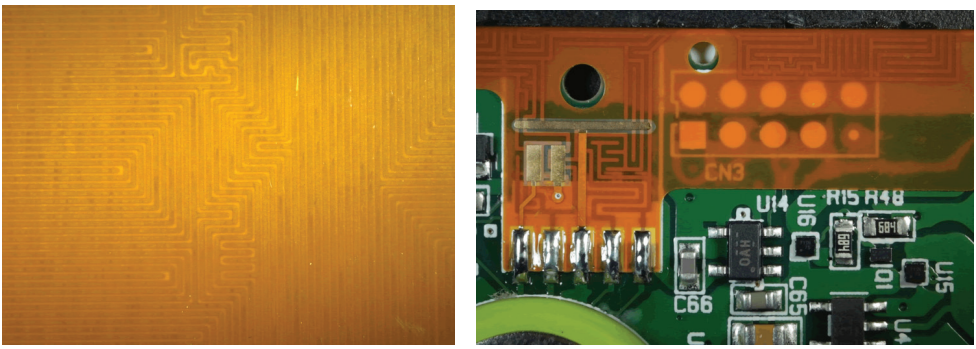


Рис. 4.8. Защитная пленка с проводящим слоем[1]

При попытке доступа к защищаемой ей части печатной платы проводники разрушаются и происходит срабатывание тампера.

2. *Магнитные*. Похожи по принципу на контактные, с тем отличием, что для детектирования вскрытия корпуса используются геркон и постоянный магнит.
3. *Оптические*. Основаны на применении датчиков света. Корпус устройства делается из непрозрачного пластика, и при его вскрытии свет приводит к срабатыванию датчика, расположенного на плате.

В зависимости от типа применяемых тамперов необходимо вырабатывать нужное решение по защите от их срабатывания. Например, сделав предположение, что в устройстве нет датчиков рентгеновского излучения, можно провести рентгенографию устройства и попытаться определить места в корпусе, просверлив которые, можно замкнуть контактные датчики. Или вскрывать устройство в темноте, если есть предположение об использовании оптических датчиков. В подобных устройствах каждый случай уникален, и иногда может потребоваться серия экспериментов с соответствующим количеством «окирпиченных» устройств.

Тамперы редко применяются во встраиваемых системах. Они всегда дополняются специальными схемотехническими решениями и технологиями, затрудняющими анализ устройства. Рассмотрим основные из подобных применяющихся решений.

Криптопамять и Secure Element

Если на устройстве хранятся какие-то чувствительные данные, например ключи шифрования, производитель может использовать специальные микросхемы криптопамяти (crypto memory IC). Это недорогие микросхемы малой емкости (как правило, до нескольких килобайт), позволяющие надежно хра-

[1] https://www.pentestpartners.com/security-blog/tamper-proofing-review-the-izettle-card-payment-terminal/.

нить чувствительные данные. Помимо защищенного от несанкционированного доступа хранилища данных, в подобных микросхемах встроены функции аутентификации, безопасного обмена ключами и надежный генератор псевдослучайных чисел. Микросхемы криптопамяти защищаются от потенциальных инвазивных и неинвазивных атак на этапе проектирования кристалла. Например, содержат в себе схемы контроля питания и при обнаружении попыток VCC-glitch атак стирают данные.

Типичным представителем микросхемы криптопамяти является Atmel AT88SC0808C, структурная схема которой представлена на рисунке.

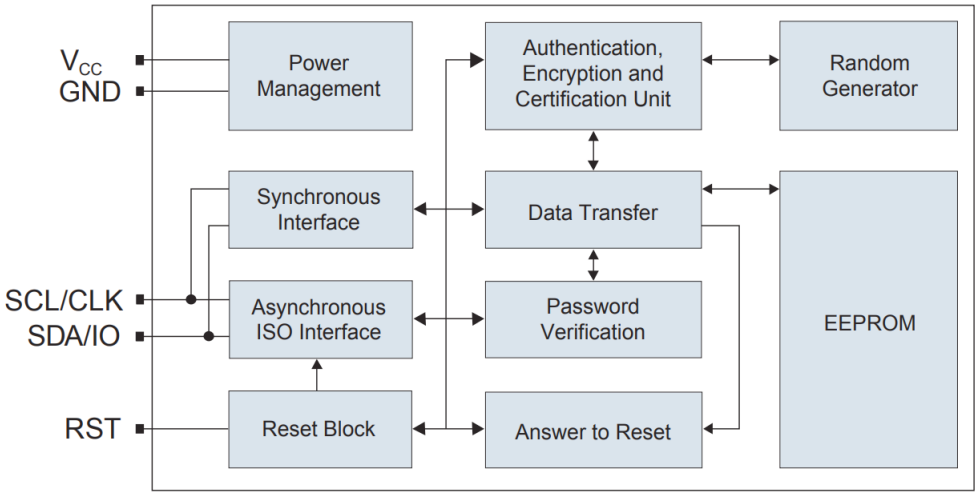


Рис. 4.9. Структурная схема микросхемы криптопамяти[1]

Применение микросхем криптопамяти существенно сокращает поверхность атаки на цифровое устройство. Но архитектурные ошибки механизмов защиты и их эксплуатация все еще возможны, особенно если микросхема обменивается симметричными ключами шифрования с микроконтроллером на каком-то этапе работы устройства.

Для примера рассмотрим технологию RPMB (Replay Protected Memory Block), появившуюся в спецификации eMMC4.41 в 2010 году (**https://www.jedec.org/standards-documents/docs/jesd84-a441**). Данная технология определяет отдельную область данных (раздел RPMB) объемом в несколько мегабайтов в обычной микросхеме eMMC. Доступ к разделу RPMB осуществляется с помощью специального защищенного протокола, использующего разделяемый симметричный ключ длиной 256 бит и алгоритм HMAC-256, обеспечивающие защиту передаваемых данных и подпись операций чтения/записи, предотвращающую replay-атаки. Пример транзакции записи в раздел RPMB микросхемы eMMC показан на рисунке:

[1] https://www.microchip.com/content/dam/mchp/documents/OTH/ProductDocuments/DataSheets/Atmel-2024S-CryptoMem-AT88SC0808C-Datasheet-Summary.pdf.

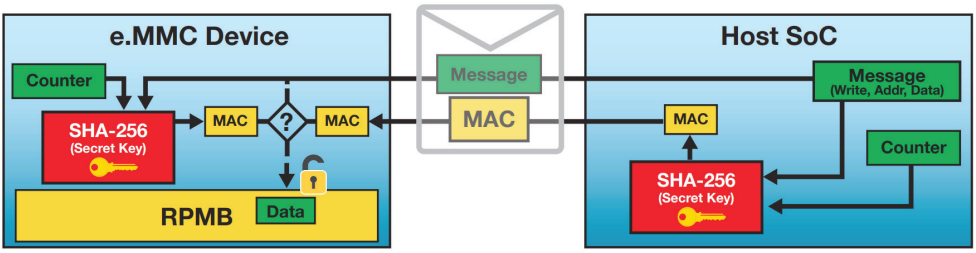


Рис. 4.10. Схема записи в раздел RPMB микросхемы eMMC[1]

В спецификации eMMC указано, что разделяемый симметричный ключ должен записываться в безопасном окружении, например при производстве устройства. Неправильное использование технологии RPMB (например, инициализация ключа при первом включении устройства пользователем) может позволить перехватить симметричный ключ и в дальнейшем получить доступ к данным, хранящимся в разделе RPMB.

Развитием идеи микросхем криптопамяти являются решения класса Secure Element, имеющие разные маркетинговые названия. В их составе есть полноценный криптопроцессор, реализующий широкий набор криптографических функций (шифрования и хеш-функций). Криптопроцессор исполняет собственную защищенную прошивку. Некоторые криптопроцессоры позволяют исполнять сторонний код (как правило, на базе встроенной виртуальной машины java) для расширения своих функциональных возможностей. Решения класса Secure Element позволяют организовать на устройстве аппаратный корень доверия. Одним из самых известных примеров применения микросхем типа secure element является TPM (Trusted Platform Module) в современных ноутбуках, структурная схема которых представлена на рисунке:

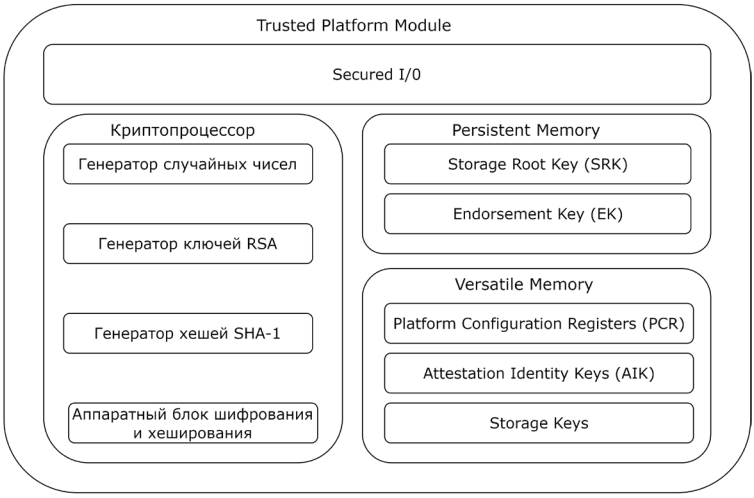


Рис. 4.11. Структурная схема модуля TPM

[1] https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/white-paper/white-paper-emmc-security.pdf.

Secure Element широко применяются в современных мобильных устройствах (смартфоны и планшеты), а также в устройствах, обрабатывающих критические данные, доступ к которым необходимо надежно защищать. Поэтому они часто становятся объектом интереса исследователей безопасности, подробнее о результате работы которых можно узнать в публикациях «Vulnerabilities in the TPM 2.0 reference implementation code» (**https://blog.quarkslab.com/vulnerabilities-in-the-tpm-20-reference-implementation-code.html**) и «FROM STOLEN LAPTOP TO INSIDE THE COMPANY NETWORK» (**https://dolosgroup.io/blog/2021/7/9/from-stolen-laptop-to-inside-the-company-network**).

Trusted Execution Environment

К технологиям класса Trusted Execution Environment (TEE) относятся еще более мощные, по сравнению с Secure Element, решения. Они бывают внешние, например микросхемы Apple T1/T2 или Google Titan, внутренние, расположенные на кристалле, такие как, например, Secure Enclave в SoC Apple, или «виртуальные», такие как ARM TrustZone.

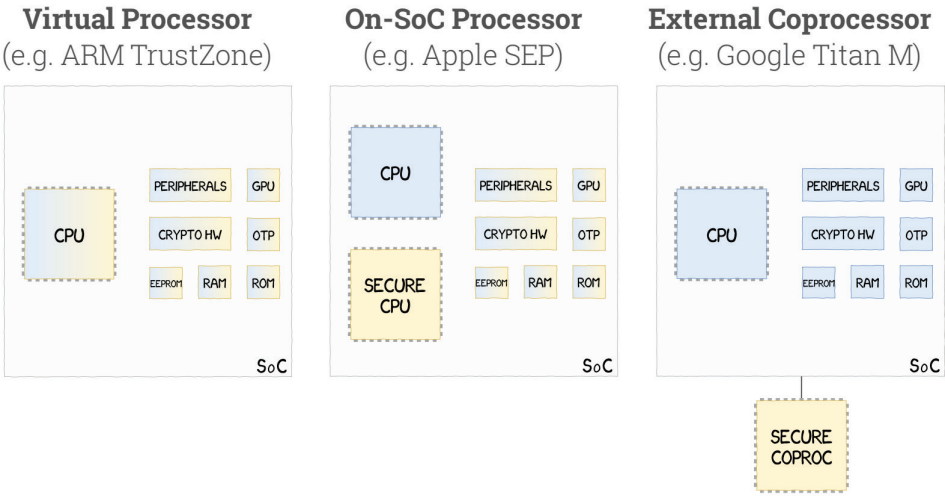


Рис. 4.12. Различные реализации TEE[1]

Рассмотрим подробнее официальное описание Apple SE (**https://support.apple.com/ru-ru/guide/security/sec59b0b31ff/web**) как пример хорошо построенной и описанной системы Secure Element см. рис. 4.13).

«Secure Enclave – это выделенная защищенная подсистема, встроенная в системы на кристалле Apple. Подсистема Secure Enclave изолирована от основного процессора для обеспечения дополнительного уровня защиты и предназначена для защиты конфиденциальных данных пользователя даже в случае взлома ядра процессора приложе-

[1] https://i.blackhat.com/USA-19/Thursday/us-19-Peterlin-Breaking-Samsungs-ARM-TrustZone.pdf.

ний. Эта подсистема построена по тому же принципу, что и вся система на кристалле: загрузочное ПЗУ задает аппаратный корень доверия, модуль AES отвечает за эффективные и безопасные криптографические операции, а память защищена. Несмотря на то что Secure Enclave не имеет собственного хранилища, Secure Enclave использует механизм безопасного хранения информации в подключенном хранилище отдельно от flash-памяти NAND, используемой процессором приложений и операционной системой. Начиная с SoC версий A11 и S4 сопроцессор Secure Enclave включает модуль защиты памяти и зашифрованную память с функциями антиповтора (anti-replay), безопасную загрузку, отдельный генератор случайных чисел и собственный модуль AES».

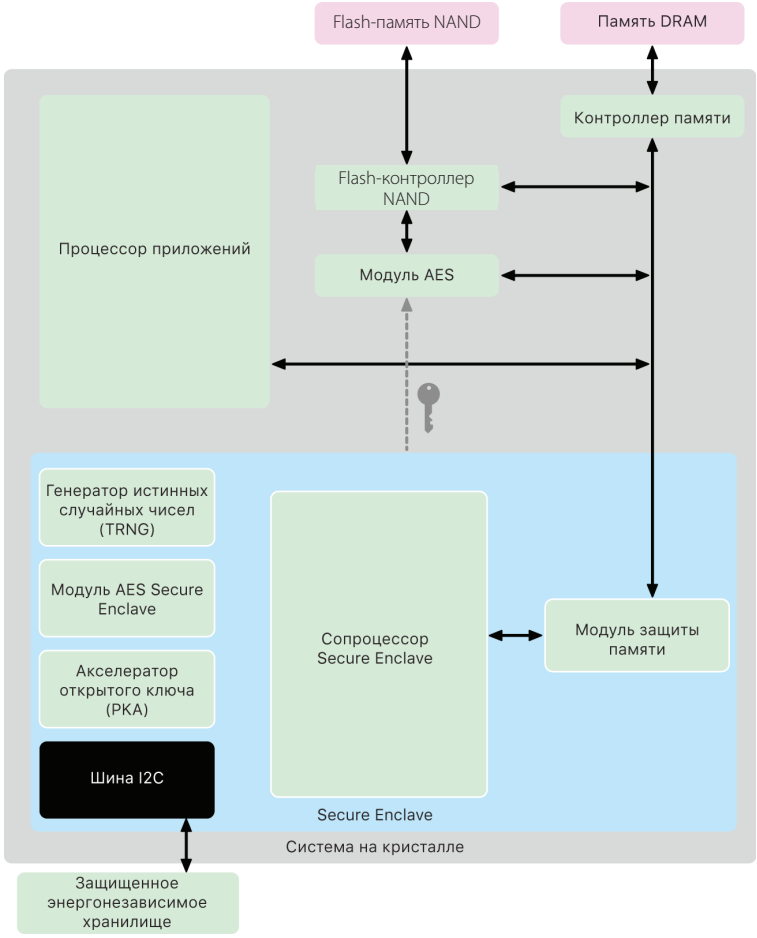


Рис. 4.13. Структурная схема Apple SE[1]

Все решения TEE объединяет одна основная идея – создание доверенной среды для выполнения кода. Рассмотрим ARM TrustZone в качестве примера реализации подобной технологии, широко применяющейся во встраиваемых системах. Фактически ARM TrustZone реализует два режима работы процес-

[1] https://support.apple.com/ru-ru/guide/security/sec59b0b31ff/web.

сора (Secure и Non-Secure) с поддержкой аппаратного контроля доступа через механизм Secure Monitor. Технология ARM TrustZone позволяет реализовать ряд мер (разделить память доверенных и недоверенных программ, разделить доступ к периферии) для создания надежного барьера между режимами Secure и Non-Secure.

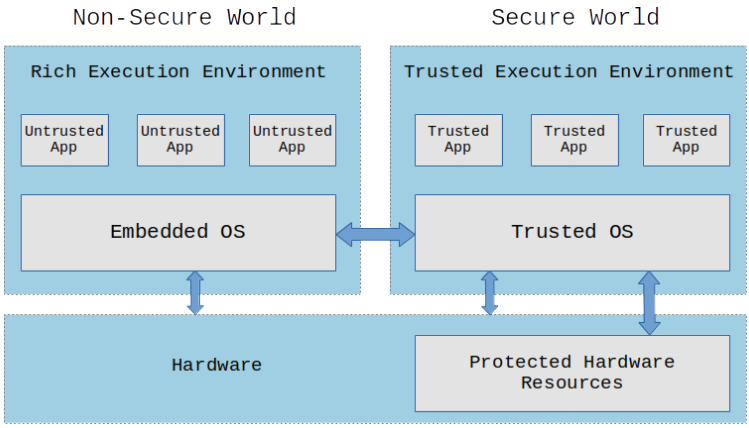


Рис. 4.14. Схема ARM TrustZone[1]

Типичное применение технологии ARM TrustZone предполагает запуск полноценной ОС в недоверенной среде выполнения кода (Non-Secure) и компактный, специализированный на безопасности код доверенной ОС (или функций ОС) в более безопасной среде (Secure). Такой подход позволяет контролировать использование различных данных и периферии на устройствах на базе ARM и должен предотвращать несанкционированный доступ к прошивке и устройству.

Переключение между мирами осуществляет Secure Monitor, а для обмена данными между мирами существует несколько механизмов, показанных на схемах:

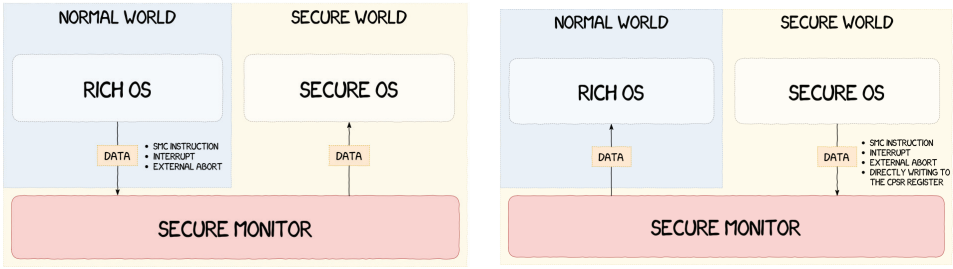


Рис. 4.15. Механизмы обмена данными между мирами ARM TrustZone[2]

Большинство ошибок реализации кроются в большой фрагментарности компонентов безопасности TEE, разрабатываемых разными фирмами: производителями микросхемы и ядра, разработчиками ОС и приложений.

[1] https://sergioprado.blog/introduction-to-trusted-execution-environment-tee-arm-trustzone/.

[2] https://i.blackhat.com/USA-19/Thursday/us-19-Peterlin-Breaking-Samsungs-ARM-TrustZone.pdf.

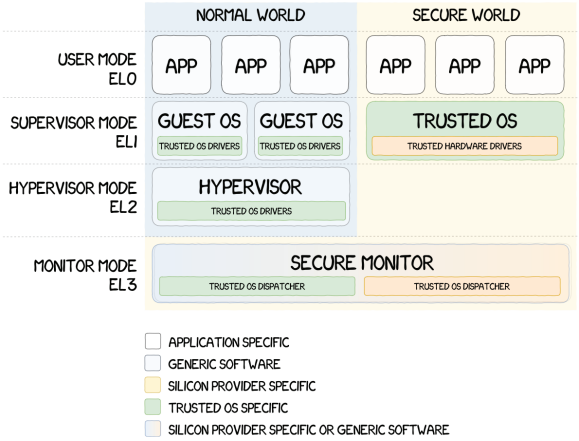


Рис. 4.16. Фрагментарность компонентов ARM TrustZone[1]

Далеко не всегда получается выстроить коммуникацию и документацию должным образом, чтобы каждый разработчик понимал свою зону ответственности при разработке безопасного решения. Поэтому для преодоления механизмов защиты устройств, использующих TEE, следует в первую очередь искать архитектурные ошибки в реализации механизмов защиты. Наиболее распространенными ошибками в реализации TEE являются ошибки в реализации обмена данными между мирами, а также разрыв цепочки доверенной загрузки на этапе Secure World за счет нахождения алгоритмических ошибок в реализации цепочки.

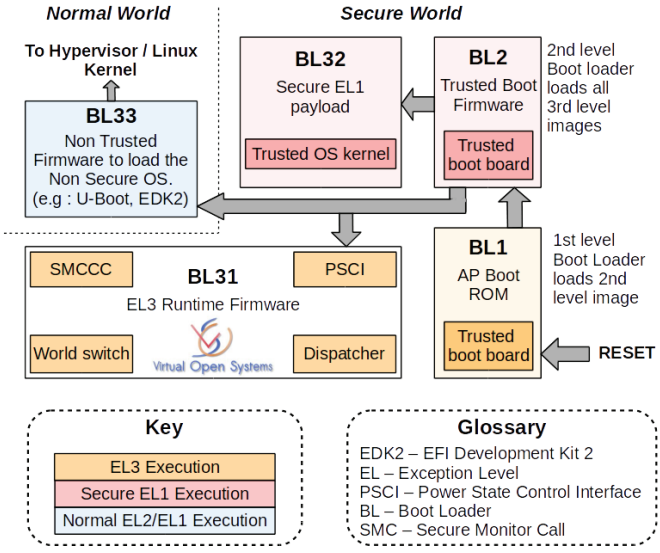


Рис. 4.17. Пример цепочки доверенной загрузки микроконтроллеров с архитектурой ARM[2]

[1] https://i.blackhat.com/USA-19/Thursday/us-19-Peterlin-Breaking-Samsungs-ARM-TrustZone.pdf.

[2] http://community.arm.com/docs/DOC-9306.

Нахождение уязвимостей в TEE является непростой и интересной задачей при исследовании современных встраиваемых систем. Тем не менее существует большое количество публикаций, показывающих возможность ее решения:

- Amlogic S905 SoC: bypassing the (not so) Secure Boot to dump the BootROM (**https://fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html**);

- Downgrade Attack on TrustZone (**https://ww2.cs.fsu.edu/~ychen/paper/downgradeTZ.pdf**);

- Attack Secure Boot of SEP (**https://github.com/windknown/presentations/blob/master/Attack_Secure_Boot_of_SEP.pdf**);

- Crouching T2, Hidden Danger (**https://ironpeak.be/blog/crouching-t2-hidden-danger/**);

- Unbox Your Phone (**https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c**);

- Bypassing Secure Boot using Fault Injection (**https://www.blackhat.com/docs/eu-16/materials/eu-16-Timmers-Bypassing-Secure-Boot-Using-Fault-Injection.pdf**);

- Breaking Samsung's ARM TrustZone (**https://www.blackhat.com/us-19/briefings/schedule/#breaking-samsungs-arm-trustzone-14932**);

- CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management (**https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang**).

SRAM PUF

Каждый экземпляр микросхемы микроконтроллера немного отличается друг от друга за счет несовершенства технологического процесса. Для работы микроконтроллера в штатном режиме эти изменения не критичны и никак не влияют на выполняемые функции. Эти отличия в характеристиках физических структур (например, транзисторов) можно использовать для создания уникального значения, присущего только конкретной микросхеме. Именно это свойство легло в основу технологии физически неклонируемых функций (Physically unclonable function) на основе состояния SRAM-памяти. Существуют и другие варианты построения PUF (не на основе SRAM), однако для примера мы остановимся именно на SRAM PUF. Принцип получения уникального стабильного отпечатка (пула данных) для микроконтроллера показан на рисунке.

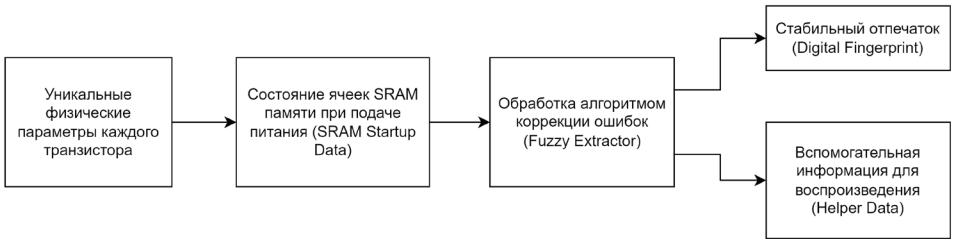


Рис. 4.18. Принцип получения уникального пула данных для SRAM PUF

Память SRAM – это энергозависимый тип памяти, то есть хранить информацию в ячейках памяти можно, только пока есть питание. Следовательно, при каждом включении устройства SRAM будет содержать неинициализированные данные. Эти первоначальные значения SRAM-ячеек (SRAM Startup Data) будут случайными и уникальными для каждой микросхемы ввиду того, что транзисторы, из которых изготовлены эти ячейки, отличаются физически. Так как значения в некоторых ячейках могут отличаться от включения к включению, а для работы функций SRAM PUF нужны стабильные данные, применяются коды коррекции ошибок. В результате работы данных кодов из нестабильных значений слепка SRAM памяти получается стабильное уникальное значение (отпечаток), характерное для каждого конкретного экземпляра микроконтроллера.

SRAM PUF может применяться для персонализации устройств, организации доверенной загрузки или защиты пользовательских ключей и данных. Конкретная реализация PUF (в том числе SRAM PUF) зависит от производителя микроконтроллера. Подробнее про SRAM PUF можно прочитать в статье «PUF for the Commons: Enhancing Embedded Security on the OS Level» (**https://arxiv.org/pdf/2301.07048.pdf**).

Как можно обходить механизмы защиты, построенные на базе PUF? Без специализированного и дорогостоящего оборудования, доступного только крупным лабораториям, можно надеяться лишь на архитектурные или логические ошибки реализации конкретной имплементации алгоритма защиты.

Watchdog

Один из моих любимых механизмов, с которым сталкиваются большинство исследователей, попытавшихся запустить свой код на устройстве и потерпевших поражение. При этом watchdog (он же – сторожевой таймер) ни в коем случае не является механизмом защиты от исследователей. Его основное назначение – вывести устройство из зависания с помощью перезагрузки микроконтроллера.

Срабатывание сторожевого таймера для исследователя, решившего написать патч для прошивки, выглядит как внезапная перезагрузка устройства без каких-либо причин. Код патча многократно перепроверяется на наличие потенциальных ошибок, способных вызвать исключение и перезагрузку

микроконтроллера, но ничего не помогает. Конечно, в основном это справедливо для bare metal прошивок и в случае, если написанный патч долгое время не возвращает управление в главный цикл (например, выводит дамп памяти в uart).

Watchdog-таймер реализован в большинстве представленных на рынке микроконтроллеров в виде отдельного аппаратного блока. Пример структурной схемы сторожевого таймера из состава микроконтроллера STM32F4 показан на рисунке.

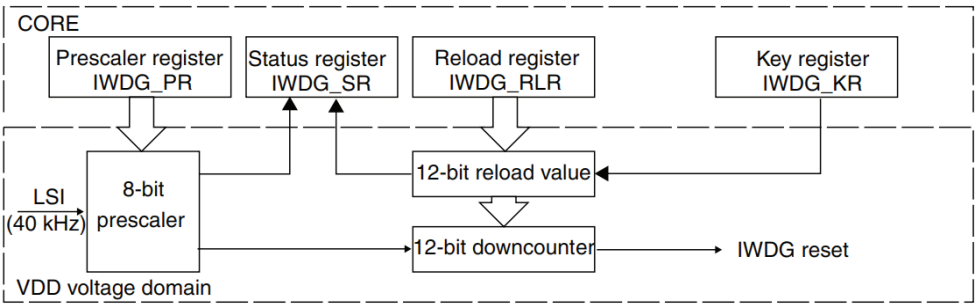


Рис. 4.19. Пример реализации Watchdog для микроконтроллеров семейства STM32F4[1]

Его принцип работы прост: в определенный регистр микроконтроллера (reload register) заносится начальное значение счетчика, которое в дальнейшем уменьшается с определенной частотой, настраиваемой с помощью делителя (из регистра prescale register). По достижении 0 генерируется сигнал, приводящий к перезагрузке микроконтроллера. Для управления таймером, в том числе сбросом значения счетчика в начальное состояние, существует регистр (key register), запись специальной константы в который переинициализирует счетчик.

Я описал самую простую вариацию сторожевого таймера. Существуют более сложные реализации с большим количеством настраиваемых параметров, позволяющих отлавливать зависания программы без перезагрузки всего микроконтроллера. Конкретную реализацию сторожевого таймера и возможные варианты его работы стоит изучить по документации производителя микроконтроллера.

Существуют полноценные внешние (по отношению к микроконтроллеру) схемы сторожевого таймера, как правило имеющиеся в устройствах, требующих повышенной надежности. Принцип работы подобных схем похож на блок сторожевого таймера в составе микроконтроллера – в их основе лежит микросхема таймера, генерирующая сигнал ресета для микроконтроллера, если за период работы таймера микроконтроллер одним из своих выводов не сбросил счетчик внутри микросхемы таймера. Если в ходе первичного анализа устройства вы обнаружите микросхему таймера, например LM555, стоит учесть вероятность реализации подобной схемы.

[1] https://www.st.com/resource/en/reference_manual/rm0368-stm32f401xbc-and-stm32f401xde-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

Level Up!

В этой главе мы рассмотрели основные механизмы защиты встраиваемых систем от модификации и как их можно нейтрализовать. Часть этих механизмов основана на архитектурных и физических особенностях реализации микроконтроллеров и устройств целиком. Несмотря на то что производители микроконтроллеров встраивают все больше технологий, служащих для повышения безопасности использования их решений, а разработчики становятся грамотнее и используют эти технологии и подходы безопасной разработки, исследователям встраиваемых систем все еще удается найти много уязвимостей. Надеюсь, после прочтения этой главы вы сможете написать работающий патч прошивки, даже при наличии механизмов защиты от исследования, узнать больше о логике работы устройства в динамике и в итоге получить полный контроль над устройством.

Well Done

Пора подводить итоги нашего погружения в процесс исследования встраиваемых систем. Надеюсь, исследователям стало немного понятнее, с чего можно начать процесс исследования устройства, а embedded-разработчикам – на что обратить внимание при проектировании устройства для его защиты. Возможно, кому-то эта книга даже поможет выбрать свой профессиональный путь или найти интересное хобби.

Если ранее вы не имели дело с исследованиями устройств, то вы еще в начале пути, и я вам немного завидую, ведь эмоции от первых успехов всегда одни из самых ярких. Предложенная методология представляет один из вариантов пути исследования электронных устройств, который показал свою эффективность более чем за 10 лет практической работы. Тезисно поговорим, что было рассказано в пяти главах книги.

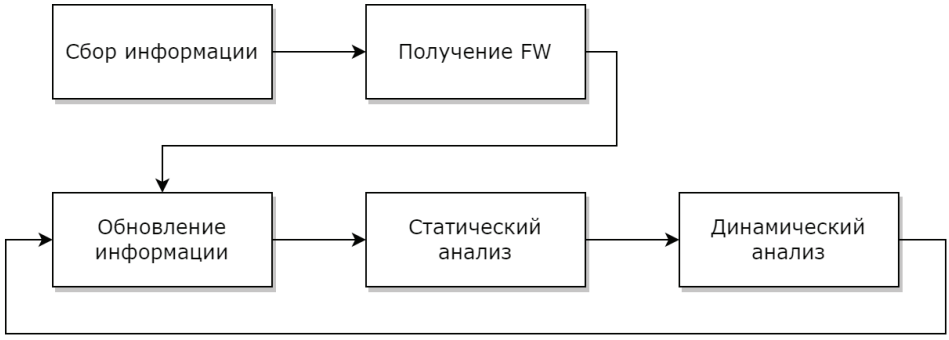


Рис. 316. Вариант пути проведения исследований встраиваемых систем

В главе *«Level 0. Первоначальный анализ»* мы рассмотрели:

- почему важно собирать всю доступную информацию об устройстве, как и где ее искать и почему эти данные помогут в дальнейшем исследовании;
- из чего состоит устройство, как правильно вскрыть его корпус и что такое печатная плата устройства, как производят устройства на заводах;
- какие ключевые компоненты лежат в основе большинства цифровых устройств, по каким интерфейсам они взаимодействуют между собой и с «внешним миром»;
- что такое структурная и функциональная схема устройства и зачем ее делать.

На выходе главы у нас сложилось понимание, из чего состоит наше устройство, какие информационные потоки в нем обрабатываются и какое оборудование нам нужно для проведения дальнейших исследований.

В главе «*Level 1. Добываем прошивку*» мы узнали про:

- распространенные методы получения прошивок из ПЗУ, обновлений или через отладочные интерфейсы;
- неинвазивные атаки и как они помогают в получении прошивки.

После прочтения этой главы стало понятно, как подступиться к процессу получения прошивки исследуемого устройства.

Получив прошивку, приступили к главе «*Level 2. Начинаем статический анализ*», в ней мы рассмотрели очень много тем:

- почему нужно разбираться в истории развития архитектур построения ЭВМ;
- какие бывают распространенные процессорные ядра;
- как выглядит процесс производства микроконтроллера;
- как определить архитектуру микроконтроллера исследуемого устройства;
- из чего состоит прошивка и как ее правильно загрузить в дизассемблер;
- что стоит искать в дизассемблере в первую очередь после загрузки прошивки;
- какие распространенные ОС используются во встраиваемых системах;
- как эмулировать код прошивки.

Очень большой объем информации для людей, ранее не имевших дела с исследованием или разработкой встраиваемых систем. В принципе, этой информации достаточно для проведения простых исследований. Но часто надо получить информацию от устройства в процессе его работы, т. е. в динамике, этому была посвящена глава «*Level 3. Настраиваем связь с внешним миром (динамический анализ)*». В ней мы рассмотрели:

- какую информацию можно получить с помощью динамического анализа;
- как собрать исследовательский стенд и автоматизировать рутинные действия;
- через какие распространенные интерфейсы (проводные и беспроводные) можно получить информацию от работающего устройства;
- какие нестандартные подходы к отладке и получению информации могут помочь, если ничто другое не сработало;
- как написать патч прошивки, позволяющий получить дополнительную информацию об устройстве.

В этой главе мы смогли получить требуемую информацию об устройстве, и теперь процесс исследования должен пойти гораздо быстрее. Или не должен, если нам помешали механизмы защиты, применяющиеся во встраиваемых системах. Им была посвящена глава «*Level 4. Механизмы защиты встраиваемых систем*». В ней мы рассмотрели:

- механизмы защиты встраиваемых систем от реверс-инжиниринга;
- какие аппаратные решения используют разработчики для защиты;
- в каких местах разработчики устройств чаще всего допускают ошибки при проектировании механизмов защиты.

На этом основная часть книги заканчивается, полученные знания должны помочь в исследовании встраиваемых систем. Информация, представленная в данной книге, показывает лишь часть подходов, применяющихся при исследовании устройств. Она должна натолкнуть вас на мысль, как можно расширить поверхность атаки за счет адаптации подходов или их рекомбинации. Не забывайте, что главное при проведении исследований – это накапливать опыт и расширять свой технический кругозор. Они должны подсказать вам правильный путь исследования именно вашего устройства.

Технологии развиваются очень быстро, и будут появляться новые способы защиты цифровых устройств и способы их компрометации. Не забывайте отслеживать технологические тренды, конференции и выставки, посвященные компьютерной безопасности и разработке встраиваемых систем. И исследователи, и разработчики, которые всегда в курсе новых трендов (возможно, еще даже не получивших воплощение в реальных технологиях), будут иметь преимущество и смогут решить самые сложные для своего времени технические задачи. Желаю успехов и тем, и другим, но «болею» за исследователей :)

В приложении приведен краткий обзор базового оборудования, необходимого для проведения исследований встраиваемых систем.

Приложение

Для проведения исследований цифровых устройств необходим набор базового оборудования, без которого будет сложно добиться результата. В этом разделе я приведу список необходимого и доступного большинству исследователей оборудования, которое можно использовать для проведения исследований как простых, так и более сложных цифровых устройств. Начнем с основы – организации рабочего места.

Рабочее место

Если есть возможность выделить для сборки/разборки и операций монтажа/демонтажа микросхем отдельный стол – это замечательно (особенно если столешница будет антистатической). Но такая возможность есть далеко не у всех. Поэтому исходим из того, что наше рабочее место – это наш обыкновенный рабочий стол, на котором нет ничего лишнего. Для защиты рабочего места при выполнении работ по разборке устройства (или вообще всегда) стоит использовать специальные прорезиненные самовосстанавливающиеся коврики. Они бывают разных размеров (рекомендую использовать размер А3 и больше) и легко находятся в любых онлайн-магазинах по словам «коврик для резки».

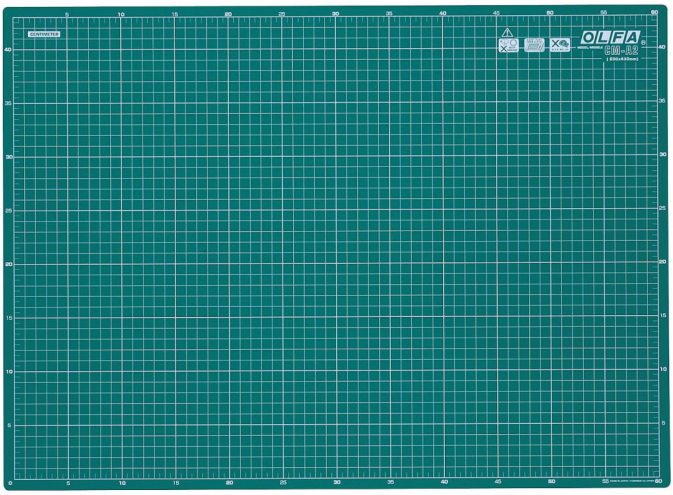


Рис. П.1. Защитный самовосстанавливающийся коврик для стола[1]

Для пайки потребуется взять силиконовый коврик, устойчивый к воздействию температуры. Его также легко купить в любом магазине, и если вы довольно аккуратны, можно выполнять все работы, включая сборку и разборку устройства, на нем.

[1] https://www.vseinstrumenti.ru/product/zaschitnyj-kovrik-olfa-a2-ol-cm-a2-1252811/.

Рис. П.2. Силиконовый коврик для пайки[1]

Обязательно стоит уделить внимание одежде, в которой вы будете проводить манипуляции с печатной платой устройства. Одежда из синтетики и шерсти существенно увеличивает риск повреждения электронных компонентов платы устройства статическим электричеством. Даже если в устройстве есть ESD-защита, она ставится только на линиях интерфейсных разъемов, т. е. тех, которые доступны при закрытом корпусе устройства, а мы собираемся его вскрывать. Поэтому лучше всего использовать обычную хлопковую одежду.

Полезные ручные инструменты

Для разборки корпуса устройства, демонтажа печатных плат, шлейфов из корпуса и других видов работ, которые приходится выполнять при исследовании устройств, необходимы различные виды ручных инструментов:

- набор отверток;
- набор пластиковых инструментов для разбора корпуса;
- пинцеты;
- скальпель/лопатки.

Кратко опишу, на какие особенности стоит обратить внимание при выборе подобного инструмента.

В наборе отверток обязательно должно быть хорошее разнообразие сменных насадок. Некоторые производители стараются затруднить разборку своего устройства и используют нераспространенные головки винтов для сборки корпуса. Еще один фактор, на который следует обратить внимание при выборе набора отверток, – толщина ручки и длина бит. Корпуса некоторых устройств имеют весьма глубокие каналы, в которых находятся крепежные винты. Набор отверток со стандартной длиной бит в 2–3 см и толстым корпусом ручки иногда не позволяет открутить глубоко посаженные винты. На фото далее показан набор отверток с удлиненными битами, лишенными данного недостатка.

[1] https://www.vseinstrumenti.ru/product/kovrik-dlya-pajki-si28-30h45-sm-deko-065-0418-5040612/.

Рис. П.3. Набор отверток с удлиненными битами[1]

Чтобы не повредить корпус устройства при его разборке, стоит использовать специальные наборы инструментов, выполненные из мягкой пластмассы. Их можно найти по запросу «инструмент для разбора телефонов».

Рис. П.4. Инструмент для вскрытия корпусов[2]

Также не советую выбрасывать старые пластиковые карты и гитарные медиаторы – они могут быть весьма полезными при вскрытии глубоко расположенных защелок корпуса.

Для точных работ (вскрытия защелок разъемов гибких шлейфов, установки элементов на печатную плату и т. п.) необходимо иметь набор пинцетов разных форм. При выборе учитывайте, что рано или поздно пинцет будет подвергнут нагреванию, поэтому вместо пластиковых лучше брать пинцеты, выполненные из нержавеющей стали, покрытые специальным составом с антистатическими свойствами (ESD-safe).

[1] https://nanchtools.com/products/22-in-1-screwdriver-repair-set-by-nanch.

[2] https://www.ifixit.com/products/prying-and-opening-tool-assortment.

Рис. П.5. Набор пинцетов с антистатическим покрытием[1]

Иногда при разборке корпуса устройства стоит задача точно надрезать какую-то защелку, подрезать клей или компаунд, на котором могут быть закреплены элементы корпуса либо антенны. Для этих задач лучше всего воспользоваться скальпелем со сменными лезвиями. Также он отлично помогает, если надо перерезать дорожку или зачистить часть маски на плате устройства.

Рис. П.6. Макетный скальпель со сменными лезвиями[2]

Отличной идеей будет купить набор металлических лопаток, который может пригодиться для вскрытия корпуса или выполнения каких-то работ, где пластиковый инструмент не подходит.

Рис. П.7. Набор металлических лопаток[3]

1 https://www.ifixit.com/products/precision-tweezers-set.
2 https://www.vseinstrumenti.ru/product/nozh-skalpel-kantselyarskij-kwb-6-lezvij-14920-1518013/.
3 https://www.ifixit.com/products/metal-spudger-set.

Вместо специализированного набора лопаток можно купить набор нержавеющих медицинских стоматологических инструментов. В нем содержатся различные крюки, лопатки и т. п. инструменты, иногда существенно упрощающие работу по разборке устройства.

Для фиксации печатной платы, пайки разъемов и других операций, где нужно что-то придерживать, удобно использовать разнообразные держатели (также называемые «третья рука»). Пример показан на фото ниже:

Рис. П.8. Инструмент «третья рука»[1]

Для фиксации именно печатных плат существуют специальные держатели, их конструктив очень разнообразен, они могут быть пружинные, магнитные, зажимные и т. п. Внешний вид одной из самых распространенных конструкций показан на фото:

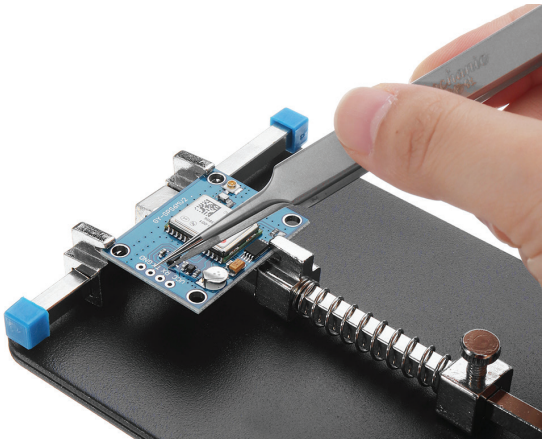


Рис. П.9. Держатель печатной платы[2]

[1] https://www.ifixit.com/products/helping-hands.

[2] https://aliexpress.ru/item/1005003322812018.html.

При выборе стоит ориентироваться на размер печатной платы устройства и отзывы реальных пользователей, лучше всего с профильных форумов по ремонту электроники.

Микроскоп

Современные цифровые устройства представляют из себя печатную плату с поверхностным монтажом элементов весьма малого размера, порой всего пару миллиметров. Без микроскопа сложно даже провести первичный анализ устройства, не говоря о выполнении каких-то операций, связанных с модификацией печатной платы устройства. Микроскопы бывают чисто оптические или оптико-электронные, вторые, в свою очередь, могут подключаться к ПК или иметь собственный монитор. Дешевые электронные микроскопы могут иметь едва различимую задержку вывода изображения, которая раздражает некоторых людей. Оптические микроскопы лишены данной проблемы, но стоят гораздо дороже, и далеко не каждый имеет возможность установки камеры (для получения изображения или фотографии просматриваемого объекта). Хорошим вариантом для базового набора будет цифровой микроскоп с поддержкой FullHD или б/у оптический микроскоп.

Рис. П.10. Цифровой микроскоп[1]

Мультиметр

Наверняка у вас уже есть какой-то мультиметр. Главное, чтобы в нем работал режим «прозвонки» и измерение сопротивлений (хотя бы с точностью 5 %). Для начала этого будет достаточно. Если мультиметра еще нет – выбирайте модель, исходя из отзывов на форумах ремонтников аппаратуры и своего бюджета. Есть возможность взять хороший инструмент – выбирайте из ассортимента известных фирм. В любом случае не забудьте купить хорошие провода с тонкими щупами. Только не берите универсальные, со сменными щупами, в них почти никогда не бывает стабильного контакта.

[1] https://aliexpress.ru/item/1005001572737660.html.

Рис. П.11. Щупы для мультиметра с тонкими жалами[1]

Еще два важных момента, на которые стоит обратить внимание при выборе мультиметра, – это минимальное время задержки появления звукового сигнала и напряжение в режиме прозвонки электрических цепей. Некоторые мультиметры имеют задержку в 0,5–1 с от момента замыкания цепи щупами до появления звукового сигнала, что мешает при анализе схемы устройства. Дешевые мультиметры могут проверять схему напряжением 3.3 В, которое может вывести из строя некоторые низковольтные компоненты. Хорошие мультиметры имеют напряжение прозвонки 1 В и даже менее.

Отладчики

Самое желанное, что хочет найти любой исследователь устройства при вскрытии его корпуса, – это разъемы отладочных интерфейсов. Следовательно, нужны специальные устройства – отладчики, но сначала нужно определить распиновку отладочного разъема. Для этого существует JTAGulator. Цена на него немного «кусается», но можно собрать аналогичный «переборщик» самому, вопрос в затраченном времени и его стоимости. Хорошие open source проекты, с которых можно начать разработку своего переборщика, – это **https://github.com/szymonh/JTAGscan** и **https://github.com/szymonh/SWDscan**.

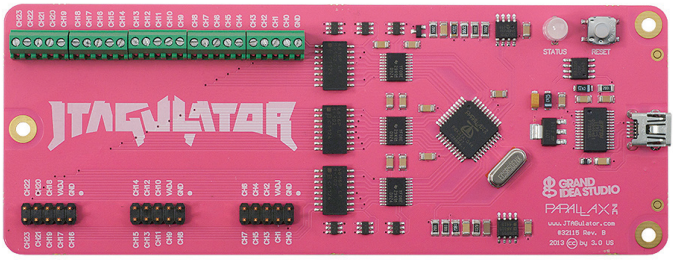


Рис. П.12. Устройство поиска контактов JTAG – JTAGulator[2]

[1] https://www.ozon.ru/product/shchup-tokoizmeritelnyy-sam-soberu-1000-v-20-a-301213333/.

[2] http://www.grandideastudio.com/jtagulator/.

Для проведения базовых исследований достаточно универсального отладчика J-Link фирмы Segger (у него есть огромное количество клонов, доступных для заказа на распространенных маркетплейсах).

Рис. П.13. Отладчик J-link[1]

Также стоит приобрести одну из распространенных плат на основе чипа FTDI 2232, позволяющую эмулировать множество отладочных интерфейсов, включая JTAG и SWD.

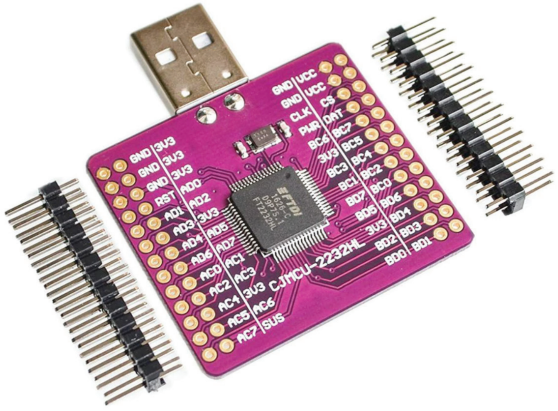


Рис. П.14. Отладчик на базе микросхемы FTDI 2232[2]

Оба решения отлично работают с проектом OpenOCD, поддерживают отладку и считывание ПЗУ большинства встречающихся в цифровых устройствах микроконтроллеров.

Программаторы микросхем памяти

Для считывания и перепрограммирования микросхем памяти необходим программатор, и чем большее количество микросхем он поддерживает, тем лучше. Цены на программаторы могут варьироваться в очень широких пределах – от

[1] https://www.segger.com/products/debug-probes/j-link/.

[2] https://aliexpress.ru/item/1005003070594525.html.

нескольких сотен рублей до сотен тысяч. Отличным вариантом, достаточным для большинства исследований, будет программатор XGecu T56 с расширенным набором адаптеров. Обращайте внимание на комплект поставки, их существует несколько видов. Лучше всего сразу взять комплект с максимально широким списком адаптеров для BGA.

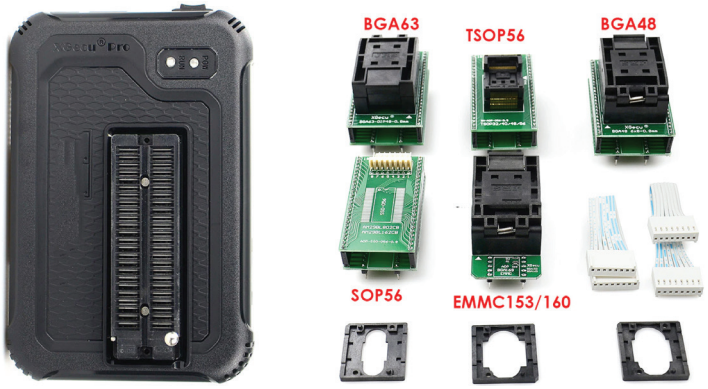


Рис. П.15. Внешний вид программатора XGecu T56 с набором адаптеров для считывания различных типов микросхем[1]

Логический анализатор

Один из самых полезных при исследовании устройств инструментов. При выборе логического анализатора надо ориентироваться на максимальную частоту захвата сигнала и размер буфера. Стабильно вы сможете захватить сигнал с частотой в 4 раза меньше максимально поддерживаемой. Если в ТТХ логического анализатора указана максимальная частота 400 МГц, не стоит рассчитывать на захват сигнала более 100 Мгц. Для большинства базовых исследований цифровых устройств подойдет логический анализатор DSlogic Plus, как раз имеющий частоту захвата 400 МГц, 16 каналов и большой список поддерживаемых декодеров протоколов.

Рис. П.16. Логический анализатор начального уровня DSLogic Plus[2]

[1] http://xgecu.com/.

[2] https://aliexpress.ru/item/4000386257930.html.

Если есть возможность, можно купить сразу более старшую версию DSLogic U3Pro16, поддерживающую частоту захвата в 1 ГГц. У этой модели есть версия DSLogic U3Pro32 с 32 каналами захвата, но такое количество каналов нужно редко. Альтернативой устройствам DSLogic могуть быть анализаторы Saleae Logic Pro, умеющие также работать в режиме осциллографа.

Осциллограф

Не менее важный, чем логический анализатор, инструмент для анализа цифровых устройств. Как и в случае с логическими анализаторами, главными параметрами являются частота захвата сигнала, полоса пропускания и размер буфера. И снова оптимальным выбором для большинства исследователей будет USB-осциллограф фирмы DreamSource Lab – DSCope U3P100. Он позволяет анализировать сигналы с частотой до 100 МГц, что является достаточным для анализа почти всех сигналов, встречающихся в простых устройствах.

Рис. П.17. Осциллограф начального уровня DSCope U3P100[1]

Для анализа скоростных сигналов такой частоты будет явно недостаточно, но осциллографы, поддерживающие частоты в несколько сотен мегагерц или даже гигагерц, стоят совсем других денег. И осциллографы, и логические анализаторы могут быть выполнены в виде самостоятельного устройства с собственным экраном и органами управления. Такие устройства могут быть удобнее в использовании, но стоят значительно дороже. Кстати, подобные модели – это тоже цифровое устройство. Для ряда моделей можно превратить младшую модель в более старшую, незначительно модифицировав прошивку (**https://www.made2hack.com/how-to-hack-a-rigol-ds1054z-digital-oscilloscope/**).

Конвертеры интерфейсов (USB 2 everything)

В базовом наборе исследователя цифровых устройств обязательно должны быть конвертеры, как минимум следующих интерфейсов:

- USB в U(S)ART (несколько экземпляров), для 1.8 В (в идеале также для 1.2 В), 3.3 В и 5 В;
- USB в SPI;

[1] https://www.dreamsourcelab.com/shop/oscilloscope/dscope-u3p100/.

- USB в GPIO, для 1.8 В, 3.3 В и 5 В;
- USB в I2C.

Часть подобных переходников легко собрать на базе Arduino или любой другой отладочной платы. Можно использовать платы на базе универсального чипа-конвертера FTDI FT2232 (они существует во множестве различных вариантов), или купить «мультитул» для работы с различными интерфейсами (JTAG, SWD, SPI, I2C и т. п.) Bus Pirate (показанный на рисунке), или использовать для этих целей Flipper Zero.

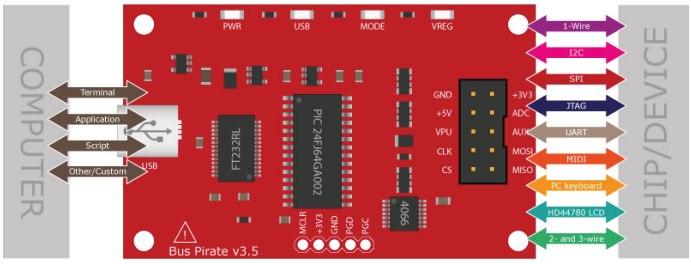


Рис. П.18. Схематичный внешний вид BusPirate со списком поддерживаемых интерфейсов[1]

Отладочные платы

Набор отладочных плат позволит решить большинство задач, возникающих при исследовании устройств, например:

- сделать нестандартный конвертер интерфейсов;
- реализовать атаку на JTAG;
- сделать MITM-атаку на интерфейс;
- реализовать подборщик JTAG и т. д.

В базовый набор обязательно должны быть включены несколько отладочных плат:

1) из семейства Arduino;

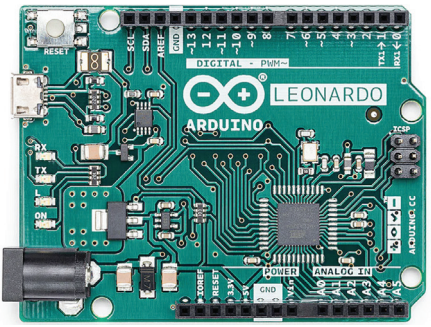


Рис. П.19. Плата Arduino Leonardo[2]

[1] https://www.seeedstudio.com/Bus-Pirate-v3-6-universal-serial-interface-p-609.html.
[2] https://store.arduino.cc/products/arduino-leonardo-with-headers.

2) Raspberry Pi;

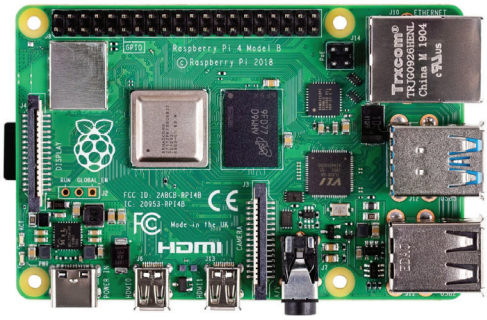


Рис. П.20. Плата Raspberry Pi4[1]

3) что-нибудь на базе STM32Fx;

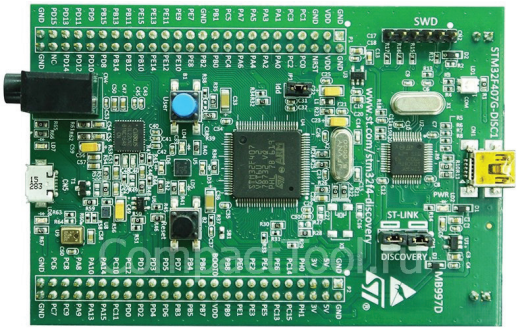


Рис. П.21. Плата STM32F407 Discovery[2]

4) на базе FPGA начального уровня (Xilinx Artix-7, Spartan-7 или Intel Cyclone 5).

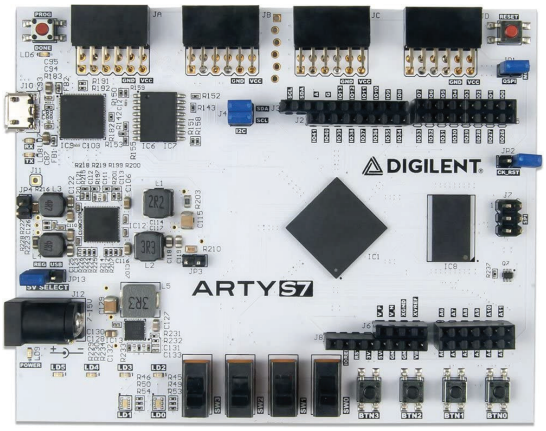


Рис. П.22. Плата с FPGA Arty S7[3]

[1] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[2] https://compacttool.ru/otladochnaya-plata-stm32f4discovery.

[3] https://digilent.com/reference/programmable-logic/arty-s7/.

Задач, которые надо сделать «здесь и сейчас» при исследовании устройств, много. Большинство из них успешно решаются (или уже решены) с помощью отладочных плат.

SDR

Полноценные модели SDR (например, BladeRF или HackRF) весьма недешевы. Если есть возможность и планируете исследовать устройства с разнообразными беспроводными интерфейсами, однозначно стоит присмотреть модель, позволяющую, кроме приема, передавать данные. В базовый набор я рекомендую включить самую простую версию SDR (выполненную на базе чипа TV-приемника), позволяющую только просматривать радиоэфир. Характеристики такого SDR-приемника уступают полноценным SDR-радио, но он послужит отличным инструментом для изучения основ радиопередачи и формирования требований к полноценному инструменту.

Рис. П.23. RTL-радио[1]

Все для пайки

В этом разделе я не буду описывать продвинутое оборудование, с помощью которого можно осуществлять сложные операции по монтажу-демонтажу SoC и т. п. Рассмотрим необходимые инструменты для выполнения базовых операций, таких как пайка проводников, выводных микросхем малого размера или BGA eMMC.

Первым делом нужна паяльная станция. Так как паять при исследовании устройств придется довольно часто в любом случае, крайне не советую экономить на паяльной станции. Конечно, иногда паять можно и обычным паяльником, но скорее это будет похоже на мучение и удовольствия от процесса вы точно не получите. Лучше всего взять качественную индукционную паяльную станцию хорошего китайского бренда, например фирмы Quick (но если есть возможность – лучше брать JBC или Metcal).

[1] https://www.rtl-sdr.com/buy-rtl-sdr-dvb-t-dongles/.

Рис. П.24. Паяльная станция[1]

Для выпаивания микросхем обязательно нужна термовоздушная паяльная станция («фен»), главное, чтобы она стабильно держала температуру выдуваемого воздуха. Опять же, рекомендую начать с моделей фирмы Quick.

Рис. П.25. Термовоздушная паяльная станция[2]

Место, где будет происходить пайка, должно быть хорошо проветриваемым. Если паять придется часто и в помещении отсутствует нормальная вентиляция – присмотритесь к автономным вытяжкам для пайки. Скорее всего, вам хватит компактного настольного варианта, хотя существуют и промышленные варианты, качественно удаляющие дым.

[1] https://technica-m.ru/catalog/articul/payalnaya-stanciya-quick-203h-lead-free.

[2] https://technica-m.ru/catalog/articul/payalnaya-stanciya-termovozdushnaya-quick-2008.

Минимальный набор необходимых расходников для пайки:

- припой (дополнительно можно взять паяльную пасту);
- флюс;
- медная оплетка для удаления припоя;
- жидкость для удаления флюса, лучше в баллончике;
- спирт (с флаконом-дозатором);
- ватные палочки, диски и салфетки;
- силиконовые провода нескольких цветов;
- провода типа ПЭЛВ и МГТФ;
- термоусадка;
- контактные гребенки на 1.27 мм, 2 мм и 2.54 мм.

Используемое для исследований ПО

В тексте книги уже встречались упоминания ПО, используемого при проведении исследований встраиваемых систем. В данном разделе я приведу список ПО (ориентируясь на ОС Windows), использующегося при проведении исследований устройств, но не включающего в себя ПО вендора для управления какими-то инструментами и оборудованием:

- дизассемблер (IDA Pro или Ghidra). Думаю, тут комментарии не нужны;
- HEX-редактор с поддержкой плагинов и опцией подсчета контрольных сумм большим количеством алгоритмов, например 010 Editor;
- Pulse View (**https://sigrok.org/wiki/PulseView**) – open source ПО для просмотра диаграмм, собранных с осциллографа, логического анализатора и т. п. Поддерживает разные форматы входных данных. Может быть полезно при собственной реализации сниффера какого-нибудь интерфейса;
- Wireshark (**https://www.wireshark.org/**) – open source ПО для анализа протоколов и различного трафика (сетевого, USB и т. д.);
- OpenOCD (**https://openocd.org/**) + GDB (**https://www.sourceware.org/gdb/**) – open source ПО управления различными отладчиками;
- binwalk (**https://github.com/ReFirmLabs/binwalk**) + плагин cpu_req (**https://github.com/airbus-seclab/cpu_rec**) – open source ПО для анализа структуры двоичных файлов и определения процессорных архитектур;
- QEMU (**https://www.qemu.org/**) – open source эмулятор, поддерживающий большой набор различных архитектур;
- Unicorn Engine (**https://www.unicorn-engine.org**) – open source эмулятор, основанный на QEMU, позволяющий проанализировать отдельные ветви подпрограммы. Есть плагины для популярных дизассемблеров;
- Frida (**https://frida.re/**) – open source набор утилит для динамической инструментации кода, особенно полезен при исследовании приложений для Android и iOS;
- PlatformIO (**https://platformio.org/**) – open source платформа (фреймфорк, IDE, Library Manager и т. д.) для разработки устройств. Может выступать заменой распространенных IDE (в том числе Arduino IDE) для быстрого прототипирования или создания отладочной оснастки на базе devbord'ов;
- HTerm (**https://www.der-hammer.info/pages/terminal.html**) или Terminal v1.93b by Br@y++ (**https://www.sites.google.com/site/terminalbpp/**) – ПО терминала последовательного порта с возможностью быстро менять параметры подключения.

Плагины для дизассемблера (IDA Pro):

- FirmLoader (**https://github.com/Accenture/FirmLoader**) – разметка адресного пространства для всех периферийных блоков микроконтроллера с помощью сегментов;
- findcrypt (**https://github.com/polymorf/findcrypt-yara**) – поиск криптографических (и не только) констант.

Список использованных источников

1. https://arxiv.org/ftp/arxiv/papers/1609/1609.04327.pdf.
2. https://cfp.recon.cx/2023/talk/HCJHBW/.
3. https://commons.wikimedia.org/w/index.php?curid=4809738 (released by Everaldo Coelho (https://en.wikipedia.org/wiki/User:Everaldo_Coelho) and YellowIcon (http://www.yellowicon.com/) under the LGPL license).
4. https://cyphunk.files.wordpress.com/2010/02/blackbox-jtag-reverse-engineering-tmbinc.pdf.
5. https://developer.arm.com.
6. https://en.wikipedia.org/wiki/Complex_instruction_set_computer.
7. https://en.wikipedia.org/wiki/Harvard_architecture.
8. https://en.wikipedia.org/wiki/MIPS_architecture.
9. https://en.wikipedia.org/wiki/Reduced_instruction_set_computer.
10. https://en.wikipedia.org/wiki/Von_Neumann_architecture.
11. https://fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html.
12. https://freertos.org/.
13. https://github.com/Accenture/FirmLoader.
14. https://github.com/airbus-seclab/cpu_rec.
15. https://github.com/jxwleong/jtag-boundary-scan.
16. https://github.com/polymorf/findcrypt-yara.
17. https://github.com/ReFirmLabs/binwalk
18. https://github.com/windknown/presentations/blob/master/Attack_Secure_Boot_of_SEP.pdf.
19. https://habr.com/ru/company/macloud/blog/555730/.
20. https://habr.com/ru/post/190012/.
21. https://habr.com/ru/post/190012/.
22. https://ironpeak.be/blog/crouching-t2-hidden-danger/.
23. https://learn.microsoft.com.
24. https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c.
25. https://recon.cx/2012/schedule/events/236.en.html.
26. https://ru.bmstu.wiki/Synopsys_ARC.
27. https://semiwiki.com/eda/307494-the-semiconductor-ecosystem-explained/.
28. https://thunderclap.io/thunderclap-paper-ndss2019.pdf.

29. http://wiki.markodelgroup.ru/lib/exe/fetch.php?media=тарасов_южанин.pdf
30. https://ww2.cs.fsu.edu/~ychen/paper/downgradeTZ.pdf.
31. https://www.aisec.fraunhofer.de/content/dam/aisec/ResearchExcellence/woot17-paper-obermaier.pdf.
32. https://www.aisec.fraunhofer.de/en/FirmwareProtection.html.
33. https://www.altium.com.
34. https://www.analog.com.
35. https://www.beyondlogic.org/usbnutshell/usb1.shtml.
36. https://www.blackhat.com/docs/eu-16/materials/eu-16-Timmers-Bypassing-Secure-Boot-Using-Fault-Injection.pdf.
37. https://www.blackhat.com/eu-17/briefings/schedule/index.html#intel-me-flash-file-system-explained-8484.
38. https://www.blackhat.com/us-19/briefings/schedule/#breaking-samsungs-arm-trustzone-14932.
39. https://www.cl.cam.ac.uk/~sps32/cardis2016_sem.pdf.
40. https://www.corelis.com.
41. https://www.cs.tau.ac.il/~tromer/mobilesc/mobilesc.pdf.
42. https://www.dediprog.com.
43. https://www.esat.kuleuven.be/cosic/publications/thesis-182.pdf.
44. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.
45. https://www.fpga4fun.com/JTAG1.html.
46. https://www.greynoise.io/blog/debugging-d-link-emulating-firmware-and-hacking-hardware.
47. https://www.gushiciku.cn/pl/pKYq.
48. https://www.jedec.org.
49. https://www.jedge.com/code/ESCAM_QF100_IP_camera_boot_log.txt.
50. https://www.keil.com.
51. https://www.langer-emv.com.
52. https://www.mdpi.com/2076-3417/9/15/3110.
53. https://www.mdpi.com/2410-387X/4/2/15#B28-cryptography-04-00015.
54. https://www.microchip.com.
55. https://www.nexustechnology.com.
56. https://www.nexustechnology.com.
57. https://www.nuand.com.
58. https://www.nxp.com.
59. https://www.olimex.com.
60. https://www.onfi.org.
61. https://www.researchgate.net/publication/311490102_Body_Biasing_

Injection_Attacks_in_Practice.
62. https://www.riscure.com.
63. http://www.robot.bmstu.ru/files/GOST/gost_2.710-81.pdf.
64. https://www.st.com.
65. https://www.thegoodpenguin.co.uk/blog/pcie-dma-attack-against-a-secured-jetson-nano-cve-2022-21819/.
66. https://www.trustedfirmware.org/projects/mcuboot/index.html.
67. https://www.usb.org.
68. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.
69. https://www.usenix.org/legacy/event/sec08/tech/full_papers/halderman/halderman.pdf.
70. https://www.xilinx.com/products/boards-and-kits/arty.html.
71. www.denx.de/wiki/U-Boot/.
72. www.feaser.com/openblt/doku.php.
73. www.ghs.com/products/rtos/integrity.html.
74. www.newae.com.
75. www.semiwiki.com/eda/307494-the-semiconductor-ecosystem-explained/.
76. www.windriver.com/products/vxworks.
77. www.zephyrproject.org.
78. *Питерсон У., Уэлдон Э.,* Коды, исправляющие ошибки / пер. с англ. под ред. Р. Л. Добрушина и С. И. Самойленко. – М.: Мир, 1976.
79. *Харрис Д. М., Харрис С. Л.*, Цифровая схемотехника и архитектура компьютера. – М.: ДМК Пресс, 2022.

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com.

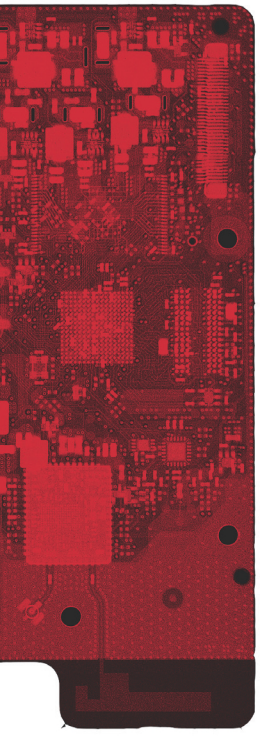
Усанов Алексей Евгеньевич

Реверс-инжиниринг встраиваемых систем

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 24,05.
Тираж 200 экз.

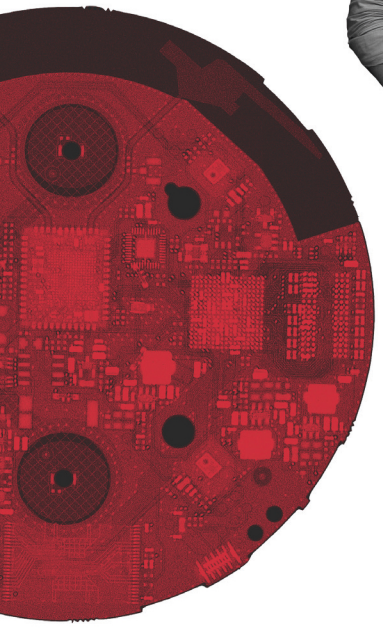
Веб-сайт издательства: **www.dmkpress.com**



Книга поможет найти ответы на вопросы:

- Как проводить исследование встраиваемых систем?
- Из каких компонентов они состоят?
- Как получить прошивку устройства и какие уязвимости могут в этом помочь?
- Чем отличается реверс-инжиниринг прошивок и ПО?
- Что нужно для динамического анализа устройства?
- Как обходить защиту встраиваемых систем от исследования?

Алексей Усанов – руководитель направления исследований безопасности аппаратных решений в компании Positive Technologies. Работал преподавателем на кафедре Информационная безопасность МГТУ им. Н. Э. Баумана. Более 15 лет занимается исследованиями встраиваемых систем и системной разработкой.



Работал с различными классами устройств: от бытовой техники и устройств потребительской электроники до промышленных систем передачи данных и контроля технологических процессов. Накопленный опыт в сфере кибербезопасности, желание передать его начинающим исследователям и разработчикам стали драйвером к написанию книги, которую вы держите в руках.

ISBN 978-5-93700-231-0