Залата Руслан Николаевич
Генеральный директор
ООО «Фабмикро»
г. Тюмень
февраль 2022г.
rz@fabmicro.ru

Разработка процессорного модуля на базе микросхемы 1892ВА018 СнК «СКИФ» и его первый запуск

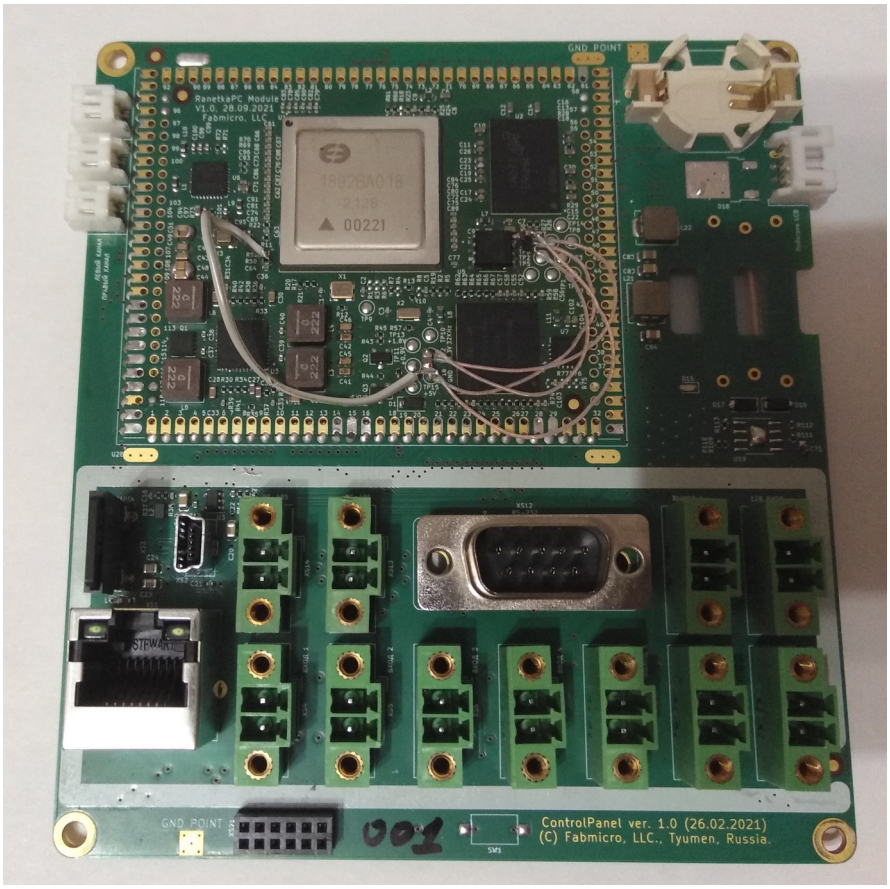

Рис.1. Процессорный модуль «RanetkaPC» на базе СнК «СКИФ» смонтирован на несущей плате изделия «ControlPanel». Инженерный образец. ООО «Фабмикро», февраль 2022.

ОГЛАВЛЕНИЕ

1. Вступление

В первой половине 2021 года в РФ резко обострились дебаты на тему импортозамещения, инициированные рядом изменений внесенных ранее в декабре 2020г в постановление N719 относительно закупок вычислительной техники, и бурно обсуждаемых на различных площадках и IT-форумах, таких как Elbrus Tech Day, YADRO Лекторий, а так же у ряда популярных блогеров. Основной посыл нововведений состоял в том, что вся вычислительная техника, закупаемая за бюджетные средства, должна иметь в основе микропроцессор, разработанный или произведенный в России или имеющий статус такового. Насколько такое решение Правительства годное и исполнимое — тема отдельного разговора, которую я не хотел бы сейчас затрагивать. Так или иначе, мы, в нашей небольшой компании, пришли к выводу, что данное решение Правительства может приоткрыть пусть не окно, но хотя бы форточку возможностей для мелких разработчиков электроники, которым является наша компания. Имея за плечами опыт разработки изделий на основе импортных (в основном китайских) систем-на-кристалле, мы решили, что на рынке могут быть востребованы одноплатные ПК, аналогичные Raspberry Pi и выполненные на отечественном СнК, по возможности близкие по габаритам и pin-to-pin совместимые с «малиной», а также процессорные модули, устанавливаемые на несущую плату, позволяющие упростить разработку конечного изделия и вынести сложную топологию за его границы.

В данном случае процесс разработки мы решили начать именно с процессорного модуля, так как у нас уже имелся ряд изделий, построенных по модульному принципу, в которых процессорный модуль на отечественном СнК может быть применен с пользой. По мимо этого, разработать и отладить модуль значительно легче, чем конечное изделие, а затраты на изготовление печатных плат инженерных образцов модуля существенно ниже.

Этому проекту мы дали название «RanetkaPC». «Малина» есть, «бананы», «ананасы» и «апельсины» — тоже. Пусть будет еще и «ранетка». К тому же, ранетка — пожалуй самое популярное плодовое растение в нашем городе (Тюмень), все центральные улицы города засажены им еще со времен СССР, что очень приятно радует глаз, особенно по весне, когда цветут ранетки, и осенью — когда деревья покрываются сочными плодами.

2. Проблема выбора

Первым делом встал вопрос — на каком СнК проектировать такое изделие и какова будет его конечная цена. Не секрет, что отечественные производители микроэлектроники не стесняются брать за неё приличные деньги, и на то есть веские основания, но очень хотелось сделать изделие доступным.

На слуху уже были Эльбрусы с их самобытной VLIW архитектурой, Байкал BE-T1000 на ядрах MIPS и недавно вышедший BE-M1000 с восемью ядрами ARM Cortex-A57. Так же в прессе просачивалась информация о ряде СнК от таких производителей, как НТЦ Модуль, «Миландр», НИИСИ, НИИЭТ и НТЦ «Элвис». В целом, какой-то выбор предоставлялся.

СнК Эльбрус-2С3 отпал сразу — общая аура секретности вокруг технической документации на эти процессоры и проприетарные средства разработки отбивали всякое желание иметь дело с его производителем. К тому же заявленная потребляемая мощность в 30Вт была слишком высока для нашего одноплатного ПК.

СнК Байкал BE-T1000 на архитектуре MIPS оказался слаб в части производительности. А вот Байкал BE-M1000 неожиданно удивил — 68000 пунктов по Coremark. Правда потребляет он при этом те же 30Вт что и Эльбрус-2С3. Теоретически, с этим можно жить, если правильно спроектировать охлаждение. Мы запросили КП и даташит на этот СнК у производителя. КП получили быстро, один «камень» при штучных заказах нам предлагался по 30 000 рублей с НДС. Однако с получением технической документации и референс-дизайна на это изделие вышла неурядица — производитель микросхемы «Байкал Электроникс» попросил нас подписать договор о не разглашении (NDA), на что получил наш отказ. Многие скажут, что NDA — это нормальная общемировая практика. Да, это так, и это один из сдерживающих факторов развития Open Source Hardware. Подписав тогда этот NDA, я бы сейчас не смог написать эту статью. Байкал — не наш выбор.

Стали перебирать и перечитывать информационные листки от других, менее раскрученных производителей. На мой взгляд, неплохо выглядят некоторые СнК от НТЦ Модуль. Например, СБИС 1879ВМ8Я, содержащий 4 ядра ARM Cortex-A5, работающих на частоте 800МГц, и кучу нейровычислителей NMC4. Продукция ПКК «Миландр» больше походит на микроконтроллеры для embedded приложений и к нашей задаче оказалась слабо применима.

В какой-то момент мне попалась информация про СнК 1892ВМ14Я производства НЦТ «Элвис» — два ядра ARM Cortex-A9, работающих на

частоте 816МГц, очень близок к Allwinner A20/T2, на котором у нас уже есть свой процессорный модуль. В этом СнК содержится вся необходимая нам периферия: GPU Mali, два порта SD/MMC, MIPI DSI, MIPI CSI, аппаратные кодеки для видео. Но самое главное — на сайте доступна вся документация, SDK и готовые сборки Linux, есть репозиторий на Github-е. В общем, мы запросили КП и начали обговаривать технические моменты. В ответ на запрос КП, мне перезвонил представитель «Элвиса» и предложил рассмотреть еще пару перспективных (находившихся в разработке) СнК их компании: 1892ВА018 «СКИФ» и 1892ВМ248 «RoboDeus». Оба эти СнК из серии «Multicore» представляют из себя гетерогенные многоядерные вычислители, т.е. содержат вычислительные ядра как общего назначения, так и некоторое количество специализированных, в том числе — DSP ядер. Так СнК «RoboDeus» содержит четыре MISP64 ядра работающих на частоте 1.5ГГц + четыре ядра DSP Elcore-50 и позиционируется производителем для средств видео-аналитики и нейросетевых вычислений. СнК «СКИФ» содержит четыре ARM Cortex-A53 на частоте до 2.0 ГГц + два DSP Elcore-50 и предлагается для использования в мобильных средствах связи - смартфоны, планшеты, цифровые радиостанции. Оба СнК содержат весь спектр «стандартной» для таких решений периферии — GPU, VPU, DSI, CSI, SPI, I2C, I2S, PCIe и т.д. В Скифе предлагается еще отдельный вычислитель для SDR (Software-Define Radio) с двумя интерфейсами JESD 204B, с помощью которого подключаются высокоскоростные ЦАП и АЦП, что позволяет реализовывать различные протоколы радиосвязи, в том числе WIFI и GNSS в софте.

Обсудив и взвесив различные варианты, мы решили остановиться на микросхеме 1892ВА018 СнК «СКИФ». Запросили и получили на неё коммерческое предложение — цена за один «камень» на июль 2021 года составляла 23 000 рублей с НДС. Срок начала отгрузки — октябрь 2021г. «Руководство» на 2100 с плюсом страниц и референс-дизайн платы «MCOM03 Bring-Up-Board» нам прислала служба технической поддержки «Элвиса» без подписания каких-либо NDA. Сборки начального загрузчика, U-Boot и ядра Linux для MCOM-03 доступны на Github-е.

*MCOM03 — кодовое название микросхемы 1892ВА018, используется в «Руководстве», в схемотехнике и программном обеспечении.

3. Система-на-кристалле 1892ВА018 «СКИФ»

И так — встречайте, СнК «СКИФ» производства НТЦ «Элвис». С подробным описанием технических характеристик этого СнК можно ознакомиться на сайте НТЦ «Элвис» по адресу: https://elvees.ru/chip/processors-multicore/skif. Ниже я позволю себе привести только некоторые, на мой взгляд самые интересные, из них:

- Центральный процессор (CPU):
    - 4-х ядерный кластер ARM Cortex-A53 с тактовой частотой 1,2 ГГц (2,0 ГГц макс — прим автора); L1 кэш - 32 кбайт, L2 кэш - 1 Мбайт;
    - контроллер прерываний ARM GIC500;
    - 128-бит SIMD/FPU сопроцессоры NEON.
- Доверенный контур и служебная подсистема:
    - ядро RISC0 32 бит MIPS, 600 МГц;
    - механизм доверенной загрузки.
- Цифровой блок для связных и мобильных приложений:
    - ядро RISC1 32 бит MIPS, 600 МГц;
    - высокопроизводительное сдвоенное DSP ядро с тактовой частотой ядра 600 МГц:
    - навигационное ядро GNSS поддержка 4 стандартов: ГЛОНАСС/GPS/BeiDou/GALILEO;
    - поддержка функций для реализации алгоритмов на базе нейросетей;
    - цифровые фильтры и акселераторы;
    - накристальная память 2 Мбайт;
- Мультимедиа и обработка видео:
    - ISP ядро:
        - поддержка 2 потоков 4K @ 30 или 1 потока 4K @ 60;
    - Видеопроцессор VPU ARM Mali V61:

        - HEVC/H.264;
        - формат данных: 10/8 бит 4.2.2 и 4.2.0;
        - поддержка 1 потока 4K @ 60 или 2 потоков 4K @ 30;
        - поддержка JPEG/MPEG.
    - Графический процессор GPU PowerVR Series8XE GE8300:
        - тактовая частота 550 МГц;
        - поддержка OpenGL, OpenCL, OpenVG;
        - OpenCL API;
        - поддержка Vulkan;
- Интерфейсы:
    - 2 контроллера DDR памяти: DDR3/ LPDDR3/ DDR4/ LPDDR4, 32 бита, 3200 Мбит/с на каждую линию с поддержкой ECC;
    - 2 контроллера PCIe: конфигурация линий 2 x 4; поддержка PCI Express 3.0 скорость на линию 8 ГТ/с;
    - 2 контроллера Ethernet 1 Гбит/с;
    - 2 контроллера JESD 204B, 4 линии, до 12,5 Гбит/с на линию;
    - 2 контроллера USB 3.0 DRD;
    - 2 контроллера SD/eMMC 4.5;
    - порт видеовывода для подключения дисплеев: MIPI DSI или RGB, поддержка разрешения 4K@30, поддержка HDR;
    - 2 контроллера QuadSPI NOR Flash (XiP);
    - сторожевой таймер WDT.
- Технология изготовления: КМОП, 28 нм, процесс TSMC.
- Температурный диапазон: -60...+85 °C.
- Встроенные датчики температуры и напряжения.
- Типовое потребление: до 5-7 Вт.

- Контроллер гибкого управления энергопотреблением, поддержка батарейного питания и режима сна.
- Основные напряжения питания: 3.3 В/1.8 В/0.9 В.
- Тип корпуса: 1936 HFCBGA, 23 мм x 23 мм, шаг по выводам 0,5 мм.

Обратите внимание на тип корпуса этой микросхемы: HFCBGA с числом выводов 1936 и шагом 0.5мм. К этому моменту мы еще не имели опыта проектирования плат, содержащих микросхемы с таким количеством выводов. Для того, чтобы «выйти» из под такой микросхемы нам потребовалось 10 слоёв металлизации с шириной/зазором проводника 0.1/0.075мм и диаметром переходного отверстия 0.3/0.1мм, но подробнее об этом читайте в главе *20. Разработка печатной платы процессорного модуля «RanetkaPC»*. А сейчас предлагаю немного подробнее рассмотреть внутренности микросхемы 1892ВА018 СнК «СКИФ».

4. Внутренняя структура 1892ВА018 СнК «СКИФ»

Рассмотрим структуры микросхемы 1892ВА018 СнК «СКИФ» (MCOM03) которая проиллюстрирована на рисунке 2 ниже.

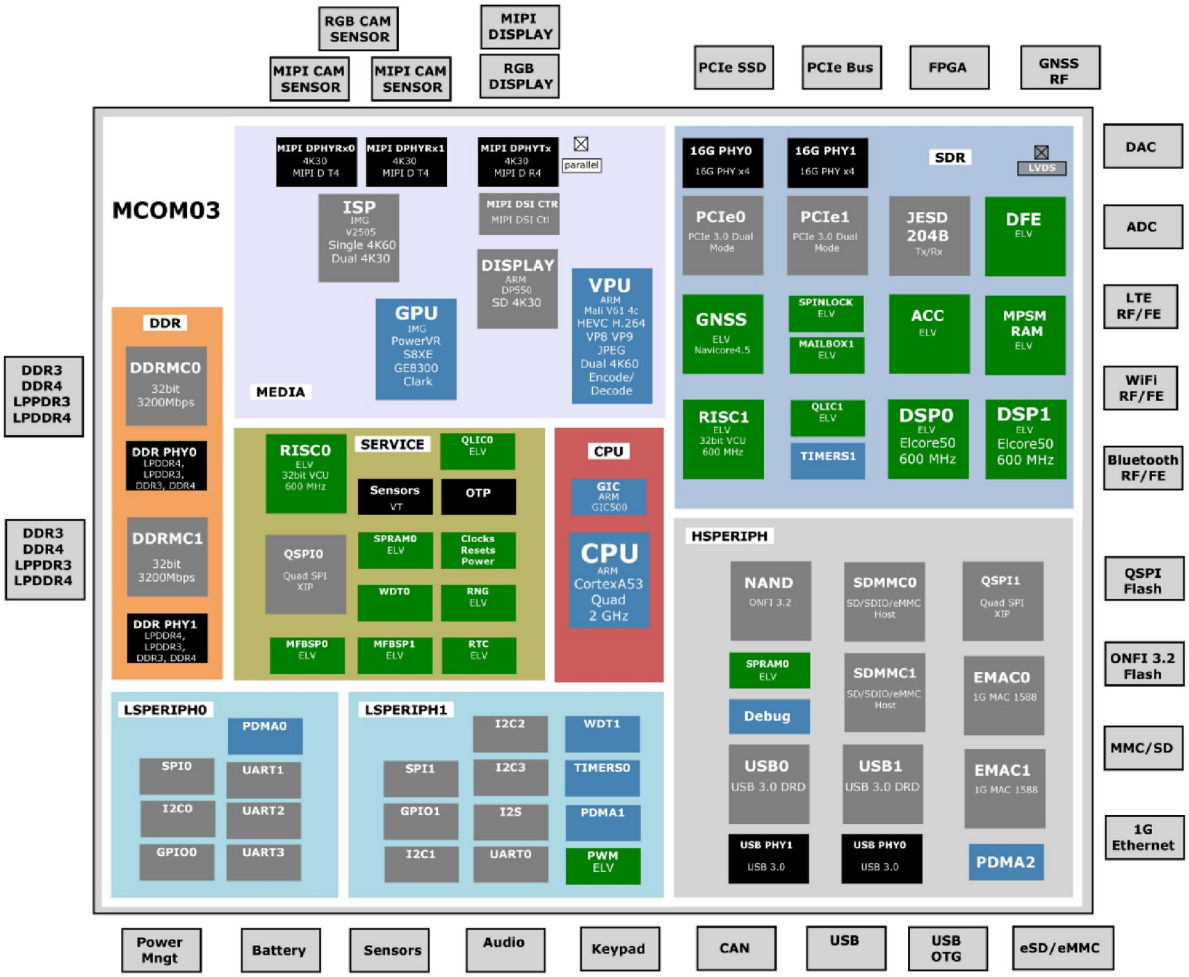

Рис.2. Структура и функциональные блоки СнК «СКИФ».

Видно, что все функциональные блоки СнК объединены в несколько групп, которые, называются «подсистемами» (subsystems или subs). Каждая подсистема имеет свой набор управляющих и конфигурационных регистров, а так же свой набор тактирующих сигналов, свой сигнал сброса, одну или несколько линии питания. Выделяются следующие подсистемы:

- служебная подсистема (service_subs);
- подсистема SDR (sdr_subs);
- подсистема портов динамической памяти (ddr_subs);
- подсистема центрального процессора (cpu_subs);

- подсистема мультимедиа (media_subs);
- подсистема высокоскоростной периферии (hsperiph_subs);
- подсистема низкоскоростной периферии #0 (lsperiph0_subs);
- подсистема низкоскоростной периферии #1 (lsperiph1_subs).

Базовой подсистемой микросхемы, т.е. подсистемой ответственной за формирование опорных частот, сброса, управления питанием для других подсистем и содержащей логику и регистры системных настроек микросхемы, а также блоки, работающие при старте и начальной загрузке микросхемы, является служебная подсистема (service_subs).

Система коммутации верхнего уровня, в виде набора соединений и коммутаторов, логики и регистров их настройки, логики формирования частот и сброса также выделена в отдельную подсистему.

Карта распределения адресного пространства микросхемы СнК «СКИФ» (МСОМ03) приведена в таблице ниже.

Таблица 1. Карта распределения адресного пространства СнК «СКИФ»

Начальный адрес	**Конечный адрес**	**Назначение**	**Размер**
C0_0000_0000	FF_FFFF_FFFF	PCIe1 High	256 ГБ
80_0000_0000	BF_FFFF_FFFF	PCIe1 High	256 ГБ
10_0000_0000	7F_FFFF_FFFF	Зарезервировано	448 ГБ
0C_0000_0000	0F_FFFF_FFFF	DDRMC1	8 ГБ
08_0000_0000	0B_FFFF_FFFF	DDRMC0	8 ГБ
07_8000_0000	07_FFFF_FFFF	DDR Low Mirror (нет прямого доступа)	2 ГБ
05_C000_0000	07_7FFF_FFFF	PCIe1 Mid	7 ГБ
04_0000_0000	05_BFFF_FFFF	PCIe0 Mid	7 ГБ
01_0000_0000	03_FFFF_FFFF	Зарезервировано	12 ГБ
00_8000_0000	00_FFFF_FFFF	DDR Low	2 ГБ
00_7000_0000	00_7FFF_FFFF	PCIe1 Low	256 МБ
00_6000_0000	00_6FFF_FFFF	PCIe0 Low	256 МБ
00_5000_0000	00_5FFF_FFFF	QuadSPI (QSPI1)	256 МБ

00_4000_0000	00_4FFF_FFFF	QuadSPI (QSPI0)	256 МБ
00_0000_0000	00_3FFF_FFFF	Внутренняя память	1 ГБ

Из таблицы 1 видно, что СнК может адресовать 16 ГБ синхронной динамической памяти по 8 ГБ на каждый их двух каналов DDRMC0 и DDRMC1. Согласно «Руководству», каналы DDR поддерживают режим работы с чередованием (interleaving) для ускорения доступа к памяти.

Так же следует упомянуть, что часть аппаратных блоков микросхемы может адресовать только 32 бита. Для таких устройств можно настроить смещение физического диапазона 4ГБ (0x00_0000_0000 – 0x00_FFFF_FFFF), адресуемого устройством, в области памяти старше адреса 0x00_FFFF_FFFF.

С адреса 00_0000_0000 по 00_0000_FFFF находится область статической памяти SPRAM0 размером 64КБ доступной для загрузки и исполнения программ на ядре RISC0.

5. Система обеспечения безопасности микросхемы 1892ВА018 СнК «СКИФ»

Важной и интересной особенностью микросхемы 1892ВА018 СнК «СКИФ» является система обеспечения безопасности. По мимо того, что микросхема аппаратно совместима с технологией ARM TrustZone и следует концепции Trusted Execution Environment (TEE), в ней предприняты дополнительные меры.

Все подсистемы микросхемы поделены между тремя контурами безопасности:

- «доверенный» - во главе с ядром RISC0 (MISP) обеспечивает процесс начальной загрузки, а так же контроль доступа между подсистемами. В «доверенном» контуре содержится «доверенное хранилище ключевой информации» (OTP), ключи из которого могут быть использованы для верификации и загрузки доверенной операционной системы, которая в свою очередь обеспечивает загрузку операционной системой общего назначения и контроль над ней.

- «связной» - во главе с ядром RISC1 (MIPS) содержит специализированные вычислители, в том числе два ядра DSP Elcore-50, вычислитель для GNSS и интерфейс высокоскоростных ЦАП/АЦП. Данный контур исполняет связное программное обеспечение.

- «контур общего назначения» - содержит четыре вычислительных ARM ядра CPU0-3 общего назначения и всю остальную периферию. Является аппаратной базой для операционной системы общего назначения (Linux, Android, и т.д.).

Ниже приведена схема разделения подсистем на контуры, показаны возможные варианты взаимодействия между ними.

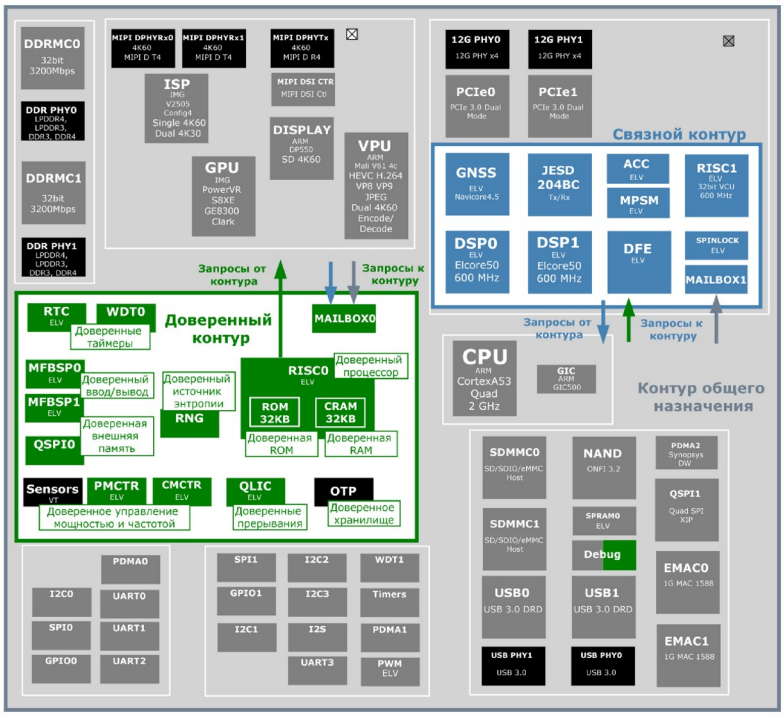

Рис.3. Контуры безопасности СнК «СКИФ».

Взаимодействие между контурами осуществляется посредством механизма «почтовых ящиков» (MAILBOX-ов). Почтовые ящики представляют собой набор набор FIFO. При необходимости передать сообщение, одна из сторон записывает сообщение в соответствующее FIFO, а другая считывает из него. При появлении данных во входном FIFO почтового ящика, вычислительное ядро, обслуживающее данный контур, получает сигнал прерывания, по которому производится считывание сообщения. Ответное сообщение помещается в выходное FIFO этого же почтового ящика. При этом (цитирую «Руководство»):

- Любые запросы в «доверенный» контур от «общего» или «связного» контура разрешены только через доверенный почтовый ящик MAILBOX0.
- Любые запросы в «связной» контур от «общего» контура разрешены только через почтовый ящик MAILBOX1.
- Запросы от «доверенного» контура имеют прямой неограниченный доступ в «связной» и «общий» контур.
- Запросы от «связного» контура имеют прямой неограниченный доступ в «общий» контур.

В системе коммутации микросхемы СнК существует несколько уровней доступа и, соответственно, уровней запросов. В порядке убывания уровня доступа:

- Доверенный уровень (trusted) – запросы от «доверенного» контура и внутри него от его компонентов.
- Связной уровень (sdr) – запросы от «связного» контура и внутри него от его компонентов.
- Безопасный уровень (secure) – запросы от компонентов «общего» контура, в рамках архитектуры ARM TrustZone помеченных как безопасные.
- Общий уровень (non-secure) – запросы от компонентов «общего» контура, в рамках архитектуры ARM TrustZone помеченные как небезопасные.

Области памяти имеют такие же уровни по доступу, отмечающие, какого уровня запросы допустимы к этой области. Более высокий уровень доступа для запроса означает, что для него допустимы обращения к областям памяти с уровнями равными, либо ниже его. Т.е. для запросов trusted допустимы обращения к областям trusted, sdr, secure и non-secure, для запросов sdr к областям sdr, secure, non-secure и т.д.

Для областей памяти существует также особый признак, касающийся доступа – securable, это означает что области памяти, изначально по сбросу, имеющей другой уровень доступа можно присвоить уровень доступа secure. Это работает как в случае повышения уровня, когда области памяти с уровнем доступа non-secure можно сделать secure, так и в случае понижения уровня – некоторые области доверенного (trusted) и связного (sdr) контура можно сделать доступными для secure обращений.

Отметим, что микросхема 1892ВА018 поддерживает режим загрузки, при котором все эти дополнительные меры обеспечения безопасности могу быть отключены подачей логической «1» на специальный вывод VS_EN микросхемы.

Авторизация загрузчика (проверка электронной подписи) на данном СнК по умолчанию отключена. Включение авторизации осуществляется подачей логической «1» на вывод BS_EN микросхемы.

Наша промежуточная цель на этапе «подъема» разработанной нами платы процессорного модуля «RanetkaPC» является загрузка и запуск ОС Linux. Образ рабочей сборки AltLinux был предоставлен нам службой технической поддержки «Элвиса», но вот сам процесс загрузки оказался совсем не тривиален и с ним пришлось детально разобраться.

6. Загрузка микросхемы 1892ВА018 СнК «СКИФ»

Процесс загрузки микросхемы 1892ВА018 радикально отличается от всего того, с чем мы имели дело ранее. Основное отличие состоит в том, что первым, после подачи питания и снятия состояния RESET, всегда запускается служебное MIPS ядро RISC0 из доверенного контура. Это ядро исполняет т.н. первичный загрузчик, который находится в встроенной ROM (32K) памяти расположенной в доверенном контуре и «прошитой» в процессе производства микросхемы, либо сразу переходит к исполнению загрузчика из NOR flash, если выбран соответствующий режим. Остальные вычислительные ядра в этом момент остановлены и обесточены. В задачи первичного загрузчика входит инициализация остальных вычислительных ядер, инициализация (трэйнинг) SDRAM, поиск и размещение загрузчика следующего уровня и размещение его в SDRAM, запуск вычислительного ядра общего назначения.

Следующая особенность — это выбор способа загрузки. Микросхема 1892ВА018 имеет три служебные линии BOOT[0:2], выведенные на корпус для задания режима загрузки. Логические состояния по входу этих линий определяют дальнейшее поведение ROM загрузчика, за исключением режима ***0b000***. В таблице ниже приведены все режимы (варианты) загрузки.

Таблица 2. Режимы загрузки СнК «СКИФ»

BOOT[2:0]	**Источник начальной загрузки**
0b000	Загрузка напрямую через QSPI0
0b001	Загрузка через ROM RISC0/QSPI0
0b010	Загрузка через ROM RISC0/MFBSP0
0b011	Загрузка через ROM RISC0/UART0
0b100	Загрузка через ROM RISC0/SDMMC0
0b101	Загрузка через ROM RISC0 с быстрым стартом CPU из QSPI1

0b110	резерв
0b111	Режим NoBoot (RISC0 не загружается в ожидании сеанса отладки)

Если в момент подачи питания на входных линиях (пинах) микросхемы на сигнальной линии BOOT2 присутствует логический «0», на BOOT1 присутствует «0» и на BOOT0 присутствует «0», то выбран режим ***0b000***. В этом режиме ядро RISC0 начинает исполнение с физического адреса ***0x00_4000_0000***. Из таблицы распределения адресного пространства (см. выше) можно увидеть, что с этого адреса располагается область памяти QSPI0 - нулевой порт QuadSPI NOR Flash памяти. Как правило, микросхемы NOR flash памяти с интерфейсом QSPI обладают возможностью исполнения кода прямо из изнутри, то есть без копирования кода в статическую или динамическую память. Эта особенность называется XiP (eXecution in Place). Другими словами, в режиме ***0b000*** загрузчик исполняется прямо из flash памяти, минуя ROM.

Если в момент подачи выбран любой другой режим, то RISC0 всегда передает управление загрузчику вшитому в ROM, который располагается по адресу ***0x00_1fc0_0000***. Этот загрузчик так же анализирует состояние сигнальных линий BOOT[2:0] и выполняет соответствующую процедуру загрузки.

Особый интерес представляет режим загрузки ***0b011*** - «Загрузка через ROM RISC0/UART0». В данном режиме, после подачи питания, ядро RISC0 исполняет код из ROM, программа которого переходит в командный (CLI) режим, т.е. ожидает ввода команды из последовательного порта UART0 — классический «монитор». Простейший интерпретатор команд «монитора» позволяет выполнить ряд операций:

- вывести дамп области памяти, заполнить область значением;
- загрузить блок из UART0 и поместить его в память;
- передать управление по адресу.

Данный режим «монитора» интересен тем, что имея в распоряжении только подключенный последовательный порт и линии питания, можно протестировать СнК и даже загрузить в него и исполнить свою собственную «bare metal» программу на MIPS ядре RISC0, при этом загрузка кода программы осуществляется в специализированный блок памяти SPRAM0, находящийся по адресу ***0x00_0000_000*** и размером 64K. Загруженной таким образом программе

доступно 32K статической памяти SRAM расположенной с адреса ***0x00_1fa0_0000***.

Далее я покажу, как мы воспользовались этой возможность для тестирования СнК при первом запуске нашего изделия и это сильно упростило нам работу. Хочу выразить отдельную благодарность инженерам НТЦ «Элвис» за такую полезную возможность!

7. Загрузка СнК «СКИФ» — более подробно

Рассмотрим немного более подробно режим загрузки ***0b000*** - «Загрузка напрямую через QSPI0» и проследим весь процесс загрузки до запуская ядра ОС общего назначения (ядра ОС Linux).

Как я уже отмечал выше, в этом режиме при подаче питания первым делом запускается MIPS ядро RISC0, которое начинает исполнять код с физического адреса ***0x00_4000_0000*** — это адресное пространство аппаратно маппируется на содержимое NOR flash памяти, подключенной к порту QSPI0. В NOR flash памяти должен быть размещен загрузчик SBL (Simple Boot Loader), состоящий их последовательно идущих друг за другом нескольких программных модулей:

- первым располагается сам SBL — код для MISP32;
- далее следует DDRINIT — код для AArch64;
- далее следует TF-A — код для AArch64;
- и завершает цепочку U-Boot — код для AArch64.

SBL (Simple Boot Loader), как следует из названия, представляет собой примитивный загрузчик, в задачи которого входит инициализация DDR контроллеров, после чего SBL помещает в SDRAM код TF-A и U-Boot и запускает ARM ядро. SBL исполняясь на MIPS производит следующие действия:

1. Инициализирует основные сигналы тактирования и поднимает частоту для RISC0 до 1.2ГГц;
2. Подает питание на контроллеры DDR памяти и на ARM ядра;
3. Копирует код модуля DDRINIT из QSPI0 в статическую память SPRAM0 которая маппируется на адресное пространство ***0x00_0000_0000***, эта память внутри ядра RISC0 видна с адреса ***0x80000000***;
4. Стартует ARM ядро CPU0 таким образом, что это ядро начинает исполнять код в SPRAM0, где находится DDRINIT;
5. Дожидается результата исполнения DDRINIT на CPU0 и если он успешный, то -

6. Копирует из QSPI0 в SDRAM начиная с физ адреса ***0x80000000*** код модуля TF-A;
7. Копирует из QSPI0 в SDRAM начиная с физ адреса ***0x80080000*** код модуля U-Boot;
8. Перезапускает CPU0 и зависает в вечном цикле, в то время как CPU0 исполняет код TF-A, который в свою очередь передает управление коду U-Boot.

Тут следует отметить, что в конфигурациях вычислительных систем с межконтурной защитой, вместо SBL должно быть что-то более серьезное — то, что называется «доверенной операционной системой». Эта доверенная ОС на начальных этапах выполняет все те же функции, что и SBL, но после запуска TF-A переходит в режим контроля и обслуживания запросов от ОС общего назначения. К сожалению, в наше распоряжение такой операционной системы предоставлено не было, соответственно опробовать все прелести и тонкости этого механизма пока что не предоставилось.

DDRINIT - это программный модуль, который исполняясь на ARM ядре, производит загрузку начальных параметров в блоки управления контроллеров динамической памяти DDRMC0 и DDRMC1. После чего, DDRINIT последовательно инициирует процедуру тренировки и калибровки временных параметров физических трансиверов при работе с конкретными микросхемами DDR памяти (PHY training). Взаимодействие с контроллерами DDRMC0 и DDRMC1 производится через MAILBOX так, как ядро ARM и контролеры памяти находятся в различных контурах.

Для ускорения задачи, DDRINIT имеет возможность считать уже ранее рассчитанные калибровочные параметры из микросхемы EEPROM на шине I2C, если таковая предусмотрена, и передать их контроллерам.

TF-A или **«ARM Trusted Firmware A»** - это программный модуль распространяемый по лицензии BSD-3 одноименным консорциумом во главе с ARM, Ltd., представляет собой референсную имплементацию системы защиты внутри ARM ядра. Если сильно не вдаваться в подробности, то основной смысла таков:

TF-A исполняясь на ARM ядре в режиме EL3 (Execution Level 3) — самый высокий уровень исполнения, имеет доступ к абсолютно всем аппаратным возможностям ARM. TF-A производит инициализацию служебных векторов прерываний и исключений, конфигурирует MMU таким образом, чтобы защитить (ограничить) доступ к ряду областей памяти и аппаратуре при исполнении кода работающего на пониженных уровнях (EL2, EL1, EL0),

загружает и запускает свою (ARM-овскую) доверенную ОС, после чего запускает загрузчик ОС общего назначения U-Boot с пониженым EL2 (или EL1) уровнем исполнения. Таким образом U-Boot и ОС Linux функционируют уже с пониженным уровнем привилегий «под присмотром» доверенной ОС. Доверенные ОС нас пока не интересуют, по этому в TF-A оставим заглушку по умолчанию, которая стартует U-Boot.

Для желающих погрузиться в этот удивительный мир уровней исполнения и безопасности ARM, приведу ссылку на документ под названием «ARM Trusted Firmware Design». Вот небольшая выдержка из этого документа:

The cold boot path in this implementation of the ARM Trusted Firmware is divided into five steps (in order of execution):

- *Boot Loader stage 1 (BL1) AP Trusted ROM*
- *Boot Loader stage 2 (BL2) Trusted Boot Firmware*
- *Boot Loader stage 3-1 (BL3-1) EL3 Runtime Firmware*
- *Boot Loader stage 3-2 (BL3-2) Secure-EL1 Payload (optional)*
- *Boot Loader stage 3-3 (BL3-3) Non-trusted Firmware*

U-Boot, популярный загрузчик ОС общего назначения, своего рода стандарт «de-facto» в embedded и мобильной индустрии. Задача U-Boot — поиск, загрузка и запуск ядра ОС общего назначения (в нашем случае ОС Linux) и конфигурационного файла к нему (Device-Tree). U-Boot обладает гибким скриптовым языком, умеет работать с сетью и множеством различной периферии, в том числе SPI, I2C, MMC/SD карты, NAND и NOR Flash и даже SATA. Честно говоря, возможности последних версий U-Boot настолько многообразны, что U-Boot можно рассматривать как однозадачную ОС общего назначения с сетевым стеком и некоторыми зачатками графического интерфейса.

8. Первый запуск СнК «СКИФ»

Более подробно аппаратное устройство разработанного нами процессорного модуля «RanetkaPC» на базе СнК «СКИФ» я рассмотрю ниже. Здесь же упомяну, что модуль был спроектирован таким образом, чтобы максимально облегчить его запуск: для включения модуля достаточно подвести сигналы GND, TX и RX порта UART0 от модуля к «USB2Serial»

преобразователю, а так же подать напряжение +5V от источника питания на соответствующий вход модуля. Даже если плата модуля имеет какие-то дефекты монтажа, СнК «СКИФ», получив питание, перейдет в режим загрузки ***0b000***, т.е. начнет исполнение кода SBL из QSPI0, а в терминале мы сможем наблюдать диагностический вывод от программных модулей SBL.

Но как начать запуск, если во Flash памяти процессорного модуля, который только что вышел из печи, отсутствует какой либо код для данного СнК? Здесь нам и придет на помощь режим загрузки ***0b011*** - «Загрузка через ROM RISC0/UART0». Напомню, что в этом режиме СнК стартует программу «монитор» из своего ROM прошитого при его производстве, программа переходит в режим командной строки и начинает общаться с разработчиком через порт UART0.

Запаяем перемычки BOOT[2:0] на плате процессорного модуля таким образом, чтобы выставить режим ***0b011***, подключим к рабочей машине USB2Serial преобразователь и подсоединим его к модулю, запустим утилиту эмуляции терминала ***cu*** (далее я буду использовать утилиту cu(1) — «call Unix» из стандартной поставку ОС FreeBSD) с параметрами порта **115200 8N1** и подадим питание 5V на модуль. Вот что мы увидим в терминале:

```
rz@butterfly:/home/rz/mcom03 $ cu -l /dev/ttyU0 -s 115200

RnD Center "ELVEES", JSC
MCom-03, Bootrom rev.1.0
#
```

Введем команду ***help*** и посмотрим, каким инструментарием мы располагаем:

```
#help
List of commands:
        help
        dump
        set
        run
        boot
        sdopen
        mfbspopen
        qspiopen
        devdump
        devsetp
        devcommit
        otpdump
        otpcommit
```

Проверим, доступна ли нам SPRAM0, попробуем считать её содержимое с адреса ***0x80000000***:

```
#dump 0x80000000
0x80000000: 0xcfcdfeef, 0xf7f9dfaf, 0x4c516ddb, 0x3e120fce
0x80000010: 0x5be2d54f, 0xfedc7dde, 0x2f0a382b, 0x3e697820
```

Как видно, dump возвращает нам какие-то данные. Попробуем изменить содержимое одной ячейки памяти и еще раз считать её:

```
#set 0x80000000 0xaabbccdd

#dump 0x80000000
0x80000000: 0xaabbccdd, 0xf7f9dfaf, 0x4c516ddb, 0x3e120fce
0x80000010: 0x5be2d54f, 0xfedc7dde, 0x2f0a382b, 0x3e697820
```

Видим, что данные записываются в память и успешно считываются обратно.

Тут встает логичный вопрос: каким образом область памяти SPRAM0, которая расположена с физического адреса 0x00_0000_0000 размером 64КБ, стала доступна ядру RISC0 по 32-битному адресу 0x80000000. Я задал его в службу технической поддержки и ниже привожу ответ инженера «Элвиса», цитирую:

*В режиме загрузки **0b011** процессор (имеется в виду RISC0) находится в режиме Kernel. При этом если три старших бита виртуального адреса равны 0b100 , выбирается 32-разрядное виртуальное адресное пространство **kseg0**. Это область размером 512 MB, которая расположена внутри границ, определяемых адресами **0x8000_0000** и **0x9FFF_FFFF**. Ссылки к kseg0 не отображаются, а физический адрес получается вычитанием **0x8000_0000** из виртуального адреса. Таким образом при обращении по виртуальному адресу **0x8000_0000** обращение идет в физический адрес **0x00_0000_0000**, что соответствует стартовому адресу памяти SPRAM0.*

Вспомним это, когда будем загружать нашу тестовую программу «**hello-world**» и прошивать NOR flash.

Теперь посмотрим, что содержится в OTP памяти, для этого воспользуемся командой ***otpdump***. OTP — это память в которой сохраняются

ключи для верификации загрузчиков и прочая сопутствующая информация, связанная с безопасностью микросхемы. Мы не собираемся использовать эту возможность, но знать о ней не помешает.

```
#otpdump

OTP [virtual 0x9fa00000]:
0x00: ZeroCount         :   0x0000 (0)
0x02: Flags             :   0x0000
      [ ForceSign       :   0 ]
      [ ForceEncrypt    :   0 ]
      [ DisableLog      :   0 ]
      [ EnableWatchdog  :   0 ]
0x04: Serial            :   0x00000000
0x08: DUK               :   0x00000000000000000000000000000000
0x18: RootKeyDigest     :   0x00000000000000000000000000000000
                              00000000000000000000000000000000
0x38: RevocationList    :   0x00000000(0x00000000)    ERROR
0x3c:                       0x00000000(0x00000000)    ERROR
0x40:                       0x00000000(0x00000000)    ERROR
0x44:                       0x00000000(0x00000000)    ERROR
0x48:                       0x00000000(0x00000000)    ERROR
0x4c:                       0x00000000(0x00000000)    ERROR
0x50:                       0x00000000(0x00000000)    ERROR
0x54:                       0x00000000(0x00000000)    ERROR
0x58: FW_counter        :   0x0000000000000000
```

9. Собираем и исполняем «Hello,World» на СнК «СКИФ»

Следующим этапом попробуем загрузить свою небольшую программу **hello-world** в SPRAM0 через UART0 и исполнить её на ядре RISC0. Для этого требуется подготовить бинарник в формате HEX. Данный формат был введен в обиход компанией Intel в далеком 1973 году и с тех пор часто используется для сохранение слепков ROM, прошивок микроконтроллеров и еще много для чего.

Файл формата HEX представляет собой ASCII текст, каждая строка которого содержит небольшой блок данных в hexadecimal формате, адрес для размещения этого блока и его контрольную сумму. Каждая строка начинается с символа «:» (двоеточие), что позволяет легко детектировать и парсить данные, поступающие через последовательный интерфейс. Детальное описание Intel HEX формата можно прочитать в Wikipedia.

Если мы будем подавать такие строки с блоками данных в HEX начинающиеся с символа «:» в порт UART0, то программа ROM «монитор» будет декодировать их, изымать данные и сохранять по указанному адресу. Таким образом, нам потребуется скомпилировать свою программу hello-world

для архитектуры MIPS32 без использования каких либо библиотек (режим компиляции bare-metal); слинковать её таким образом, чтобы текст (код) программы располагался с адреса ***0x80000000***; преобразовать получившийся бинарник в HEX формат и побайтово скопировать его в порт UART0 нашего модуля. Передать управление по любому адресу (в том числе в место загрузки нашей программы) можно с помощью команды ***run*** **<*addr*>** монитора. Команда ***run*** без параметра передает управление по адресу ***0x80000000***.

Начнем с того, что выкачаем **Codescape GNU IMG Bare Metal Toolchain** для архитектуры MIPS32 по следующей ссылке:

https://codescape.mips.com/components/toolchain/2019.02-05/Codescape.GNU.Tools.Package.2019.02-05.for.MIPS.IMG.Bare.Metal.CentOS-6.x86_64.tar.gz

Развернем его командой ***tar*** в каталог ***/opt*** на рабочей Linux (x86_64) машине:

```
rz@devbox:~$ tar -zxpf Codescape.GNU.Tools.Package.2018.09-
03.for.MIPS.IMG.Bare.Metal.CentOS-6.x86_64.tar.gz -C /opt
```

Создадим скрипт следующего содержания для установки переменных окружения для кросс-компиляции:

```
rz@devbox:~$ cat /opt/setenv_gcc7_mips.sh

export ARCH=mips
export CONFIG_ARCH=mips
export CROSS_COMPILE=/opt/mips-img-elf/2018.09-03/bin/mips-img-elf-
export SYSROOT=/opt/mips-img-elf/2018.09-03/

export INSTALL_MOD_PATH=output
export DESTDIR=output
export  CFLAGS="-Wno-misleading-indentation -Wno-int-to-pointer-cast -Wno-
pointer-to-int-cast -I${SYSROOT}/include -I${SYSROOT}/usr/include "
export LDFLAGS="-L${SYSROOT}/lib -L${SYSROOT}/usr/lib -L${SYSROOT}/lib64"
export LIBS=""
export CC=${CROSS_COMPILE}gcc
export AS=${CROSS_COMPILE}as
export ASM=${CROSS_COMPILE}gcc
export LD=${CROSS_COMPILE}ld
export PKG_CONFIG_EXECUTABLE=/usr/bin/arm-linux-gnueabihf-pkg-config
export PKG_CONFIG_SYSROOT_DIR=${SYSROOT}
export PKG_CONFIG_PATH=${SYSROOT}/lib/pkgconfig
```

Загрузка переменных окружения для данного сеанса осуществляется командой:

```
rz@devbox:~$ . /opt/setenv_gcc7_mips.sh
```

Выкачаем готовый пример bare-metal утилит с репозитория НТЦ «Элвис» на Github-e:

```
rz@devbox:~$ git clone https://github.com/elvees/mcom03-baremetal-tools.git
```

Подправим исходный код программы «hello-world», содержащейся в файле ***mcom03-baremetal-tools/hello-world/main.c*** — добавим номер порта и линии GPIO, к которому у нас подключен светодиод, а так же код инициализации этого GPIO, как показано ниже.

```
/*
 * Copyright 2021 RnD Center "ELVEES", JSC
 */

#include <stdint.h>

#include <regs.h>
#include <uart.h>

int main(void)
{
        uint32_t gpio_state = 1;

        REG(LSPERIPH1_SUBS_PPOLICY) = PP_ON;  // Enable LSPERIPH1
        REG(TOP_CLKGATE) |= BIT(6);  // Enable clock to LSPERIPH1
        while ((REG(LSPERIPH1_SUBS_PSTATUS) & 0x1f) != PP_ON) {
        }

        REG(LSP1_UCG_CTRL4) = 0x2;  // GPIO_DBCLK CLK_EN
        REG(LSP1_UCG_CTRL6) = 0x2;  // UART_CLK CLK_EN

        REG(GPIO1_SWPORTB_CTL) |= 0xc0;  // UART0 in hardware mode

        REG(GPIO1_SWPORTD_CTL) = 0;  // GPIO1_PORTD in GPIO mode
        REG(GPIO1_SWPORTD_DDR) |= 0x1;  // GPIO1_PORTD_0 to output

        REG(GPIO1_SWPORTA_CTL) = 0;  // GPIO1_PORTA in GPIO mode
        REG(GPIO1_SWPORTA_DDR) |= 0x10;  // GPIO1_PORTA_4 to output

        uart_init(XTI_FREQUENCY, 115200);
        while (1) {
                REG(GPIO1_SWPORTD_DR) = gpio_state;
                        REG(GPIO1_SWPORTA_DR) = (gpio_state << 4); //
GPIO1_PORTA_4
                uart_puts("Hello, world!\n");
                gpio_state = !gpio_state;
                for (volatile int i = 0; i < 10000; i++) {
```

```
                        asm volatile ("nop");
                }
        }

        return 0;
}
```

Выполним сборку нашей программы:

```
. /opt/setenv_gcc7_mips.sh
cd  mcom03-baremetal-tools
mkdir build
cmake ..
make
```

В результате в каталог ***build*** будут собраны две программы: ***hello-world*** и ***spi-flasher***, каждая в двух вариантах: для исполнения из flash памяти (XiP) порта QSPI0 и для исполнения путем загрузки в SPRAM0 через UART0:

```
rz@devbox:~/mcom03-baremetal-tools/build$ ll */*.hex
-rw-rw-r-- 1 rz rz  2166 Mar 21 00:05 hello-world/hello-world-mips-ram.hex
-rw-rw-r--  1  rz  rz   2166  Mar  21  00:05  hello-world/hello-world-mips-
xip0.hex
-rw-rw-r-- 1 rz rz 26916 Mar 21 00:05 spi-flasher/spi-flasher-mips-ram.hex
-rw-rw-r--  1  rz  rz  26916  Mar  21  00:05  spi-flasher/spi-flasher-mips-
xip0.hex
```

Загрузим файл ***hello-world-mips-ram.hex*** через терминал в порт UART0 нашего модуля и выдадим монитору команду ***run*** на исполнение:

```
rz@butterfly:/home/rz/mcom03 $ cu -l /dev/ttyU0 -s 115200
Connected
RnD Center "ELVEES", JSC
MCom-03, Bootrom rev.1.0
#
~>Local file name? hello-world-mips-ram.hex
752 bytes


#run
Starting 0x80000000...
Hello, world!
Hello, world!
...
```

Как видим, исполняясь, программа производит вывод в последовательный порт строки текста, при этом моргает светодиод, подключенный к порту GPIO1_PORTA_4 на внешнем периметре нашего модуля.

Если кто не совсем понял, как происходит передача содержимого файла из локальной файловой системы в порт, поясню: утилита ***cu*** из стандартного дистрибутива ОС FreeBSD имеет возможность побайтовой передачи содержимого файла в последовательный порт. Для того чтобы вызвать эту функцию, необходимо нажать последовательность из двух символов **~>** , на вопрос ***Local file name?*** задать имя локального файла и два раза нажать ***Ввод***. Процесс передачи окончится сообщением числа переданных байт: ***752 bytes***.

10. Сборка SBL, DDRINIT, TF-A и U-Boot

Чтобы продолжить работу, а именно — начать загрузку ОС общего назначения, нам понадобиться загрузчик SBL. Напомню, что SBL содержит следующие программные модули, которые исполняются последовательно: сам SBL, DDRINIT, TF-A и U-Boot. Разные модули SBL выполняются на вычислительных ядрах различных архитектур внутри СнК «СКИФ», краткое описание каждого приводилось выше.

Для того чтобы выполнить полную сборку блоба (получить файл ***sbl-mips.bin***), нам потребуется целых два toolchain-а: **Codescape GNU IMG Bare Metal** toolchain для MIPS32 и **GCC 8.2 Toolchain** для Aarch64 (ARM64). Выкачаем и установим недостающий toolchain для AArch64 согласно приведенной ниже инструкции:

1) Скачиваем GCC 8.2 toolchain для архитектуры AArch64 (little-endian) с официального сайта ARM, Ltd., ссылка на страницу сайта:

https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads/8-2-2018-08

Нам потребуются следующие файлы:

gcc-arm-8.2-2018.08-x86_64-aarch64-linux-gnu.tar.xz

sysroot-glibc-8.2-2018.08-x86_64-aarch64-linux-gnu.tar.xz

runtime-gcc-8.2-2018.08-x86_64-aarch64-linux-gnu.tar.xz

2) Распаковываем toolchain так, чтобы пакеты библиотек **sysroot** и **runtime** оказались внутри каталога **/opt/gcc-arm-8.2-2018.08-x86_64-aarch64-linux-gnu**:

```
# cd /opt

# tar -xzpf ~/Downloads/gcc-arm-8.2-2018.08-x86_64-aarch64-linux-gnu.tar.xz

# cd /opt/gcc-arm-8.2-2018.08-x86_64-aarch64-linux-gnu

# tar -zxpf ~Downloads/sysroot-glibc-8.2-2018.08-x86_64-aarch64-linux-gnu.tar.xz

# tar -zxpf ~Downloads/runtime-gcc-8.2-2018.08-x86_64-aarch64-linux-gnu.tar.xz
```

2) Создаем скрипт ***/opt/setenv_gcc8_arm64.sh*** **,** устанавливающий переменные окружения для кросс-компиляции:

```
rz@devbox:~/linux-kernels/linux-mcom03$ cat /opt/setenv_gcc8_arm64.sh

export ARCH=aarch64
export
CROSS_COMPILE=/opt/gcc-arm-8.2-2018.08-x86_64-aarch64-linux-gnu/bin/
aarch64-linux-gnu-
export SYSROOT=/opt/gcc-arm-8.2-2018.08-x86_64-aarch64-linux-gnu
export INSTALL_MOD_PATH=output
export DESTDIR=output
export CFLAGS="-Wno-misleading-indentation -Wno-int-to-pointer-cast -Wno-
pointer-to-int-cast -I${SYSROOT}/include "
export LDFLAGS="-L${SYSROOT}/lib -L${SYSROOT}/lib64"
export CC=${CROSS_COMPILE}gcc
export CXX=${CROSS_COMPILE}g++
export CPP=${CROSS_COMPILE}cpp
export LD=${CROSS_COMPILE}ld
export AS=${CROSS_COMPILE}as
export ASM=${CROSS_COMPILE}as
export PKG_CONFIG_EXECUTABLE=/usr/bin/arm-linux-gnueabihf-pkg-config
export PKG_CONFIG_SYSROOT_DIR=${SYSROOT}
export PKG_CONFIG_PATH=${SYSROOT}/lib/pkgconfig
```

Теперь нам потребуется выкачать исходные коды каждого из программных модулей, входящих в состав SBL, сделаем это путем клонирования соответствующих репозиториев с Github-а:

```
$ git clone https://github.com/elvees/mcom03-sbl.git
$ git clone https://github.com/elvees/ddrinit.git mcom03-ddrinit
```

```
$ git clone -b mcome02 --depth 1 https://github.com/elvees/arm-trusted-firmware.git mcom03-arm-trusted-firmware
$ git clone -b mcom03 --depth 1 https://github.com/elvees/u-boot.git mcom03-u-boot
```

Собирать модули будем в обратной последовательности. Нам важно, чтобы ***mcom03-sbl*** собирался последним, так как именно в нем происходит линковка всех бинарников в единый блоб и сборка финального файла ***sbl-mips.bin*** используемого для прошивки во flash память.

Собираем U-Boot для архитектуры AArch64:

```
$ . /opt/setenv_gcc8_arm64.sh
$ cd ~/mcom03-u-boot
$ make mcom03bub_defconfig
$ make
```

В результате получаем файл ***~/mcom03-u-boot/u-boot.bin***

Собираем TF-A для архитектуры AArch64:

```
$ . /opt/setenv_gcc8_arm64.sh
$ cd ~/mcom03-arm-trusted-firmware
$ make PLAT=mcom03
```

В результате получаем файл
~/mcom03-arm-trusted-firmware/build/mcom03/release/bl31.bin

Собираем DDRINIT для архитектуры AArch64:

```
$ . /opt/setenv_gcc8_arm64.sh
$ cd ~/mcom03-ddrinit
$ make
```

В результате получаем файл ***~/mcom03-ddrinit/output/ddrinit.bin***

И самым последним собираем SBL для архитектуры MIPS32. В процессе сборки SBL будет собран финальный блоб. Чтобы это состоялось, перед сборкой SBL нам потребуется подредактировать файл ***CMakeLists.txt*** и исправить в нём пути к собранным ранее ***.bin*** файлам. Сборка блоба происходит средствами линковщика ***ld***, для чего в файле ***link.ld.in*** для каждого из бинарных модулей заведена отдельная поименованная секция с глобальной ссылкой.

Ниже приведен фрагмент файла ***CMakeLists.txt***, который требуется подредактировать вручную:

```
set(DDRINIT_PATH "~/mcom03-ddrinit/output/ddrinit.bin")
if(NOT DDRINIT_PATH)
    message(FATAL_ERROR "DDRINIT_PATH is not specified")
endif()

set(TFA_PATH
"~/mcom03-arm-trusted-firmware/build/mcom03/release/bl31.bin")
if(NOT TFA_PATH)
    message(FATAL_ERROR "TFA_PATH is not specified")
endif()

set(UBOOT_PATH "~/mcom03-u-boot/u-boot.bin")
if(NOT UBOOT_PATH)
    message(FATAL_ERROR "UBOOT_PATH is not specified")
endif()
```

Процедура сборки типовая:

```
$ . /opt/setenv_gcc7_mips.sh
$ cd ~/mcom03-sbl
$ vi CMakeLists.txt ### < --- исправить пути к .bin файлам
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Финальный блоб будет находится в файле ***~/mcom03-sbl/build/sbl-mips.bin***

11. Прошиваем SBL загрузчик во flash память на порту QSPI0

Следующим этапом попробуем выяснить, работает ли у нас NOR flash память. Для этого нам нужно загрузить и исполнить программу **spi-flasher** из пакета **mcom03-baremetal-tools**. После сборке, проведенной выше, данная программа будет находится в файле ***spi-flasher-mips-ram.hex***.

По аналогии с **hello-world**, подключимся терминалом к нашему модулю, загрузим и запустим файл ***spi-flasher-mips-ram.hex***:

```
rz@butterfly:/home/rz/mcom03 $ cu -l /dev/ttyU0 -s 115200
Connected
```

```
#~>Local file name? spi-flasher-mips-ram.hex
9552 bytes

#run
Starting 0x80000000...
QSPI Flasher
#
```

Программа spi-flasher готова к работе и принимает свой набор команд:

```
#help
help    - show this message
qspi    - select active QSPI controller: qspi <id>
erase    - erase sector: erase <offset_in_bytes>
write    - turn to write mode (requred binary data) : write <offset>
read    - read data from SPI flash: read <offset> <size> [text|bin]
readcrc    - return CRC16 of data: readcrc <offset> <size>
custom    - send custom command byte and read answer: custom <cmd> <size>
```

Попытаемся считать идентификатор микросхемы flash памяти подключенной к порту QSPI0 отправив следующие команды:

```
# qspi 0
# custom 0x9f 3
```

Команда ***qspi 0*** выбирает порт QSPI0, а команда ***custom*** посылает кастомную команду с номером ***0x9F*** микросхеме flash памяти и дожидается ответа из неё 3-х байт. Вот какой ответ мы получим от микросхемы flash памяти на нашем модуле «RanetkaPC»:

```
ef 40 16
#
```

Это говорит нам о том, что микросхема flash памяти отвечает на запросы. Её ID ={0xef, 0x40, 0x16}, что соответсвует значению из даташита на используемую нами микросхему Winbond W25Q32. А значит можно попытаться «прошить» в неё SBL загрузчик.

Для того чтобы осуществить запись загрузчика в микросхему flash памяти, нам потребуется пакет программ **mcom03-flash-tools**, который в свою очередь состоит из следующих утилит:

1. ***mcom03-flash*** — flont-end утилита для прошивки написанная на языке Python для ОС общего назначения. Взаимодействует с СнК MCOM03 через UART0, загружает и запускает ***spi-flasher-mips-***

ram.hex, после чего, используя командный интерфейс, по-блочно передает и записывает файл из локальной файловой системы в NOR flash на одном из портов СнК: QSPI0 или QSPI1.

2. ***mcom03-read-flash*** — front-end утилита для чтения содержимого flash памяти и сохранения в локальный файл.

Выкачать данный пакет можно с Github репозитория следующей командой:

```
$ git clone https://github.com/elvees/mcom03-flash-tools.git
```

Установка пакета конвенциональна для программ на Python-е:

```
cd mcom03-flash-tools
pip3 install . --user

# or if you want to hack the code
pip3 install -e . --user
```

Подробная инструкция по сборке и установке, как обычно, находится в файле README.rst.

Здесь следует сделать небольшое замечание. В файле ***mcom03_flash_tools/__init__.py*** имеется массив структур, описывающих параметры различных микросхем flash памяти с QSPI интерфейсом, с которыми может работать утилита. Используемой нами микросхемы Winbond W25Q32 в этом массиве не было, по этому пришлось добавить следующую строку:

```
FlashType("W25Q32", 4 * MiB, 64 * KiB, 256, [0xEF, 0x40, 0x16]),
```

Если этого не сделать, то утилита ***mcom03-flash,*** считав идентификатор из микросхемы памяти, сообщит о том, что данная микросхема не поддерживается и на этом закончит свое исполнение. Формат структуры очень простой: *строка названия микросхемы, размер микросхемы в байтах, размер erase блока, размер страницы для записи* и *массив байтов идентификатора микросхемы*. Вся эта информация легко добывается из даташита на микросхему, либо угадывается опытным путем. :)

Разумеется, после внесения изменений, нужно выполнить установку пакета еще раз.

Попробуем прошить во flash память загрузчик SBL, который хранится в локальном файле ***sbl-mips.bin*** на локальной машине (о том как собрать SBL из исходных кодов я расскажу ниже). Для этого выполним следующую команду:

```
rz@mustodon:/home/rz/mcom03 $ mcom03-flash -v -p /dev/ttyU0 flash qspi0
sbl-mips.bin
```

В моём случае ***/dev/ttyU0*** это последовательное устройство, созданное конвертором USB2Serial, к которому подключен порт UART0 СнК. Параметр ***-v*** включает режим «verbose», т.е. утилита будет многословно сообщать нам о ходе процесса записи во flash память.

Вот как выглядит сам процесс записи:

```
Uploading flasher to on-chip RAM...

RnD Center "ELVEES", JSC
MCom-03, Bootrom rev.1.0
#Sending flasher...

#9568 bytes


#run
Starting 0x80000000...
QSPI Flasher
#qspi 0
Selected QSPI0
#custom 0x9f 6
ef 40 16 00 00 00
#Found W25Q32 memory on QSPI0
Flash size: 4 MiB, erase sector: 64 KiB, page: 256 B
Erasing 644508 bytes, rounded to 655360 bytes (10 sectors, starting from
0)...
[                    ]   0.0%erase 0
OK
[██                  ]  10.0%erase 65536
OK
[████                ]  20.0%erase 131072
OK
[██████              ]  30.0%erase 196608
OK
[████████            ]  40.0%erase 262144
OK
[██████████          ]  50.0%erase 327680
OK
[████████████        ]  60.0%erase 393216
```

```
OK
[████████████       ]  70.0%erase 458752
OK
[████████████████   ]  80.0%erase 524288
OK
[████████████████████ ]  90.0%erase 589824
OK
Erase: 1.9 s (323 KiB/s)
Writing to flash 629.40 KB...
write 0
Ready for data

#Write: 80.5 s (8 KiB/s)
Checking...
readcrc 0 644508
0x6ec6
#Check: 1.9 s (334 KiB/s)
Total: 84.3 s
```

Утилита ***mcom03-flash*** закончила своё выполнение без ошибок, запись во flash память и последующая проверка CRC выполнена успешно.

12. Загрузка SBL и U-Boot

На предыдущем этапе мы собрали и «прошили» содержимое файла ***sbl-mips.bin*** во flash память подключенной к порту QSPI0 СнК «СКИФ».

Для того чтобы инициировать процесс загрузки с SQPI0 flash, необходимо обесточить процессорный модуль и перепаять перемычки ***BOOT[2:0]***, для задания режима ***0b000***. После подачи питания, если не возникло никаких проблем с инициализацией аппаратуры, мы увидим следующую картину:

```
rz@mustodon:/home/rz $ cu -l /dev/ttyU0 -s 115200
Connected

DDRMC0: Initialized successfully within 2082649 us, 1 ranks, speed 3200
MT/s
Ranks:                                                  1
Die size:                                               -230055451
Rank size:                                              2048 MB
Full size:                                              2048 MB
Full SDRAM Width:                                       146 bits
Primary SDRAM Width:                                    121 bits
```

```
ECC SDRAM Width:                                  128 bits
Device Width:                                     16 bits
Number of Row Addresses:                          65536
Number of Column Addresses:                       1024
Number of Bank Addresses:                         8
Bank Group Addressing:                            1
Clock Cycle Time (tCK):                           625 ps
Minimum Clock Cycle Time (tCK min):               4457 ps
Maximum Clock Cycle Time (tCK max):               1602 ps
CAS# Latency Time (tAA min):                      12500 ps
Minimum Four Activate Windows Delay (tFAW):       40000 ps
RAS to CAS Delay Time (tRCD min):                 30000 ps
Row Precharge Delay Time (tRP min):               101875 ps
Active to Precharge Delay Time (tRAS min):        42500 ps
Normal Refresh Recovery Delay Time (tRFC1 min):   35390000 ps
2x Mode Refresh Recovery Delay Time (tRFC2 min):  30196875 ps
4x Mode Refresh Recovery Delay Time (tRFC4 min):  30936250 ps
Short Row Active to Row Active Delay (tRRD_S min):  57500 ps
Long Row Active to Row Active Delay (tRRD_L min):  144375 ps
Long CAS to CAS Delay Time (tCCD_L min):          20000 ps
Minimum Active to Auto-Refresh Delay (tRC):       63750 ps

DDRMC1: Failed to get configuration: Failed to configure DIMM
Total DDR memory size 2048 MiB
Memory interleaving: disabled

U-Boot 2021.01-ge8f2ec33ac (Feb 07 2022 - 21:22:13 +0000)

Model: MCom-03 Bring-Up Board r1.3.0
DRAM:  1 GiB
MMC:   sdhci0@10220000: 0, sdhci1@10230000: 1
Loading Environment from SPIFlash... jedec_spi_nor s25fl128s@0:
unrecognized JEDEC id bytes: ef, 40, 16
*** Warning - spi_flash_probe_bus_cs() failed, using default environment

In:    serial0@1750000
Out:   serial0@1750000
Err:   serial0@1750000

Warning: ethernet@10200000 (eth0) using random MAC address -
5a:91:76:00:a1:be
eth0: ethernet@10200000
Warning: ethernet@10210000 (eth1) using random MAC address -
7e:52:8c:db:29:1b
eth1: ethernet@10210000
Hit any key to stop autoboot:  0
```

Из выше приведенного лога интерес представляет следующее:

1) Контроллер динамической памяти DDRMC0 успешно провел тестирование (трейнинг) микросхемы памяти и готов работать на скорости 3200 MT/s, о чем свидетельствует следующее сообщение:

```
DDRMC0: Initialized successfully within 2082649 us, 1 ranks, speed 3200
MT/s
```

Сразу скажу, что на такой скорости микросхема LPDDR4 памяти в нашем процессорном модуле работает нестабильно и скорость пришлось занизить до 1600 MT/s. С причинами мы сейчас разбираемся.

2) Контроллер динамической памяти DDRMC1 ничего не обнаружил, что совершенно справедливо, так как в нашем модуле использована только одна микросхема памяти и один контроллер.

```
DDRMC1: Failed to get configuration: Failed to configure DIMM
```

3) Исполнение дошло до вторичного загрузчика U-Boot, который успешно стартовал, обнаружил **1 ГБ** динамической памяти и ряд периферии, в том числе QSPI flash, MMC0, MMC1, MAC0 и MAC1:

```
U-Boot 2021.01-ge8f2ec33ac (Feb 07 2022 - 21:22:13 +0000)

Model: MCom-03 Bring-Up Board r1.3.0
DRAM:  1 GiB
MMC:   sdhci0@10220000: 0, sdhci1@10230000: 1
...
Warning: ethernet@10200000 (eth0) using random MAC address -
5a:91:76:00:a1:be
eth0: ethernet@10200000
Warning: ethernet@10210000 (eth1) using random MAC address -
7e:52:8c:db:29:1b
eth1: ethernet@10210000
```

Почему U-Boot видит только 1ГБ оперативной памяти, вместо имеющихся 2ГБ — вопрос интересный. Мы задали его в службу технической поддержки «Элвиса» и получили ответ примерно следующего содержания: «есть проблема, над её разрешением работают». Проблема, как я понимаю, связана с DDRMC контроллером, по этому в TF-A временно встроено искусственное ограничение. Проблему обещали разрешить программным способом — перепрошивкой ПО в DDRMC.

4) U-Boot готов к поиску и загрузке ядра ОС Linux.

На самом деле при первом запуске SBL у нас все было не так радужно.

Во-первых, выяснилось, что DDRINIT не поддерживал память формата LPDDR4, соответственно не мог детектировать микросхему памяти MT53D512M32D2DS, установленную на нашем процессорном модуле и не мог проводить трейнинг DDR трансивера. Эта проблема решилась после общения с технической поддержкой — нам прислали патч для DDRINIT из двух строк:

```
diff --git a/src/phy/phy-lpddr4.c b/src/phy/phy-lpddr4.c
index c9fffe2..8fa89c4 100644
--- a/src/phy/phy-lpddr4.c
+++ b/src/phy/phy-lpddr4.c
@@ -273,9 +273,9 @@ void phy_training_params_load(int ctrl_id, struct
ddr_cfg *cfg)

-       params.CsPresentChA = 3;
+       params.CsPresentChA = (CONFIG_DRAM_RANKS == 2) ? 3 : 1;
        params.EnabledDQsChB = 16;
-       params.CsPresentChB = 3;
+       params.CsPresentChB = (CONFIG_DRAM_RANKS == 2) ? 3 : 1;
```

Во-вторых, обнаружилось, что после загрузки U-Boot, СнК потребляет достаточно большой ток и сильно нагревается без радиатора. Проблему решил патч для U-Boot, который останавливает и выключает незадействованные аппаратные блоки, в частности, SDR и два DSP ядра.

В-третьих, как я уже отметил, хоть трейнинг памяти на скорости 3200 MT/s и заканчивался успешно, но дальнейшая загрузка ядра ОС Linux почти в половине случаев застревала при инициализации и запуске дополнительных ARM ядер (напомню, в данном СнК всего четыре ARM ядра, в процессе загрузки используется только одно — CPU0). Опытным путем было установлено, что память работает стабильно на скорости 1600 MT/s. На этой скорости успешно стартует ОС AltLinux и проходит ряд нагрузочных тестов, но об этом позже.

13. Подготавливаемся к загрузке ОС AltLinux

Следующим логичным шагом является загрузка ОС общего назначения — Linux (AltLinux в нашем случае). Чтобы было откуда загружать AltLinux, мы смонтировали наш процессорный модуль «RanetkaPC» на несущую плату изделия «ControlPanel», посадочное место на которой было изначально предназначено для другого нашего процессорного модуля — «Module A20» на базе СнК Allwinner A20. В процессе работы над модулем «RanetkaPC» мы постарались сделать его совместимым, насколько это представлялось возможным, с модулем «Module A20» в рамках доступной нам периферии внутри СнК «СКИФ». На рис.1 вначале статьи как раз приведена фотография такой конструкции. В результате у нас появился доступ к слоту SD-карты, USB порту и прочей интересной периферии, часть из которой до сих пор находится в стадии тестирования и подгонки драйверов.

Служба технической поддержки «Элвиса» выдала нам ссылку на скачивание образа SD карты с установленной ОС AltLinux — вот такой файл:

```
-rw-r--r--    1 rz rz 2807212980 Jan 26 13:15 mcom03-altlinux-sdcard-2021-09-08.bz2
```

Ссылки на образы и документацию приведены в конце данной статьи.

Образ AltLinux был успешно развернут на SD-карту размером 16ГБ с помощью команды ***dd***:

```
$ sudo dd bs=8192 if=mcom03-altlinux-sdcard-2021-09-08.bz2 of=/dev/sdb status=progress
```

Перед тем как осуществить загрузку, нам потребовалось выполнить еще одну важную процедуру — выполнить сборку ядра ОС Linux версии ***4.19.106-mcom03*** из исходных кодов и создание адаптированного под наш процессорный модуль конфигурационного файла Device-Tree (DTB). Без правильного DTB в процессе загрузки ядро Linux-а может натолкнуться на «непреодолимые препятствия».

Сборка ядра Linux выполняется достаточно конвенциально, в несколько шагов:

1) Клонируем исходный код ядра Linux адаптированного для микросхемы MCOM-03 из репозитория «Элвиса» на Github-а:

```
$ git clone -b mcom03 --depth 1 https://github.com/elvees/linux.git linux-mcom03
```

2) Конфигурируем и собираем ядро Linux:

```
$ . /opt/setenv_gcc8_arm64.sh
$ cd linnux-mcom03
$ make ARCH=arm64 mcom03_defconfig
$ make ARCH=arm64
```

3) Создаем DTS файл для нашего модуля «RanetkaPC» следующего содержания:

```
rz@devbox:~/linux-kernels/linux-mcom03$ cat arch/arm64/boot/dts/elvees/mcom03-RanetkaPC_Module-1.0.dts

// SPDX-License-Identifier: GPL-2.0+
/*
 * Copyright 2021 RnD Center "ELVEES", JSC
 */

/dts-v1/;

#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/pwm/pwm.h>

#include "mcom03.dtsi"

/ {     model = "MCom-03 RanetkaPC_Module ver 1.0";
        compatible = "elvees,mcom03-bub-r1.3.0";

        chosen {
                stdout-path = "serial0:115200n8";
                bootargs = "earlycon console=ttyS0,115200";
        };

        sdhci1_supply: regulator@0 {
                compatible = "regulator-fixed";
                regulator-name = "sdhci1-supply";
                regulator-min-microvolt = <1800000>;
                regulator-max-microvolt = <1800000>;
                regulator-always-on;
        };

        rgb_panel_supply: regulator@1 {
```

```
                compatible = "regulator-fixed";
                regulator-name = "rgb-panel-supply";
                regulator-min-microvolt = <3300000>;
                regulator-max-microvolt = <3300000>;
                regulator-always-on;
        };

        rgb_panel_backlight: rgb-panel-backlight {
                compatible = "gpio-backlight";
                gpios = <&gpio1d 2 GPIO_ACTIVE_HIGH>;
                default-on;
        };

        rgb_panel: panel@0 {
                compatible = "mi,mi0700s6t";
                backlight = <&rgb_panel_backlight>;
                power-supply = <&rgb_panel_supply>;
                status = "okay";

                port {
                        rgb_panel_in: endpoint {
                                remote-endpoint = <&rgb_panel_out>;
                        };
                };
        };
};

&arch_timer {
        clock-frequency = <27456000>;
};

&xti_clk {
        clock-frequency = <27456000>;
};


&cpu_clk {
        clock-frequency = <1400256000>;
};

&uart0 {
        status = "okay";
};

&uart1 {
        status = "okay";
};

/* Wired to PBD connector */
&uart2 {
        status = "okay";
};

&sdhci1 {
```

```
        status = "okay";

        /* Disable High Speed support
         * Disable 1.8V support
         * Disable SDR50, SDR104 and DDR50 support
         */
        sdhci-caps-mask = <0x7 0x4200000>;
        sdhci-caps = <0x0 0x0>;
        disable-wp;
};

&sdhci0 {
        status = "okay";
        non-removable;
        bus-width = <8>;
        vqmmc-supply = <&sdhci1_supply>;
        disable-wp;
};

&emac0 {
        status = "okay";
        phy-handle = <&ethernet_phy0>;
        reset-gpios = <&gpio0c 6 GPIO_ACTIVE_LOW>;

        /* Use large delays because 10 uF capacitor is connected to reset pin.
         */
        reset-delay-us = <1000>;
        reset-post-delay-us = <200000>;

        ethernet_phy0: ethernet-phy@1 {
                reg = <0x1>;

                /* This is max GTX_CLK skew (0b11111 * 60ps/step)
                 * for KSZ9031 Ethernet PHY
                 */
                txc-skew-ps = <1860>;
                interrupt-parent = <&gpio1a>;
                interrupts = <6 IRQ_TYPE_LEVEL_LOW>;
        };
};

&usb0 {
        status = "okay";
};

&usb1 {
        status = "okay";
};

&qspi {
        status = "okay";
};

&i2c0 {
```

```
        clock-frequency = <400000>;
        status = "okay";

        pinctrl-0 = <&i2c0_mux_data &i2c0_mux_smbalert &i2c0_mux_smbus>;
};

&i2c1 {
        clock-frequency = <400000>;
        status = "okay";
};

&i2c2 {
        clock-frequency = <400000>;
        status = "okay";
};

&i2c3 {
        clock-frequency = <400000>;
        status = "okay";
};

&dp {
        status = "okay";
};


&media {
};

&display_encoder {
        status = "okay";
        video-output = "rgb";

        panel = <&rgb_panel>;

        ports {
                port@2 {
                        reg = <2>;

                        rgb_panel_out: endpoint {
                                remote-endpoint = <&rgb_panel_in>;
                        };
                };
        };

};

&qlic0 {
        status = "disable";
};

&elcore0 {
        status = "disable";
};
```

```
&elcore1 {
        status = "disable";
};

&pcie0 {
        status = "okay";
};

&pcie1 {
        status = "disabled";
};

&isp {
        status = "okay";
};
```

4) Выполняем компиляцию DTS в DTB:

```
$ make ARCH=arm64 dtbs
```

5) Копируем результирующие файлы в раздел boot на SD-карту с AltLinux-ом:

```
$ sudo mount -t vfat /dev/sdb1 /mnt
$ sudo cp arch/arm64/boot/Image /mnt
$ sudo cp arch/arm64/boot/dts/elvees/mcom03-RanetkaPC_Module-1.0.dtb /mnt
$ sudo umount /mnt
```

На этом с подготовительными работами закончено и можно приступать к загрузке ОС AltLinux.

14. Загрузка ОС AltLinux

Вставим SD-карту в слот и перезапустим модуль «по питанию». Напомню, что СнК находится в режиме загрузки ***0b000*** — «Загрузка напрямую через QSPI0», т.е. происходит исполнение кода SBL прямо из NOR flash на порту QSPI0.

Дождемся пока исполнение перейдет к U-Boot и в терминале появится сообщения:

```
Hit any key to stop autoboot:  3
```

Нажмем любую клавишу, чтобы U-Boot не приступил к исполнению «вшитого» в него при компиляции скрипта и попадем в командную строку U-Boot:

```
Hit any key to stop autoboot:  0
=>
```

Наша задача состоит в том, чтобы последовательно загрузить в память ядро ОС Linux и файл конфигурации (.dtb) с SD-карты и передать управление ядру с нужными параметрами командной строки.

1) Выясним, работает ли у нас SD/MMC интерфейс, для этого воспользуемся командами ***mmc list***, ***mmc dev*** и ***mmc info***:

```
=> mmc list
sdhci0@10220000: 0 (eMMC)
sdhci1@10230000: 1 (SD)

=> mmc dev 0
switch to partitions #0, OK
mmc0(part 0) is current device

=> mmc info
Device: sdhci0@10220000
Manufacturer ID: 45
OEM: 100
Name: DG400
Bus Speed: 25000000
Mode: MMC legacy
Rd Block Len: 512
MMC version 5.1
High Capacity: Yes
Capacity: 7.3 GiB
Bus Width: 4-bit
Erase Group Size: 512 KiB
HC WP Group Size: 8 MiB
User Capacity: 7.3 GiB WRREL
Boot Capacity: 4 MiB ENH
RPMB Capacity: 4 MiB ENH
Boot area 0 is not write protected
```

```
Boot area 1 is not write protected

=> mmc dev 1
switch to partitions #0, OK
mmc1 is current device
=> mmc info
Device: sdhci1@10230000
Manufacturer ID: 27
OEM: 5048
Name: SD16G
Bus Speed: 50000000
Mode: SD High Speed (50MHz)
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 14.5 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
```

Видно, что U-Boot обнаружил два SD/MMC устройства ***mmc 0*** и ***mmc 1***. Нулевое — это запаянная на процессорный модуль микросхема eMMC (DG400 объёмом 7.3 GiB), а первое — вставленная в слот SD-карта (SD16G фактическим объёмом 14.5 GiB).

2) Командой ***fatls*** просмотрим содержимое загрузочного раздела на SD карте который представляет собой FAT32 файловую систему:

```
=> fatls mmc 1
            alt/
            extlinux/
 13936648   Image
  6291456   vmlinux
    26277   mcom03-RanetkaPC_Module-1.0.dtb

3 file(s), 2 dir(s)
```

Здесь у нас располагается ядро ОС Linux (файл ***Image***), конфигурационный файл для ядра (файл ***mcom03-RanetkaPC_Module-1.0.dtb***), а так же ряд других файлов и подкаталогов которые находились на развернутом образе. Нам потребуются только первые два файла.

3) Загрузим в оперативную память с адреса ***0x82000000*** ядро ОС Linux с помощью команды ***fatload***:

```
=> fatload mmc 1 0x82000000 Image
```

```
13936648 bytes read in 975 ms (13.6 MiB/s)
```

4) Загрузим в оперативную память с адреса ***0x88800000*** файл конфигурации ядра (Device-Tree):

```
=> fatload mmc 1 0x88800000 mcom03-RanetkaPC_Module-1.0.dtb
26277 bytes read in 16 ms (1.6 MiB/s)
```

5) Установим переменную окружения U-Boot отвечающую за параметры командной строки Linux ядра:

```
=> setenv bootargs earlycon console=ttyS0,115200 root=/dev/mmcblk1p2 rw
rootwait initcall_debug=1 drm.debug=0x1f loglevel=8 video=DPI-1:800x480
```

Здесь мы указываем ядру слудующее:

- ***root*** - монтировать rootfs с устройства ***mmcblk1p2***, т.е. второй раздел на MMC1;
- ***rootwait*** - перед монтированием rootfs нужно дождаться готовности устройств;
- включаем максимум диагностических сообщений, в том числе сообщение на каждый вызов процедур внутри драйверов;
- **earlycon** - инструктируем ядро о том, что вывод диагностических сообщений нужно осуществлять специальным способом — работая с «железом» напрямую, а не через драйвер, иначе часть сообщений будет утеряна.

6) Используя команду ***booti*** передадим микрофон ядру ОС Linux:

```
=> booti 0x82000000 - 0x88800000
Moving Image from 0x82000000 to 0x82080000, end=82e23000
## Flattened Device Tree blob at 88800000
   Booting using the fdt blob at 0x88800000
   Loading Device Tree to 00000000bf745000, end 00000000bf74e6a4 ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
```

```
[    0.000000] Linux version 4.19.106-g54f4366ee-dirty (rz@devbox) (gcc
version 8.2.1 20180802 (GNU Toolchain for the A-profile Architecture 8.2-
2018-08 (arm-rel-8.23))) #20 SMP Fri Feb 18 15:38:50 UTC 2022
[    0.000000] Machine model: MCom-03 RanetkaPC_Module ver 1.0
[    0.000000] earlycon: uart0 at MMIO32 0x0000000001750000 (options
'115200n8')
[    0.000000] bootconsole [uart0] enabled
[    0.000000] cma: Reserved 128 MiB at 0x00000000b7400000
[    0.000000] On node 0 totalpages: 262144
[    0.000000]   DMA32 zone: 4096 pages used for memmap
[    0.000000]   DMA32 zone: 0 pages reserved
[    0.000000]   DMA32 zone: 262144 pages, LIFO batch:63
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv1.1 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: MIGRATE_INFO_TYPE not supported.
[    0.000000] psci: SMC Calling Convention v1.1
[    0.000000] random: get_random_bytes called from
start_kernel+0xa0/0x44c with crng_init=0
[    0.000000] percpu: Embedded 22 pages/cpu s51672 r8192 d30248 u90112
[    0.000000] pcpu-alloc: s51672 r8192 d30248 u90112 alloc=22*4096
[    0.000000] pcpu-alloc: [0] 0 [0] 1 [0] 2 [0] 3
[    0.000000] Detected VIPT I-cache on CPU0
[    0.000000] Speculative Store Bypass Disable mitigation not required
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages:
258048
[    0.000000] Kernel command line: earlycon console=ttyS0,115200
root=/dev/mmcblk1p2 rw rootwait initcall_debug=1 drm.debug=0x1f loglevel=8
video=DPI-1:800x480
[    0.000000] Dentry cache hash table entries: 131072 (order: 8, 1048576
bytes)
[    0.000000] Inode-cache hash table entries: 65536 (order: 7, 524288
bytes)
[    0.000000] Memory: 885048K/1048576K available (8638K kernel code,
1194K rwdata, 2948K rodata, 768K init, 329K bss, 32456K reserved, 131072K
cma-reserved)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
[    0.000000] ftrace: allocating 29486 entries in 116 pages
[    0.000000] rcu: Hierarchical RCU implementation.
[    0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
[    0.000000] GICv3: GIC: Using split EOI/Deactivate mode
[    0.000000] GICv3: Distributor has no Range Selector support
[    0.000000] GICv3: no VLPI support, no direct LPI support
[    0.000000] ITS: No ITS available, not enabling LPIs
[    0.000000] GICv3: CPU0: found redistributor 0 region
0:0x0000000001180000
[    0.000000] arch_timer: cp15 timer(s) running at 27.45MHz (phys).
[    0.000000] clocksource: arch_sys_counter: mask: 0xffffffffffffff
max_cycles: 0x6550ab484, max_idle_ns: 440795202486 ns
[    0.000003] sched_clock: 56 bits at 27MHz, resolution 36ns, wraps every
4398046511095ns
...
[    0.334934] smp: Bringing up secondary CPUs ...
[    0.350055] Detected VIPT I-cache on CPU1
```

```
[    0.350087] GICv3: CPU1: found redistributor 1 region
0:0x00000000011a0000
[    0.350121] CPU1: Booted secondary processor 0x0000000001 [0x410fd034]
[    0.360435] Detected VIPT I-cache on CPU2
[    0.364408] GICv3: CPU2: found redistributor 2 region
0:0x00000000011c0000
[    0.364423] CPU2: Booted secondary processor 0x0000000002 [0x410fd034]
[    0.374687] Detected VIPT I-cache on CPU3
[    0.381794] GICv3: CPU3: found redistributor 3 region
0:0x00000000011e0000
[    0.381809] CPU3: Booted secondary processor 0x0000000003 [0x410fd034]
[    0.381872] smp: Brought up 1 node, 4 CPUs
[    0.430510] SMP: Total of 4 processors activated.
[    0.435544] CPU features: detected: GIC system register CPU interface
[    0.442447] CPU features: detected: 32-bit EL0 Support
[    0.448226] CPU: All CPU(s) started at EL2
...
[    2.564194] EXT4-fs (mmcblk1p2): recovery complete
[    2.568931] EXT4-fs (mmcblk1p2): mounted filesystem with ordered data
mode. Opts: (null)
[    2.576900] VFS: Mounted root (ext4 filesystem) on device 179:18.
[    2.599291] devtmpfs: mounted
[    2.603911] Freeing unused kernel memory: 768K
[    2.608387] Run /sbin/init as init process
[    2.935442] random: fast init done
[    3.568437] drm_mcom03_mode_valid: requested_rate = 33000000, real_rate
= 33000000
[    3.576054] drm_mcom03_mode_valid: requested_rate = 33000000, real_rate
= 33000000
[    3.583600] drm_mcom03_mode_set: type = 0, cmos0_clk_clock = 148500000,
mode->clock * 1000 = 33000000, ret = 0
[    3.593498] drm_mcom03_enable: type = 0, cmos0_clk_clock = 148500000
[    4.026070] systemd[1]: System time before build time, advancing clock.
[    4.045788] systemd[1]: Failed to find module 'autofs4'
[    4.181753] systemd[1]: systemd v246.14-alt1 running in system mode.
(+PAM +AUDIT +SELINUX +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP
+GCRYPT +GNUTLS +ACL +XZ +LZ4 +ZSTD +SECCOMP +BLKID +ELFUTILS +KMOD +IDN2
-IDN +PCRE2 default-hierarchy=hybrid)
[    4.204397] systemd[1]: Detected architecture arm64.
...
Welcome to Simply Linux 9.1 (Destiny)!
altlinux login:
```

Ура товарищи!

Не будут сильно вдаваться в процесс загрузки ОС Linux, приведу лишь некоторые ключевые момент:

1) Перед передачей управления ядру ОС Linux, U-Boot осуществил релокацию (перенос) образа ядра с адреса загрузки ***0x82000000*** на адрес ***0x82008000***. Для этого есть несколько причин: во-первых, требуется выравнять

код по границе 2БМ из-за особенностей работы виртуальной памяти, во-вторых, добавлено смещение 0x80000 указанное в заголовке ядра.

2) U-Boot распаковал загруженный DTB (Flattened Device-Tree в бинарном формате) в полноценную структуру данных и разместил её почти в самый конец доступной оперативной памяти, по адресу ***0x00000000bf745000*** (напоминаю, что на данный момент нам доступен только 1ГБ ОЗУ).

3) Запуск ядра ОС Linux осуществляется на одном вычислительном ядре (CPU0). На ранних стадиях инициализации аппаратуры, ядро ОС Linux проводит поиск и запуск остальных трех вычислительных ядер. Результат мы видим в следующих сообщениях:

```
[    0.430510] SMP: Total of 4 processors activated.
[    0.448226] CPU: All CPU(s) started at EL2
```

4) ОС Linux на всех четырех вычислительных ядра работает в режиме Execution Level 2 (EL2). Это говорит о том, что над ядром ОС общего назначение находится супервизор, в данном случае TF-A, работающий в режиме исполнения EL3.

5) Ядро ОС Linux успешно осуществило монтирование корневой файловой системы с устройства ***mmcblk1p2***, которое содержит файловую систему типа EXT4FS.

```
[    2.564194] EXT4-fs (mmcblk1p2): recovery complete
[    2.568931] EXT4-fs (mmcblk1p2): mounted filesystem with ordered data
mode. Opts: (null)
[    2.576900] VFS: Mounted root (ext4 filesystem) on device 179:18.
```

6) Процесс ***/sbin/init*** успешно стартовал и начал исполнение ***rc*** скриптов, в том числе запустился системный демон ***systemd***.

7) Процесс ***login*** запущен и мы видим приглашение к входу в систему.

15. Просмотр основных параметров системы

Войдем в систему под пользователем **altlinux** (с пустым паролем), после чего перейдем в режим супер-пользователя командой **su -** (символ «минус» обязателен при работе в AltLinux) и немного осмотримся:

```
[root@altlinux ~]# cat /proc/cpuinfo
processor   : 0
BogoMIPS    : 54.91
Features    : fp asimd crc32 cpuid
CPU implementer   : 0x41
CPU architecture: 8
CPU variant : 0x0
CPU part    : 0xd03
CPU revision      : 4

processor   : 1
BogoMIPS    : 54.91
Features    : fp asimd crc32 cpuid
CPU implementer   : 0x41
CPU architecture: 8
CPU variant : 0x0
CPU part    : 0xd03
CPU revision      : 4

processor   : 2
BogoMIPS    : 54.91
Features    : fp asimd crc32 cpuid
CPU implementer   : 0x41
CPU architecture: 8
CPU variant : 0x0
CPU part    : 0xd03
CPU revision      : 4

processor   : 3
BogoMIPS    : 54.91
Features    : fp asimd crc32 cpuid
CPU implementer   : 0x41
CPU architecture: 8
CPU variant : 0x0
CPU part    : 0xd03
CPU revision      : 4
```

Видно, что операционной системе доступно 4 вычислительных ядра ARMv8 с поддержкой Advanced SIMD, блоком вычислений с плавающей точкой (FP) и аппаратным вычислителем CRC. Производительность каждого ядра оценена в **54.91 BogoMIPS**, но об этом чуть позже.

Запустим утилиту ***top*** и посмотрим сколько у нас имеется оперативной памяти и как она задействована:

```
[root@altlinux ~]# top
top - 18:14:55 up 4 min,  2 users,  load average: 1,11, 1,95, 0,94
Tasks: 150 total,   1 running, 149 sleeping,   0 stopped,   0 zombie
%CPU(s):  0,2 us,  0,0 sy,  0,0 ni, 99,8 id,  0,0 wa,  0,0 hi,  0,1 si,  0,0 st
MiB Mem :    993,1 total,    347,2 free,    324,2 used,    321,6 buff/cache
MiB Swap:      0,0 total,      0,0 free,      0,0 used.    593,2 avail Mem
```

Выясним, что за дистрибутив ОС Linux нам выдали:

```
root@altlinux ~]# cat /etc/os-release
NAME="Simply Linux"
VERSION="9.1 (Destiny)"
ID=altlinux
VERSION_ID=9.1
PRETTY_NAME="Simply Linux 9.1 (Destiny)"
ANSI_COLOR="1;36"
CPE_NAME="cpe:/o:alt:slinux:9.1"
HOME_URL="https://www.basealt.ru/products/simplylinux/"
BUG_REPORT_URL="https://bugs.altlinux.org/"
```

```
[root@altlinux ~]# ps ax | grep xfce
    433 ?        Ssl    0:01 xfce4-session
    568 ?        Sl     0:00 /usr/lib64/xfce4/xfconf/xfconfd
    857 ?        Sl     0:00 xfce4-panel --display :0.0 --sm-client-id
2b22d59c6
    871 ?        Sl     0:00 /usr/lib64/xfce4/panel/wrapper-2.0
/usr/lib64/xfce4
    876 ?        Sl     0:00 /usr/lib64/xfce4/panel/wrapper-2.0
/usr/lib64/xfce4
    877 ?        Sl     0:03 /usr/lib64/xfce4/panel/wrapper-2.0
/usr/lib64/xfce4
    878 ?        Sl     0:00 /usr/lib64/xfce4/panel/wrapper-2.0
/usr/lib64/xfce4
    879 ?        Sl     0:00 /usr/lib64/xfce4/panel/wrapper-2.0
/usr/lib64/xfce4
    880 ?        Sl     0:00 /usr/lib64/xfce4/panel/wrapper-2.0
/usr/lib64/xfce4
    881 ?        Sl     0:00 /usr/lib64/xfce4/panel/wrapper-2.0
/usr/lib64/xfce4
    910 ?        Ssl    0:00 xfce4-power-manager --restart --sm-client-id
2e896e
    939 ?        Ssl    0:00 /usr/lib64/xfce4/notifyd/xfce4-notifyd
```

Видно, что у нас полноценный десктопный вариант AltLinux-а с запущенным **xfce4** в качестве оконного менеджера. Из этого так же следует, что Xorg обнаружил и задействововал встроенный видеоадаптер.

Посмотрим в файла ***/var/log/Xorg.0.lo***g и выясним, что у нас запущен Xorg 1.20.8, в качестве дисплейного драйвера использован modesetting (универсальный драйвер) без поддержки OpenGL:

```
[    31.453] (II) modeset(0): Refusing to try glamor on llvmpipe
[    31.500] (EE) modeset(0): glamor initialization failed
[    31.500] (II) modeset(0): ShadowFB: preferred NO, enabled NO
[    31.509] (II) modeset(0): Output DPI-1 has no monitor section
[    31.517] (II) modeset(0): EDID for output DPI-1
[    31.517] (II) modeset(0): Printing probed modes for output DPI-1
[    31.517] (II) modeset(0): Modeline "800x480"x59.6   33.00  800 979
1009 1055  480 501 502 525 -hsync (31.3 kHz eP)
[    31.517] (II) modeset(0): Output DPI-1 connected
[    31.517] (II) modeset(0): Using exact sizes for initial modes
[    31.517] (II) modeset(0): Output DPI-1 using initial mode 800x480 +0+0
[    31.517] (==) modeset(0): Using gamma correction (1.0, 1.0, 1.0)
[    31.517] (==) modeset(0): DPI set to (96, 96)
```

К сожалению, нам не удалось подключить LCD дисплей на порт DPI нашего модуля, чтобы опробовать все прелести работы в графической среде, так как обнаружилась несовместимость по уровням сигналов: СнК «СКИФ» выдает сигналы напряжением 1.8V, в то время как все имеющиеся у нас LCD принимают сигнал минимум от 3.0V. Очевидно, что найти дисплей с уровнями сигналов 1.8V будет крайне не просто, а значит потребуется ввести в схему модуля микросхемы преобразователя уровней, что скажется на максимальной частоте развертки и вообще усложнит и удорожает схему включения LCD.

Изменить уровни выходных сигналов для DPI не представляется возможным, это стало понятно после общения со службой технической поддержки «Элвиса». Честно говоря, я был крайне разочарован таким решением инженеров «Элвиса».

В СнК «СКИФ» имеется четыре датчика температуры, по одному на каждую из подсистем. Попробуем прочитать их текущие значения:

```
[root@altlinux ~]# cat /sys/devices/platform/*/*.pvt/hwmon/hwmon*/temp*
54987
cpu
54465
sdr
54639
media
54987
service
```

Видно, что температура микросхемы составляет около 54.8°C. При этом на микросхеме установлен небольшой алюминиевый радиатор. В целом, это допустимо, при том что в данном случае, ARM ядра СнК работают на частоте **1.4ГГц** и питании **0.9V**.

Попутно выяснили, что в процессе работы ОС Linux наш модуль «RanetkaPC» потребляет ток до **0.65A** по линии питания **5V**, итого **3.25W**.

16. Проблема с архитектурным таймером

Теперь вернемся к BogoMIPS-ам. Изначально, до версии Linux 3.6, параметр BogoMIPS использовался операционной системой для внутренних циклов задержки (delay loops), для расчета минимального кванта времени используемого при переключении контекста (jiffies), а так же для еще ряда служебных нужд. BogoMIPS рассчитывался (калибровался) каждый раз при загрузке операционной системы, как количество холостых циклов процессора за один квант времени и выражался в «миллионах циклов процессора в секунду». Разумеется, такой подсчет не является точным с научной точки зрения и поэтому имеет в названии приставку **Bogo** от английского слова «bogus» (кривой, убогий). Придумал этот параметр, кстати, сам Линус Торвальдс в 1993 году.

Однако с появлением технологий динамического изменения частоты вычислительного ядра, а так же технологии big.LITTLE, использование такого параметра, как BogoMIPS, стало наталкиваться на проблемы, и с версии Linux 3.6 начали использовать другой параметры — **lpj** или **loops_per_jiffies**. Расчет **lpj** так же производится при загрузки системы, но не полагаясь на производительность вычислительного ядра, а на частоту системного таймера,

частота которого всегда постоянна и, зачастую, заранее известна. Параметр BogoMIPS остался, но он теперь почти не связан с количеством циклов процессора, а является своего рода индексом (показателем) производительности вычислительного ядра на максимальной тактовой частоте.

Более подробно об этой особенности ОС Linux можно прочитать в статье на Wikipedia: https://en.wikipedia.org/wiki/BogoMips

Чтобы правильно рассчитать BogoMIPS (а точнее уже lpj — loops_per_jiffies, так как мы используем ядро 4.19), системе требуется такая штука, которая называется «Архитектурный Таймер». Архитектурный таймер, в ядре ОС Linux называется **arch_timer,** это аппаратный таймер внутри микросхемы СнК или микропроцессора, частота которого заранее известна, стабильна (не зависит от режимов работы вычислительных ядер) и существенно ниже самой низкой тактовой частоты процессора, при этом счетчик таймера постоянно и монотонно нарастает и не переполняется.

Архитектурный таймер участвует не только в переключении контекста, но и для выдачи системного времени от начала запуска вычислительного ядра, того самого, которое возвращает системный вызов ***gettimeofday()***, а так же выводит утилита ***date***, разумеется, после корректировки смещения полученного от RTC (часов реального времени) или по протоколу NTP от сервера точного времени. Иными словами, архитектурный таймер является опорой всей операционной системы.

И тут нас ожидал интересный подвох. Я обратили внимание на то, что параметр **lpj**, который отображается при загрузке ядра ОС Linux, несколько отличался от того, что следовало бы ожидать. Дело в том, что в нашем процессорном модуле в качестве источника опорной частоты использован кварцевый резонатор на частоту **27.456МГц**, в точном соответствии с «Руководством» на микросхему СнК «СКИФ». Однако, в логе мы наблюдали следующее:

```
[root@altlinux ~]# dmesg | grep MIPS
[    0.013755] Calibrating delay loop (skipped), value calculated using
timer frequency.. 54.00 BogoMIPS (lpj=270000)
```

Это сразу навело нас на мысль о том, что архитектурный таймер либо неверно настроен, либо ядру операционной системы сообщили неверную

частоту этого таймера. А это может создавать ряд проблем, одна из которых — неправильное вычисление системного времени, а значит у нас будут проблемы с корректным измерением производительности системы, т.е. различные тесты (бенчмарки) просто будут неверны. Это предположение тут же подтвердилось простым замером времени по команде ***date*** и сравнению его с «настенными» часами — системные часы «спешили» примерно на 1 секунду за минуту (или на 1:01 минуту за час), что соответствует разнице в опорных частотах между нашим кварцевым резонатором (27.456MHz) и тем что ожидает система в качестве опорной частоты для arch_timer (27.000MHz).

В качестве архитектурного таймера на платформах AArch64, принято использовать встроенный таймер в System Control Processor (CP15), частота которого равна опорной частоте микросхемы и, насколько мне известно, изменению не поддается. Покопавшись немного в исходных кодах ядра Linux, было установлено, что значение опорной частоты для arch_timer загружается из Device-Tree из параметра **clock-frequency** в структуре **xti_clk**. И действительно, заглянув в файл ***arch/arm64/boot/dts/elvees/mcom03.dtsi*** было обнаружено следующее:

```
xti_clk {
        #clock-cells = <0>;
        compatible = "fixed-clock";
        clock-frequency = <27000000>;
};
```

Решение проблемы было достаточно простым: добавить в файл ***mcom03.dtsi*** одноименную ссылку на структуру xti_clk:

```
xti_clk: xti_clk {
    ...
};
```

и уже в своём кастомном конфиге ***mcom03-RanetkaPC_Module-1.0.dts*** изменить значение опорной частоты на правильное:

```
&xti_clk {
        clock-frequency = <27456000>;
};
```

После перекомпиляции DTB и перезагрузки системы с новым конфигурационным файлом мы видим ожидаемый результат:

```
[root@altlinux ~]# dmesg | grep timer
[    0.000000] arch_timer: cp15 timer(s) running at 27.45MHz (phys).
[    0.013755] Calibrating delay loop (skipped), value calculated using timer
frequency.. 54.91 BogoMIPS (lpj=274560)
```

Замер по «настенным» часам показал очень точное вычисление системного времени, а значит можно приступать к различным тестам нашего процессорного модуля.

17. Тест производительности подсистемы памяти

Наиболее простой тест, который можно произвести в Unix системе, не имеющей ничего, кроме набора стандартных утилит — это тест производительности памяти, с помощью утилиты ***dd***. В вызове утилиты ***dd*** попросим её подготавливать и заполнять нулями блоки памяти большого объеме и посмотрим с какой скоростью это происходит:

```
[root@altlinux ~]# dd if=/dev/zero of=/dev/null bs=819200 status=progress
125371187200 байт (125 GB, 117 GiB) скопирован, 27 s, 4,6 GB/s
154218+0 записей получено
154218+0 записей отправлено
126335385600 байт (126 GB, 118 GiB) скопирован, 27,2066 s, 4,6 GB/s
```

Как видно, подсистема динамической памяти обеспечивает пропускную способность в **4.6 GB/s** (гигабайт в секунду). Напомню, что скорость работы микросхемы памяти LPDDR4 на нашем процессорном модуле на данный момент составляет чуть более 1600 MT/s (миллионов передач в секунду), а тактовая частота вычислительных ядер ARM составляет 1.4 GHz.

Чтобы понять, насколько полученная цифра хороша или плоха, приведу результат этого же теста со своего рабочего ноутбука с микропроцессором AMD Ryzen 5 с тактовой частотой 3.3 GHz и LPDDR4 памятью, работающей на скорости 3200 MT/s в режиме **interliveing** под управлением ОС FreeBSD 13.1:

```
rz@butterfly:/home/rz $ sudo dd if=/dev/zero of=/dev/null bs=819200
status=progress
653813350400 bytes (654 GB, 609 GiB) transferred 15.003s, 44 GB/s
829191+0 records in
```

```
829191+0 records out
679273267200 bytes transferred in 15.604429 secs (43530798652 bytes/sec)
```

Разница почти в 10 раз. Безусловно, данный тест весьма примитивен, скрывает ряд нюансов и не может претендовать на точность, тем не менее, он позволяет быстро оценить производительность подсистемы оперативной памяти.

Не будем останавливаться на достигнутом и произведем замер производительности оперативной и кэш памяти с помощью теста **STREAM Benchmark**. Данный тест производит вычисление пропускной способности подсистемы оперативной памяти, а так же расчет векторов различной длины, чтобы выявить влияние «объема задачи» на скорость её вычисления, оценивая размер и пропускную способность подсистемы кэш памяти. Тест STREAM Benchmark имеет реализацию на двух языках — на языке Си и на языке FORTRAN. Более детально о тесте STREAM Benchmark можно прочесть на http://www.cs.virginia.edu/stream/ref.html

Выкачаем исходный код тест STREAM на рабочую Linux машину с Github-а методом клонирования и подготовим среду для кросс-компиляции:

```
rz@devbox:~$ git clone https://github.com/jeffhammond/STREAM.git
rz@devbox:~$ cd STREAM/
rz@devbox:~$ . /opt/setenv_gcc8_arm64.sh
```

Подредактируем ***Makefile*** — уберем из него сборку версии теста, написаной на языке FORTRAN, т.е. удалим вызов цели **stream_f.exe** и запустим сборку:

```
rz@devbox:~/STREAM$ make
```

В результате получим файл ***stream_c.exe***. Для удобства переименуем его в ***stream_c-arm64.exe*** и скопируем его на SD-карту с ОС AltLinux, прямо в раздел boot (FAT32). Осуществим загрузку процессорного модуля с этой карты, как показано в главе *14. Загрузка ОС AltLinux*.

Подключим терминал к процессорному модулю, войдем в систему, получим права супер-пользователя, подмонтируем первый раздел SD-карты (boot раздел, имеет тип FAT32), на котором у нас находится бинарник с тестом STREAM и запустим тест:

```
[root@altlinux ~]# mount /dev/mmcblk1p1 /mnt
[root@altlinux ~]# /mnt/stream_c-arm64.exe
```

Результат выполнения теста STREAM выглядит следующим образом:

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 4
Number of Threads counted = 4
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 51665 microseconds.
   (= 51665 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:            4018.3     0.040066     0.039818     0.040534
Scale:           4038.6     0.040074     0.039618     0.040634
Add:             3528.1     0.068950     0.068026     0.070036
Triad:           3751.0     0.065299     0.063983     0.066056
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Из финальной таблицы видно, что скорость копирования данных из памяти и обратно составляет **4018.3 MB/s** (мегабайт в секунду), что достаточно

близко к показаниям теста, полученного утилитой ***dd***. Утилита ***dd*** выдает несколько больший результат, вероятно из-за того, что производит запись в одну и ту же область памяти, что ускоряется кэшем. Но это не точно.

Ниже для сравнения приведена сводная таблица выполнения тестов STREAM на различных платформах, источник:
https://forums.raspberrypi.com/viewtopic.php?t=271121

	Copy	Scale	Add	Triad
PIII 650MHz	175.2	176.1	203.6	200.8
Athlon 1400MHz	507.1	634.1	692.8	692.7
Pi B+	738.7	222.2	306.5	293.4
Pi Zero	838.5	290.5	398.4	385.7
Pi 2B v1.1	2021.9	1041.3	739.0	581.9
Pi 3B+	**2683.8**	2500.2	2216.0	1962.2
Pi 4B 2GB	**5311.6**	5581.7	4562.3	4380.8
Pi 4B 4GB	5511.8	5531.7	4835.8	4829.5
Odroid N2	6868.2	6854.6	6665.7	6715.2
Jetson Nano	7170.4	6753.6	7145.0	7015.7
i3 550 3.2GHz	7992.4	7838.9	8747.0	8692.6
Xeon Gold 6162	10211.3	11678.0	13283.3	13097.5
Ryzen7 1700	28368.6	17044.8	18710.6	18770.3

Из таблицы видно, что производительность подсистемы памяти на нашем процессорном модуле существенно выше, чем у одноплатного ПК Raspberry Pi модели 3B+ (2673.8 MB/s), но отстает от модели 4B (5311.6 MB/s). Есть все основания полагать, что подняв скорость работы микросхемы памяти LPDDR4 до заявленных в даташите 3200 MT/s, по этому показателю наш процессорный модуль сможет превзойти Raspberry Pi 4B.

18. Тест производительности CPU

Следующий тест который мы провели — это тест **CoreMark Benchmark**. Позволю себе несколько слов об этом тесте.

Тест CoreMark был разработан в конце 2000-х годов и предложен в качестве промышленного стандарта для измерения производительности вычислительных ядер (CPU), взамен широко использовавшегося тогда теста Dhrystones. Тест содержит набор алгоритмов, широко используемых в современном программном обеспечении, в том числе манипуляции с матрицами, работа со связанными списками, алгоритмы конечных автоматов,

расчет CRC и т.д. Все это «перемешано» в определенных пропорциях в виде одного большого теста. CoreMark написан на «чистом Си», без использования каких либо библиотек, за исключением stdlib, поэтому может быть легко портирован и скомпилирован почти для любой вычислительной платформы, для которой существует компилятор с языка Си.

CoreMark выдает результат в виде неких единиц (scores) которые могут быть рассчитаны для одного или нескольких вычислительных ядер и нормированы к тактовой частоте — мегагерцам (по аналогии с DMIPS у теста Dhrystones). Более подробно о тесте CoreMark можно прочесть в статье:

https://en.wikipedia.org/wiki/Coremark

Традиционно выкачаем исходные коды теста Coremark с Github-а методом клонирования на рабочую машину с ОС Linux и подготовим среду для кросс-компиляции в AArch64:

```
rz@devbox:~$ git clone https://github.com/eembc/coremark.git

rz@devbox:~$ . /opt/setenv_gcc8_arm64.sh
```

Соберем тест сначала для одного ядра:

```
rz@devbox:~$ cd coremark
rz@devbox:~/coremark$ make link
rz@devbox:~/coremark$ cp coremark.exe  coremark-arm64.exe
```

Переименуем результирующий бинарник ***coremark.exe*** в файл с другим именем ***coremark-arm64.exe*** и соберем еще один, но уже для четырех ядер:

```
rz@devbox:~/coremark$ make clean
rz@devbox:~/coremark$ make XCFLAGS="-DMULTITHREAD=4 -DUSE_FORK" link
rz@devbox:~/coremark$ mv coremark.exe coremark-arm64-mt.exe
```

Переименуем результирующий бинарник из **coremark.exe** в **coremark-arm64-mt.exe** — все это исключительно для удобства.

Скопируем полученные бинарники на SD-карту с ОС AltLinux прямо в раздел boot и осуществим загрузку процессорного модуля с этой карты, как показано в главе *14. Загрузка ОС AltLinux*.

Примечание. При сборке указывается цель (target) **link**, что производит компиляцию и линковку исполняемого файла, но не запуск самого теста.

Подключим терминал к процессорному модулю, войдем в систему, получим права супер-пользователя и подмонтируем первый раздел SD-карты, на котором у нас расположены бинарники с тестом Coremark:

```
[root@altlinux ~]# mount /dev/mmcblk1p1 /mnt
```

Запустим сначала тест для одного ядра:

```
[root@altlinux ~]# /mnt/coremark-arm64.exe

2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 13198
Total time (secs): 13.198000
Iterations/Sec   : 4546.143355
Iterations       : 60000
Compiler version : GCC8.2.1 20180802
Compiler flags   : -O2 -DPERFORMANCE_RUN=1  -lrt
Memory location  : Please put data memory location here
                   (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xbd59
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 4546.143355 / GCC8.2.1 20180802 -O2 -DPERFORMANCE_RUN=1  -
lrt / Heap
```

и следом тест для четырех ядер:

```
[root@altlinux ~]# /mnt/coremark-arm64-mt.exe

2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 13252
Total time (secs): 13.252000
Iterations/Sec   : 18110.473891
Iterations       : 240000
Compiler version : GCC8.2.1 20180802
Compiler flags   : -O2 -DMULTITHREAD=4 -DUSE_FORK -DPERFORMANCE_RUN=1  -
lrt
Parallel Fork : 4
Memory location  : Please put data memory location here
                   (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[1]crclist       : 0xe714
[2]crclist       : 0xe714
```

```
[3]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[1]crcmatrix     : 0x1fd7
[2]crcmatrix     : 0x1fd7
[3]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[1]crcstate      : 0x8e3a
[2]crcstate      : 0x8e3a
[3]crcstate      : 0x8e3a
[0]crcfinal      : 0xbd59
[1]crcfinal      : 0xbd59
[2]crcfinal      : 0xbd59
[3]crcfinal      : 0xbd59
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 18110.473891 / GCC8.2.1 20180802 -O2 -DMULTITHREAD=4 -
DUSE_FORK -DPERFORMANCE_RUN=1  -lrt / Heap / 4:Fork
```

Цифры которые нас инересуют это

```
Iterations/Sec   : 4546.143355
```

для одного ядра, и

```
Iterations/Sec   : 18110.473891
```

для всех четырех.

Видно, что присутствует почти линейное увеличение производительности от количества задействованных вычислительных ядер.

Пересчитаем эти цифры в индекс CoreMark/MHZ, то есть разделим на частоту в мегагерцах:

```
4546.143355 / 1400 = 3.24724525357 (single core)
18110.473891 / 1400 = 12.9360527793 (multi core)
```

Данные показатели близки к заявленной «по даташиту» производительности на СнК «СКИФ», которая составляет **11.5** Coremarks/MHZ (multi core).

Для сравнение приведу аналогичные показатели от нескольких популярных моделей одноплатных ПК Raspberry Pi:

Raspberry Pi 3 BCM2837B0 @ 1.2GHz:

В абсолютных значениях: 3841 / 15364

В приведенных на MHZ: 3.20 / 12.80

Raspberry Pi 4 BCM2711 @ 1.5GHz:

В абсолютных значениях: 8066 / 33072

В приведенных на MHZ: 5.37 / 22.048

Если сравнивать наше изделие с Raspberry Pi 3 в единицах приведенных на MHZ, то СнК «СКИФ» в них имеет почти такую же производительность, как и СнК BCM2837B0 на плате Raspberry Pi 3, что неудивительно, так как оба СнК содержат одинаковые вычислительные ядра общего назначения ARM Cortex-A53 и одинаковое их количество. Но если говорить об абсолютных значениях, то СнК «СКИФ» на нашем процессорном модуле показывает производительность на 18% больше чем СнК на плате Raspberry Pi 3, за счет более высокой тактовой частоты.

Одной из задач для дальнейших исследований для нас — это попытаться разогнать СнК «СКИФ» до частоты 1.8ГГц или даже предельных 2.0ГГц, оценить стабильность работы, потребляемую и рассеиваемую мощность.

19. Устройство процессорного модуля «RanetkaPC»

Как я уже отмечал, разработку нашего одноплатного ПК мы начали с разработки его упрощенного варианта в виде процессорного модуля, по возможности с минимальным «обвесом» и минимальной площадью. За форм-фактор модуля взяли уже имеющийся у нас формат: плата размером **70x56мм** с расположенными по периметру 122 торцевых металлизированных контактных площадок, позволяющих монтировать модуль на несущую плату, как методом пайки, так и путем установки на штыревые разъемы, для чего в контактных площадках модуля предусмотрены отверстия (см. рисунок 4 ниже).

Рис. 4. Процессорный модуль «RanetkaPC» версии 1.0, инженерный образец. ООО «Фабмикро», Февраль 2022.

Помимо собственно СнК «СКИФ», процессорный модуль «RanetkaPC» содержит:

1) Одну микросхему синхронной динамической памяти стандарта LPDDR4, подключенную на контроллер DDRMC0 СнК по 32-битной шине (2x16). На момент написания данной статьи, производителями LPDDR4 памяти выпускались микросхемы объемом 1ГБ, 2ГБ и 4ГБ. На инженерном образце установлена микросхема MT53D512M32D2 производства Micron, объемом 2ГБ.

2) Одну микросхему NOR flash памяти, подключенную к порту QSPI0. На инженерном образце установлена микросхема W25Q32, объемом 4МБ. Данная микросхема памяти предназначена для хранения загрузчика SBL и загрузчика U-Boot.

3) Одну микросхему NAND flash памяти стандарта eMMC 5.0, подключенную к порту SDMMC0. На инженерном образце установлена микросхема SDINBDG4-8G-I2 производства SanDisk, объемом 8ГБ. На данный носитель может быть установлена ОС общего назначения.

4) Микросхему аудиокодека TLV320AIC3110 для преобразования интерфейса I2S0 в аналоговой одноканальных вход и стерео выход звуковой частоты, с частотой дискретизации до 192КГц.

5) Микросхему АЦП TSC2046E, предназначенную преимущественно для подключения 4-х проводной резистивной сенсорной панели (RTP), либо функционирующей как низкоскоростной двухканальный АЦП общего назначения.

6) Систему обеспечения питания на базе микросхемы ADP5054A, содержащей четыре настраиваемых (но не программируемых) DC/DC преобразователя.

7) Кварцевый резонатор с частотой 27.456МГц, для задания опорной частоты СнК.

На внешние контакты модуля выведены сигнальные линии следующих интерфейсов СнК:

- порт SDMMC0 для подключения внешней SD-карты или микросхемы eMMC;

- два порта последовательной шины USB0 и USB1;
- три последовательных порта UART0, UART1 и UART2;
- порт шины CAN0;
- три двухпроводные шины I2C0, I2C1 и I2C2;
- сигналы RGMII порта EMAC0;
- сигналы DPI порта MIPI DSI;
- сигналы DVP порта MIPI CSI;
- порт PCIe0 (2 lane);
- Порт JESD0;
- 16 линий GPIO;
- линии АЦП для подключение резистивной сенсорной панели;
- аналоговый вход и выход звуковой частоты;
- ряд служебных сигналов.

20. Разработка печатной платы процессорного модуля «RanetkaPC»

С 2019 года, все свои разработки РЭА мы полностью ведем в свободном программном комплексе для проектирования печатных плат KiCAD. Причин для этого несколько, основная конечно же — финансовая. Стоимость рабочего места таких САПР как Altium Designer или Cadence OrCAD/Allegro попросту неподъемная для небольшой фирмы, «не хватающей звезд с неба». У нас имеется приобретенная постоянная лицензия на Cadence OrCAD, но набор функционала катастрофически ограничен. В то же время, KiCAD за последние годы продемонстрировал серьезный прогресс и его последняя версия 6.0.2, на мой взгляд, ничем не хуже того-же Altium-a.

KiCAD позволяет вести разработку печатных плат, содержащих до 32 слоёв металлизации, имеет гибкую систему правил и ограничений (constraints), позволяет трассировать дифференциальные пары, СВЧ и высокочастотные цепи

с автоматическим расчетом длины цепи и импеданса, имеется возможность выравнивания цепей. KiCAD имеет функцию проверки правил и ограничений (DRC), имеет связку с симулятором цепей NGSpice, содержит встроенный просмоторщик Gerber-ов с некоторыми элементами редактора. Помимо этого, KiCAD «из коробки» содержит обширную библиотеку электронных компонентов и футпринтов к ней и почти все, с чем приходилось сталкиваться, присутствует в этой библиотеке.

Еще одна причина перехода на KiCAD — принципиальный отказ от Windows и остального проприетарного софта. На моём рабочем месте установлена ОС FreeBSD, мои коллеги используют AltLinux. По мимо этого есть желание участвовать в открытых проектах (OSHW), где KiCAD уже занял прочные позиции. Есть намерения опубликовать всю КД на разработанный нами процессорный модуль.

Процессорный модуль «RanetkaPC» был разработан в САПР KiCAD 5.1. Разработка началась в июле 2021 года и к концу сентября у нас уже были готовы Gerber-файлы для отправки на производство. В процессе разработки печатной платы не раз вставал вопрос: где её изготавливать и под какие нормы подгонять, однако жизнь диктовала свои требования — выйти из под микросхемы 1892ВА018 с шагом 0.5мм было не так просто. В первом приближении у нас получилось 10 слоев металлизации с шириной проводника 0.075/0.075мм.

Так как мы в основном работаем с зеленоградским Резонитом, то первый дизайн платы отправили им на просчет и тут же получили «от ворот поворот» — наша плата не проходила по ряду технологических требований. Тут нас посетила мысль — не отправить ли плату на и изготовление в Китай, ведь «все так делают», пора уже и нам осваивать эту схему производства. В июле 2021 года я отправлял на пробный просчет дизайн платы другого нашего процессорного модуля с подобной топологией в одну новосибирскую компанию (не хочу приводить её название), которая занимается предоставлением посреднических услуг по производству печатных плат на китайских заводах. Тогда я получил вполне сносное коммерческое предложение — в районе 14 тысяч рублей за серию из 10 плат аналогичного размера. В общем, отправили им нашу «Ранетку», а тем временем принялись подгонять дизайн под требования Резонита — после общения с резонитовским технологом возникло ряд мыслей, как можно «загрубить» топологию.

Получив КП от новосибирской посреднической компании, мы слегка прифигели — за 10 плат «ранетки» с нас попросили более 40 тысяч рублей, со сроком поставки 4 недели. На вопрос — «почему два месяца назад было в 3 раза дешевле» ответ был примерно такой: «во-первых, не были учтены ряд моментов, во-вторых, у китайского завода была промоакция, которая закончилась». Вот такие дела!

Немного подумав, решили что экономии почти никакой нет, а риски с Китаем непонятные (в случае дефекта замаешься претензии писать), поэтому стали налегать на доработку под требования Резонита. Пришлось увеличивать размер переходных отверстий и размер проводника, в итоге вышли на следующие топологические нормы: 10 слоёв металлизации, минимальная ширина проводника составила 0.1мм с зазором 0.075мм, а диаметр минимального переходного отверстия — 0.3мм, с отверстием под сверло 0.1мм.

На рис. 5а и 5б ниже проиллюстрированы слои металлизации TOP и BOTTOM уже доработанной версия платы (V1.1). На слоях просматривается трассировка сигнальных линий LPDDR4, а так же «поле» под микросхемой СнК «СКИФ» усеянное переходными отверстиями.

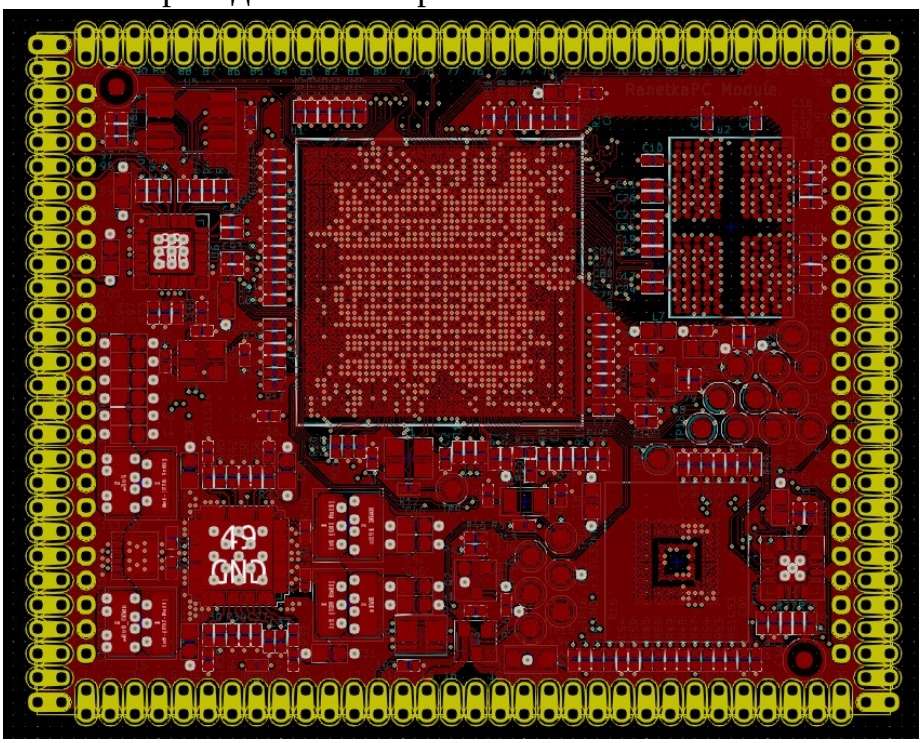

Рис. 5а. Печатная плата процессорного модуля «RanetkaPC».

Слой металлизации TOP.

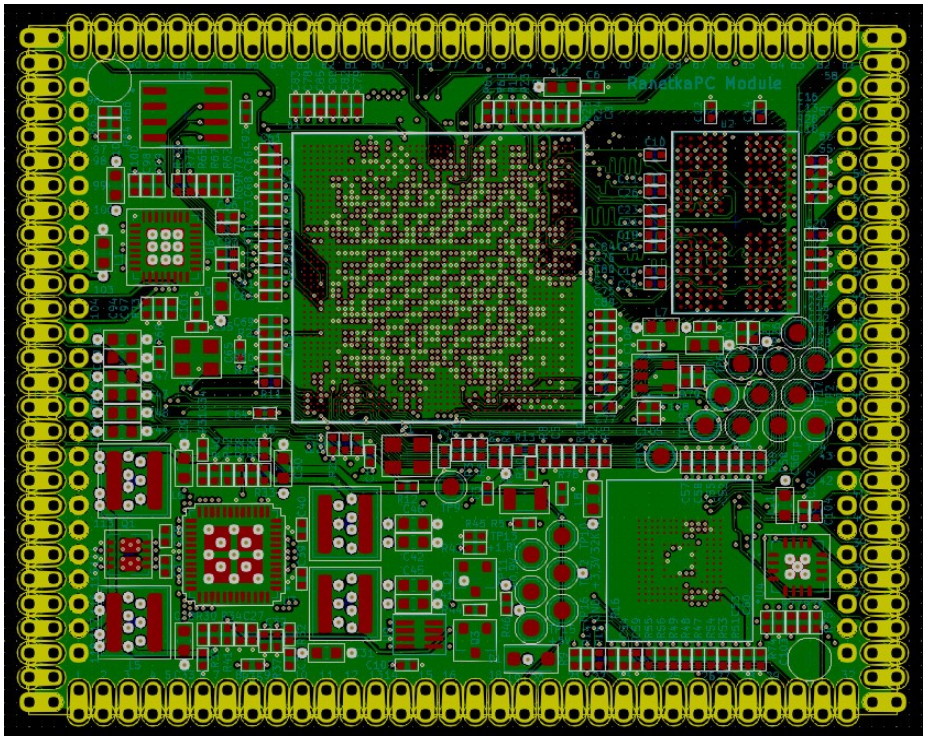

Рис. 5б. Печатная плата процессорного модуля «RanetkaPC».

Слой металлизации BOTTOM.

Резонит принял плату к срочному производству и выставил нам счет со сроком производства 8 рабочих дней. В счете от Резонита порадовал параметр «сложность», указанный как «предельная».

Срок исполнения: 8 рабочих дней с момента поступления оплаты.

№	Наименование товара	Кол-во	Ед.изм.	Цена	Сумма (руб.)
1	Печатная плата RanetkaPC_Module_V1.0	2	шт	18707.35	37414.70
	Новая; 70x56мм; Фрез;FR4-1,36-18мкм; 1021; маска - Зел/Зел маркировка - Бел; ImmersGold; E-test; Переходные отв : <Закрыты маской>; Сложность-Предельная;				

Итого без НДС :	**37414.70**
Итого НДС 20 % :	**7482.94**
Всего к оплате :	**44897.64**

Отгрузка: доставка включена в стоимость

Всего к оплате : Сорок четыре тысячи восемьсот девяносто семь рублей 64 копейки, в т.ч.: НДС - Семь тысяч четыреста восемьдесят два рубля 94 копейки.

Рис. 6. Фрагмент счета на оплату производства печатных плат модуля «RanetkaPC». Сложность-Предельная.

Пока платы изготавливались, к нам начали поступать заказанные ранее электронные компоненты по этому проекту. Проблем тут никаких не возникло, кроме одной — отсутствие самой микросхемы СнК 1892ВА018. Небольшую партию этих СнК мы проплатили в сентябре и «Элвис» обещал нам поставить их к концу октября 2021г, но что-то видимо пошло не так — микросхемы задержались. Посовещавшись, мы решили, что останавливаться не стоит, и как только поступили платы от Резонита, тут же запустили сборку одного инженерного образца без установки микросхемы СнК. В самом конце октября 2021г у нас уже был получен первый инженерный образец процессорного модуля, правда без самого его сердца — без процессора. :-)

Отсутствие СнК на плате позволило нам протестировать систему питания, проверить на выносливость микросхему DC/DC преобразователя ADP5054A и микросхему силовых транзисторов IRFHM8363, в паре с которой должна работать ADP5054A. Мы убедилось в том, что примененное техническое решение годное, DC/DC может стабильно выдавать ток в 4.5A по шине питания 0.9V, что требовалось для питания ядра СнК согласно «Руководству». Проверили еще несколько узлов и на этом изыскания остановились.

Таким образом, с середины ноября 2021г мы фактически остановили работы над проектом из-за отсутствия микросхем 1892ВА018 СнК «СКИФ». Разумеется «Элвис» обещал вот-вот отправить нам эти заветные микросхемы, но их всё не было и не было, в какой то момент мы даже начали терять надежду — денег отдали за них не мало, по 23 000 рублей за штуку, переживать было за что. Но чудо случилось, микросхемы прибыли к нам в январе 2022 года, сразу после новогодних праздников и таким образом у нас случился еще один праздник!

21. «Анбоксинг» микросхем 1892ВА018

Получив микросхемы мы первым делом осуществили «анбоксинг». :-)

Микросхемы прибыли к нам в картонной коробочке (рис. 7а), каждая микросхема располагалась в индивидуальном антистатическом пакетике, который был помещен в еще один пакет с наклейкой. К микросхемам

прилагался листок, озаглавленный как «Этикетка» (рис. 7б) за подписью главного конструктора, на этикетке указаны ТУ **АЕНВ.431280.469ТУ** и дата производства **2128** (28-я неделя 2021г).

Рис. 7а. Упаковка с образцами микросхем 1892ВА018 СнК «СКИФ».

Рис. 7б. Содержимое упаковки: микросхемы 1892ВА018 и этикетка к ним.

22. Финальная сборка процессорного модуля «RanetkaPC»

После «анбоксинга» встал вопрос как действовать далее — устанавливать ли микросхему СнК на пасту, а это означало полную сборку еще одной платы в станке, или же сэкономить время и детали, и установить микросхему на уже собранную плату с помощью «ремонтной» ИК паяльной станции. Во втором варианте отпугивало то, что огромное количество «шаров», следующих с мелким шагом, может плохо запаяться на ИК станции. Так же смущала металлическая крышка на микросхеме, которая может отражать часть ИК излучения и не позволить шарам прогреться и, соотвественно, качественно оплавиться.

Желание поскорее включить и начать работу с процессорным модулем преодолело наши страхи. Наш главный инженер-конструктор взял в руки недопаянную, но уже частично протестированную плату процессорного модуля,

микросхему СнК, смазал обе «детали» тонким слоем флюса Flux Plus EFD 6-412-A так, чтобы не было излишков, которые могут «вскипать» в процессе пайки и тем самым сместить микросхему со своего посадочного места, совместил их «на глаз», установил в ИК станцию Achi IR12000 и запустил процесс. Через 15 минут плата была полностью готова.

Осмотр платы под микроскопом после монтажа СнК, в том числе заглядывание в торец микросхемы, не выявил каких либо видимых проблем — все просматривающиеся крайние ряды шаров стояли так, как было им положено — ровно и слегка приплюснуто, что указывало на наличие паяного соединения. Тем не менее, хотелось убедиться в том, что под микросхемой не образовалось короткого замыкания между шарами. Подача напряжения питания на плату с таким дефектом могла бы привести к моментальному выходу из строя дорогостоящего СнК, причем это могло произойти незаметно, а это масса потерянного времени и сил при попытке запустить такое изделие. Немного подумав, мы решили свозить плату к стоматологам и сделать ей «рентген».

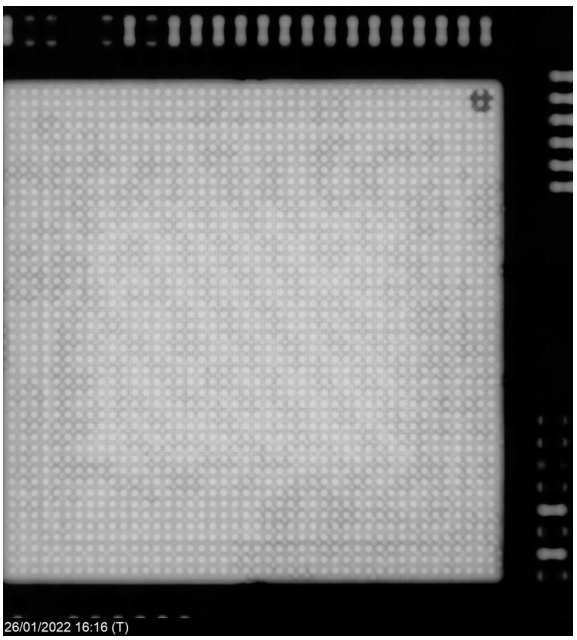

Рис. 8. Изображение смонтированной на плате процессорного модуля микросхемы СнК «СКИФ», полученное на сенсоре стоматологического рентгеновского аппарата.

Изображение полученное со «стоматологического» рентгеновского аппарата (рис. 8), конечно же не идет ни в какое сравнение с профессиональными системами компьютерной томографии, применяемыми для неразрушающего контроля РЭА, но тем не менее, на нём отчетливо видны все шары и отсутствие КЗ между ними. Металлическая крышка на микросхеме СнК сильно «аттенюирует» излучение и не позволяет рассмотреть точность совмещения шаров с контактными площадками, но этого и не требовалось.

Так как при установке микросхемы СнК использовался безотмывочный флюс, то решили плату лишний раз в ультразвуковую ванну не опускать, а приступать к первому включению, процесс которого описан выше в главе *8. Первый запуск СнК «СКИФ»*.

23. Проблемы запуска процессорного модуля «RanetkaPC» и СнК «СКИФ»

Разумеется, при запуске процессорного модуля не все шло так гладко, как описано в статье выше. Далее я опишу только самые проблемные места, с которыми нам пришлось столкнуться.

Запаяв перемычки BOOT в положение **0b011** и подав питание на модуль, мы сразу получили «уши» от ROM монитора СнК «СКИФ» в порту UART0. Это сильно воодушевило! Воспользовавшись исходным кодом программы «**hello-world»,** мы протестировали различные выводы GPIO и традиционно «поморгали светодиодом». Далее встал вопрос прошивки SBL в NOR flash, и тут нас ждала первая засада — утилита **«flasher»** отказывалась «видеть» микросхему NOR flash памяти на порту QSPI0. После нескольких часов возни с осциллографом и детального изучения «Руководства», мы выяснили, что порт QSPI0 может работать только с сигналами уровня 1.8V, в то время как используемая нами микросхема W25Q32JVS требует уровней 3.3V (VDDmin = 3.0V, Vih = 0.7*VDD). Надо заметить, что порт QSPI1 вполне способен работать как при 1.8V, так и при 3.3V. Данный факт был зафиксирован в «Руководстве», но «очень мелким шрифтом», который мы проглядели при разработке схемы модуля. Чтобы двигаться далее, нам пришлось приподнять вывод питания у микросхемы NOR flash и запитать его от навешанного рядом LDO на 2.65V. Это сработало — микросхема стала отзываться, что позволило успешно прошить в неё SBL.

Как только стал загружаться SBL, тут же наткнулись на следующую проблему — DDRINIT не видит микросхему LPDDR4 памяти на порту DDRMC0. Пару дней было убито на то, чтобы перебрать различные варианты настроек таймингов порта. Делается это при компиляции DDRINIT путем запуска команды ***make menuconfig***. Результата это не дало, что наводило на следующие мысли: либо память запаяна с дефектом, либо есть ошибки в схемотехнике или в топологии. Просвечивание микросхемы памяти «рентгеном» не проявил каких либо очевидных дефектов монтажа, что совсем уже опечалило. Но в процессе изучения исходников DDRINIT обнаружилась полезная функция — вывод диагностической информации (печать в последовательный порт всего обмена с контроллером DDRMC0). Включив вывод диагностики стало понятно, что микросхема памяти работает и даже отвечает на запросы, только вот с самими запросами «что-то не так». Мы обратились в службу техподдержки «Элвиса» и на следующий же день получили небольшой патч для DDRINIT. Патч сработал, DDRINIT стал детектировать микросхему памяти, правильно определять её объем и даже

проводить трейнинг трансивера на самой максимальной скорости — 3200 MT/s. Позже выяснилось, что DDRINIT лукавил, и на такой скорости работы памяти ядро Linux подвисает в процессе загрузки. Но процесс был сдвинут мертвой точки и мы уже имели возможность созерцать U-Boot.

С U-Boot-ом обнаружились сразу две проблемы. Как только U-Boot загружался, мы тут же фиксировали резкое увеличение потребляемого модулем тока, а микросхема СнК при этом начинала сильно разогреваться. До этого мы запускали модуль без радиатора на СнК, поэтому решили приклеить радиатор и проверить с ним — ситуация несколько улучшилась, но вопрос с лишней потребляемой мощностью сильно смущал. Еще раз связались с технической поддержкой «Элвиса», объяснили ситуацию и через пару дней получили очередной патч — для U-Boot, который отключал невостребованные подсистемы внутри СнК (в частности — прожорливый SDR).

Другая проблема состояла в том, что U-Boot детектировал только 1GiB динамической памяти из 2GiB доступных. Немного полистав исходный код U-Boot и TF-A, мы обнаружили искусственное программное ограничение, установленное в 1GiB. Задали вопрос в «Элвис» и получили ответ — в микросхеме СнК есть проблема с DDRMC контроллерами, не позволяющая полноценно задействовать всю доступную динамическую память. Со слов их инженера, данная проблема «сейчас решается» и скорее всего будет устранена программным методом. Но это не точно.

Объема ОЗУ 1GiB вполне достаточно для загрузки ОС Linux, поэтому стали двигаться дальше. И тут нас ждала следующая засада — ядро Linux произвольно зависало в различных точках при инициализации аппаратуры. Сначала грешили на «перегрев», сделали замер температуры и пришли к выводу, что дело не в перегреве. Решили понижать скорость работы LPDDR4 и таким образом выяснили, что Linux стабильно работает только при скоростях DDR 1600 MT/s и ниже. Возникло подозрение на то, что для увеличения скорости обмена необходимо повышать напряжение питания ядра и памяти. Решили оставить это исследование «на потом».

Запустив на процессорном модуле AltLinux и немного осмотревшись, обнаружили проблему с системным временем (с arch_timer) которая описана в главе *16. Проблема с архитектурным таймером.* Чтобы понять, что происходит и почему у нас «спешат» системные часы, ушло еще пару дней.

После устранения проблемы с системны временем, перед запуском тестов производительности, решили выяснить, на какой всё же реальной частоте у нас работают вычислительные ядра и каков её предел. Углубляясь в эту «кроличью нору» выяснили, что в Linux-е для MCOM03 отсутствует «гувернер», т.е. механизм для динамического управления частотой вычислительных ядер и напряжением питания попросту не имплементирован, а частота задана

фиксированным значением настроек PLL, которые прописаны в исходниках U-Boot. Я быстро набросал патч для U-Boot, который позволял задавать настройки PLL через параметр частоты в ***.config*** файле U-Boot-а. Этот патч позволил проверить работоспособность СнК «СКИФ» на различных частотах. Предельной стабильной в нашем случае оказалась частота **1400256000** Гц (PLL множитель: 50) при установленном на плате кварцевом резонаторе в **27.456** МГц. К слову сказать, Linux загружается и на частоте 1.5ГГц, но при увеличении нагрузки начинает произвольно подвисать или «паниковать». Скорее всего, так же не достаточно напряжения питания, как и в случае с динамической памятью — будем исследовать этот вопрос отдельно. Очень хочется раскочегарить СнК хотя бы до 1.8ГГц.

Покончив с замерами производительности решили двигаться далее — попытаться подключить LCD дисплей с интерфейсом DPI (RGB24), соответствующий разъем и прочий обвес уже присутствовал на несущей плате. После долгого и изнурительного изучения исходных кодов для подсистемы **media**, исправлении нескольких откровенных багов, связанных с конфигурированием ряда clock-ов, нам удалось составить правильный DTS файл и получить на выходах DPI интерфейса требуемые сигналы. Но тут нас ждала еще она засада от разработчиков СнК. Засада состоит в том, что уровни выходных сигналов дисплейного интерфейса DPI привязаны к уровням входных сигналов камерного интерфейса CSI и составляют 1.8V (все сильно завязано на питание подсистемы **media**). Подавляющее большинство LCD дисплеев предназначены для работы с уровнями 3.3V, иногда мне встречались дисплеи с 2.5V, но я ни разу не видел дисплеев с 1.8V. Чтобы подключить дисплей, нам требуется вводить в схему преобразователи уровней (level shifter), а это радикальным образом портит весь наш замысел: все наши изделия, на которые мы предполагали устанавливать процессорный модуль «RanetkaPC», придется перепроектировать. Либо придется вводить преобразователи уровня на плату процессорного модуля, на котором не так много свободной площади под еще три или четыре микросхемы, что увеличит его стоимость и вероятно сильно снизит максимальную частоту развертки.

Еще одна проблема которую мы не смогли разрешить — это запуск часов реального времени (Real-Time Clock, RTC). На подавляющем большинстве современных СнК, оснащенных часами RTC, питание этого блока (шина Vbat) предусматривается от широкого диапазона напряжений — от 1.3V до 3.6V, либо одним фиксированным напряжением 1.3V. Сделано это для того, чтобы RTC можно было запитать напрямую от «часовой» батареи, либо от батареи через один DC/DC с низким током утечки. Как выяснилось постфактум, в СнК «СКИФ» все не так. По какой-то причине блок RTC в СнК «СКИФ» запитывается от шести выводов, причем на одну их часть подается напряжение

0.9V, а на другую — 1.8V. Почему так — не понятно. Попытки добиться разъяснение от службы технической поддержки «Элвиса» по этому вопросу не увенчалась успехом — инженер который отвечал мне по электронной почте банально процитировал содержимое «Руководства» сославшись на таблицу 51.24 (см. рис. 9). Если следовать букве «Руководства», то нам потребуется ввести в схему процессорного модуля целых два DC/DC преобразователя питающие линии BVDD, BAT_VDDPST и BAT_VDD от часовой батареи. Также мы не смогли получить информацию о том, какой ток потребляется по этим линиям, что важно знать при проектировании схем батарейного питания. В ближайшее время попытаемся еще раз зайти к «Элвису» с этим вопросом.

Таблица 51.24. Питание домена BAT

Название выводов	Вывод на корпусе	Назначение	Объединяемые линии питания	Номиналы напряжений, В	Число выводов
BVDD	AU31, AU32, AV31, AV32	Напряжение питания домена питания PD_BAT		0.9	
BAT_VDDPST	BA30	Напряжение питания периферии CMOS КП подсистемы bat_domain		1.8	
BAT_VDDO	AW28	Напряжение питания периферии LVDS КП подсистемы bat_domain		1.8	

Рис. 9. Фрагмент «Руководства» на микросхему 1892ВА018. Питание домена BAT.

24. Перспективы

На данный момент мы еще не завершили весь план тестирования нашего процессорного модуля «RanetkaPC» и СнК «СКИФ» в его основе, вероятно, что проявятся еще какие-то проблемы. Тем не менее, мы решили двигаться далее и приступили к разработке версии 1.1 платы модуля, которая учтет предыдущий опыт, т.е. будет содержать «работу над ошибками», а так же ряд нововведений. Одно из них - посадочное место под вторую микросхему памяти LPDDR4. Пообщавшись с потенциальными пользователями мы пришли к выводу, что максимальный объем ОЗУ равный 4ГБ, которого можно достичь одной микросхемой памяти, по современным меркам слишком мал. Также мы планируем ввести в схему питания программную регулировку напряжения ядра, что позволить разгонять вычислительные ядра по необходимости, и наоборот — снижать потребление мощности при простое или работе на низких частотах.

В целом, если закрыть глаза на проблемы с контроллерами динамической памяти DDRMC («Элвис» утверждает, что проблема будет разрешена), то СнК «СКИФ» производит очень хорошее впечатление. Производительность ядер ARM вполне на уровне современных импортных аналогов. Ядра DSP Elcore50 нам протестировать еще не довелось, но их наличие может оказаться очень

весомым дополнением при решении ряда задач по обработки видео и аудио сигналов, а так же в приложениях с ИИ.

Если у Вас имеются какие-то конструктивные предложение по процессорному модулю, есть интерес к его использованию в своих изделиях — прошу писать мне на электронную почту. Так же я готов ответить на вопросы по функционированию СнК «СКИФ» в меру своих скромных познаний.

С уважением,
Генеральный директор ООО «Фабмикро»
Залата Руслан Николаевич <rz@fabmicro.ru>

I. Ссылки на ресурсы по микросхеме 1892ВА018 «СКИФ» и процессорному модулю «RanetkaPC»

1) Репозиторий исходных кодов на Github:
https://www.github.com/elvees

2) Образы для прошивки SD и QSPI:
https://dist.elvees.com/mcom03/altlinux/2022.02/

3) Buildroot для плат MCom-03 BuB
https://dist.elvees.com/mcom03/buildroot/20220216/

4) Документация по программному обеспечению (HTML):
https://dist.elvees.com/mcom03/altlinux/2022.02/doc

5) Руководство по микросхеме 1892ВА018 (PDF):
https://box.elvees.com/index.php/s/WiJqGYs4EtQ7Tmr#pdfviewer

6) Схема внешних соединений процессорного модуля «RanetkaPC» V1.0:
http7s://www.fabmicro.ru/pub/RanetkaPC_Module/RanetkaPC_Module_V1.0-page-1.pdf

8) Схема расположения выводов процессорного модуля «RanetkaPC» V1.0:
https://www.fabmicro.ru/pub/RanetkaPC_Module/RanetkaPC_Module-V1.0.svg

9) 3D модель процессорного модуля «RanetkaPC» V1.0:
https://www.fabmicro.ru/pub/RanetkaPC_Module/RanetkaPC_Module_V1.0.step